

**Der Fakultät für Wirtschaftswissenschaften der
Universität Duisburg-Essen, Campus Essen, vorgelegte Dissertation
zum Erwerb des Grades Dr. rer. nat.**

Contribution Barriers to Open Source Projects

Dipl.-Inform., Dipl.-Math. Christoph Hannebauer,
geboren in Offenbach am Main

Datum der mündlichen Prüfung: 2016-10-19

Erstgutachter: Prof. Dr. Volker Gruhn
Zweitgutachter: Prof. Dr. Klaus Echtele

Abstract

Contribution barriers are properties of Free/Libre and Open Source Software (FLOSS) projects that may prevent newcomers from contributing. Contribution barriers can be seen as forces that oppose the motivations of newcomers. While there is extensive research on the motivation of FLOSS developers, little is known about contribution barriers. However, a steady influx of new developers is connected to the success of a FLOSS project.

The first part of this thesis adds two surveys to the existing research that target contribution barriers and motivations of newcomers. The first exploratory survey provides the indications to formulate research hypotheses for the second main survey with 117 responses from newcomers in the two FLOSS projects Mozilla and GNOME. The results lead to an assessment of the importance of the identified contribution barriers and to a new model of the joining process that allows the identification of subgroups of newcomers affected by specific contribution barriers.

The second part of the thesis uses the pattern concept to externalize knowledge about techniques lowering contribution barriers. This includes a complete categorization of the existing work on FLOSS patterns and the first empirical evaluation of these FLOSS patterns and their relationships. The thesis contains six FLOSS patterns that lower specific important contribution barriers identified in the surveys.

Wikis are web-based systems that allow its users to modify the wiki's contents. They found on wiki principles with which they minimize contribution barriers. The last part of the thesis explores whether a wiki, whose content is usually natural text, can also be used for software development. Such a Wiki Development Environment (WikiDE) must fulfill the requirements of both an Integrated Development Environment (IDE) and a wiki. The simultaneous compliance of both sets of requirements imposes special challenges. The thesis describes an adapted contribution process supported by an architecture concept that solves these challenges. Two components of a WikiDE are discussed in detail. Each of them helps to lower a contribution barrier. A Proof of Concept (PoC) realization demonstrates the feasibility of the concept.

Contents

Abstract	II
Contents	III
List of Definitions	VII
1 Introduction	1
1.1 Models of FLOSS Communities	1
1.1.1 The Onion Model	2
1.1.2 Joining Scripts	3
1.1.3 Steinmacher et al.'s Model	4
1.1.4 Technical Aspects of the Joining Process	5
1.2 Contributor Motivation	8
1.2.1 Theoretical Works	8
1.2.2 Empirical Works	8
1.2.3 Attractiveness	10
1.3 Contribution Barriers	10
1.3.1 Rationale of the Definition	11
1.3.2 Specific Contribution Barriers	12
1.3.3 New Developers in General Software Development Projects	13
1.3.4 Joining FLOSS Projects	14
1.3.5 Exploring Contribution barriers	15
1.3.6 Summary of Contribution Barriers	19
1.3.7 The Effect of Lowered Contribution Barriers	20
1.4 Research Goals	22
1.4.1 Identification of Contribution Barriers	23
1.4.2 Lowering Contribution Barriers	23
1.4.3 Wiki Development Environments	24
2 Contribution Barriers to FLOSS Projects	25
2.1 Exploratory Survey	25
2.1.1 Demography	26
2.1.2 Developers' Motivation	27
2.1.3 Contribution Barriers	30
2.1.4 Discussion	32
2.1.5 Conclusion	33
2.2 Main Survey	34
2.2.1 FLOSS Projects	35

Contents

2.2.2	Research Hypotheses	36
2.2.3	Participant selection	38
2.2.4	Demography	42
2.3	Contributor Motivation	46
2.3.1	Usage Motivation	46
2.3.2	Open Question on Modification Motivation	47
2.3.3	Closed Question on Modification Motivation	49
2.3.4	Open Question on Submission Motivation	50
2.3.5	Closed Question on Submission Motivation	52
2.3.6	Comparison to Related Work	53
2.4	Contribution Barriers	54
2.4.1	Open Question on Modification Barriers	55
2.4.2	Closed Question on Modification Barriers	56
2.4.3	Open Question on Submission Barriers	58
2.4.4	Closed Question on Submission Barriers	60
2.4.5	Discussion	61
2.5	Improvement Suggestions	63
2.5.1	Results	64
2.5.2	Discussion	66
2.6	Analysis	66
2.6.1	Language and Contribution Barriers	66
2.6.2	Experience and Contribution Barriers	67
2.6.3	Projects and Contribution Barriers	69
2.6.4	Own Need, Motivation, and Occupation	69
2.6.5	Motivation and Contribution Barriers	72
2.7	Threats to Validity	76
2.7.1	Construct Validity	76
2.7.2	Content Validity	77
2.7.3	Internal Validity	77
2.7.4	External Validity	79
2.8	A New Model for Joining FLOSS Projects	79
2.9	Conclusion	80
3	FLOSS Pattern Languages	82
3.1	Introduction	83
3.1.1	Classification of FLOSS Patterns	83
3.1.2	Related Work on Pattern Languages and Pattern Maps	84
3.1.3	Pattern Maps for the FLOSS Pattern Language	85
3.2	Categories of FLOSS Patterns	86
3.2.1	Integrate FLOSS Assets	89
3.2.2	Start a New Project from Scratch	91
3.2.3	Publish Closed Sources	92
3.2.4	Licensing	94
3.2.5	Architecture	96

3.2.6	Cultivate a User Community	98
3.2.7	Tools for New Developers	102
3.2.8	Empower the Most Suitable	105
3.3	Patterns Lowering Contribution Barriers	108
3.3.1	Low Contribution Barriers	109
3.3.2	Preconfigured Build Environment	113
3.3.3	Unit Tests for Contributors	117
3.3.4	Low-hanging Fruit	120
3.3.5	Bazaar Architecture	123
3.3.6	Exposed Architecture	128
3.4	Evaluation	132
3.4.1	Mozilla Firefox	133
3.4.2	Rack	138
3.4.3	Ruby on Rails	141
3.4.4	Spree	144
3.4.5	Rake	148
3.5	Discussion	151
3.5.1	FLOSS Patterns and the Success of FLOSS Projects	151
3.5.2	Validation of Relationships	154
3.5.3	Newly Observed Relationships	157
3.6	Conclusion	158
4	Wiki Development Environments	160
4.1	WikiDE Concept	160
4.1.1	Requirements	160
4.1.2	Existing Wiki Systems for Programming	165
4.1.3	WikiDE Contribution Process	169
4.1.4	Architecture	170
4.2	CI Services	175
4.2.1	Current Limits of FLOSS CI	176
4.2.2	Security Analysis of CI Systems	178
4.2.3	Secure Build Servers	182
4.2.4	Validation	186
4.2.5	Limitations	187
4.2.6	Related Work	188
4.2.7	Summary	188
4.3	Reviewer Recommendations	189
4.3.1	Related Work	190
4.3.2	Expertise Explorer	195
4.3.3	Empirical Evaluation	199
4.3.4	Results	201
4.3.5	Threats to Validity	206
4.3.6	Conclusion and Future Work on Reviewer Recommendations	208

Contents

4.4	Proof of Concept Realization	209
4.4.1	Environment Overview	209
4.4.2	Software Forge	209
4.4.3	Immediate Feedback	212
4.4.4	Reviewer Recommendations	214
4.4.5	Limitations	214
4.5	Summary	216
5	Conclusion	217
5.1	Identification of Contribution Barriers	217
5.2	FLOSS Patterns	218
5.3	Wiki Development Environments	219
5.4	Future Work	219
	Primary Sources	221
	Secondary Sources	236

List of Definitions

contribution barrier Definitions on pages 7, 10

contributor motivation Definition on page 7

FLOSS Definition on page 1

FLOSS Pattern Definition on page 82

FLOSS Project Definition on page 2

modification barrier Definitions on pages 7, 11

modification motivation Definition on page 7

multitenant CI system Definition on page 176

newcomer Definition on page 5

Own Need Definitions on pages 27, 47

social barrier Definition on page 16

Stupid Tax Definitions on pages 52

submission barrier Definitions on pages 7, 11

submission motivation Definition on page 7

usage barrier Definitions on pages 7, 11

usage motivation Definition on page 7

WikiDE Definitions on pages 24, 160

1 Introduction

Users of Free/Libre and Open Source Software (FLOSS) do not only receive an executable program but also its source code and thereby the freedom to modify the program to their demands. Thus, developers of FLOSS give away some of their power to the users without direct compensation. It might be astonishing that the FLOSS phenomenon even exists, on first sight against economic pressure. Nevertheless, some FLOSS projects like Linux [P*Lin15] and Apache [P*The15a] have been very successful and lead their respective markets [Gar15; QSu16].

Accordingly, the FLOSS phenomenon has also attracted research attention. One area of research focuses on the communities around FLOSS projects, as they drive development and marketing of the FLOSS project. This chapter outlines the current state of research on the FLOSS phenomenon in general and more specifically on how and why contributors join FLOSS projects. Furthermore, it derives own assumptions and models about FLOSS as the foundation for later chapters. Section 1.1 describes models that have been used in research to model FLOSS communities in general and more specifically how new contributors join a FLOSS project. Next, Section 1.2 summarizes the current state of research on motivations of FLOSS developers. Section 1.3 presents and discusses the research on contribution barriers to software projects, especially FLOSS projects. The chapter concludes with the research questions of this thesis in Section 1.4.

1.1 Models of FLOSS Communities

There are four different terms that mostly describe the same phenomenon: Free Software, Open Source Software, Free Open Source Software (FOSS), and Free/Libre and Open Source Software. The Free Software Foundation (FSF) uses the term Free Software to underline that its users have several freedoms [P*Fre15]. The Open Source Initiative (OSI) uses the term Open Source Software [P*Ope04] and underlines the superiority of its development model over traditional closed source software development. However, both definitions base solely on the license used for the software. Debates about the correct term are sometimes emotional and consequently the term FOSS tries not to take sides in the debate and instead tries to be a neutral term for the phenomenon. “Free” has a double meaning in English: First, it can refer to something available without cost. Second, it can refer to some kind of freedom. Although Free Software can often be acquired for free, the FSF refers to freedom. The term FLOSS tries to capture this by adding “libre” as an unambiguous reference to the freedoms of users of FLOSS. This thesis will use FLOSS without committing to any special ideology or license.

Since FLOSS licenses grant freedoms to modify and redistribute FLOSS, there is no legal entity that holds the exclusive development rights for any FLOSS. As an exception, there may be a legal entity holding the rights on the name and logos of a FLOSS artifact. Although there is

no legal reason to see any entity as the owner of a FLOSS artifact, there is usually a community around it: A maintainer manages modifications to the FLOSS artifact and distributes its source code and compiled versions to its users. This community and its organizational structure together with its technical infrastructure and the source code belonging to the FLOSS artifact shall be called a *FLOSS Project*. The source code of the FLOSS project belongs to a specific application or library, both of which are referred to as components in this thesis.

1.1.1 The Onion Model

Ye and Kishida [YK03] proposed that the community of a FLOSS project resembles an onion with its layers. Figure 1.1 depicts this model. The core of the community are maintainers and core developers who decide about the direction of the FLOSS project. This includes a vote on which modifications to the application will be accepted and which will be rejected. The next layer around the core developers are the co-developers who contribute source code to the FLOSS project. In contrast, the next layer of active users contribute only other types of resources to the FLOSS project, like documentation, translations, and technical support for other community members. The outermost layer are passive users who do not contribute to the FLOSS project but only use its application. Outside are all people who do not use the FLOSS project at all.

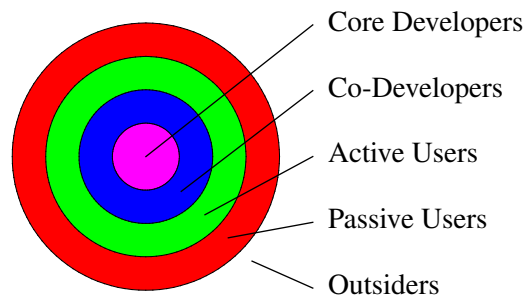


Figure 1.1: The Onion Model of FLOSS communities

The onion model assumes that an outsider moves into the FLOSS project layer by layer. Thus, the outsider first uses the application, then joins a mailing list as an active user, afterwards contributes code, and eventually the community may welcome the former outsider as a core developer.

Research on FLOSS commonly uses the onion model [Duc05; Cro+05; CH06b; Rul06; Mas+09; Gel10], often with small variations to the number of layers, but there is also criticism [JS07], especially that employed developers skip the active user layer and contribute code as their first action in the FLOSS project [Her+06]. Jergensen et al. [JSW11] also found empirical evidence holding against the onion model, but with another explanation: While the first FLOSS projects hosted their infrastructure on their own, today most FLOSS projects rely on software forge services [Rie+09] like SourceForge [P*Sla15] and GitHub [P*Git15c]. They speculated that while the onion model does not apply to single FLOSS projects hosted by a software forge

service, it may still be applicable to each software forge service as a whole.

The role migration in a software forge service as a whole has previously been analyzed using Markov Chains on the example of the SourceForge community. In the corresponding study [DR08a], the roles were defined differently to the onion model, though, distinguishing between developers and founders. One of their results implies that the role distribution on SourceForge was nearly stable in 2002: The fraction between gained and lost members is nearly equal for each role.

Another example of research on migration within the community of FLOSS projects is Robles and Gonzalez-Barahona's study of core developers [RG06]. They analyzed how the group of core developers of 19 FLOSS project changed over time and found three categories of behavior: In 3 FLOSS projects, there was a stable group of core developers. In 9 FLOSS projects, there was more fluctuation and the group of core developers changed. The 7 remaining FLOSS projects were a mix, with some developers being core developers over the whole study period and some others coming and going.

1.1.2 Joining Scripts

Von Krogh et al. [KSL03] introduced the concept of a FLOSS project's "joining script". The joining script is a process that new developers should follow in order to be granted commit access to the project's Version Control System (VCS). In terms of the onion model, this corresponds to the transition from active user to co-developer. Using the concept of the joining script to analyze the Freenet project, von Krogh et al. found that users' early mails to the mailing list allowed a significant prediction of whether the posters would eventually become co-developers or not: If the first post contained a bug description with accompanying code, the poster was more likely to become a co-developer; however, if the first post was just an introduction or suggestion for improvements, the likelihood that the poster would become a co-developer was lower.

This observation supports Raymond's first rule of Open Source development: "Every good work of software starts by scratching a developer's personal itch." [Ray00] Thus, software developers start to modify the source code of a FLOSS project if they experience a problem with the application that they want to fix and only then describe the problem and submit a patch.

Typically, the joining script is not fully documented. Thus, new developers need to learn the joining script via observation or trial-and-error. This learning can be cumbersome [JS07]. Von Krogh et al. analyzed only the Freenet project and did not claim that their analysis of Freenet's joining script also applies to other FLOSS projects.

More recently, Zhou and Mockus [ZM12] found empirical data that seem to contradict the hypothesis that the first posts of promising developers usually contain code fixing their personal problems. While they also analyzed whether users' first posts can predict future contributions, they did not analyze mailing lists like von Krogh et al. did. Instead, they used data from the issue trackers of the GNOME and Mozilla projects. Zhou and Mockus used the term Long Term Contributor (LTC) for posters that stayed on the project for at least three years and caused more than the 10th percentile of changes per year. They used the term One Time Contributor (OTC) for posters with only a single post in their work for the project. Zhou and Mockus showed that users have significantly higher odds of becoming LTCs instead of OTCs (65 % higher at GNOME, and 112 % higher at Mozilla), if their first post is a comment to an existing thread instead of the start

1 Introduction

of a new thread, and assumed that if a user's first post was a comment instead of a patch, it shows a pro-community attitude.

Zhou and Mockus did not comment on this contradiction with von Krogh et al.'s work, but one cause may be differences in the joining scripts used by GNOME and Mozilla, as opposed to Freenet and the FLOSS projects Raymond observed. Another explanation may be differences between the definitions of LTCs and co-developers: LTCs are defined only by their activity in the issue tracker, regardless of whether they contribute code or not. Co-developers in contrast are defined only by their code contributions. The onion model also assumes that a co-developer has previously been an active user and therefore must have participated in discussions without contributing code, and the first post is the transition between passive and active user in the onion model.

In a case study of three FLOSS projects, Jensen and Scacchi [JS07] found different paths that developers may take from outsiders towards the developer roles for each FLOSS project. They further distinguish different subtracks for each path. This implies the existence of multiple joining scripts for each FLOSS projects, upon which new developers may choose or for external reasons be dragged to.

1.1.3 Steinmacher et al.'s Model

Steinmacher et al. [SGR14] proposed a new model to better chart how new developers join a FLOSS project. Figure 1.2 depicts the elements of this model. The model purposefully omits non-code contributions and defines the stages outsider, newcomer, contributor, and member in order of increasing importance for the FLOSS project. Four different forces either help or hinder a developer's migration towards the more productive states:

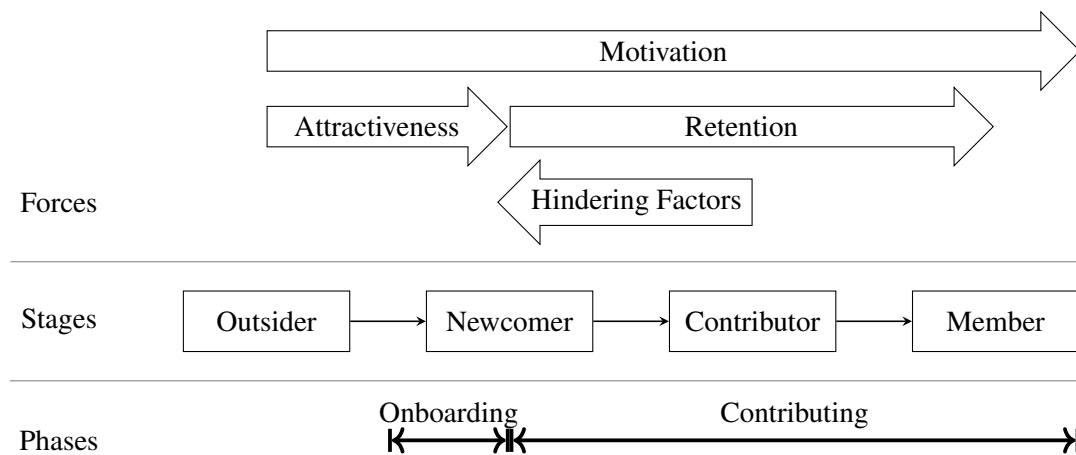


Figure 1.2: Steinmacher et al.'s FLOSS joining model [SGR14]

- A FLOSS project's *Attractiveness* determines whether the FLOSS project attracts both users as well as developers. As a force, it helps Outsiders to become Newcomers.

- *Motivation* comprises the reasons why developers contribute to the FLOSS project. Therefore, motivation helps developers in all states to reach the next stages.
- According to Steinmacher et al.'s textual description, *Hindering Factors* hinder newcomers to become contributors and contributors to become members, although their diagram suggests that Hindering Factors do not have influence on outsiders becoming newcomers. Figure 1.2 reflects this discrepancy.
- A FLOSS project uses its *Retention* to let newcomers and contributors overcome *Hindering Factors* and increase their motivation.

The origin of these forces is important for their differentiation. Attractiveness and Retention are attributes of the FLOSS project, while Motivation depends on the specific developer. Both developers and the FLOSS project may cause Hindering Factors.

Additionally, the model defines two different phases that the developers can be in. In the Onboarding phase, a newcomer tries to place a first source code contribution. The Contributing phase starts with the first source code contribution. Hence, all changes to the source code stem from developers in the Contributing phase. Thus, only these developers forward the FLOSS project with their work on the source code.

The model's limitation to source code contributions has advantages and disadvantages. As a disadvantage, the model cannot differentiate between different ways to join a project if these ways differ only in terms of non-code contributions. Previous contributions as active user according to the onion model might influence how the four defined forces effect a newcomer. An advantage is the model's universal applicability: Every developer joining the project must go through the stages defined in the model.

This thesis adopts Steinmacher et al.'s definition of *newcomer*: A person interested to contribute source code to the FLOSS project, but who has not yet been successful with the contribution: No source code modification from the newcomer has yet been merged into the main code base.

1.1.4 Technical Aspects of the Joining Process¹

The specific activities required for the first code contribution to a FLOSS project have received less research attention than the general processes. Thus, this subsection derives a model for the technical aspects of the first code contribution. In terms of the models presented above, the first code contribution corresponds to the role migration from active user to co-developer in the onion model, the execution of the joining script, and to the onboarding phase in Steinmacher et al.'s model.

Figure 1.3 presents this process as a Unified Modeling Language (UML) activity diagram. UML swim lanes separate the tasks of the newcomer from those of the core developers. The process shall apply to most FLOSS projects and therefore abstracts from tasks that may be specific to individual FLOSS projects. Furthermore, the process focuses on the newcomer and simplifies the core developers' tasks. For example, the diagram does not specify how the core developers select the reviewers of a patch among them and which methods they use for the review.

¹A preliminary version of this section was published previously [GH12b; GH12a].

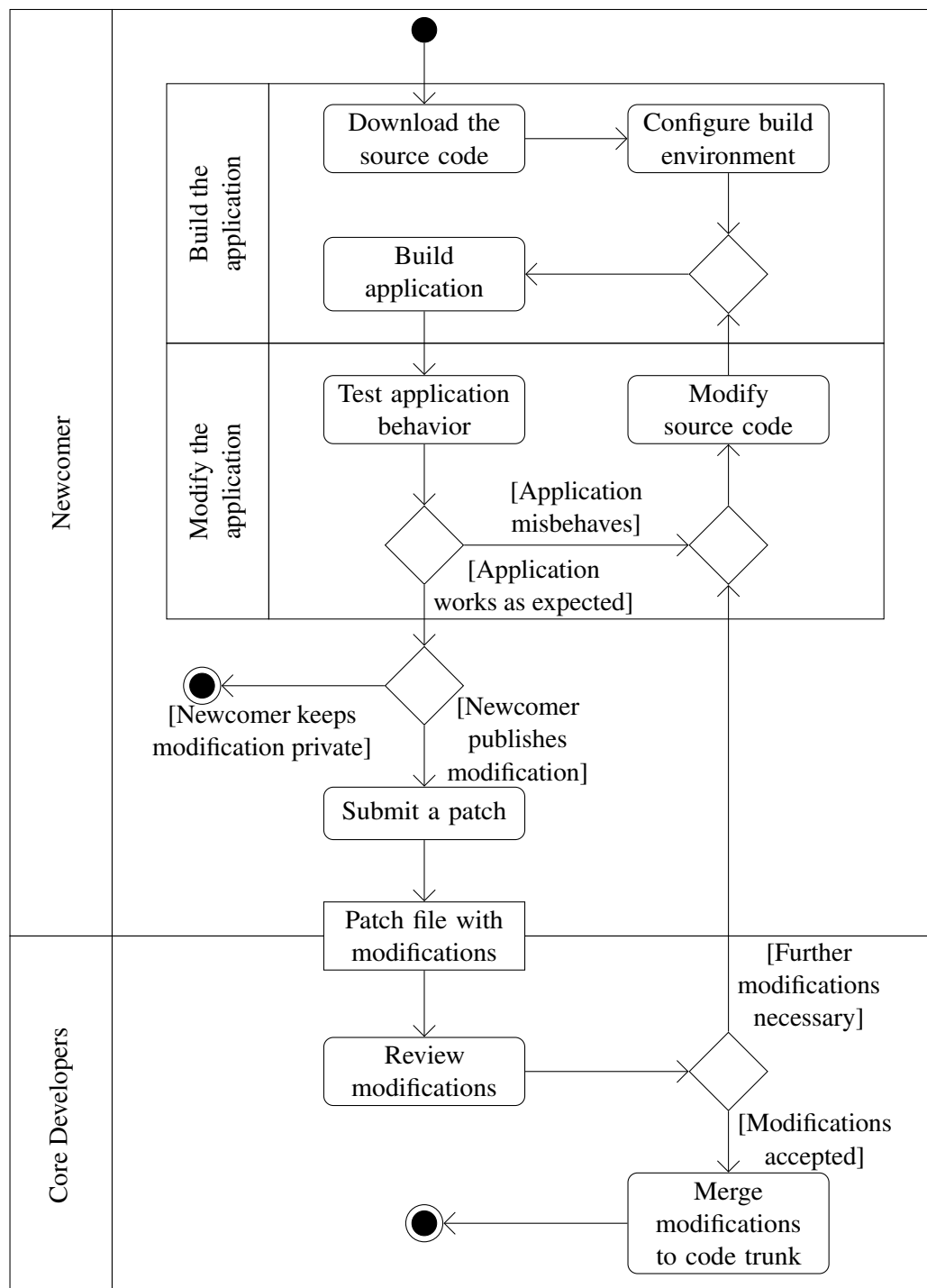


Figure 1.3: Activity diagram of a new software developer changing code in a FLOSS project

The joining process involves the three major activities *Build the application*, *Modify the application*, and *Submit a patch*. Each of these activities involves sub-activities.

For the first major activity, Build the application, there are three sub-activities: *Download the source code*, afterwards *Configure build environment*, which may also involve setting up other parts of the Integrated Development Environment (IDE), and then *Build application*. Even applications written in interpreted programming languages need an interpreter that may require configuration. Of course, some newcomers may first only want to execute the application and only afterwards think about modifying it. Nevertheless, the mentioned activities are still required, but possibly with a different intention.

Afterwards, when newcomers Modify the application, they *Test application behavior* and may notice differences to the specification. This specification will often exist only as their personal expectation, for example when the application fails or when they need a feature that the application is missing. However, they might also have read specification documents in the form of tickets in the FLOSS project's issue tracker. In any case, as long as the application misbehaves, the newcomer will *Modify source code* of the application and then repeat Build application and Test application behavior until the application works as expected.

As the last activity, the newcomer *Submit a patch* with the modifications back to the FLOSS project. Depending on the FLOSS project, this might literally mean the creation of a patch file in diff format [HM76], but can also be something like a pull request in GitHub [P*Git15c]. The FLOSS project's Core developers *Review modifications* in this patch and decide whether to accept or reject the patch. The details of this process, like the number of reviews necessary and who exactly is eligible to review which patch depends on the FLOSS project. A core developer with appropriate rights will incorporate the modifications of an accepted patch into the main VCS of the FLOSS project in the activity *Merge modifications to code trunk*. The newcomer has the opportunity to rework rejected patches in further iterations of Modify the application and Build the application until they are accepted.

Each of the three major activities may have its own motivations and difficulties. As each activity necessarily requires the previous, one possible motivation for the earlier activities Build the application and Modify the application may be to reach the latter. For example, some newcomers may want to Build the application only because later they want to Modify the application. This is not always the case, though.

Build the application is necessary for using the application. Many FLOSS projects offer precompiled binaries, so most users will not have to perform this activity themselves. Modify the application can be a goal in its own right, when newcomers need a modified behavior for their own. Consequently, newcomers have motivations to use the application, to modify the application, and to submit their modifications back to the FLOSS project. These different types of *contributor motivations* will be called *usage motivations*, *modification motivations*, and *submission motivations*. Analogously, the difficulties related to the corresponding activities shall be *usage barriers*, *modification barriers*, and *submission barriers*. Modification barriers and submission barriers are the two kinds of *contribution barriers*, which will be defined more precisely in Section 1.3.

1.2 Contributor Motivation

This section outlines the state of research on FLOSS contributor motivation. A detailed description of all research work is outside the scope of this thesis, as FLOSS contributor motivation has been a domain of extensive research over the course of the last about 15 years. Von Krogh et al. [Kro+12] summarized the state of research on FLOSS contributor motivation as of 2011 with more details in a literature review. This section describes the main concepts and studies, especially in regard to new contributors.

Most, though not all research on FLOSS contributor motivation relies on self-determination theory (SDT) [GD05]. Like some of its predecessor theories, SDT distinguishes intrinsic and extrinsic motivations for an activity. Intrinsic motivations justify the activity itself, while extrinsic motivations justify the activity by its output. The joy of performing an activity is an example of an intrinsic motivation, while payment or praise for an activity are extrinsic motivations. While intrinsic motivations are always autonomous, extrinsic motivations differ in their degree of autonomy. A higher degree of autonomy results in a stronger motivation. Extrinsic motivations may be internalized and then have a higher degree of autonomy. Consequentially, some studies on contributor motivation have internalized extrinsic motivations as an additional class of motivations next to intrinsic and purely extrinsic motivations.

Theoretical works about FLOSS contributor motivation include the application of economical theories on FLOSS [LT02] and literature reviews [Kri06; Kro+12]. All other works known to me rely on empirical work which are on the one hand surveys of FLOSS developers and on the other hand analysis of public data from FLOSS projects like mailing lists. Theoretical and empirical works are difficult to compare, as the type of result of a study strongly depends on its methodology. Therefore this section presents theoretical and empirical works in separate subsections. The section concludes with how FLOSS projects can stimulate their contributors' motivations through their attractiveness.

1.2.1 Theoretical Works

Early work on developer motivation includes Lerner and Tirole's theoretical application of economic theories to the FLOSS phenomenon [LT02]. They explored reasons to explain the FLOSS phenomenon without stressing altruistic motivations and found egoistic reasons to work in FLOSS projects. They argued that there is a signaling incentive for FLOSS developers, as they increase their market value if they demonstrate their programming skills in a FLOSS project.

Ye and Kishida [YK03] agreed that altruism is not the main motivation for FLOSS developers, but they advanced the view that learning and social recognition are the most important intrinsic and extrinsic motivations for FLOSS developers. They derived this position from the onion model described in Section 1.1.1 and substantiate their claims with documented examples.

1.2.2 Empirical Works

Empirical surveys can be distinguished into open surveys, in which any interested FLOSS developer may participate, and closed surveys, which target a specific audience like the developers of a specific FLOSS project. While open surveys may suffer from a strong self-selection bias,

closed surveys only represent the situation in a specific and possibly atypical FLOSS project. Another type of empirical work is the analysis of public data about FLOSS projects such as VCS and issue tracker logs or mailing lists. Analysis of public data allows only indirect conclusions about the developers' motivations. As a consequence of these different data sources, the empirical results often contradict or seem to contradict each other and there is no unified theory of FLOSS contributor motivation.

According to survey results of Hertel et al. [HNN03] and Hars and Ou [HO01], the enjoyment of programming is a major motivator for most developers. Besides this, pragmatic reasons such as needing the modification for their own project were most often mentioned not just in Hertel et al.'s, but also Lakhani and Wolf's survey [LW03]. Improvement of one's own programming skills was also frequently cited in these studies, and seems to be an important factor in starting FLOSS project involvement according to Gosh [Gho05]. David et al. found particularly high numbers of developers driven by the belief that source code should be open, and they should return something to the community for using it [DWA03].

As Krishnamurthy pointed out [Kri06], these surveys do not differentiate between different types of tasks within FLOSS projects. Types of tasks include but are not limited to *usage* of the component, *modification* of the source code, and *submission* of resulting patches as described in Section 1.1.4. Exceptions include a survey of the motivation of active users instead of the developers within a FLOSS project [LH03], an analysis of the social structure in FLOSS projects [CH05b], and Benbya and Belbaly [BB10] also addressed this in a more recent survey of Sourceforge developers. Mair et al. [Mai+15] differentiated two types of tasks, source code contributions and mailing list participation in the community of the statistical FLOSS project R. However, they considered only three motivations: intrinsic, extrinsic, and hybrid. They showed that hybrid motivations have stronger effects on participation, and particularly extrinsic motivation is even negatively associated with mailing list participation.

Surveys that distinguish between modification motivations and submission motivations are rare, with Shah [Sha06] being the only exception I am aware of. In a qualitative empirical study using mailing lists and 88 interviews as data sources, she found that the primary usage and modification motivations are the developers' own need, as they need the modification for themselves. Another identified reason is enjoyment of the coding task. However, the modification motivation of *newcomers* was "need" in 42 out of 45 cases in her data set, i.e. they modified the application because they wanted to use the modification for themselves. Submission motivations were different and comparable to existing studies. Shah acknowledged that there are developers who modify the application but do not submit their modifications back to the FLOSS project, which is a consequence of the separation between modification and submission motivations.

Finally, most motivation surveys, again with Shah [Sha06] as the only exception, only looked at developers that are already developers in FLOSS projects, i.e. people who have mastered the onboarding phase. Their results are therefore likely biased towards the views of a small, but very visible minority of expert contributors, while the group of newcomers is much less visible and thus harder to reach.

1.2.3 Attractiveness

According to Steinmacher et al.'s model of a FLOSS community as described in Section 1.1.3, attractiveness is a property of a FLOSS project and a force that influences outsiders to become users and newcomers of the FLOSS project. When focusing on contributors of source code, attractiveness describes the properties of a FLOSS project that nourish contributor motivations of newcomers.

Ye and Keshida [YK03] postulated that Learning is an important contributor motivation. Hence, they suggested to design FLOSS projects in a way that enables learning, as that would increase the attractiveness of these FLOSS projects. Specifically, they suggested to favor programming languages that many developers want to learn over programming languages that many developers have mastered already.

Smaller FLOSS projects typically do not host their own technical infrastructure and instead rely on software forge services like Sourceforge [P*Sla15] and GitHub [P*Git15c] for their infrastructure [Rie+09]. Simmons and Dillon [SD06] proposed that software forge services should publish more metadata about their hosted FLOSS projects. With these metadata, interested developers will find it easier to acquire an overview over a FLOSS project and therefore have a higher chance to identify FLOSS projects that are interesting for them.

Steinmacher et al. gave the choice of license as one example for a factor influencing the attractiveness of a FLOSS project [SGR14]. However, current research contradicts each other on whether the choice of license has an effect on attractiveness and which type of licenses are more attractive [SAM06; CMP07; CF09; SSN09]. Other examples of attractiveness are project visibility, project age, and the number of developers [SGR14].

1.3 Contribution Barriers

Von Krogh et al. first introduced the term contribution barrier for hurdles that prevent newcomers to join a FLOSS project [KSL03]. This thesis uses the following definition:

A contribution barrier is a property of a FLOSS project that has the ability to prevent motivated people from contributing source code modifications to the FLOSS project.

There are four types of research related to contribution barriers. First, a study may research a specific problem of FLOSS that is a contribution barrier. These studies usually do not refer to the contribution barrier concept, as the specific problem is an obvious research problem by itself. Second, there are studies that focus on the addition of new team members to existing software development projects and the problems with these additions. Since FLOSS projects are special cases of software development projects, these studies describe contribution barriers. Third, some studies describe how new developers join FLOSS projects. Fourth are studies that explicitly try to uncover or collect contribution barriers.

Subsection 1.3.1 discusses the rationale and implications of the above definition of the term contribution barrier and related definitions. The remaining part of this section describes related work of the four named types of research in detail.

1.3.1 Rationale of the Definition

The definition of contribution barrier as stated at the beginning of this section is purposefully limited to source code contributions and excludes other types of contributions such as support to other users, documentation, or administration of a web site. These types of contributions also have their barriers, but they can be very different to the contribution barriers as defined here and therefore require a separate analysis that is outside the scope of this thesis.

The definition presupposes motivated people. This separates the broad field of contributor motivation as discussed in Section 1.2 from contribution barriers.

A contribution barrier only needs the *ability* to prevent contributions and does not need to have prevented actual contributions to be seen as a contribution barrier. This is necessary as otherwise the context would influence which properties of FLOSS projects are contribution barriers and which are not. If the definition would omit this generalization, changes to the management of a FLOSS project would constitute new contribution barriers only after the changes had prevented contributions. Thus, the change of management itself would not impact contribution barriers directly, which would complicate discussions of those changes. Additionally, it would be impossible to discuss contribution barriers independently from specific FLOSS project contexts. Nevertheless, although a FLOSS project's context is independent from the *existence* of contribution barriers, the context has a strong influence on their *importance*. Thus, the context has to be taken into account when evaluating contribution barriers.

Some research, for example that of Davidson et al. [Dav+14a], takes personal barriers into account. Personal barriers fulfill the definition of contribution barriers except that they are properties of the potential contributor and not of the FLOSS project. These personal barriers have been excluded from the definition because they either originate in a FLOSS project's contribution barrier, in which case their separate consideration as personal barriers would be redundant, or they are not caused by the project, in which case there is nothing a FLOSS project could do about them.

Finally, while the definition is limited to FLOSS projects, a closed-source software development project may have properties that are equivalent to contribution barriers of FLOSS projects. Consequently, these properties may hinder development in the closed-source software development project. Thus, research results on contribution barriers can be applicable beyond the scope of FLOSS, but FLOSS is the medium of research. For example, a complex source code structure is difficult to understand for newcomers in both FLOSS and closed source projects. By definition, it is a contribution barrier only for FLOSS projects, but it is likely that closed source developers experience the same kind of difficulties as FLOSS developers when dealing with a complex source code structure.

As announced in Section 1.1.4, analogously to the structure of motivations, there are usage, modification, and submission barriers. Only modification and submission barriers constitute contribution barriers, as these directly hinder the contribution. Usage barriers will only be considered if they are characteristically more important for the modification than for the usage. For example, problems with the download of the application binaries is a usage barrier and will not be considered as contribution barrier, as the download is not even necessary for the modification – a newcomer may want to compile the application from source code anyway. On the other hand, if the FLOSS project develops a library, the libraries' compilation can be

1 Introduction

considered a contribution barrier, even if some users compile the library only for usage in their own applications. Borderline cases are rare, though, and the differentiation between pure usage barriers and contribution barriers is usually not difficult in practice.

The other end of the community spectrum also has its barriers: Occasional source code contributors, the co-developers, may have to overcome hurdles before they gain commit and review rights and become core developers. Section 1.1.2 already described related research from Zhou and Mockus on predictors for LTCs [ZM12]. These types of hurdles can be assumed to be very different to contribution barriers and are therefore also out of the scope of this thesis.

1.3.2 Specific Contribution Barriers

Section 1.1.2 describes the joining script as the activity an outsider has to perform to become a developer of the FLOSS project. The joining script is an informal, social process and therefore at most only partially documented. Adherence to the joining script can involve effort and therefore constitutes a contribution barrier by itself. [KSL03]

Furthermore, Jensen and Scacchi [JS07] observed that the joining script can be so sophisticated that mere understanding the joining script can be difficult for newcomers. Thus, understanding the joining script can be a contribution barrier. They further claim that this contribution barrier can be on a par with the technical contribution barriers.

FLOSS projects employ policies enforcing that submitted patches fulfill given requirements before the patches are merged into the main code repository. Successful code reviews from core developers of the patch are an omnipresent requirement. Rigby et al. [RGS08] described that these requirements may impose submission barriers. Slow code reviews frustrate the submitter. Furthermore, first contributions often do not adhere structural requirements. For example, a single patch file may include changes solving multiple issues, which may be against the policy.

The first and most difficult task when working on a new project is its first build, according to Fitzpatrick [SF09, p. 70f]. The first build requires setting up the IDE, solving issues with platform compatibility and library dependencies. Phillips et al. [PZB14] showed that builds often require substantial effort. The analyzed commercial companies maintained dedicated teams for the build. This shows that the effort required for the build may surpass the effort a possibly voluntary FLOSS developer may be willing to invest.

Midha et al. [Mid+10] proved statistically on 450 FLOSS projects that a low cyclomatic complexity [McC76] correlates with a high number of contributions from new developers. Therefore, a high cyclomatic complexity is a contribution barrier.

In an analysis of 1.4 million developers on GitHub, Terrell et al. [Ter+16] found evidence that gender-related factors influence the probability that a patch submission is accepted. The first result shows that women's patches are more likely to be accepted than men's. However, they argue that this is not related to the FLOSS project but to the developer and therefore does not qualify as a contribution barrier as per the definition of this thesis. They also show that acceptance rate drops if the author's gender is recognizable and that this effect is stronger for women than for men. Thus, sensitivity to the gender of newcomers and possibly to their self-portrayals is a contribution barrier.

1.3.3 New Developers in General Software Development Projects

Brooks [Bro95, Chap. 2] analyzed the effect of new developers on the schedule of a software development project. The new developers require training, so existing trained developers are busy mentoring the new developers and are less productive during that time. Brooks's Law simplifies this dilemma as "Adding manpower to a late software project makes it later".

Sim and Holt [SH98] observed seven patterns occurring when new developers join a software development project. They derived these patterns in an exploratory case study, in which they interviewed four new developers in software development project. They also described a course that they introduced for new developers. They believed that the course remedies some of the obstacles identified in the patterns. The exploratory study tried to carve out possible problems of new developers joining a software development project but did not quantify them. Consequently, it is unclear which of the patterns are specific to the software development project of their case study and which apply generally. Nevertheless, they found as a common theme among the seven patterns that some tasks are more frustrating than others and this does not depend very much on their difficulty or the time required to solve them, but in their perceived usefulness. For example, pattern three states that administrative tasks not directly related to the programming job are frustrating, while even difficult programming tasks do not decrease the motivation to work.

This finding may be even more important in the context of FLOSS, where there are often no external incentives for the work like the salary in closed-source software development projects. If a volunteer developer must perform a frustrating task in order to join a FLOSS project, the developer possibly refrains from joining the FLOSS project. Thus, frustrating, required tasks are by definition contribution barriers.

Begel and Simon [BS08] also stated that a problem that hinders new software developers from being productive, i.e. programming, "frustrates them" [BS08]. They observed eight new software developers at Microsoft over a period of about two months. These new software developers joined Microsoft immediately after leaving University and therefore had no or almost no experience in professional software development. Hence, the focus of their study lay on novices joining a software development project and not on experts joining a software development project, which would presumably better represent the situation of an onboarding newcomer in a FLOSS project. However, they identified some difficulties that may also apply to expert newcomers. Concretely, they collated the difficulties into the five categories Communication, Collaboration, Technical, Cognition, and Orientation. A central, cross-domain difficulty is asking others for help. The decision when to ask others for help is a Communication and Cognition difficulty, and whom to ask for help is an Orientation difficulty. The new developers feared that asking too many questions could be interpreted as a sign of incompetence, and therefore had a tendency to spend hours on problems that could have been solved quickly with the help of a more senior teammate. Collaboration difficulties comprised reluctance to urge other teams to finish prerequisite legwork, and reluctance to decline tasks if they exceeded the new developer's time capacities. The later difficulty resulted in bad prioritizations. In the Technical category, the study concluded that "tools [...] were often a source of difficulty". Participants had trouble to use the VCS, they had no access to test systems, and the development environment was unfamiliar. They also had difficulties to locate the position in the source code to be modified to implement a given change request.

1 Introduction

In a qualitative study based on Grounded Theory, Dagenais et al. [Dag+10] interviewed 18 experienced software developers at International Business Machines Corporation (IBM) who recently joined an ongoing software development project. They determined “Orientation Aids” that helped the developers naturalize in a software development project, and “Obstacles” that made the naturalization harder for them. They classified items in the four categories “Early experimentation”, “Internalizing structures and cultures”, “Progress validation”, and “Cross-Factor”. There are three obstacles in the early experimentation phase: First, setting up a working IDE took between one week and two months. One of the interviewed developers dubbed this setup time as “lost”, so this developer perceived the IDE setup as less desirable than the main job, which is programming. The second obstacle are tools unique to the project, which is closely related to the first obstacle. Third, and contrary to Sim and Holt, upfront courses were seen as a cumbersome duty and slowing down the naturalization with the technical and social environment. A lack of documentation was the only obstacle for internalizing structures and cultures. The category progress validation comprises the three obstacles “Inadequate/No Feedback”, “Sensitive Tasks”, and “unprofessional/no feedback”, but the paper referred to them only superficially and never defined them, so it remains unclear what these obstacles are exactly and how they differ from each other.

Yates [Yat14] also used Grounded Theory to analyze mentoring. She approached mentoring from the program comprehension perspective and described current problems and improvements of onboarding sessions, in which experts transfer knowledge to new developers. Improvement suggestions were mostly micro-optimizations of onboarding sessions, but they also hint at general contribution barriers: Setting up the development environment is difficult for new developers, and each project uses its own terminology that new developers have to learn before they can communicate effectively.

1.3.4 Joining FLOSS Projects

Herraiz et al. [Her+06] analyzed the joining scripts of the GNOME project [P*The15e]. They recognized that there are multiple different ways to join GNOME as a developer. In particular, 7 of 8 analyzed volunteers joined GNOME adhering the onion model described in Section 1.1.1: They first post a message on the mailing list and only afterwards they report a bug, submit a source code modification, and eventually gain commit rights to the VCS. All members of a second group of 12 employees and university staff members used a different sequence, starting almost simultaneously with a message in the mailing list, bug reports, source code submissions, and VCS commits. However, the results may be seen as debatable, since the raw data presented in the paper seem to contain exceptions to those sequences described in the test.

Weiss et al. [WMZ06] showed that developers sometimes join FLOSS projects in groups: there are cases where multiple developers that collaborate in one FLOSS project join another FLOSS project together. In these cases, social contribution barriers are presumably lower than in cases where an outsider wants to join a FLOSS project, because only in the latter cases all developers are unknown to the outsider.

Bird et al. [Bir+07] found that joining a FLOSS project requires the acquisition of project-specific skills. This acquisition takes time. Table 1.1 shows the median times newcomers need for their first patch submission and acceptance for the three FLOSS projects Bird et al. had analyzed.

They also showed that motivation to join a project declines over time after initial contact with the project. Although the study did not explicitly state this, consequentially FLOSS projects that require long naturalization will gain less new developers.

Table 1.1: Time needed to acquire project-specific skills [Bir+07]

FLOSS project	Median time after the first email to the mailing list until ...	
	... first patch submission	... first patch acceptance
Postgres	2. month	3. month
Apache	2. month	10. month
Python	6. month	13. month

Jergensen et al. [JSW11] found evidence that the majority of FLOSS developers dedicatedly use the VCS system, and no social mediums like mailing lists. One explanation they proposed is that these developers may have socialized in related FLOSS projects and therefore were already well-known in the community of the FLOSS project. This indicates that prior exposure to FLOSS projects reduces the effect of social contribution barriers.

Practitioners have also reflected on joining scripts. Turnbull [Tur14] exposed the key parts of typical joining scripts from a practitioner’s view. He emphasized the importance of polite communication both of newcomers as well as of existing developers of a FLOSS project, and explained that lack of kindness can be a contribution barrier. The technical part may sometimes be the easier part of a contribution to a FLOSS project than engaging in its culture and community. He pointed out that the setup of the development environment is the first step of a code modification. This must therefore be “incredibly easy”, as newcomers may not have acquired a high enough modification motivation yet to overcome considerable modification barriers. He argued that contribution barriers can be lower when the first modifications are on documentation and tests: Both are often neglected in FLOSS projects and thus the community welcomes this type of contributions. Another modification barrier is the lack of readability of the code, and code comments can lower this modification barrier.

1.3.5 Exploring Contribution barriers

Von Krogh et al. [KSL03] first described the concept of a “contribution barrier” in their analysis of the Freenet project. They explained that for each module of a FLOSS project, the contribution barrier is a set of the following four hurdles that prevent newcomers from modifying that module:

1. The module itself can be difficult to modify.
2. The programming language of the module may be unfamiliar to newcomers or just unsuited for the task.
3. The architecture of the FLOSS project may lack Application Programming Interfaces (APIs) to add new modules.

1 Introduction

4. Tight coupling between modules may require newcomers to modify more modules than they are willing to.

These contribution barriers were derived from interviews of developers of the Freenet FLOSS project. These contribution barriers are therefore specific to Freenet and to the time of the interviews.

Practitioners also reflected about contribution barriers. Naramore [Nar10] asked the developers among her Twitter followers who had not yet contributed to a FLOSS project via a Twitter poll for their reasons to abstain from contributing. She received 264 responses. The three most frequent replies had the following order: First, they did not have enough time. Second, they were “not sure where or how to contribute”. Third, they lacked confidence in their skills. She also suggested some techniques to lower these contribution barriers.

Sethanandha et al. [SMJ10a] analyzed contribution processes in FLOSS projects. First, they reviewed current literature and then examined 10 FLOSS projects. The literature review yielded four specific submission barriers, specifically

- differences in tool sets between projects,
- long delays for reviews,
- lost patches that never receive a review, and
- rejected patches.

Steinmacher et al. collected contribution barriers [Ste+14a; Ste+15a] from different sources: They mined the mailing list as well as the issue tracker of the FLOSS project Hadoop and interviewed 11 people who failed joining Hadoop [Ste+13], monitored nine students joining FLOSS projects in a controlled experiment [Ste+14b], analyzed existing research in a systematic literature review for contribution barriers [SSG14; Ste+15b], and interviewed 36 developers of nine different FLOSS projects [Ste+14a].

With the results of the individual studies, Steinmacher et al. created a hierarchical classification for 58 individual contribution barriers. Contrary to the definition in this thesis, the classification contains a category *Newcomers' Characteristics* that includes 17 barriers not depending on the FLOSS project but only on the newcomers. Aside from *Newcomers' Characteristics*, five categories remain. *Reception Issues* and *Cultural Differences* comprise *social barriers* related to the communication with the FLOSS project's community. Contribution barriers in the category *Documentation problems* cover different kinds of missing, incorrect, incomprehensible, and inaccessible documentation. Notably, problems with code comments also fall into this category. Steinmacher et al. also proposed the category *Newcomers need orientation* that describes how newcomers may have difficulties to understand the technical and organizational structure of the FLOSS project. The largest category of contribution barriers is *Technical Hurdles* depicted in Figure 1.4.

As shown in Figure 1.4, Technical Hurdles comprise a hierarchy consisting of five sub-categories. *Local Environment Setup Hurdles* contains the only technical hurdle for which Steinmacher et al. found evidence in all four data sources, “Building workspace locally”. The

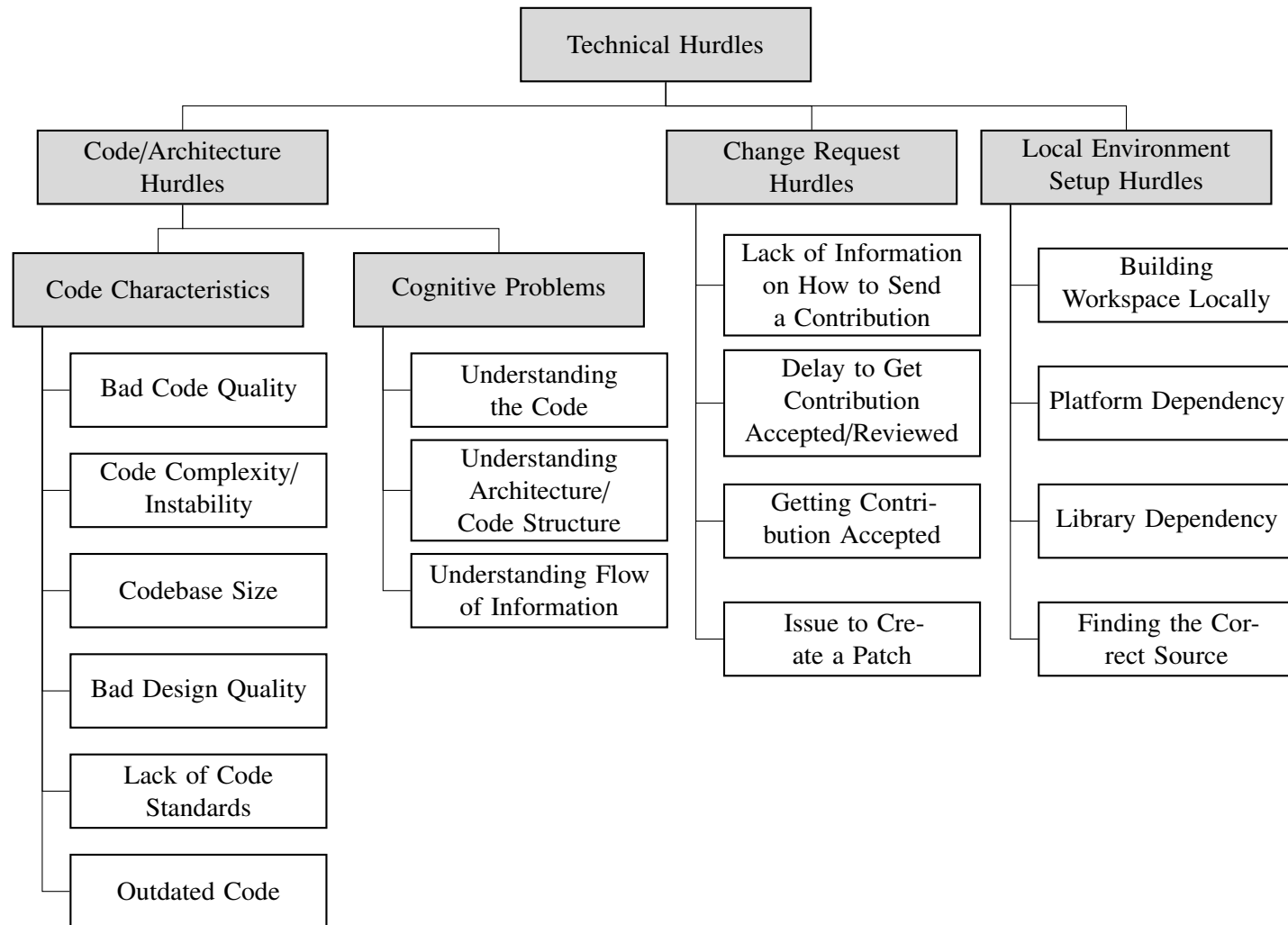


Figure 1.4: Steinmacher et al.'s contribution barrier category Technical Hurdles [Ste+14a]

1 Introduction

category *Change Request Hurdles* encompasses technical submission barriers occurring after the source code modification is already complete. Another technical hurdle with substantiated evidence is “Understanding the code”. However, its category name *Cognitive Problems* suggests that Steinmacher et al. saw this more as a problem of the newcomers instead of a contribution barrier that the FLOSS project induces. Possible sources inside the FLOSS project for these problems of contributors are barriers in the category *Code Characteristics*. This category comprises modification barriers such as “Bad Code Quality” and “Codebase Size”, for both of which Steinmacher et al. presented sound evidence.

Davidson et al. specialized on older developers in FLOSS projects, which includes those older than 50 years by their definition. As part of this research, they also collected contribution barriers that these older FLOSS developers experience. They interviewed 11 older FLOSS developers and 6 FLOSS maintainers [Dav+14b], and they performed a diary study with 4 older FLOSS developers in the process of joining their first FLOSS project [Dav+14a].

In the interviews, 5 of the 11 older FLOSS developers stated that they experienced technical challenges on their first contribution, and 2 mentioned social challenges on their first contribution. The technical challenges included difficulties understanding the source code, and adopting to the development environment. [Dav+14b]

The diary studies identified lack of communication as the most important contribution barrier, followed by problems setting up the development environment, and insufficient documentation. Other contribution barriers were mentioned, but much less frequently. [Dav+14a]

Gousios et al. [GPD14] mined GitHub for data about patch submissions, or “pull requests” in GitHub terminology. This included reasons for rejections. After a rejected patch, a developer may improve the patch or submit a patch for another issue, and therefore still join the FLOSS project as a developer. A patch rejection nevertheless constitutes a contribution barrier [SMJ10a], as co-developers are likely not to submit a patch again after rejection [Bay+12]. Thus, the reasons for rejection also indicate contribution barriers.

Of all patches identified as rejections (excluding the *merged*, *deferred*, and *unknown* reasons described in the paper), the reasons for rejection were

- in 43 % of the cases, another developer was currently developing or already had developed the same feature,
- in 24 % of the cases, the submitted patch was erroneous or otherwise incorrect,
- in 17 % of the cases, the patch did not fit the FLOSS project’s strategy, and
- in 16 % of the cases, the submission deviated from the FLOSS project’s submission procedures.

Tsay et al. [TDH14] also mined GitHub to study patch submission, but focused on predictors for acceptance of a patch. The predictors split into three groups: First are predictors directly related to the patch, like the number of files that have been modified. Second are predictors related to the contributor. Third are predictors related to the FLOSS project. In all categories, the study found multiple statistically highly significant predictors. The three project-related predictors are the age of the FLOSS project, its number of developers, and a measure for the popularity of the FLOSS project among the developer community. All three predictors decrease the chances

of acceptance statistically highly significantly. The FLOSS project's popularity has the highest impact of the three predictors. Among the most influential predictors in general is the social distance between contributor and the FLOSS project's community. As one interpretation, new developers who have not yet arrived in the FLOSS project's community may have difficulties to get their contribution accepted.

Gousios et al.'s [GPD14] finding that the cause for 43 % of the rejections is redundant development of a patch by multiple developers yields another interpretation: Developers watching a FLOSS project closely notice more quickly if there is a parallel development and may resolve the redundancy before anybody submits a patch. Thus, those developers do not submit redundant patches to be rejected.

1.3.6 Summary of Contribution Barriers

There are multiple possible classifications of contribution barriers. This thesis mostly differentiates between modification and submission barriers. Another remarkable classification distinguishes between social and technical contribution barriers. Other classes in this classification are political and juridical contribution barriers, for example requiring the contributor to sign a contract that the contributor's employer does not allow or a country's export restrictions to source code implementing strong cryptography. However, political and juridical contribution barriers seem to have almost no relevance in research.

There are systematic studies of contribution barriers, but they rely on a thin empirical foundation. Anyhow, Steinmacher et al. [Ste+15a] aggregated multiple of those studies to come to more reliable conclusions. Assured or even quantitative propositions about contribution barrier incidence are not possible from existing research, but some contribution barriers stand out:

- Installation, configuration, and familiarization with the tools in the development environment. This applies particularly to tools unique to the software development project. This barrier exists not only for FLOSS projects, but is also well documented for closed source software development projects. [SF09; Dag+10; SMJ10a; Tur14; Ste+14a; Dav+14b; Dav+14a; Yat14]
- Redundant work is a frequent reason for rejected patches. Surprisingly, only one study indicates this problem, although the study founds on a large data set. [GPD14]
- Low quality source code impedes contributions [KSL03; Tur14; Ste+14a]. Specifically a high cyclomatic complexity is a contribution barrier [Mid+10]. Although this barrier is not limited by design to FLOSS projects, I found no research connecting this problem to closed source projects. However, the barrier is difficult to detect in small data sets as they are typically available to studies of closed source systems.
- Understanding the joining script and adhering it is a contribution barrier by itself [KSL03; JS07; Ste+14a].
- The patch review process involves different steps that can be contribution barriers [RGS08; SMJ10a; Ste+14a].

1 Introduction

- Several studies observed lack of documentation and difficulties with communication or general understanding [Ste+14a; Dav+14b; Dav+14a]. It remains unclear whether other, more specific contribution barriers are the underlying cause for these observations. This may be a modification or a submission barrier, and existing research offers little pointers to one direction or the other, because they most commonly do not distinguish between the two types of contribution barriers.

Generally, tasks become contribution barriers if they are frustrating. Some tasks are not frustrating and therefore do not act as contribution barriers, even if they are time-consuming. [SH98; BS08]

The onboarding phase takes between 3 and 13 months in median from the first involvement with the FLOSS project till the first acceptance of a patch [Bir+07]. The specific duration depends on the FLOSS project and on the newcomer. During the onboarding phase, existing developers need to spend resources to support the newcomers with onboarding. This costs resources that cannot be used for actual development. This is also a consequence of Brooks's Law [Bro95, Chapter 2], which will be discussed in the next section in more detail.

1.3.7 The Effect of Lowered Contribution Barriers

Steinmacher et al. [Ste+15a] discussed two potential problems of lowered contribution barriers. First, a higher number of co-workers in a cooperative work increases the coordination overhead and may therefore ultimately decrease productivity. Second, contribution barriers may act as gatekeepers that filter out those newcomers who lack technical or social skills or otherwise just do not fit into the FLOSS project's community. This section incorporates Steinmacher et al.'s discussion of these problems including counterarguments and further reflects on the effects of lowered contribution barriers.

Brooks's Law

Brooks [Bro95, Chapter 2] discussed the problem of increased coordination overhead by adding manpower to software projects. He argued that there are generally partitionable and unpartitionable tasks. While the former will speed up when assigning additional workers to them, the latter will not benefit. Furthermore, adding workers has two disadvantages: New workers need training before they perform well, and a task may need intercommunication between its workers. Both negatively affect existing workers, because they must train the new workers and coordinate with them instead of their actual work. This lost performance may be greater than the added performance of the new workers, at least in the short term, so overall performance may actually decline after adding new workers to a task.

Raymond argued that Brooks's Law applies only to the core developers of a FLOSS project and that co-developers need only very little coordination, as they work on easily separable subtasks [Ray00]. Capiluppi and Adams [CA09] and later Adams et al. [ACB09] found empirical support for Raymond's argument against the intercommunication problem of Brooks's Law in FLOSS projects. This is a result of the community structure of FLOSS projects: Some are scale-free networks, in which communication follows a power distribution over the community [Xu+05;

WMZ06]. Nevertheless, the loss of productivity through the training phase does exist [ACB09]. However, the training phase is the lesser of the two problems [Bro95, p. 18].

Other online communities also have barriers for participation. Especially wikis are also similar to FLOSS development, as they enable their users to contribute to the project. Assuming that communities of FLOSS and wiki projects have similar mechanics, it is interesting to look at the equivalent question for wikis: Do wikis suffer from a communication overhead when the number of contributors grows? Kittur et al. showed that the coordination overhead in Wikipedia increased over time while Wikipedia has been growing [Kit+07b]. Kittur and Kraut showed that this also applies to other wikis: More contributors result in more inefficiency [KK10].

Contrary to Brooks's Law, research has shown that a high influx of new developers is related to the success of FLOSS projects [Sch+08; CSD10; CLM03; CAH03]. Additionally, Brooks argued that complex applications need large development teams to be successful, even if they are less efficient [Bro95, Chapter 3].

Contribution Barriers as Gatekeepers

Overcoming technical contribution barriers requires technical skill, and only those newcomers with social skills overcome the social contribution barriers. Thus, contribution barriers keep out newcomers with a lack of technical or social skills. Having a smaller group of skilled developers may be better than a larger group of mostly less skilled developers. Thus, the contribution barriers may positively act as gatekeepers for the FLOSS project. [Duc05]

Indeed, Sackman et al. found high performance differences of 25 to 1 between skilled and unskilled programmers [GS67; SEG68]. Unskilled programmers would therefore aggravate the problem of communication overhead described above, as their minor contributions cannot compensate the communication overhead necessary for their integration into the development team. However, Prechelt showed that the original figures of 25 to 1 were based on methodological problems and the interpersonal variation is in fact only between 2 to 1 and 4 to 1 [Pre99]. Consequently, technical contribution barriers have at most a low value as gatekeepers for technical skills.

Social contribution barriers presumably also do more harm than good: On the one hand, Ducheneaut [Duc05] argued that contribution barriers help filter out those individuals who do not fit into the FLOSS project. On the other hand, he admitted that contribution barriers are also problematic for the useful newcomers. Furthermore, advertising social requirements more explicitly, for example rules of netiquette [Ham95], would reduce the social contribution barriers that these social requirements cause: newcomers could socialize more quickly and with less problems.

Again, a comparison to wikis may yield additional insights. Do wiki contribution barriers have a positive value as gatekeeper or do they lower the performance of a wiki project? Kittur et al. found that a group of very involved core contributors in Wikipedia provided an initial thrust to the project, but later on, casual contributors became more and more important for the overall performance of Wikipedia [Kit+07a]. Kittur and Kraut also proved that article quality in Wikipedia improved with higher numbers of contributors, and that this is only possible with increased communication [KK08]. This is an analog to Linus's Law of FLOSS development [Ray00, Rule 8]. Thus, more contributors in Wikipedia increase the quality but also the communication

1 Introduction

overhead, but the output of the additional contributors is still greater than the communication overhead they induce.

The Long Tail of Contributors

A number of contribution barriers identified in this section affect only newcomers. Regular developers may contribute without the need to overcome the contribution barriers. Newcomers presumably differ in their tolerance to contribution barriers. This tolerance depends partially on the work time a newcomer is willing to invest into the FLOSS project. Newcomers who are willing to contribute only little to the FLOSS project have a low tolerance, but their contribution is small anyway. Thus, one might conclude that lowering the contribution barrier will bring only little benefit to the FLOSS project. However, all newcomers with small contributions taken together may provide a large accumulated contribution. This subsection presents an argument for this point of view.

Figure 1.5 shows a simplified diagram of the different developers' contributed work time to a given FLOSS project in a given time frame. Figure 1.5 is not based on empirical data and is only intended to visualize the rationale of the argument. The y-axis shows the work time each individual developer is willing to spend on the FLOSS project in a given time frame, and assumingly at the same time the tolerance to contribution barriers. The developers on the x-axis are ordered by the time they are willing to spend on the FLOSS project in the given time frame, i.e. by the value on the y-axis. There are c full-time developers and $d - c$ new part-time developers. Of course, this is one of the simplifications, as there may be part-time developers who are not new to the FLOSS project. For new part-time developers, the effective work time used to actually improve the FLOSS project is reduced by the time needed to overcome the contribution barriers. If the time needed for the contribution barriers is higher than the time a part-time developer is willing to spend on a FLOSS project, this part-time developer will not spend any work time to actually improve the FLOSS project. When the contribution barriers are reduced, new part-time developers can spend more time on actual improvements, because they need less time for the contribution barriers. Reducing the contribution barriers also increases the number of new part-time developers, as there are some part-time developers who are willing to spend more time on the FLOSS project than the time needed for the reduced contribution barriers, but not for the original contribution barriers. In Figure 1.5, this increased number of developers after lowering the contribution barrier is n .

Thus, even small reductions to the contribution barriers may increase the number of contributors more than proportional. Lowered contribution barriers may therefore change the community structure of FLOSS projects towards much higher fractions of occasional contributors. According to Linus's Law, this improves the quality of the application of the FLOSS project [Ray00, Rule 8].

1.4 Research Goals

The preceding part of the chapter gave an overview of the current state of research on FLOSS communities and their joining processes. It focused on contributor motivations and contribution barriers. This section describes three open research questions that this thesis is going to answer.

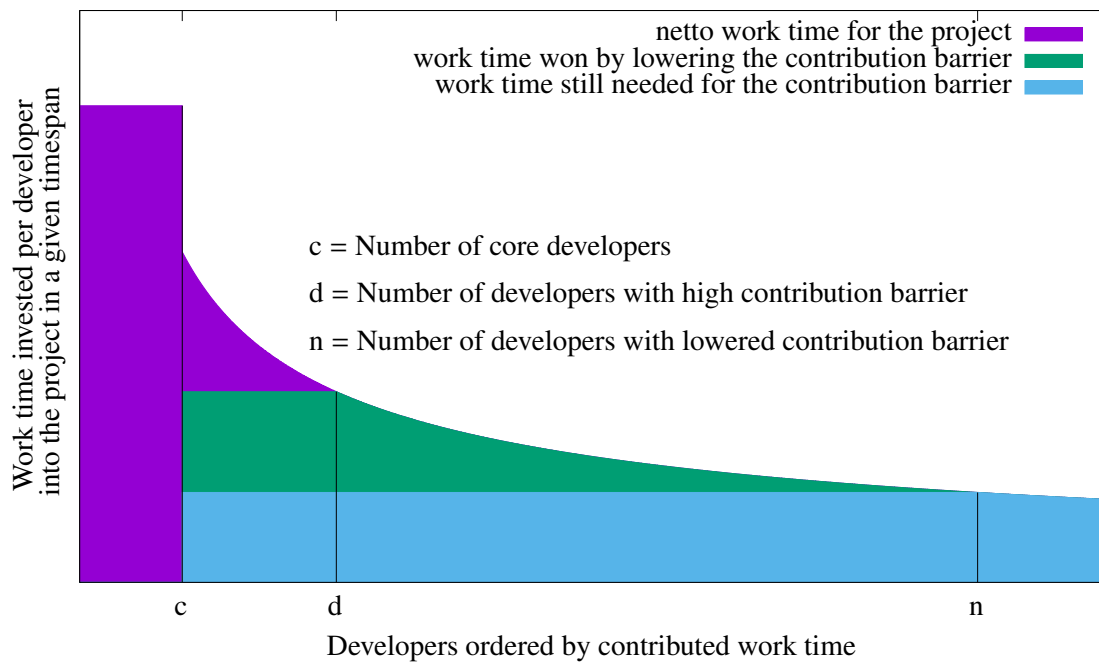


Figure 1.5: Model of the distribution of work time contributors are willing to spend on the FLOSS project

1.4.1 Identification of Contribution Barriers

While the existing research identified individual contribution barriers, there is no complete and empirically founded theory of contribution barriers and their importance. This is the goal of the first research question of this thesis:

Research Question 1. *Which contribution barriers exist and which are important for newcomers to FLOSS projects?*

Chapter 2 approaches this first research question with surveys of newcomers to FLOSS projects. These allow a quantitative analysis of contribution barriers.

1.4.2 Lowering Contribution Barriers

As described previously, lowering contribution barriers is desirable. The results of the first research question show which contribution barrier are relevant, but not how they might be lowered or circumvented. This leads to the second research question:

Research Question 2. *Which practices or techniques lower contribution barriers?*

Patterns are a format to present problems and their solutions in a comprehensible way. This makes them well suitable to structure available management techniques for FLOSS projects, especially those that lower contribution barriers.

In fact, different groups of authors have authored patterns dealing with the management of FLOSS projects, e.g. [Wei09; Lin10]. However, these FLOSS patterns are scattered among various publications that have drawn only few connections between them. Yet, patterns cannot stand on their own but are part of a pattern language [Ale79] and are not completely understandable without the context of a pattern language [CH05a].

Thus, this thesis compiles a pattern language for FLOSS in Chapter 3. It details FLOSS patterns that lower contribution barriers and links them to the contribution barriers identified in Chapter 2. These FLOSS patterns and the FLOSS pattern language are an answer to the second research question.

1.4.3 Wiki Development Environments²

Wikis are web applications that store and show content to its users and provide the means to let users edit the content. They treat all users as contributors [LC01; Ebe+08]. This is the wiki principle.

The content of wikis is usually text. Wikipedia [P*Wik16d] is an example for a wiki with textual content. Textual content is not a requirement for the wiki concept to work, though: OpenStreetMap [P*Ope16c] is a wiki with maps as content. OpenStreetMap enables its users to edit the maps using only their web browsers. Thus, OpenStreetMap proves that the wiki concept is not restricted to textual content. Another example for a wiki with more than only textual content is FlowWiki [Jun+11]. FlowWiki allows its users to modify workflows. These workflows can interact with each other. This leads to the questions whether such an integrated system based on wiki principles could also be used for software projects:

Research Question 3. *Can wiki principles be used to minimize contribution barriers?*

Approaching the third research question, this thesis explores how to adopt the wiki principle to software development. This results in a Wiki Development Environment (WikiDE), which is both an IDE and a wiki system. Such a WikiDE must fulfill the requirements of both an IDE and a wiki system, which imposes special conceptual challenges.

Individual components necessary for the realization of a WikiDE already exist. These components are not easily integrated, though. A WikiDE needs an architecture that integrates these components, possibly with adaptations, into a single platform.

WikiDEs reduce or lower contribution barriers identified in the first chapters of this thesis. WikiDEs realize some of the FLOSS patterns lowering contribution barriers and even advance some of them. They also tackle contribution barriers for which isolated countermeasures described by FLOSS patterns do not suffice. However, some FLOSS patterns complement WikiDEs, and can be implemented in addition to them.

Chapter 4 derives which requirements a WikiDE must fulfill and discusses how to realize a WikiDE. This involves the general architecture of a WikiDE and how to adapt individual components to the requirements of a WikiDE. A Proof of Concept (PoC) realization documents the feasibility of the approach.

²A preliminary version of this section was published previously [GH12b].

2 Contribution Barriers to FLOSS Projects

This chapter addresses the research question described in Section 1.4.1: Which contribution barriers decrease a newcomer's willingness to contribute source code to a FLOSS project?

This main research question implies more specific research goals. Which contribution barriers exist? Which contribution barriers are easier to overcome and which are more difficult? What factors influence whether a specific contribution barrier affects a specific newcomer?

To approach these open questions, this chapter presents two surveys and their analyses. A first, exploratory survey in Section 2.1 provides qualitative insights about the nature of contribution barriers. These insights allow a more precise formulation of the above-stated research questions, the formulation of testable hypotheses about contribution barriers, and thereby clear the path for the second, quantitative survey. The structure of this main survey is outlined in Section 2.2. Sections 2.3 to 2.5 detail the results of the main survey. These results allow statistical tests of the previously formulated hypotheses. Section 2.6 presents and discusses the results of these statistical tests. Threats that endanger the validity of the survey and applied countermeasures are the subject of Section 2.7. A new model for joining FLOSS projects based on the surveys' findings is the subject of Section 2.8. Section 2.9 summarizes and concludes the chapter.

2.1 Exploratory Survey¹

The goal of the exploratory survey was to get a realistic impression of the challenges developers face as they take the first steps toward contributing to a FLOSS project. Mailing lists, issue trackers, and other public archives of project communication are biased by the large amount of experienced FLOSS developers active on these channels.

Instead, an online questionnaire was sent to 664 professional software developers from six organizations (five software companies and one university). These developers work on commercial software development projects across a variety of technologies and domains. Most of these people are very experienced developers who use FLOSS in their projects, but their employers have no particular policy or focus on making active contributions to FLOSS projects, so there is no institutionalized support for making contributions, and any such endeavors stem from the employees' own motivation. Such an environment is a rather common starting point for most potential FLOSS contributors.

22 participants out of this audience responded to the invitation. The invitation explained that anyone who has modified the source code of a FLOSS application may fill out the questionnaire. A possibly large fraction of the invited software developers had never modified the source code of a FLOSS application and therefore refrained from participating in the survey. This explains the relatively low return ratio of about 3 %. It means that findings need to be interpreted with a

¹A preliminary version of this section was published previously [HBG14].

2 Contribution Barriers to FLOSS Projects

grain of salt. While the low return ratio does not suffice for a representative picture, the target population was reasonably broad and the answers provide indications of common contribution barriers for further analysis.

Two kinds of contribution barriers will be relevant for this survey: First, modification barriers hinder or prevent work on the source code of the FLOSS project. Second, submission barriers hinder or prevent developers from submitting this work back to the FLOSS project. While some FLOSS developers may work on the source code of a FLOSS project only because they want to submit their work back to the FLOSS project, this is not generally true. Some software developers may as well solve the problems they currently experience with a FLOSS application and not submit those changes back to the FLOSS project.

Analogously to the contribution barriers, motivations also exist as modification motivations and submission motivations. Additionally, usage motivation determines why a developer gets involved with the FLOSS project in the first place. While usage barriers do also exist, they are not part of this survey.

In this exploratory survey, the distinction between modification and submission barriers is only implicit. The three kinds of motivations, however, are explicitly treated differently.

The questionnaire has undergone a pretest. As a result of the first pretests, the wording improved and some of the questions got simpler. Further pretests ensured comprehensibility and eventually the pretesters did not indicate any ambiguities in the questions anymore. The survey questions use the term Open Source Software (OSS) instead of FLOSS, because OSS is more familiar to the participants.

2.1.1 Demography

The questionnaire first asked respondents to categorize their FLOSS contribution experience. 4 participants that had never modified the source code of a FLOSS project were excluded from the rest of the questionnaire. The remaining participants had modified the source code of a FLOSS project. However, 5 developers did not answer any further questions, so 13 respondents who completed the whole questionnaire remained. Out of these, 4 developers indicated that they had only modified FLOSS source code, while 9 developers responded that they had also successfully submitted patches to projects.

Next, the questionnaire asked which FLOSS project they had modified, and in which year the modification was made. The participants were prompted to consider only the first modification to each FLOSS project, not succeeding modifications to the same FLOSS projects, because only the first modification involves the typical contribution barriers. Following questions in the questionnaire often referred to this FLOSS project. In this report, the question text will read “project PROJ” where the actual participants were shown the name of the project they entered in this question.

While many existing surveys focus on a small number of specific FLOSS projects and thus may be biased by that project’s particular barriers and group dynamics [ØJ07], all of this survey’s respondents indicated that they worked on different FLOSS projects, so their experiences have higher external validity. The majority of modifications was made within the last 1.5 years before the survey, while some changes also reached back as far as 2004.

2.1.2 Developers' Motivation

In order to distinguish the different forces motivating the modification of and submission to a FLOSS project, as well as those hindering contributions, the questionnaire asked all participants about their general relationship to the project, and their reasons for their first modification of its source code. Those who had also submitted their modifications back to a FLOSS project were also asked about their reasons for the submission. With the previously introduced terminology, the survey sampled usage, modification, and submission motivations of the participants using the following three questions. Participants could select as many answer items as they liked. They saw the answer items to Questions 3 and 4 in randomized order and could rank the selected answer items in order of priority. The bold labels in parentheses were not shown to the participants and will be used as references throughout the subsequent discussion.

2. What is the relation of you/your organization to project PROJ? You/your organization ...
 - a) ... use project PROJ yourself.
 - b) ... provide consulting for organizations using project PROJ.
 - c) ... include libraries of project PROJ within your own software.
 - d) ... want to make project PROJ interoperable with your/other software.
 - e) ... have some other relation to the project. (with text box)
3. What are the reasons for you or your organization to modify the source code of project PROJ?
 - a) The malfunction or missing functionality bothered me/my organization as a user of the OSS project. (**Own Need**)
 - b) The joy of programming and/or the intellectual challenge. (**Joy**)
 - c) I wanted to submit the patch. (**Submit**)
 - d) Acquiring experience in technologies used by project PROJ. (**Learning**)
 - e) I like the developers of project PROJ and like to work with them. (**Community**)
 - f) Something else (with text box) (**Other**)
4. Why did you/your organization decide to submit the patch to project PROJ? You/your organization wanted to ...
 - a) ... have your changes reviewed by others. (**External Review**)
 - b) ... return something, because you feel obliged to do so. (**Feeling Obligated**)
 - c) ... avoid the work of reintegrating your changes into new releases of the OSS project. (**Stupid Tax**)
 - d) ... gain experience in the procedures of (OSS) projects. (**Learning**)
 - e) ... publish the source code, because you believe source code should be open. (**Belief in FLOSS**)
 - f) ... gain respect from the community. (**Gain Respect**)

2 Contribution Barriers to FLOSS Projects

- g) ... do something good (altruism). (**Altruism**)
- h) ... gain publicity (for example, you might have wanted to get job offers). (**Publicity**)
- i) ... fight closed source software. (**Fight CS**)
- j) ... achieve something else (with text box). (**Other**)

The primary usage motivation (Question 2) was the use of the project itself – either as a tool they used as end users, or as a library they integrated into their own system. Other reasons such as providing consulting for the FLOSS project, making it interoperable with one’s own products, or evaluating the project played only a minor role for the respondents.

Figure 2.1 depicts how respondents ranked the answer items to Question 3. Only the highest ranks 1-3 are shown and answer items are ordered by the number of selections as first rank. The answers showed a clear picture: The majority of developers made modifications because a malfunction or missing feature bothered them (Own Need), because they wanted to acquire experience with the project’s technologies (Learning), and because they enjoyed the intellectual challenge (Joy). Satisfaction derived from working with the project’s community (Community) and submitting patches (Submit) were only minor motivators. In an interesting case, one participant wanted to “preserve part of [the participant’s own] project work spent on the usage of [the FLOSS component],” i.e. the participant refactored work originally performed outside the FLOSS component into it, so it could be reused in future endeavors. These results agree with Raymond’s observation that most work starts from “scratching a personal itch” [Ray00] and especially with Shah [Sha06] who found that Own Need is the primary modification motivation for newcomers.

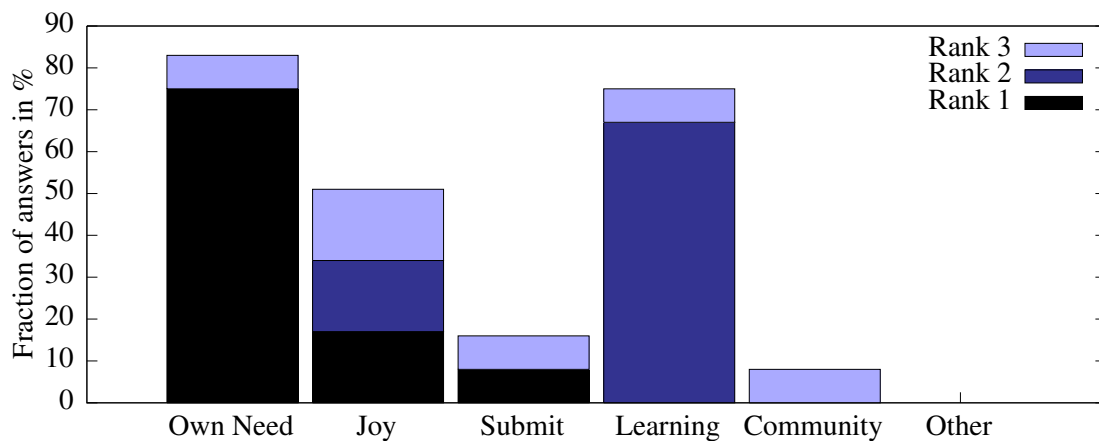


Figure 2.1: Number of ranked responses for reasons to modify the source code of a FLOSS project (Question 3)

In contrast, submission motivations (Question 4) show more variation, as Figure 2.2 indicates: Having one’s changes reviewed by others (External Review) and avoiding the work of reintegrating one’s changes into new releases of the OSS project (Stupid Tax) were the most frequently mentioned and highest-ranked reasons, with a feeling of obligation to return something (Feeling Obligated) being a close third. These three motivations are an interesting mix spanning

aspects of learning (External Review), pragmatics (Stupid Tax), and ethics (Feeling Obligated). The complementarity of these factors, between which no compromise is necessary, together supposedly creates a quite strong motivation that offsets the time and effort involved in submitting a patch.

Other motivating factors were gaining experience with FLOSS procedures (Learning), belief in the open source idea (Belief in FLOSS) and altruism (Altruism), as well as to a lesser degree community respect (Gain Respect) and personal publicity (Publicity). An aversion against closed-source software (Fight CS) did not play a role among the respondents.

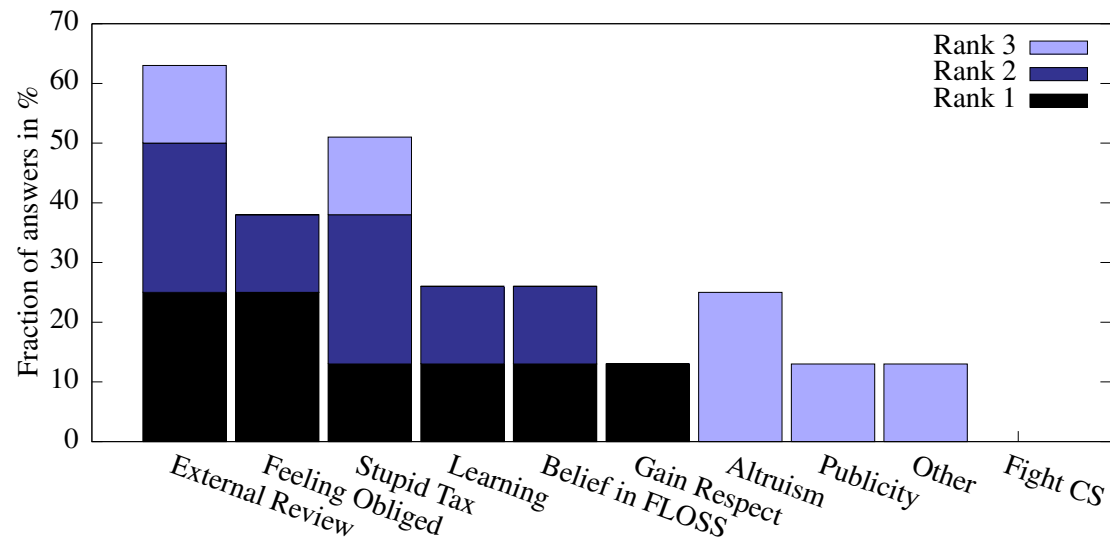


Figure 2.2: Number of ranked responses for submitting patches to a FLOSS project (Question 4)

The participants who had succeeded in submitting their patch were also asked whether the expected benefit ultimately merited the invested effort. To each of the following questions, they could respond on a five-point scale ranging from “far too much effort” to “benefit far exceeds effort”.

5. Was the benefit of submitting your patch worth its effort?
 - a) Was your personal benefit worth the effort?
 - b) Was the benefit for your organization worth the effort?

The respondents uniformly agreed that the benefits of submitting the patch were worth the effort, some even indicated that the benefits exceeded the effort – another reason for developers’ perseverance in the face of the contribution barriers discussed in the following section, once they have made the decision to contribute.

In contrast, those participants who had not tried to submit their private modification back to the FLOSS project were asked about the reason for their renunciation, and could choose any number of the following options:

6. Why didn’t you submit your modification back to project PROJ?

2 Contribution Barriers to FLOSS Projects

- a) Others would have no use for this modification.
- b) The possibility of publishing the modification just wasn't considered.
- c) Expected time effort was too high.
- d) My organization would have lost an advantage over its competitors.
- e) Something else (with text box)

The four participants who had decided against submitting justified this in Question 6 with a too high expected time effort, or the impression that others would have no use for their modification. In one case, a submission back to the FLOSS project was not provided as a patch, but exchange of ideas that the maintainer agreed to implement directly later. All developers however at least considered the possibility of submitting their patch, and losing a competitive advantage was not a concern for them, so the decision was mostly based on reasons of practicality.

2.1.3 Contribution Barriers

The next questions addressed the contribution barriers experienced by the developers. The first two question quantitatively assessed the time needed for the modification and the submission. The last set of questions were open questions and thus asked for free-text answers in order to receive unbiased impressions of the contribution barriers.

Closed Questions

Based on the tasks required for a FLOSS contribution described in Section 1.1.4 (with some tasks combined for simplification), the questionnaire first asked all respondents about the time they spent on each task. Items 7f and 7g that refer to the actual submission of the patch were hidden from those participants who had indicated that they had only privately modified but never submitted anything to a FLOSS project. Respondents could select one of the time intervals “up to 10 minutes”, “up to 1 hour”, “up to 8 hours”, “up to 40 hours”, and “over 40 hours”.

7. In the whole procedure of submitting the patch, you probably had to finish a couple of distinct tasks. Possible tasks are listed below. How much time did you spend for each of these tasks?
- a) Download the source code (or re-download current version of the source code) **(Download)**
 - b) Build executable binaries from the source code **(Build)**
 - c) Reproduce the problem you wanted to fix **(Bug Repro.)**
 - d) Find the location to change in the source code / find the cause of the problem **(Find the Code)**
 - e) Solve the problem (write code) **(Solution)**
 - f) Create a Diff/Patch file **(Patch)**
 - g) Submit a bug report including your patch **(Submission Procedure)**

h) Accomplish tasks not listed above (with text box) (**Other**)

To reveal hidden effort that the participants had to invest in order to familiarize themselves with the internals of the FLOSS project, but that had not been related directly to the patch in question, the questionnaire also inquired about the time spent on any general bootstrapping effort. Possible answers again were the same time frames as above.

8. How much time did you spend on the following tasks without relation to your patch, before you started your work on tasks directly related to your patch?
 - a) Download the source code (or re-download current version of the source code)
 - b) Build executable binaries from the source code
 - h) Accomplish tasks not listed above (with text box)

The bandwidth of answers in the survey is shown in Figure 2.3. Finding the cause of a problem and the right location in the source code (Find the Code) consistently required one of the highest efforts, with writing the code to solve the problem (Solution) a close second. Obtaining the current version of the source code (Download), reproducing the problem (Bug Repro.), creating the patch (Patch), and submitting it (Submission Procedure) took more manageable time. The highest maximum and arithmetic mean time effort was spent on building the system (Build).

The primary effort drivers Find the Code and Solution, i.e. finding and fixing the problem, are independent of FLOSS projects – they mirror the general intellectual challenge of programming, and are addressed by research on program comprehension, debugging, and related fields. Among the steps that are more specific to the context of joining a FLOSS project, the effort for building the system (Build) stands out in particular.

However, the time needed for this step also varied widely between participants. This is due to the variety of programming environments developers need to deal with. For example, one participant directly modified the source code of a productive system written in the programming language PHP: Hypertext Preprocessor (PHP). Since PHP is directly interpreted by a web server, the participant neither needed any work time to download the source code nor to build the application from source code. Still, the setup of the interpreter environment may be a major effort in other project environments as well. Another participant who pointed to the setup of the development environment as the biggest barrier was modifying a component written in Python, which is also an interpreted language. Thus, for interpreted languages, the work time needed until the *first execution* of the application may be as long as the work time needed until the *first build* for other projects. This means that some interpreted languages just shift this contribution barrier from a modification barrier to a usage barrier.

Open Questions

Following these quantitative effort estimates, the questionnaire next asked those participants who had tried to submit a patch to the FLOSS project about their qualitative experiences with the contribution barriers. This was deliberately an open question in order to receive unbiased responses of the contribution barriers that were most prominent in developers' memories:

2 Contribution Barriers to FLOSS Projects

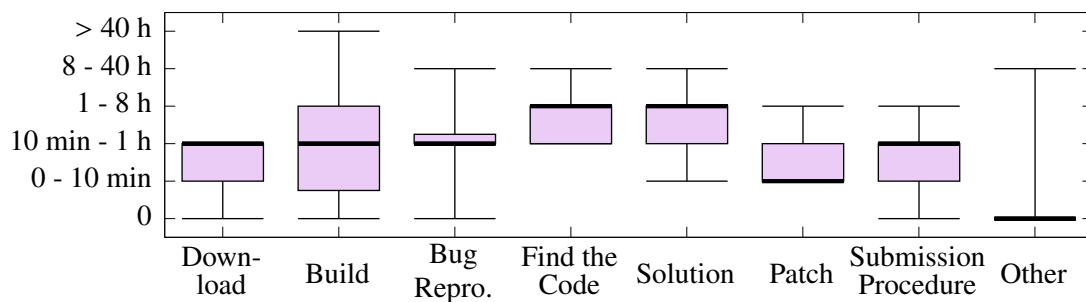


Figure 2.3: Time spent on tasks required for modifying and submitting code of a FLOSS project (Questions 7 and 8)

9. What bothered you when submitting the patch to project PROJ?

- What was the biggest barrier that you encountered while submitting a patch to project PROJ?
- Before you submitted the patch to the project PROJ, had the submission procedure of project PROJ ever discouraged/prevented you from submitting a patch? If so, what was the specific reason for you not to submit that patch?

Of the respondents who successfully submitted a patch to a FLOSS project, two answered that they did not experience any barriers. Among the remaining answers, the by far most frequently mentioned barrier in Question 9 was the setup of the development environment in one form or another.

In total, the online survey took the participants 10-15 minutes to complete. Respondents also had the opportunity to pose questions for clarification or indicate difficulties with the questionnaire, which none did.

2.1.4 Discussion

The survey results indicate that a major part of the contribution barrier is the effort invested until the first build. This is in agreement with other research as presented in Section 1.3.

At first sight, the difficulty in setting up the build system may be surprising, since the participants of this survey are professional software developers who are familiar with build systems and should already have a development environment installed and configured on their machine. There are two explanations why the familiar development environments do not suffice for the modifications of the source code of some FLOSS projects. The survey gives merely hints on which explanation best describes the situation of the participants.

First, developers may not have used the programming language of the FLOSS project before and need a new development environment for the new programming language. Often, the need for the FLOSS component may stem from project requirements, but all of this survey's participants who found the setup of the development environment to be the biggest contribution barrier also indicated that they had started to work on the FLOSS component because they wanted to acquire experience in the technologies it uses. This indicates that developers are willing to tackle the

contribution barrier even if the main reason for working on the FLOSS component is not a pressing project need, but a learning endeavor.

Second, even if the FLOSS project uses a programming language that the existing development environment supports, the core developers' platform may be different from that of the new software developers, making a platform-specific setup necessary. An example for a FLOSS project with such limitations is Mozilla: Depending on the subproject and the branch of the subproject modified, one version of the same development environment may be compatible while another is not [P*Moz12].

The respondents cited difficulties to access the source code as another contribution barrier. One participant had been discouraged from submitting patches earlier because there was "no browse access to the 'trunk' version of the source code". Making it easier for developers to get all code, tools, and information they need should therefore be part of a FLOSS project's strategy for encouraging new contributors, especially given that the technological and intellectual barriers further down the process are still hard enough, as discussed below.

As a solution to the challenges associated with building binaries from the source code, closed source projects often use Continuous Integration (CI) systems. These CI systems automatically check whether modifications are compatible with each other, and test them on a broad range of platforms. A missing CI system can be a contribution barrier, as one developer in the survey indicated:

"My productive system had a very special configuration (including dated versions of some of the components) and I was unable to test my fix against alternative configurations."

A prominent example for the use of CI systems is Microsoft executing thousands of tests for each of their security patches [P*Mic15b]. However, such a number of tests requires dedicated test servers. Large FLOSS projects may have the financial resources to afford such a CI system. Mozilla, for example, tests their modifications on different platforms [P*Moz15i]. In contrast, many smaller FLOSS projects do not maintain their own server infrastructure. Instead, they rely on software forge services such as Sourceforge [P*Sl15] and GitHub [P*Git15c] for their key infrastructure like VCSs and issue trackers [Rie+09]. However, these software forge services do not provide CI to FLOSS projects – in fact, the advent of CI systems since 1997 has not had a significant impact on development practices of FLOSS projects [DR08b] even though they may help to lower the projects' contribution barrier. Recently, platforms emerged that offer free CI services to FLOSS projects. Examples include Travis-CI [P*Tra15] (since early 2011) and BuildHive [P*Clo15] (since May 2012).

2.1.5 Conclusion

This section presented the results of an exploratory survey of professional software developers to identify contribution barriers that newcomers to FLOSS projects encounter. In contrast to previous surveys, this survey deliberately did not target typical FLOSS communication channels (mailing lists, issue trackers, etc.) which might have biased the results to a more expert-dominated view, but surveyed developers outside the established communities who were asked about their

2 Contribution Barriers to FLOSS Projects

experiences in making their first modifications and contributions in FLOSS projects. This way, the exploratory survey identified two prominent components of the contribution barrier that work as opposing forces to developers' motivation to contribute:

- Depending on the specific FLOSS project and platforms of the joining software developers, the effort necessary until the first build of an executable application from the source code is the largest contribution barrier. To achieve the first build, joining software developers have to set up a development environment and configure the build scripts for their platform, which is not trivial even for experienced developers.
- Small FLOSS projects usually cannot afford costly infrastructure such as CI systems. This also increases the contribution barrier, as joining software developers cannot be sure whether their modifications are compatible with all platforms that the FLOSS project supports.

So what is the theoretical minimal contribution barrier when the available tools improve further? Brooks [Bro87] argues that there are two types of effort in software development – essence and accidents. While essence is the core problem to be solved, accidents are all man-made problems around the core problem. Examples of accidents are copious programming languages and improper development environments. Brooks pointed out that improvements of software development tools can only reduce the effort for accidents, but never for the essence. The survey confirms that the “essence” – the actual code understanding and fixing – incurs the largest effort in FLOSS projects as well. The most prominent “accident”, however, was the effort induced by setting up build environments.

Open Research Questions

While the survey measured the time effort necessary for the individual steps of a contribution, it is not clear to what degree contributors feel reluctant to perform the necessary tasks of each step. Some tasks may require more time to perform but are joyful and therefore do not act as a contribution barrier. Thus, only tasks that newcomers feel averse to perform are contribution barriers, and only for these tasks does the time needed for their execution increase the contribution barrier. Future research should measure the contributors' averseness to each task.

The differentiation between modification and submission motivations proved useful. The survey identified a single primary modification motivation, that is using the modification for oneself. The differentiation between modification and submission should also be more strongly extended to contribution barriers.

The identified contribution barriers and motivations should be confirmed on a larger data set. A larger data set would allow statistical analyses and stronger empirical support for the results.

2.2 Main Survey²

The main survey improves on the exploratory survey described in the last section, as it strives for answers to the questions that the exploratory survey opened or could not answer. The main

²A preliminary version of this section was published previously [HG16b].

survey implements improvements in survey design based on the experiences from the exploratory survey.

Consequently, this section advances the endeavor to expose contribution barriers to FLOSS projects with a survey of 118 developers who had recently joined a FLOSS project as new software developers. These developers described the contribution barriers they had experienced when creating and submitting their first patch to either the Mozilla project or the GNOME project. Insights on contribution barriers from their answers are complemented by explicitly distinguishing modification and submission motivations.

2.2.1 FLOSS Projects

This section provides an overview over the FLOSS projects Mozilla and GNOME, as the survey's participants are newcomers in one of these two projects. Both projects have been the subject of previous research. Østerlie and Jaccheri criticised that FLOSS research focuses on too few FLOSS projects, as this disallows drawing conclusions about the FLOSS phenomenon as a whole, and some characteristics of FLOSS projects might stay unnoticed [ØJ07]. This implies that research results should be confirmed on less commonly researched FLOSS projects. However, this research targets an aspect of FLOSS development that existing research has not focussed on yet. Studying popular FLOSS projects can therefore build on existing research of these FLOSS projects and minimizes the risk that observed characteristics are specific only for exotic FLOSS projects.

Mozilla

Netscape made their Communicator product a FLOSS project in 1998 [P*Net98]. They found the Mozilla Foundation to host the Mozilla project. While the Mozilla project is well known for its browser Firefox, the Mozilla project is more diverse. Its 311 modules and submodules [P*Moz15x] comprise the mail client Thunderbird, the issue tracker Bugzilla, and the cryptographic library Network Security Services, to name three examples.

These applications usually target multiple platforms like Linux, MacOS, and Windows and also mobile platforms like Android. Developers may also choose among multiple development platforms including Unix operating systems and Windows. [P*Moz15d]

Research on Mozilla is diverse. It includes analysis of its early development process [MFH02] and tools [RM02], it serves as a case study to construct general models about FLOSS procedures [JS07; ZM12], and it is also a common data source to evaluate general Software Engineering (SE) techniques [Yin+04; CC06].

Mozilla follows a process dubbed “bug-driven” development [RM02]. Mozilla uses the pejorative term *bug* for every type of change request to their application, not only including fixing defects, but also implementations of new features. This thesis uses the more neutral term *issue* instead of *bug*. Mozilla uses the issue tracker Bugzilla, a Mozilla product itself, to keep track of all issues [P*Moz01]. Indeed, Mozilla policy enforces an issue in Mozilla's Bugzilla for every modification committed to Mozilla's main VCS trunk. This policy also enforces that every proposed code modification must receive a positive review from either the owner of the changed

2 Contribution Barriers to FLOSS Projects

module or one of the module owner's trusted delegates. This is true even for the module owners themselves, and developers are not allowed to review their own patches. [P*Moz15f]

Mozilla's Bugzilla instance [P*Moz01] keeps track of all issues, proposed code modifications, and reviews. The following describes the mandatory steps for every modification to the code: A reporter first creates an issue in Mozilla's Bugzilla instance that describes the defect or feature request. Then a developer, who may or may not be the reporter, is assigned to the issue. Developers can assign themselves to issues if they have the appropriate rights. After developers have modified code, they aggregate all modifications into a single file and attach it to the issue. This file is often in diff format [HM76], but may for some modules be the Uniform Resource Locator (URL) to a pull request on GitHub [P*Git15c]. Attached files may receive flags of multiple types in Bugzilla. Of special importance are the "r" flags that indicate whether a patch has been reviewed or not. The developer who created the attachment flags this attachment with an "r?" flag together with the name of a core developer. This core developer is then expected to review the attachment and report the review result by changing the attachment's flag either to "r-" or "r+" for a declined or accepted code modification, respectively. In case of an accepted code modification, a committer merges the modification into the main VCS branch for the module. [Bay+12]

Any of these mandatory steps may require additional work like discussions on the code or questions to the reporter on how to reproduce a failure. A participant may also inadvertently hamper this process, for example when developers do not add the "r?" flag to their attachments because they do not know that this is necessary.

GNOME

GNOME is a window manager and application collection for Unix operating systems. In 1997, developers in the GNU's Not Unix (GNU) community started the GNOME project [P*Ica97]. Today, the GNOME Foundation hosts the GNOME project as a non-profit organization. An elected board governs the community [P*GNO14]. [P*The15e]

GNOME has also been the subject of research. This includes use cases for FLOSS [Fin03], FLOSS communities [Her+06; JSW11], the project's development process [Ger03], and its ability to attract LTCs [ZM12].

An instance of Bugzilla serves as the issue tracker for the GNOME project [P*The15d]. Similarly to Mozilla, GNOME's Bugzilla instance tracks all modifications to the code, so with a few exceptions, for every code modification there must be an issue in GNOME's Bugzilla. Using the same flag feature that Mozilla uses, attachments to an issue in GNOME'S Bugzilla may receive flags when reviews happen. The flags are different to Mozilla's, though, and handled less strictly. [P*GNO14]

Besides Bugzilla, GNOME uses the VCS git, a wiki, Internet Relay Chat (IRC), and mailing lists as tools to support developers with their contribution [P*GNO15b]. A part of the wiki is dedicated to help newcomers onboard the project as developers [P*GNO15a].

2.2.2 Research Hypotheses

One important specialization of this chapter's research question is: What factors influence the contribution barriers that a newcomer experiences? This section derives research hypotheses

from literature and results from the exploratory survey. These research hypotheses will be tested with statistical methods in Section 2.6.

The programming language has influence on different aspects of a modification: First, each programming language comes with its own tool chain. Some programming languages have better tool support, others are relatively new and therefore the community had less time to develop good tools, or the community may simply be smaller. Second, the choice of programming language depends on the purpose of the modification. Changes to an application's interface may require a different programming language than changes to an application's core. For example with Mozilla, Graphical User Interface (GUI) changes may often require modifications of Extensible Markup Language (XML) codes like Hypertext Markup Language (HTML) and XML User Interface Language (XUL) or JavaScript code, while changes to Mozilla's core libraries may require modifications to C or C++ code. Changes to these core libraries may be more difficult, for example because there are more dependencies of other modules that need to be considered. Third, some programming languages are simply easier to learn and use than others [SS13; End+14]. This leads to

Hypothesis 1. *The contribution barrier for modification depends on the programming language used for the first modification.*

Despite claims that programming experience and programming performance may be uncorrelated [GS67; Bro95, p. 30], research on larger data sets confirms the intuition of a correlation [WSK08]. Thus, experienced developers should have less problems handling technical problems when they try to modify a FLOSS application, and therefore perceive lower technical contribution barriers. This leads to the following hypothesis:

Hypothesis 2. *More experience of newcomers helps to lower their contribution barriers.*

Each FLOSS project uses its own development tool chain, depending on the programming languages of its source code, the platform the FLOSS application runs on, and the platforms the developers use. Additionally, FLOSS projects differ in the attitudes and procedures they have employed. New contributors may appreciate some of these more than others; partly, because some procedures may actually speed up the joining of new contributors. To another part, the attitudes and procedures simply may or may not create a welcoming atmosphere. As a third factor, FLOSS projects have different user bases due to their supported platforms and intended use cases. For example, users of the Eclipse IDE [P*The15c] are developers themselves, while users of the Mozilla Firefox browser may not necessarily be technically savvy. When FLOSS projects recruit newcomers out of their user base, as for example according to the onion model [Cro+05] and Raymond's "first lesson" [Ray00], the skills of these newcomers differ between FLOSS projects. Consequently, the newcomers in one FLOSS project may easily overcome a contribution barrier that fits their skill set, while the newcomers in another FLOSS project perceive the same contribution barrier as an important hurdle when they lack the required skills. These factors suggest the following hypothesis:

Hypothesis 3. *Contribution barriers vary between FLOSS projects.*

The exploratory survey presented in Section 2.1 shows in Section 2.1.2 that the most important modification motivation for the respondents was their Own Need: They needed the modification

2 Contribution Barriers to FLOSS Projects

for their own use of the application. This is in agreement with Shah’s qualitative results [Sha06]. This may be a general rule for modification motivations:

Hypothesis 4. *Most newcomers modify the FLOSS project’s source code primarily because they need the modification for themselves.*

According to the Theory of Cognitive Dissonance [Fes57], a person feels discomfort if two of the person’s opinions contradict each other. This is particularly the case when new information contradicts the expectation derived from existing information. This type of discomfort is called *dissonance* and its reduction is a natural stimulus just like hunger. In order to reduce dissonance, people may take a new point of view, but they may also just discard newly gained information if it does not fit their existing views.

Before newcomers decide that they want to contribute to a FLOSS project, they have a mental model of how a FLOSS project works and especially about the procedures and reception of their contribution. Like any model, this mental model is incomplete, although some newcomers may be aware of this incompleteness. Even if they are, they have unconscious or conscious assumptions about the procedures of their contribution, as they must have founded their decision to contribute on something. Thus, their motivation to contribute derives from this model. For example, if they assume that a contribution to a FLOSS project gains respect for the contributor, then they may derive the motivation to gain respect from the FLOSS community through their contribution. Accordingly, this can be used in the other direction: Contributors’ motivations are indicators for their assumptions about the contribution before their contribution experience.

If the contribution experience contradicts the previous assumptions, dissonance arises. The mental model also includes assumptions about what parts of the contribution procedure are difficult and which are not. Hence, unexpected problems create more discomfort than expected problems. One way to reduce this dissonance is to refrain from the contribution: Either after accepting the new information about unexpected problems or through discarding existing information that served as foundation for the motivation, even if that information was in fact true. A newcomer may also ignore or accept the unexpected problems and continue with the contribution, if that is the lesser mental effort [Fes57]. Since dissonance can be a reason to refrain from contribution, dissonance influences the perception of contribution barriers.

If this application of the Theory of Cognitive Dissonance to FLOSS contributions is correct, the following hypothesis must hold:

Hypothesis 5. *Contribution barriers are less important for contributors whose motivations suggest that they expect these contribution barriers.*

2.2.3 Participant selection

The exploratory survey in Section 2.1 targeted professional developers to report about contribution barriers experienced when working on FLOSS projects. In contrast, the main survey presented in this and the following sections targeted developers who had recently joined a FLOSS project. The survey design founds on the Tailored Design Method (TDM) [Dil99] with some adaptations to accommodate to the survey’s circumstances, especially invitations via email.

As a consequence of the changed target audience in contrast to the exploratory survey, some questions had to be added or changed. For example, the questionnaire’s demographic section

included questions about the occupation that were unnecessary for the exploratory survey targeting only professional developers in a known environment. Although most parts of the questionnaire had been pretested and tested in the exploratory survey, an additional pretest for the main survey ensured the questionnaire's comprehensibility.

The survey targeted developers whose first patch was recently accepted by one of the FLOSS project Mozilla or GNOME. A data collection and filtering process depicted in Figure 2.4 and Figure 2.5 selected the participants: From April 2013 to October 2013 and from November 2013 to January 2014, scripts based on Zhou's and Mockus's download and transformation scripts [ZM12] regularly downloaded the issue tracker data from Mozilla and GNOME, respectively. The scripts assigned all accepted patches to developers. For each developer, the chronologically first acceptance of a patch indicated the date when this person, by this survey's definition, became a developer of the project. Participant candidates are those who became developers between April 2013 and October 2013 for Mozilla and between November 2013 and January 2014 for GNOME. Afterwards, two thirds of the participant candidates had to be filtered out for the following reasons: First, employees of Mozilla do not go through the regular contribution process of an outsider and therefore were filtered out. The second largest group of filtered out participant candidates had submitted patches, but the scripts had incorrectly identified them as accepted due to unusual flag combinations in the Bugzilla flag system described in Section 2.2.1. Another reason were developers with multiple accounts or renamed accounts, whom the scripts falsely identified as new developers.

After filtering, 190 developers remained as invitees to answer an online questionnaire. These subdivide in four waves for Mozilla plus two waves for GNOME. Invitations for each wave were sent at different dates, only inviting the developers whose first patch had recently been accepted, so their memories about the first patch were still fresh. The patch was accepted at most three months before the first invitation, with the exception of the invitations of GNOME developers. Figure 2.4 and Figure 2.5 also show the dates when the invitations were send out and the specific number of developers invited. The first invitation wave for Mozilla newcomers did not include all suitable newcomers of April 2013, but only 10 randomly selected newcomers. This helped to test the invitation process and fix possible technical problems.

The invitees were contacted up to three times via email. The number of contacts depended on whether they answered the survey or one of the emails, in which case they received no further emails. The emails included one \$2 gift code for amazon.com for each invitee. This accords to the TDM, which suggests to put a \$2 bank note in the invitation envelope. The TDM prescribes five different types of contacts, like postcards, letters, and priority mail. The lower possible variance in format of emails in contrast to written mail did not allow five different formats for email contacts. In the spirit of TDM, the invitation emails varied in appearance: The first email had a plain text format, the second email had a more fancy HTML format, and the third email was plain text again, but signed with a digital certificate from the University of Duisburg-Essen's Certification Authority (CA). Each email included a personalized link to the web-based questionnaire. The personalization was necessary to restrict each invitee to fill out only one questionnaire and to keep track of the invitees who had not yet filled out the questionnaire. The personalization information is not included in the responses to keep the questionnaire completely anonymous. The link contained an identifier to differentiate between Mozilla and GNOME contributors, though.

Each first email further included a manually written short summary of the invitee's first

2 Contribution Barriers to FLOSS Projects

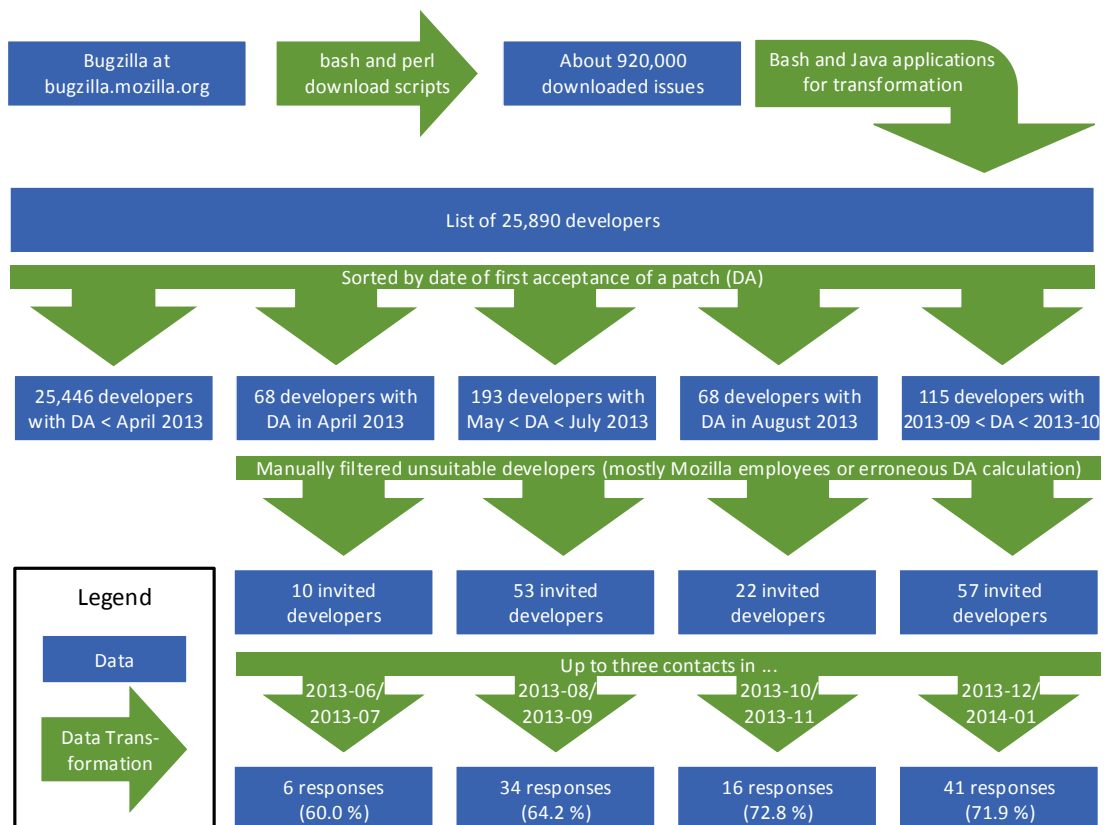


Figure 2.4: Data flow for the selection and invitation of survey participants from Mozilla

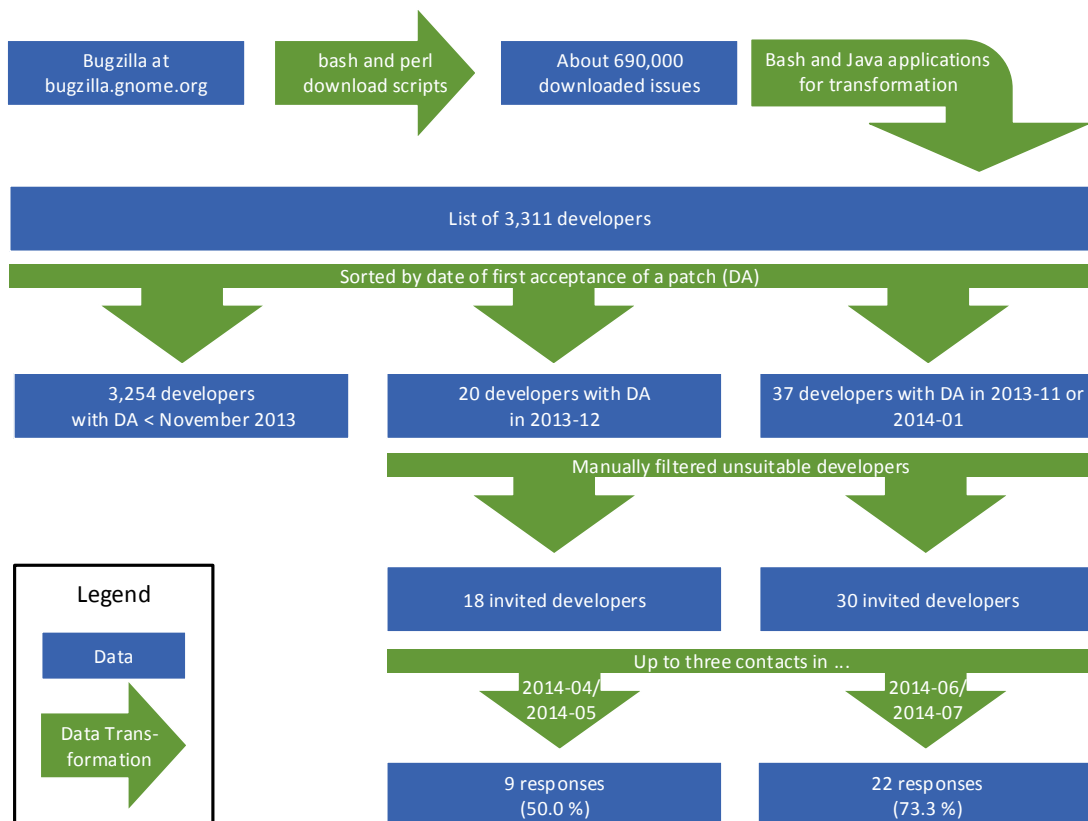


Figure 2.5: Data flow for the selection and invitation of survey participants from GNOME

2 Contribution Barriers to FLOSS Projects

accepted patch to the project. The summary proved that the invitation is no mass mailing but a personal invitation. A link to the issue for which the patch was submitted accompanied the summary. The survey referred to this specific patch in some of the questions and the link served as a reminder and clarification which patch the survey was referring to – this ensured that the selection procedure had correctly identified the first patch and also clarified cases where it is not immediately clear to the invitee which patch was the first. For example, another patch may have been written and submitted earlier, but it still had been reviewed later than the patch referred to in the email.

As shown in Figure 2.4 and Figure 2.5, a total of 97 Mozilla contributors and 31 GNOME contributors responded to the survey. 5 Mozilla respondents and also 5 GNOME respondents had answered only very few questions and especially no questions regarding contribution barriers, therefore their responses did not flow into the analysis. Additionally, 1 Mozilla contributor's answers revealed that this Mozilla contributor is in fact a Mozilla employee. As the analysis specifically targeted contributors from outside, this contributor's response also did not flow into the analysis. The 91 and 26 useful responses out of 132 and 48 invitations for Mozilla and GNOME, respectively, results in a response rate of 68.8 % and 54.2 %, in total exactly 65 %. This response rate is within the typical range of well-designed surveys [Di199, p. 3f]. More importantly, the response rate is higher than all other surveys targeting FLOSS developers to the best of my knowledge, where response rates have been at most 38.1 % [HO01; LW03; WGY07; XJS09; XJ10; BB10; Mai+15].

This paper cites respondents to the questionnaire directly and indirectly. In these citations, a symbol “[Mn]” for Mozilla or “[Gn]” for GNOME identifies the quoted respondent, where n is a number assigned arbitrarily to each respondent within the groups of Mozilla and GNOME.

2.2.4 Demography

The first questions for the participants of the questionnaire asked which programming language they wrote the patch in and the number of files they had to modify. The first questions in a questionnaire are of special importance and the following aspects that Dillman [Di199, p. 92] describes were taken into account: First, the first question should apply to everyone. This is clearly the case, as the submission of a patch was a precondition for the invitation to the survey. Second, it should be easy to understand and answer. As the participants have written the patches themselves, they will have a firm knowledge of what programming language they used. As the invitation email included a link to the ticket in the issue tracker, they could also easily look again at their patch to verify the programming language and number of involved files. Third, the first question should be interesting and relate to the general topic of the survey. Since the participants have taken the time to write and submit their patch, they can be expected to be interested in it. Since it is a creative work of their own, they are probably also rightfully proud of it, and therefore feel positive talking about it. Additionally, they must have at least a basic understanding of the programming language used for the patch, and their answer is proof of this expertise.

Besides these motivating aspects, the two questions also engaged the participants to refresh their memories about their first patch. Thus, the comments about their negative and positive experiences during modification of the source code and submission of the patch will reflect the practice in FLOSS projects more accurately.

Programming Languages

Figure 2.6 shows the answers to the question of which programming language the participants used for their modification. 3 of the 86 patches to Mozilla contained source code in two programming languages, so the percentage total in Figure 2.6 actually comprises 103 % of the responses of Mozilla contributors. The 24 surveyed GNOME contributors wrote their patches in only one programming language. The category Other comprises programming languages that only one or two respondents had used for their first patches. Thus, both projects use a wide range of programming languages, but the composition of programming languages differs between the two projects. For Mozilla, JavaScript was the most common programming language for first contributions, with a ratio of 54.7 % of all first contributions, with C++ as the second-most common with 22.1 % of first contributions. For GNOME, exactly 50 % of all first contributions used C, while VALA was the second-most common programming language for first contributions with a ratio of 20.8 %.

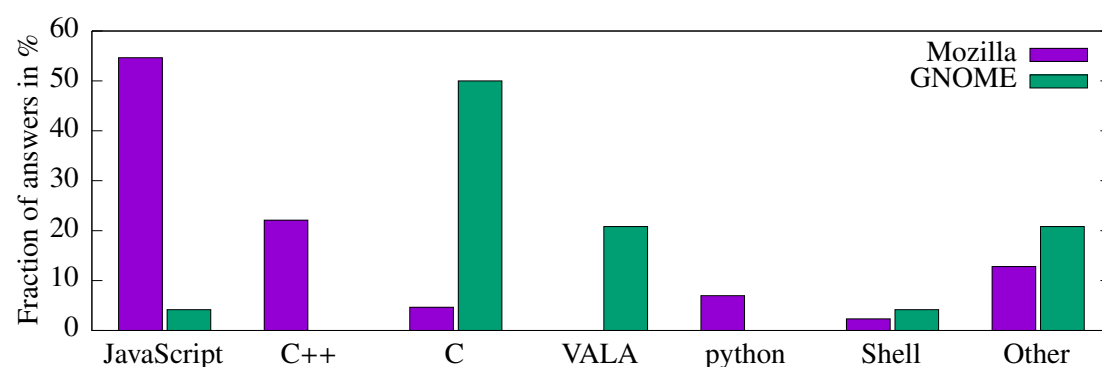


Figure 2.6: Programming languages that the survey participants wrote their first patch in

Occupation

The questionnaire also asked the respondents about their current occupation and their levels of experience in different aspects of software development. In terms of occupation, newcomers to Mozilla and GNOME are surprisingly similar, as Figure 2.7 shows. About 49.0 % of the newcomers are employees. Another 36.5 % of the newcomers are students.

This partition of occupations is similar to results from other surveys of FLOSS developers not restricted to newcomers. However, the ratio of students is higher in this survey: Only 14 % of Hars and Ou's respondents from various FLOSS projects were students [HO01]. 23 % of the Linux kernel developers in 2000 were students [HNH03]. 19.5 % of 684 FLOSS developers working on projects hosted on SourceForge.net [P*Sla15] in 2001 were students [LW03]. Of the 1488 FLOSS developers whom David et al. surveyed in 2003, 28.8 % were students [DWA03]. A later sample of 148 FLOSS developers, mainly from SourceForge.net, included at most 13 % students [WGY07]. In a more recent survey, 13.1 % of 848 R package authors responded that they were students [Mai+15].

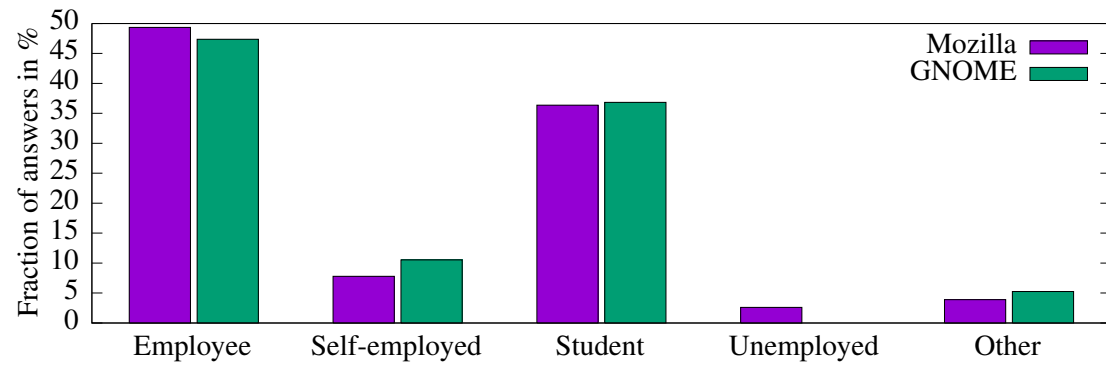


Figure 2.7: Occupation of newcomers

There are three important differences between the target population of existing research with its lower number of students among the contributors compared to the participants of this survey:

First, the existing research is mostly about 10 years older. The structure of the FLOSS contributor population may have changed in the meantime. Future researchers should try to reproduce these earlier results to decide this hypothesis.

Second, the difference might be a peculiarity of Mozilla and GNOME. Mozilla employs student programs like Google Summer of Code [P*Moz14f] and is the subject of some FLOSS university courses as answers to other questions in this survey show. However, this cannot explain all of the difference and it does not explain why GNOME also has a comparably high number of students. The difference is therefore unlikely to be project-specific, especially because earlier research confirmed their results in a wide range of different projects.

Third, this survey targets newcomers while the other research targets FLOSS developers in general. There are two explanations of why this difference in group structure should create a difference in the fraction of students. Firstly, those who are students on their first contribution may have become employees after a while and may show up as employees in later surveys. This would mean that a considerable fraction of developers choose their FLOSS projects as students and then stick with the project. However, Hertel et al.'s participants stayed with the project only for 17 months on average [HNH03], which contradicts that contributors stick for a longer time with a FLOSS project after they finish their university studies. Secondly, maybe students contribute only for a short period and then drop out of the project, while employed contributors stick with the project after their first contributions. Thus, later samples of FLOSS contributors would cover the employed contributors who are still with the project, but not the students who had left after their first contributions already. Which, if any, of these two explanations is correct, should be the subject of future research.

Experience

Table 2.1 shows how the participants self-assessed their experience in different domains relevant for their contribution. Besides the option to skip an answer, the participants had the following four possible choices for each domain:

1. The patch was your first experience with this.
2. Some experience. Needed instructions to work with this.
3. Experienced. Frequently worked on your own with this.
4. Expert. Instructed others in this.

The answers were interpreted numerically as 1 (lowest experience level) to 4 (highest experience level). In the following questions shown to the participants, PROJ was either Mozilla or GNOME, and the terms in parentheses will be used for reference in this thesis. The domains asked for are

- software development in general (**Software Development**),
- the organizational structure and processes of open source projects in general (**FLOSS processes**),
- using the software of project PROJ (**Using PROJ**),
- the organizational structure and processes of project PROJ (**Processes of PROJ**)
- programming language/development environment used in project PROJ (**Software Development Environment (SDE) of PROJ**), and
- the source code of project PROJ (**PROJ's source code**).

Table 2.1: Number of answers for each level of experience and experience domain

Experience	Software Development	FLOSS Processes	Using PROJ	Processes of PROJ	SDE of PROJ	PROJ's Source Code
4	30	6	10	0	8	0
3	31	20	40	3	43	8
2	28	36	17	31	27	27
1	6	35	28	60	17	61
Arithmetic Mean	2.895	1.969	2.337	1.394	2.442	1.448

The answers show that the participants usually considered themselves fairly good software developers, although they were mostly less experienced with the technologies used in the FLOSS project. Some are experienced users of the FLOSS project and also know about the procedures in other FLOSS projects, but for others this is completely new. Almost none, however, had more than rudimentary knowledge about the specific processes and code structure of Mozilla or GNOME.

2.3 Contributor Motivation³

After the demographic questions, the questionnaire asked about contribution barriers and only afterwards about contributor motivation. This order reduced item non-response on the contribution barrier questions, which were in the focus of this survey. For reasons of presentation, the order in this thesis is exchanged: This section presents contributor motivations, while contribution barriers are deferred to Section 2.4.

The questionnaire structures for both contributor motivations and contribution barriers are alike. Both split into the domains of code modification and submission. For all combinations, modification motivations, submission motivations, modification barriers, and submission barriers, there is an open question and a closed question. Each open question is asked before its corresponding closed question to ensure unbiased answers in the open question.

The questionnaire first asked in open questions about the motivation why they used the FLOSS application, why they modified the FLOSS application, and then why they submitted their modification as a patch to the FLOSS project. Usage can be seen as a precondition to contribution [Cro+05]. Therefore, usage motivation influences the motivation to contribute, and the questionnaire includes usage motivation.

Then the participants were presented closed questions about their motivation to use the FLOSS application, modify it, and submit their modification back to the FLOSS project. For modification and submission, the participants could pick motivations that applied for them from a list, and put them into an order that represented the importance of the selected motivations. For usage motivation, the questionnaire presented a multiple-choice checklist. This gave less insight about usage motivation than the method used for modification and submission motivation, but it was also not as much in focus of the study and cost the participants less time.

Using an open coding methodology taken from Grounded Theory [SC94], the answers to the *open* questions received tags identifying the contributor motivation that the participants mentioned. The tags have a hierarchy, so for example, wanting to help the community is a form of altruism, and altruism itself, together with the motivation to foster FLOSS in general, belongs to the more general category “Ideal”. 71 participants explained their modification motivation as an open answer, and 60 participants explained their submission motivation. 82 and 91 tags describe the answers for modification and submission motivations, respectively, so each participant gave 1.15 modification motivations and 1.52 submission motivations.

The pre-defined items for the closed questions match the motivations identified in the existing research [Kri06; LW03]. As previous research did not distinguish between modification and submission motivations, each motivation item appears either in the closed question for modification motivations or the closed question for submission motivations. Additionally, as a result of the pretests and the experiences gained in the exploratory survey, the phrasing of some items are different to those of the existing research.

2.3.1 Usage Motivation

74 participants responded to the open question of why they use project Mozilla or GNOME. Their answers were assigned 91 codes, so each answer was assigned about 1.23 codes. Figure 2.8

³A preliminary version of this section was published previously [HG16b].

shows which codes have been frequently assigned to the answers. 54.1 % of the respondents gave answers coded with *Application* and thereby expressed that they used an application that the project develops, like Mozilla Firefox. 9.5 % of the respondents use a *Library* that the project develops. 24.3 % of the participants engaged with the project with a source code *Modification* and possibly had the submission of their modification in mind. For example, they wanted “to get involved in open source” [M28]. 33.8 % of the respondents explicitly mentioned that the project was “*Open Source*” [M3, M7, M8, . . .], “FOSS” [M2], “FLOSS” [M38], or the like. 9 of these “Open Source” respondents did not specify further why specifically they came in touch with the project, admittedly because the question was not clear enough that this was asked for.

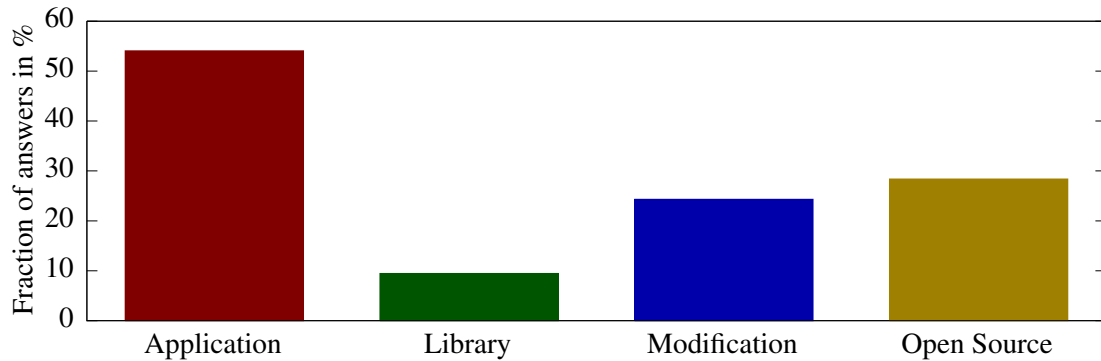


Figure 2.8: Codes of open answers for usage motivation

The pre-defined answers to the corresponding closed question are the x-axis labels in Figure 2.9. Participants could select any number of use cases. If they did not select any, this analysis excluded their answer as nonresponse. Figure 2.9 shows which fraction of the remaining 94 respondents selected each item. The answers show more clearly than the open answers that most respondents, 87.2 % specifically, use an *Application* for themselves. Another use case for 21.3 % of the respondents is using a project *Library* in their own applications. This seems to be more important for GNOME contributors than Mozilla contributors, with 45.0 % and 14.9 % of the respective respondents selecting the usage motivation Library. About 20.2 % of the respondents got in touch with the FLOSS project in order to contribute a modification to increase *Interoperability* with another software. Hence, these respondents possibly favor a competitor over the FLOSS project for usage. 5.3 % of the respondents, and especially 10.0 % of those who contributed to GNOME, provide *Consulting* for other organizations that use Mozilla or GNOME. 7.4 % of the respondents, all of them Mozilla contributors, had *Other* reasons to get in touch with Mozilla. For example, M11 participated in a college student program to contribute to Mozilla, and M60 as well as M65 merely wanted to write code without using the application.

2.3.2 Open Question on Modification Motivation

Figure 2.10 shows the four tag categories assigned to more than 7 % of the respondents’ answers to the open question for their modification motivation. There were 71 respondents to this question, so at least 6 respondents must have given answers that belong to a category to pass the threshold.

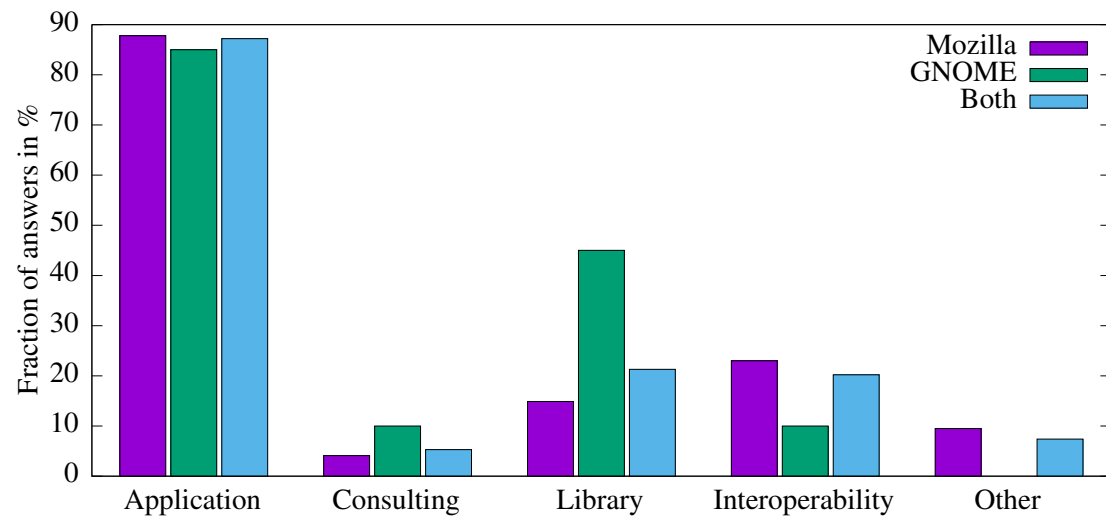


Figure 2.9: Distribution of answers for the closed question of usage motivation

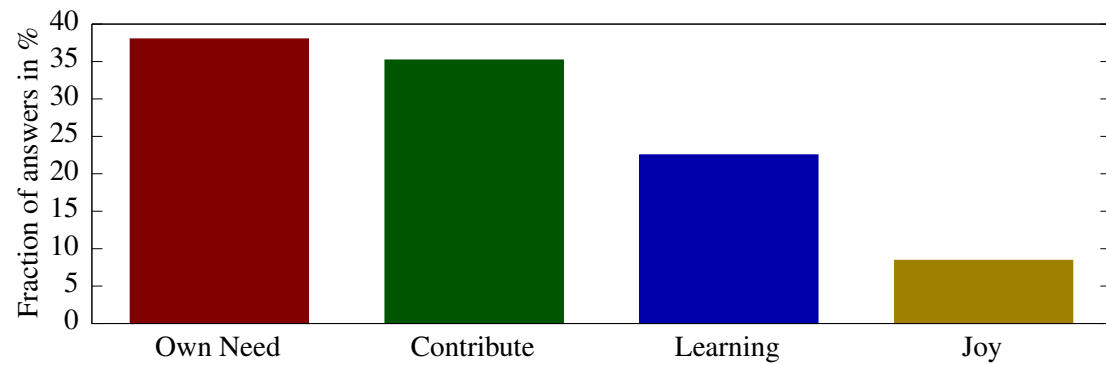


Figure 2.10: Answers to the open question for modification motivation

38.0 % of the respondents answered that they modified the FLOSS application due to their *Own Need*: Either they themselves or their employer had experienced a bug that they fixed with their modification, or they wanted a specific feature that they had implemented with their modification.

As a close second, 35.2 % of the respondents wrote that they wanted to *Contribute* to the FLOSS project to support it. In these cases, the modification is a means to an end, as only the submission of the modification to the FLOSS project eventually improves the FLOSS project.

About 22.5 % of all respondents mentioned that their motivation to modify the FLOSS project was *Learning*. M90 wanted “to learn versioning practices”, M39 wanted to gain “experience working with code written by others” and M7 simply wanted to improve “programming skills”. Respondents in this group all contributed to Mozilla and not GNOME. Learning might be especially important for Mozilla contributors, as some of them contribute to Mozilla as part of a student program like Google Summer of Code [P*Moz14f].

6 respondents, 8.5 % of all respondents, claimed that they started the modification for the *Joy* of programming.

2.3.3 Closed Question on Modification Motivation

Participants had to select modification motivations from a list and rank them in order of importance. Figure 2.11 shows how many participants selected each motivation and ranked them as one of the first three priorities. Different ranks have different colors. The motivations were presented in random order for each participant to rule out any bias because of the order. Figure 2.11 presents them ordered by the number of selections with first rank. A total of 93 participants responded to this question.

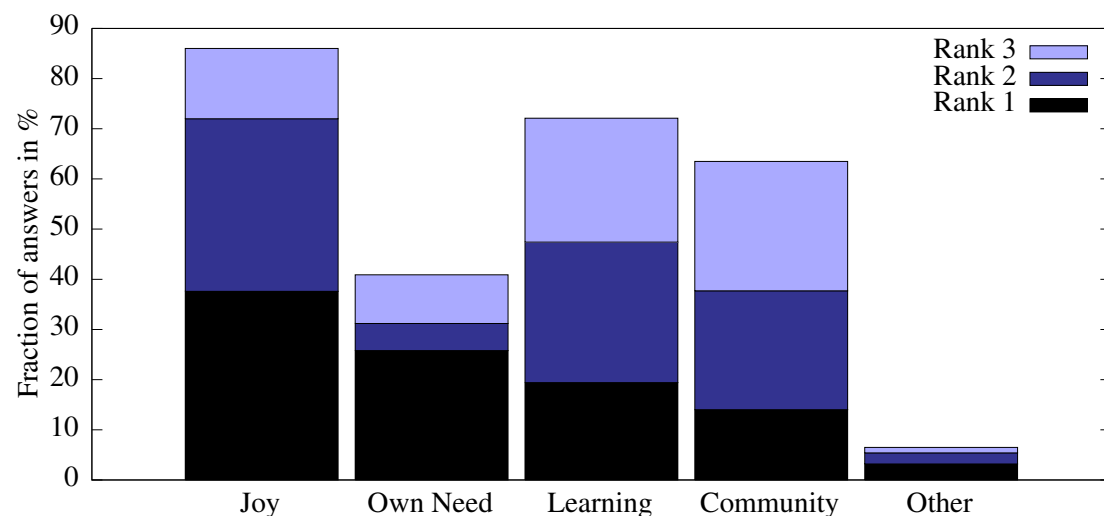


Figure 2.11: Ranked modification motivations

Contrary to the open question, “the *Joy* of programming and/or the intellectual challenge” was the most important modification motivation. For 37.6 % of the respondents, it was the primary modification motivation, and for 86.0 % of the respondents, it was one of the top three

2 Contribution Barriers to FLOSS Projects

modification motivations. Even when considering only those with Joy as primary motivation, this is more than four times as much as the 8.5 % respondents who mentioned the joy of programming in their answer to the open question. There are two explanations for this discrepancy: First, the participants may not be aware that the joy of programming is a possible modification motivation and it was only brought to their attention through the closed question. They possibly saw the joy of programming as a self-evident property of their modification that needed no mentioning. Second, they may have seen joy of programming on another level of abstraction than other motivations. They may have become software developers because they experienced joy when programming generally, but they had more specific motivations for their specific modification asked for in the questionnaire.

About 25.8 % of the respondents modified the FLOSS application primarily in order to satisfy their *Own Need*, phrased in the questionnaire as “A malfunction or missing functionality bothered me/my organization as a user of the project”. Its high importance is not surprising when considering that this was the most often mentioned modification motivation in the answers to the open question and that it was the by far most important modification motivation in the exploratory survey (see Section 2.1.2). However, in deviation to the other modification motivations, only a few respondents ranked this reason second- or third-most important, resulting in only 40.9 % of the respondents ranking this reason as one of the top three modification motivations. Then again, this is put into perspective by another 25.8 % of the respondents ranking this modification motivation fourth.

Learning is the primary modification motivation for 19.4 % and one of the top three modification motivations for 72.0 % of the respondents. More specifically, this modification motivation was labeled “acquiring experience in technologies used by project [Mozilla or GNOME]” for the participants. This is in line with the answers to the open question.

Getting in touch with the *Community* was the most important modification motivation for 14.0 % of the respondents, and one of the three most important modification motivations for 63.4 % of the respondents. The questionnaire phrased this modification motivation as “I like the developers of project [Mozilla or GNOME] and like to work with them”.

Participants could describe *Other* modification motivations in free-text form. About 3.2 % of the respondents ranked a free-text motivation as most important modification motivation, and about 6.5 % ranked it among the three most important modification motivations. Among these, four respondents repeated the frequent answer in the open question that they modified the FLOSS application just to have something to submit. Three respondents explained that they modified the FLOSS application as part of a student project like their thesis, although only one ranked this among the three most important modification motivations.

2.3.4 Open Question on Submission Motivation

Figure 2.12 shows which answers the respondents gave to the open question about their submission motivation. Analogously to the open question on modification motivation, each submission motivation mentioned in the free text of an answer was assigned a code. These codes had a hierarchy to group classes of codes. The analysis includes only codes assigned to at least 7 % of the respondents’ answers. Among these, Figure 2.12 shows two hierarchy levels, with Ideal, Personal, and Economic on the higher level of abstraction and the remaining motivations as more

specific submission motivations. The trees below the graph indicate which motivation codes belong to which more abstract category.

Unlike other open questions in this questionnaire, a considerable number of statements could not be assigned codes that unambiguously represent submission motivations, and were not included in the analysis. 13 respondents explained their submission motivation similar to M28: “This is how my patch gets in the official source tree of Mozilla”. These answers suggest that the submission is an end in itself and the respondents do not seem to consider that it would be an option to keep the modification for themselves. While this indicates altruism as a matter of course, these answers do not clearly and unambiguously imply the true motivation for submission.

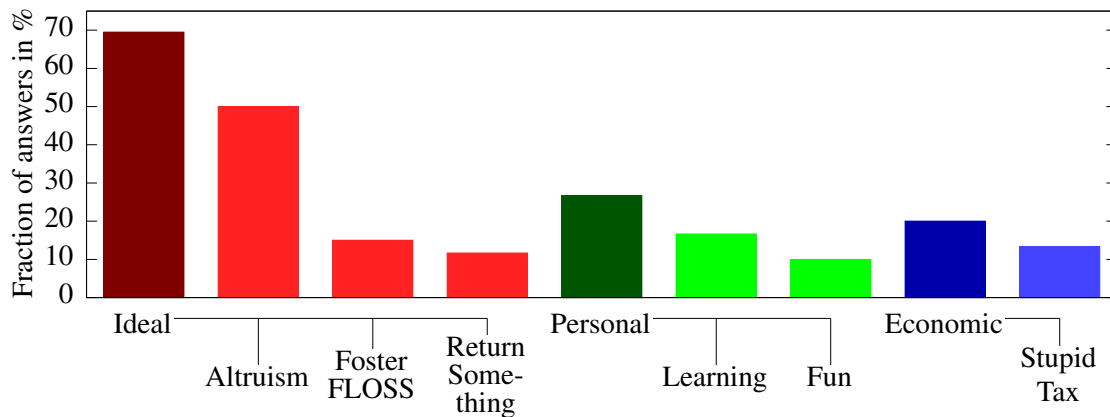


Figure 2.12: Answer frequency to open questions for submission motivation

Exactly half of the respondents justified their submission to the FLOSS project as an act of *Altruism*. M38’s reason “to make it available to everyone” is an example of a direct answer, M50 explained a bit more indirectly “to help out”. However, 20.0 % of the respondents who were in the altruist category just circumscribed that they wanted to “make it [Mozilla] better” [M8], possibly inspired by the slogan “Made to make the Web a better place” of Firefox 4 [P*Moz11] and “make the Internet an ever better place for everyone” in The Mozilla Manifesto [P*Moz08b]. Instead of accounting this as altruism, this explanation might mean that the contributor merely wants to create something beautiful like good software, without the explicit intent to make a positive impact on the software’s users. However, no respondent enunciated this non-altruistic meaning of making Mozilla better, and, to the contrary, some of these respondents seem to take it for granted that helping other users is a strong motive by itself.

15.0 % of the respondents wanted to *Foster FLOSS* in general or Mozilla in particular. About 11.7 % of the respondents felt a commitment to *Return Something* to the FLOSS project for the benefits gained through the usage of the FLOSS application. Together, the three preceding *Ideal* submission motivations accounted for about 69.5 % of the respondent’s answers.

16.7 % of the respondents submitted their modification to the FLOSS project in hope of *Learning* something about the contribution process, for example as part of a student project or in order to improve skills needed in their future careers. 10 % of the respondent submitted the modification for *Fun*. This includes cases like M49’s, who found it “very satisfying” that the “product now has my [M49’s] contribution”. Learning and Fun constitute the main components

2 Contribution Barriers to FLOSS Projects

of the major category of *Personal* submission motivations. About 26.7 % of all answers belong to this major category. Answers in this category describe intrinsic submission motivations with personal benefits the contributor expects.

The last major category describes extrinsic submission motivations for *Economic* benefits to the contributor, which 20 % of the respondents mentioned. As the only relevant submission motivation in this major category, about 13.3 % of the respondents wanted to avoid paying the so-called *Stupid Tax* [P*The12; P*Ell10] of FLOSS: Reintegrating a self-written modification into every new official release of the FLOSS application. When the modification is submitted back to the main project, their maintainers take care of the modification and the modification's developer can use the off-the-shelf version of the FLOSS application.

2.3.5 Closed Question on Submission Motivation

Figure 2.13 shows which items the respondents selected as submission motivations in the closed question. As in the closed question for the modification motivation, the participants had a randomly ordered, but pre-defined list of submission motivations that they could select and had to bring into an order of importance. Figure 2.13 regards only the three submission motivations ranked as most important. It is ordered by the fraction of respondents selecting each motivation as their primary submission motivation. In total, 90 participants responded to this question.

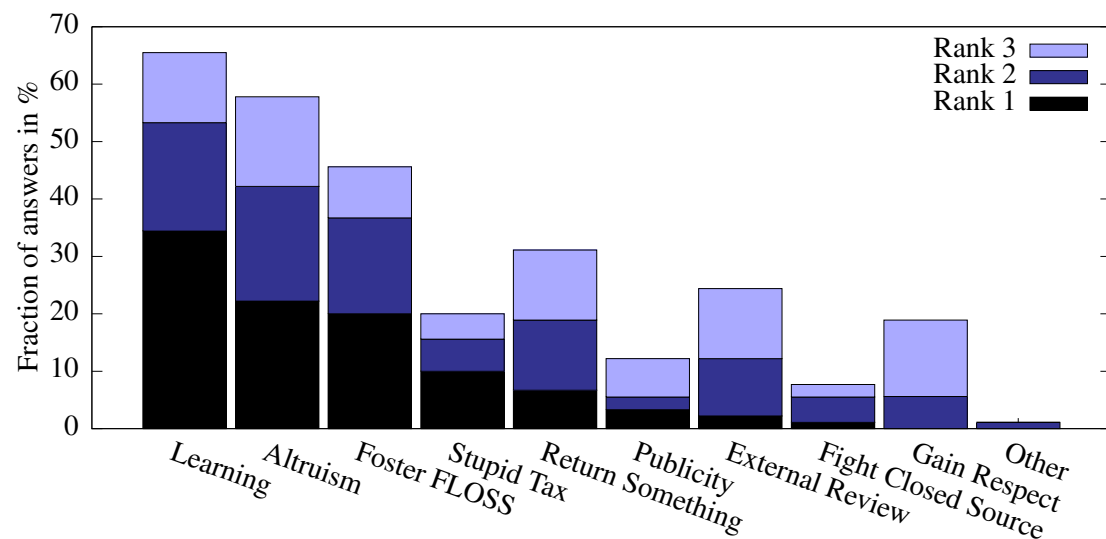


Figure 2.13: Ranked submission motivations

For 34.4 % of the respondents, the primary submission motivation was to “gain experience in the procedures of (OSS) projects”, i.e. *Learning* something. For 65.6 % of the respondents, Learning was among the three most important submission motivations, so both metrics indicate that Learning is the most important submission motivation. Interestingly, Learning appears to be more important when looking at the closed question as compared to the answers to the open question. Possibly, the respondents became aware of their motivation only through the suggested answers in the questionnaire.

As explained in Section 2.2.4, about 36.5 % of the respondents were students. This comparatively high fraction of student participants at least partially explains why Learning is so important for the respondents, as students can be expected to see Learning as an important motivation: Learning about FLOSS could be part of the students' curriculum or a desire to learn made them both students and FLOSS contributors. And indeed, 51.4 % of the student respondents ranked Learning as the most important submission motivation.

Altruism was the most important submission motivation or one of the three most important submission motivations for 22.2 % or 57.8 % of the respondents, respectively. Similarly, 20.0 % respectively 45.6 % *Foster FLOSS*, or, more specifically, “publish the source code, because [they] believe source code should be open”. Together with those 6.7 % respectively 31.1 % who wanted to *Return Something*, and the 1.1 % respectively 7.8 % who wanted to *Fight Closed Source*, these four motivations constitute Ideal submission motivations which appear to be the most important group of submission motivations. This is in line with the answers to the open question.

10.0 % of the respondents submitted their modification to the FLOSS project, primarily in order to “avoid the work of reintegrating [their] changes into new releases of the OSS project”. 20.0 % of the respondents had this *Stupid Tax* among the three most important submission motivations, so this submission motivation seems to be polarizing. Contrarily, getting an *External Review* of the modified code from experts was the primary, or one of the most important three submission motivation, for 2.2 % or 24.2 % of the respondents. This is surprising as these motivations are similar in that they are the two motivations with technical benefits to the submitter.

3.3 % respectively 12.2 % of the respondents wanted to gain *Publicity* “(for example, you might have wanted to get job offers)”. 0.0 % respectively 18.9 % wanted to gain *Respect*. Gaining respect seems to be a common additional submission motivation besides others, while publicity is a strong enough submission motivation to work as the primary submission motivation – but may have an egoistic and therefore negative sound that comes with it, and therefore participants do not pick this submission motivation unless it was very important for them.

Participants who felt that the pre-defined items did not represent their submission motivation could pick *Other* and describe this submission motivation in a text box. Only participant M84 used this to explain that the company whom M84 worked for was paid to develop the submitted feature, though this was only the second-most important motivation for M84.

2.3.6 Comparison to Related Work

This section compares the motivations of newcomers as found in this survey to motivations of FLOSS contributors in general as identified in previous research. A quantitative or even statistically sound comparison is not possible, though, as the methodology and results vary between studies. Additionally, as explained in Section 2.2.4, the demographic structure of this survey's participants is different to the demographic structure of previous surveys' participants. The reasons for these differences are not entirely clear. These differences, like the fraction of students in the set of survey participants, influence the motivations to contribute to FLOSS projects [LW03].

In Hars and Ou's survey, Learning was the primary and self-determination the secondary motivation [HO01]. Self-determination includes hedonism and therefore the Joy of Programming. Lakhani and Wolf [LW03] found learning as the third-most important reason for contribution. The

2 Contribution Barriers to FLOSS Projects

two more important were the Joy of Programming and what is dubbed Own Need in this paper. This is also in accordance with this survey's results. David et al.'s questionnaire items are too different to be compared with this survey's results [DWA03]. Hertel et al. found that hedonistic motivation had the strongest agreement from contributors, and found "pragmatic motives" and "social/political motives", to which Learning belongs, as next-most important [HNN03]. Their results are difficult to compare, though, as their factor analysis categorized motivations differently than the survey in this thesis.

Previous research identified hedonism and Learning as primary motivations for participation in FLOSS projects, although the scope of these categories varies from study to study. In terms of this survey's categories, Joy belongs to hedonism and is usually a modification motivation. Learning can be either a modification or submission motivation and previous research did not distinguish between these two different activities. This survey's results indicate that Learning is more important as a submission motivation and less important as a modification motivation. This confirms Ducheneaut's analysis that newcomers already have good programming skills and want to learn how to contribute to large projects [Duc05, p. 352].

For the participants of this survey, Altruism and similar Ideal motivations are more important than in preceding research. One reason may be that developers have Joy or Own Need as modification motivations, and Altruism is only an important motivation for the submission – submission motivations may go unnoticed in preceding research because they have only one category of motivations. As another explanation, Rullani argued that monetary and signaling-related motivations increase in importance over the time after a developer has joined a FLOSS project [Rul06]. It is therefore unsurprising to see that these types of motivations are not important in a survey of newcomers.

2.4 Contribution Barriers

As explained in Section 2.3, the questionnaire asked for contribution barriers immediately after the demographic questions described in Section 2.2.4, and before the questions for contributor motivation. The questionnaire asked in two open questions about the "biggest barriers" to *modify* the source code of the project and to *submit* the patch to the project. These are the modification and submission barriers, respectively. Subsequent to the open questions, the questionnaire presented predefined contribution barriers for code modification and patch submission. Differently to the closed questions for contributor motivation, participants were prompted to rate each predefined contribution barrier on a scale of one to five, where one means "no obstacle at all" and five is "almost a show stopper". As an alternative to the one to five rating, participants could select "not applicable". This was treated like a one in the evaluation, though, as a contribution barrier that is not applicable is also not an obstacle. The option "not applicable" was included because pretests had shown that it increased participants' understanding of the question and their answer rate. In all cases, they could explain their choice in a text box for each contribution barrier. The differences in format between the closed questions for contribution barriers and the closed question for contributor motivation take into account that every participant was motivated to modify the FLOSS application and submit the modification back, whereas contribution barriers did not necessarily occur.

Analogously to the open answers for the contributor motivation questions, the answers given to the open questions for contribution barriers received codes. Since the contribution barrier questions come earlier in the questionnaire, item non-response was lower than for the contributor motivation questions. 91 and 90 participants responded to the open question asking for modification barriers and submission barriers, respectively. These answers were assigned in total 161 and 127 codes, respectively, including 24 and 33 explicit respective mentions that the modification or submission was easy. Thus, each respondent mentioned a mean number of 1.51 and 1.04 contribution barriers, respectively.

The contribution barriers asked for in the closed questions are similar to the items in the exploratory survey. In contrast to the exploratory survey, the items are grouped in submission and modification barriers, and there are a few additional contribution barriers. They still represent typical steps that a newcomer has to go through, as presented in Section 1.1.4. The items therefore systematically comprise all possible contribution barriers. However, relevant contribution barriers might apply to multiple steps or one step might comprise multiple contribution barriers. Therefore, the open questions are important, as they might, for example, mention a specific tool as a contribution barrier that is used in multiple steps. Thus, they may also refer to cross-cutting concerns. Using this two-way approach, the results contain contribution barriers seen from multiple perspectives.

2.4.1 Open Question on Modification Barriers

Figure 2.14 shows codes for the answers to the open question for modification barriers. The graph shows all codes that at least 7 % of the respondents mentioned and omits those mentioned less often. This threshold corresponds to at least 7 mentions. This threshold applies to codes anywhere in the hierarchy, be it major codes standing on its own or subcodes that belong to another code. For each code, a vertical bar marks the fraction of respondents whose response received the code directly or as a subcode. The x-axis labels the codes and tree lines designate which codes are subcodes of others.

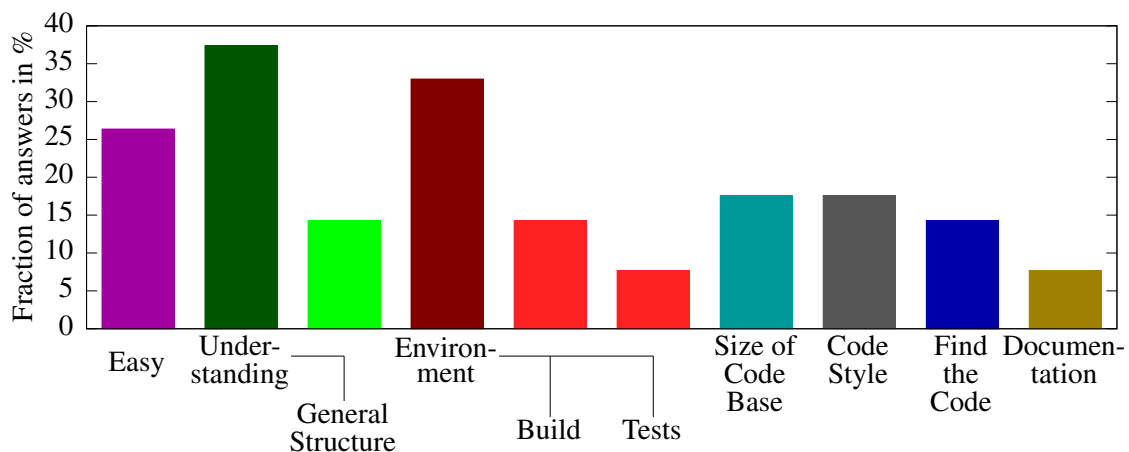


Figure 2.14: Fraction of codes assigned for the open question on modification barriers

2 Contribution Barriers to FLOSS Projects

In their answer to the open question to the biggest modification barriers, 26.4 % participants explicitly mentioned that the modification was *Easy* to do or that they did not encounter noteworthy modification barriers.

Understanding comprises different types of missing prerequisite knowledge. As the most important subcategory, understanding the *General Structure* of the project was the biggest hurdle for about 14.3 % of the participants. Interestingly, General Structure was a modification barrier only for Mozilla contributors, but not for GNOME contributors.

About 33 % of the respondents reported problems with the *Environment*, which made it the second-most mentioned main category for modification barriers. As a subgroup of Environment, 14.3 % of all respondents had a problem with the *Build* of their project, for example because it took too long. As an example for why the build can be such an obstacle, participant M64 “had to set up a whole ecosystem of interdependent projects” for the build. 7.7 % of the participants found *Tests* of their modification difficult. The group of participants with build problems and those with test problems are not distinct, though, as tests require a build and sometimes the test is the only reason for the build.

17.6 % of all respondents mentioned the *Size of the Code Base* as a hurdle, often without further explanation why this is a hurdle. Some respondents’ further comments and their answers to other questions suggest that two explanations exist: First, it is difficult to trace an observed behavior of the application to a location in the source code. Thus, it is not clear which part of the source code needs modification. 14.3 % of the participants answered even explicitly that it was difficult to *Find the Code* to be modified instead of referring to the Size of the Code Base. Second, the large code base inhibits understanding the project’s structure and therefore side effects of modifications are difficult to predict.

Another 17.6 % responded with syntactical issues of *Code Style* as the biggest barrier to code modifications. An example is the question whether to indent with four spaces or a tab. Consistent code style is a valid requirement and increases the comprehensibility of the code [OC90]. However, given the high fraction of respondents who see this as their biggest issue, the negative impact of its rigorous enforcement may outdo its benefits. This specific contribution barrier may also be seen as a submission barrier, as compliance with the code style rules is enforced only on submission.

Slightly above the threshold of noteworthiness, 7.7 % of respondents mention a lack of *Documentation* as a contribution barrier.

2.4.2 Closed Question on Modification Barriers

Table 2.2 lists the answers to the closed question on modification barriers. Participants were presented a list of modification barriers corresponding to the columns of the table and rated each of them. The table lists how many respondents rated each modification barrier at each level of importance, where 5 corresponds to the highest and 1 to the lowest importance. The bottom row lists an arithmetic mean for each modification barrier, calculated by interpreting the five levels of importance as the numbers 1 to 5.

The modification barriers were generally ranked low, with all except Find the Code ranking lower than 2.0 as arithmetic mean. This further confirms the result from the open question that a recognizable fraction of newcomers do not experience noteworthy difficulties with the

Table 2.2: Number of answers for each level of difficulty and modification barrier in the closed questions to modification barriers

Importance	Down-load	Setup	Build	Bug Repro.	Redundant Work	Find the Code	Solution	Community	Other
5	1	1	1	2	2	2	0	1	1
4	2	8	5	3	3	8	3	1	0
3	7	11	18	7	6	24	13	9	0
2	17	24	23	22	16	29	35	13	1
1	71	54	48	57	65	36	47	69	70
Mean	1.418	1.755	1.821	1.582	1.489	2.101	1.714	1.409	1.069

modification in their first contribution. However, a contributor may refrain from contribution already if only a few contribution barriers are too high while all others can be low. Therefore and because the highest contribution barrier differs from contributor to contributor, low means for each individual contribution barrier do not imply that contribution barriers had no impact.

The modification barrier *Find the Code* received the greatest arithmetic mean rank of 2.101. The category with the same denomination for the open question on modification barriers seemed much less important, but, as discussed above, those mentioning the Size of Code Base as a contribution barrier in the open question may have also had trouble to Find the Code which they wanted to modify. Interestingly, writing the actual code for the bug or feature, the *Solution* of the problem, has a lower mean rank of 1.714. Contrary to the other modification barriers, nobody found the Solution to be a “show stopper” (rank 5), and the number of answers ranking 4 on Solution is lower than its mean suggests.

Solution and possibly Find the Code comprise the essence of the contribution, the shaping of the abstract structures of the software [Bro87], while all other contribution barriers consist entirely of accidents: difficulties imposed by man-made processes and rules. Considering this, it is interesting to note that two other modification barriers have greater arithmetic means than the actual Solution. The category *Build* matches the open answer of same denominator and has an arithmetic mean of 1.821 for the closed answers. The column *Setup* represents “difficulties installing the development environment” and its rankings average on 1.747.

Contributors who had fixed a defect in the application instead of adding a new feature usually first have to find inputs on which the application fails in order to understand the cause of the failure. Issues with this *Bug Reproduction* rank on an arithmetic mean of 1.582. When interpreting the difficulty of this modification barrier, it has to be taken into account that this step does not apply to new features and also does not apply on some defects that the contributors themselves experienced as users and therefore started to work on the defects in the first place. If a defect is assigned to a newcomer, the chance that Bug Reproduction is a considerable modification barrier may be higher than the numbers suggest.

Column *Redundant Work* lists answers for “concerns of wasted modification effort, as someone else might work on the task in parallel”. GNOME and Mozilla usually assign developers to

2 Contribution Barriers to FLOSS Projects

issues in their issue tracker Bugzilla to prevent this problem. However, newcomers usually lack the rights to change issue assignments, and may therefore start working on the solution before someone with the appropriate rights can assign the issue to them. With a mean rank of 1.489, this contribution barrier is ranked less than the majority of modification barriers.

The column *Download* lists how participants ranked the “difficulties downloading the right version of the source code”. Developers may have trouble downloading the source code because they are unfamiliar with the VCS [SAA10]. Four developers explained that the download was difficult because they were unfamiliar with Mozilla’s VCS Mercurial (hg). Especially large FLOSS projects may have multiple VCS repositories, so developers have to find out which of these holds the correct version of the source code. For example, Mozilla has 20 repositories in the top level category of their VCS hg and 25 sub-categories each containing multiple further repositories [P*Moz15a]. Some repositories are hosted on GitHub instead of hg, some are just mirrored to GitHub [P*Moz15b]. This can confuse newcomers who are not yet familiar with the repository structure. With an arithmetic mean rank of 1.418, this is still a minor modification barrier.

Ranked least are problems with *Community Support*, having an arithmetic mean of 1.409. This positive impression of the community also shows in several answers to open questions, for example M18 explains that “once you’re through to IRC most of the job is done. You get pretty handy tips to modify the source code.”

Most participants either omitted the item *Other* or explicitly noted that there were no other obstacles, resulting in a mean of 1.069. This indicates that most modification barriers arise in one of the given categories. Two answers to Other were reassigned to contribution barriers for submission, because the participants’ comments for their answer described problems with the submission. The other two answers other than 1 are M72’s answer of rank 2, explaining that adapting to coding conventions was difficult, and G18’s rank 5, who was worried that component libgxps would have no future releases and therefore G18’s contribution in libgxps will never be released. This concern about wasted effort may be a more important modification barrier in FLOSS projects that are less active than Mozilla and GNOME.

2.4.3 Open Question on Submission Barriers

Figure 2.15 summarizes the results of the open question on submission barriers. Its x-axis shows codes that were assigned to answers during open coding. Again, the list contains only codes that at least 7 % of respondents mentioned, which are at least 7 respondents. For each code, the position on the y-axis marks the fraction of respondents who mentioned this submission barrier. The x-axis’ labels have a tree structure that shows whether a code stands on its own or whether it belongs to another code in the code hierarchy.

As mentioned above, respondents mentioned a mean of 1.51 modification barriers per answer, but only 1.04 submission barriers per answer. Additionally, 36.7 % of the responses explicitly stated that the submission was *Easy*, as opposed to the 26.4 % who found modification easy. This suggests that newcomers face more problems while writing their modification than while submitting it to the FLOSS project.

Still, some reoccurring submission barriers exist. As the most frequent submission barrier, 32.2 % of the respondents reported problems with the tools and the *Environment* used in the

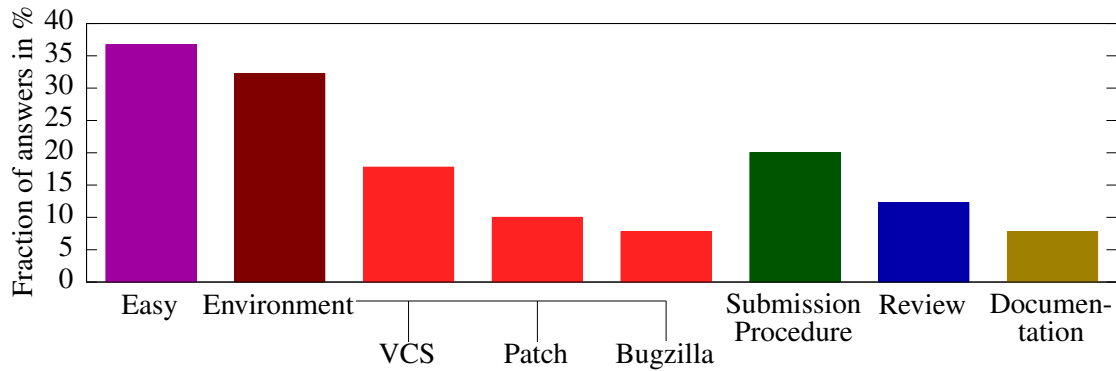


Figure 2.15: Fractions of submission barriers mentioned in the open question

submission process. More specifically, 17.8 % of all respondents experienced difficulties with the *VCS*, for example when creating a pull request on GitHub or when “dealing with an ever changing git [tree]” [M8]. Proper preparation of the contributor’s modification causes these difficulties: Depending on the project or Mozilla subproject, the modification has to be submitted as GitHub pull request or as a patch file created with the tool diff. In any case, the modification must refer to a specific version of the source code. After clearing this difficulty, the *Patch* file creation was another difficulty for 10.0 % of all respondents. For example, M64 wondered whether to “submit the patch as multiple commits for readability, or squash them as a single commit for conciseness”. 7.8 % of all responses criticize GNOME’s and Mozilla’s issue tracker *Bugzilla* explicitly. Although G17 went as far as claiming that “Bugzilla is modeled after a 1970s soviet department of motor vehicles”, other respondents explicitly praise Bugzilla and therefore counterbalance the criticism. Hence, the answers to this questionnaire show no distinct public opinion about Bugzilla, instead Bugzilla is polarizing.

20.0 % of the respondents had problems with the *Submission Procedure*. Some of the respondents explained that they just did not know where to start or what they were expected to do next. In other cases, form fields intended only for specific types of modifications confused contributors when these form fields did not apply to their kind of modification.

For about 12.2 % of the respondents, the core developers’ *Review* of their patch was among the highest submission barriers. For example, “apparently nobody noticed the patch” of M82 and those of two other respondents. Those who searched for reviewers actively had to “[f]ind the proper engineer to review the patch” [M31]. After a reviewer was assigned, willingness to contribute decreases steadily if “the review process takes too long” [M19].

7.8 % of the respondents either did not find enough *Documentation* about the submission process required for the module they contributed to, struggled to find applicable and up-to-date documentation in the bulk of other documentation, or needed too much time to read all documentation that applied to their submission.

Five responses describe purely psychic rather than technical submission barriers. For example, M16 “ended up feeling dumb” because the reviewer rejected early versions of the patch due to code style problems. M45 and M60 indicated that they were afraid to lack the skills to solve the problem, but eventually found the modification not very difficult. Admitting psychic contribution

2 Contribution Barriers to FLOSS Projects

barriers can be embarrassing, even in an anonymous questionnaire [TB10]. Therefore, some survey participants may have kept quiet about psychic contribution barriers and the number of cases with those problems may be higher than the five participants who actually wrote about these problems.

2.4.4 Closed Question on Submission Barriers

Analogously to Table 2.2 for modification barriers, Table 2.3 lists the answers to the closed question on submission barriers. Again, each column represents a submission barrier that the participants could rate on a scale from 1 to 5, where 1 is the lowest and 5 the highest importance. The rows contain the number of participants that ranked a contribution barrier with the particular importance of the row. The last row shows the arithmetic mean rank for each submission barrier.

Table 2.3: Number of answers for each submission barrier per level of difficulty in the closed questions for submission barriers

Importance	Submission Procedure	Documentation	Issue Tracker	Bureaucracy	Member Attitude	Other
5	4	0	0	1	2	0
4	6	6	1	2	0	0
3	21	12	11	6	1	0
2	22	24	24	13	12	1
1	47	51	62	65	84	63
Mean	1.980	1.710	1.500	1.402	1.222	1.016

Although the answers to the open question on submission barriers suggested that less participants had difficulties with the submission than with the modification, the submission barrier *Submission Procedure* has a relatively high mean of 1.980, making it the highest overall contribution barrier after Find the Code. When only considering the levels 4 and 5, Submission Procedure is ranked even higher than Find the Code and all other contribution barriers. This indicates that although the submission procedure works for a considerable fraction of contributors, it is a high contribution barrier for some. Respondents explained high rankings with difficulties using the VCS in the way the FLOSS project expected them to. Although partly, this seems to be a matter of personal preference: On the one hand, M47 explained that “patch-based submissions is [sic] a lot of effort compared to pull requests” as they are common on GitHub. On the other hand, M11 explained the selected importance of 4 with “little experience with git” used for the Mozilla Webmaker component to which M11 contributed.

With an arithmetic mean rank of 1.710, issues with the *Documentation*, called “Instructions on homepage” in the question, were more important than the answers to the open question suggest: While problems with the documentation were barely worth mentioning when asked in an open question, respondents rated lacking documentation regarding the submission on par with the modification barrier Solution. Respondents might have described their problems more directly in the open question, while they came aware that they could have avoided the problems if the

documentation would have been better after seeing the option Documentation in the closed question.

The *Issue Tracker*, which is Bugzilla for both projects, was ranked with a mean of 1.500 when asked how important as a contribution barrier it is. M72 ranked the Issue Tracker as a submission barrier highest among all respondents, with rank 4. M72 explained that Bugzilla dictates parts of the submission process that was especially difficult. This could also be seen as an issue of the Submission Procedure. Thus, there are only minor submissions problems distinctively caused by the Issue Tracker.

With an arithmetic mean rank of 1.402, “*Bureaucracy/Paperwork*” are usually not an issue for the participants. Those who ranked Bureaucracy high explained that it was difficult to create an account for Bugzilla. However, all contributors had to create accounts for Bugzilla, but 65 of them did not see Bureaucracy as a submission barrier. As an explanation for why this is a problem only for a few participants, Mozilla offers two types of accounts for Bugzilla: first, a login with email address and password as it is common among many websites, and second, authentication via Mozilla Persona [P*Moz15r]. Some contributors might be unaware of the choice and then select Persona even if it is badly suited for these contributors’ context like browser environment or email provider.

The participants could also rate whether the “attitude of project members” constitutes a problem for them. This *Member Attitude* received the lowest mean of 1.222 among the predefined submission barriers. The attitude of project members is therefore generally not an issue, and 10 respondents even praised the kindness of the Mozilla developers in their explanation of this ranking. Contributors to GNOME wrote no such praise, but this peculiarity may have occurred by chance.

Three answers in the category *Other* were reassigned to Submission Procedure, as these participants explained their choice with problems of patch file creation and the VCS, which other respondents regarded as a problem of the Submission Procedure. The one remaining answer not ranked 1 saw follow up work as a submission barrier, although only a minor submission barrier.

2.4.5 Discussion

When a FLOSS project grows to a large size, this manifests in different symptoms that can be interpreted as modification barriers. Consequently, participants of this study and also existing research observe multiple of these different modification barriers and their relationship is not directly visible. Manifestations in the open question for modification barriers include the visibly large Size of the Code Base and more indirectly the difficulties to acquire an overview over the organizational or technical structure of the FLOSS project. A manifestation in the closed question for modification barriers is the difficulty to Find the Code. For both questions, these manifestations are the biggest modification barriers. Steinmacher et al. categorizes difficulties of understanding as “cognitive problem” [Ste+14a], suggesting that this depends on the contributor and not on the FLOSS project and disqualifying it from this thesis’s definition of a contribution barrier as per Section 1.3. Although people are differently susceptible to difficulties of understanding, as they are to any contribution barrier, the existence and the impact of difficulties of understanding depends on the FLOSS project. The size of the FLOSS project is one example, but this also includes countermeasures and particularly a lack of countermeasures, which is also a manifestation as

2 Contribution Barriers to FLOSS Projects

a modification barrier. Possible countermeasures are better documentation of different types including code comments or a simpler architecture that is easier to understand. Steinmacher et al. found relatively high numbers of evidence for these individual manifestations that spread around different categories [Ste+14a]. This further confirms that this is an important modification barrier.

Sim and Holt [SH98] suggest that there is another individual factor besides cognitive capacity influencing whether an individual is affected by difficulties to understand the general structure of a software development project or not. They noticed that some newcomers approach a project top-down and first try to get an overview before they dive into the individual parts. Others try to understand a software system bottom-up and look at specific parts of the source code that add up more and more to an overall understanding of the software system. Those with the bottom-up approach seemed to have less trouble understanding the general structure of the project, because they could extend their comprehension in small increments.

Another important contribution barrier documented by this survey are difficulties with the technical Environment of the FLOSS project. This is both the second-most important modification barrier and was most frequently cited in the open question for submission barriers. It is also frequently mentioned in research: Sim and Holt recognized that setting up the development machines is frustrating [SH98] and Dagenais et al. identified “Long IDE installation” as hurdle for newcomers [Dag+10]. Installation issues are the second-most frequently mentioned contribution barrier in the diary studies of Davidson et al. [Dav+14a]. In their interviews, Steinmacher et al. count the contribution barrier “Building workspace locally” most often, and find evidence in their three other data sources, too [Ste+14a].

According to the closed question for submission barriers, the Submission Procedure is the greatest submission barrier. Every fifth answer to the open question for submission barriers also mentions this submission barrier. Furthermore, Jensen and Scacchi argued that understanding the submission process is a contribution barrier by itself already, which may be equivalent in importance to the technical barriers [JS07].

Steinmacher et al. [Ste+14a] also defined contribution barriers that map to problems with the Submission Procedure. These are “poor ‘how to contribute’ available” and “newcomers don’t know what is the contribution flow” in the category “newcomers need orientation”, and “lack of information on how to send a contribution” in the subcategory “change request hurdles” of “technical hurdles”. They report of only sparse evidence for these barriers given that it turned out to be important in this survey. The cause for this difference may be Steinmacher et al.’s smaller data set.

Obviously, finding a solution for the issue and implementing it in source code can be difficult. This task is the core of the modification and it can also be a modification barrier if it is too difficult. The survey answers show this, although only in the closed question on modification barriers and not in the open question. As explained above, other modification barriers are more important. Sim and Holt observed already that programming has little potential for frustration, even if it is time-consuming [SH98]. More generally, essential tasks may be less frustrating than accidental tasks, because the latter seem like unnecessary and therefore wasted effort. Steinmacher et al. identified “bad code quality” and “bad design quality” as contribution barriers and especially for the former, they found evidence in three of their four data sources [Ste+14a]. These contribution barriers may affect the Solution. This survey found no evidence for the two particular barriers “bad code quality” and “bad design quality”. Mozilla and GNOME possibly have a good Quality

Assurance (QA) and therefore do not suffer much from bad code and design quality.

The results from the survey show that the review process can be problematic, as this is the third most important submission barrier in the open question. Begel and Simon also present anecdotal evidence about problems to receive approval for a patch [BS08, Section 4.2.2]. For FLOSS projects, Sethanandha et al. agree that reviews may need intolerably much time [SMJ10a]. Baysal et al. confirm this specifically for Mozilla [Bay+12]. Steinmacher et al. distinguish between an “issue to create a patch” and a “delay to get contribution accepted/reviewed” [Ste+14a]. Thongtanunam et al. [Tho+15] present empirical evidence that confirms delays of 12 days in arithmetic mean for reviews with assignment problems. They show that, depending on the specific FLOSS project, 4 % to 30 % of all reviews have assignment problems. In summary, slow and obscure reviews are a well evidenced submission barrier.

Some practitioners [Tur14] as well as researchers [Dav+14a] see social contribution barriers as more important than technical contribution barriers. However, they present only weak empirical evidence for their claims. Jensen found empirical evidence that slow answers to newcomers’ questions constitute a contribution barrier [JKK11]. Steinmacher et al. reserve two of five categories exclusively for social contribution barriers and therefore underpin their importance [Ste+15a]. In contrast, respondents of this survey evaluated social issues as the least important contribution barrier in the closed question for both modification and submission barriers. In the open question, there has not even been a noteworthy number of mentions of social issues.

The survey asked in the closed question for modification barriers whether participants feared that others might be working on the same issue in parallel. This involves the danger that their solution would be redundant and hence superfluous. Respondents evaluated this as only a minor modification barrier. Steinmacher et al. did not even mention this in their classification of contribution barriers [Ste+14a], which further supports that contributors do not see this as a problem. However, as explained in Section 1.3.5, Gousios et al. proved that the reason for 43 % of all patch rejections on GitHub were redundant work on the same problem [GPD14]. Newcomers seem to underestimate this problem.

2.5 Improvement Suggestions

In a set of three open questions, the questionnaire asked the participants to suggest improvements for the contribution experience of Mozilla or GNOME. The three questions specifically asked how Mozilla or GNOME could better support

1. source code modifications,
2. patch submissions, and
3. issue reports.

The first two questions split the onboarding phase of newcomers into source code modification and submission, as does the rest of the survey. Question 1 asks for methods to lower modification barriers, Question 2 asks for methods to lower submission barriers, and Question 3 asks for methods lowering barriers for non-source-code contributions that are otherwise not in the focus of this survey.

2 Contribution Barriers to FLOSS Projects

The answers may obviously yield ideas to lower contribution barriers, but they are also another method to measure contribution barriers: The questions let the participants think about contribution barriers from another perspective and let them identify new or confirm known contribution barriers.

2.5.1 Results

Analogously to the open questions for contribution barriers and motivations, each statement in the answers received a code in an open coding phase. The code categorizes the type of improvement. It turned out that some, but not all, suggestions impact the contribution process or tool set as a whole. Hence, a separation of the answers into the three categories that the three questions impose was not always possible. For example, some contributors suggested changes to the VCS when asked for source code modifications, others suggested the same change when asked for patch submissions. Nevertheless, having multiple questions seems still useful in retrospective, as this encouraged the respondents to think about improvements from different angles. Additionally, some answers are indeed specific for one of the three types of barriers.

For the reason described above, Figure 2.16 shows the results for all three questions in one figure. Analogously to the open questions in the other sections, the list of codes on the x-axis shows only those codes that at least 7 % of the 65 respondents mentioned, which are at least 5 respondents. A participant counted as respondent if the participant answered at least one of the three questions. The individual codes will be discussed in the following. This discussion includes a remark if a code has been specific to one of the three questions.

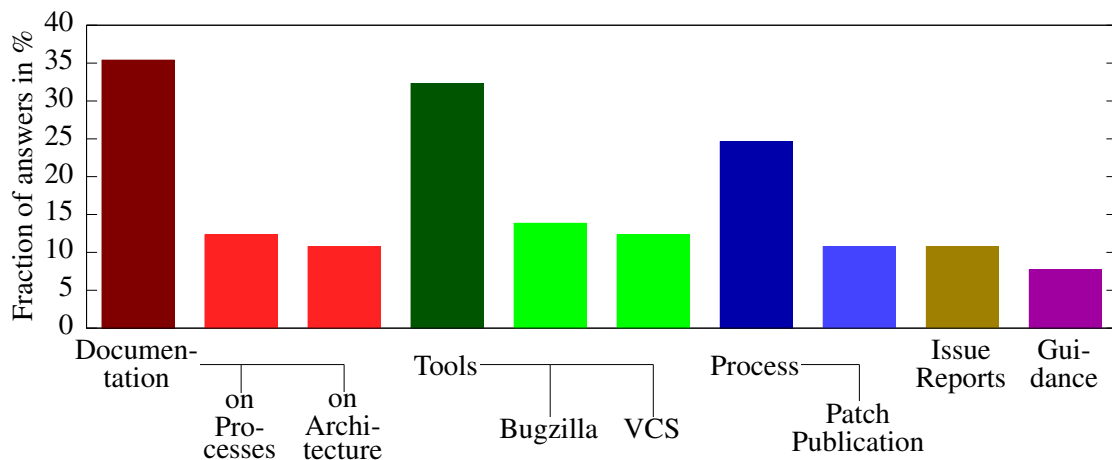


Figure 2.16: Fraction of codes assigned for the open questions for suggestions for improvements

The most common suggestion for improvement was *Documentation* with 35.4 %. This is much more frequently than in the open questions for contribution barriers. This might indicate that the existing documentation is of good quality, for example it is easy to understand, but it may not have covered the specific problem of the contributors. However, this code should be interpreted with caution: The most frequent response to the open question for modification barriers was trouble Understanding some aspect of the contribution, i.e. they were missing some information. Even if

the response was more specific, the problem usually would have been solved easier, and thus with a lower contribution barrier, if they had have some specific information prior to their contribution, like what code style rules are mandatory. A seemingly obvious improvement to solve these issues is therefore to have the specific necessary information documented. However, there may be cases with a better but less obvious solution, like simplifying the process or increasing the intuitiveness of the tools such that additional information and thus documentation becomes unnecessary.

12.3 % of all respondents specified that they missed *Documentation on Processes*. All of these answers referred to the submission processes: How should you create a patch? Where should you submit it? This seemed to be a problem mostly for GNOME contributors, while Mozilla contributors had problems with the flag system described in Section 2.2.1. As Understanding the General Structure of the project was an important modification barrier according to the open question, consequently *Documentation on Architecture* was a common suggestion for improvement, specifically for 10.8 % of all respondents. Interestingly, all these suggestions for Documentation on Architecture were responses to Question 1 and all respondents were Mozilla contributors.

As the next major code, 32.3 % of the respondents suggested improvements in the *Tools* used for the contribution. The suggestions spread on a broad range of tools, such that most individual tools fall below the 7 % threshold selected for this analysis. Two of the most mentioned were improvements to *Bugzilla* with 13.8 %, and improvements to the *VCS* with 12.3 %. The code Bugzilla comprises mostly suggestions to simplify different aspects of the issue and patch submission forms. All responses with the code VCS suggest to use git or GitHub, depending on whether the question was Question 1 or Question 2, respectively. Other suggestions include automatic code style checks, which would help with the frequently mentioned Code Style modification barrier, and easy ways to use unit tests.

Of all respondents, 24.6 % suggested improvements to the *Process* of incorporating new contributors. This process is what von Krogh et al. referred to as Joining Script [KSL03]. As the most frequently mentioned code in this category, with 10.8 % of all respondents mentioning it, is *Patch Publication*. However, all responses that mentioned Patch Publication came from Mozilla contributors, so this may be specific to the Mozilla Project. As another hint that this may be a contribution barrier specific to Mozilla, two of those respondents explicitly mention hg patch queues, an hg feature that even the hg project discourages from using [P*Mer14], but is still in use at Mozilla [P*Moz14a]. Other responses suggest faster reviews or marking issues as easy if they are suitable for newcomers.

10.8 % of all respondents suggested that *Issue Reports* should incorporate additional information that will help developers to work on the issue. Three respondents suggested to have hints at the solution in the issue report, especially which source code files a developer has to change in order to work on the issue. In some cases, a core developer with a good understanding of the code structure might very quickly assemble such a list of probable changes. This would decrease the modification barrier Find the Code, which is the greatest contribution barrier identified in this survey according to the means in the closed questions.

Another 7.7 % of all respondents wanted more *Guidance* in their contribution. [M31] suggested to have a “‘test’ project to let the participants try-run the submit process”, as potential contributors may be afraid to make mistakes publicly in an operational project. The others suggested that mentors should help newcomers onboarding. According to these suggestions, mentors should

initiate conversation with the newcomers even if they themselves do not actively ask for help. This is in line with Cox, who points out that some newcomers need help but then will become very productive members of the community [Cox98]. The suggestions with the code Guidance resemble those contribution barriers that Steinmacher et al. categorize as “Newcomers need orientation” [Ste+14a].

2.5.2 Discussion

Sim and Holt suggest courses to help newcomers onboard software development projects [SH98]. For FLOSS projects however, this will not be possible in most cases. Furthermore, Dagenais et al. found evidence that upfront courses are in fact counterproductive, as they prevent newcomers from gaining first-hand experience with the environment [Dag+10].

The suggestion to check submitted patches for coding styles is not a novel suggestion of this survey. Sethanandha et al. suggested a review system with advanced features including a check for coding styles [SMJ10b].

Some respondents suggested mentors who actively guide newcomers. Mozilla already employs a mentoring program [P*Moz14d], but they suggest that mentors should have a very active role. Sim and Holt also observed that Mentoring is useful to help newcomers onboard a software development project, but they also argue that mentoring costs much time [SH98]. Canfora et al. specifically analyzed how mentoring in FLOSS projects works [Can+12].

2.6 Analysis

This section revisits the hypotheses defined in Section 2.2.2. Using the survey answers described in the previous Sections 2.2.4, 2.3, and 2.4, the hypotheses are tested and discussed.

In this section, a result is considered statistically significant at a significance level of 0.05. Results that are statistically significant at a significance level of only 0.1 are considered statistically slightly significant.

As explained in Section 2.2.4, experience was measured on a scale from one to four, where one represents the least and four represents the most experience. As detailed in Section 2.4, contribution barriers were measured on a scale from one to five. One means that the contribution barrier had no negative effect, while a contribution barrier rated five was almost a show stopper.

2.6.1 Language and Contribution Barriers

As apparent from Section 2.2.4, the most common programming languages that the survey’s participants used for their first patch were JavaScript, C++, and C. Hypothesis 1 states that contribution barriers depend on the programming language. Thus, I first looked which programming languages occurred often and then took two groups out of the participants: The first group consisted of the 48 participants who wrote their modification in JavaScript. The other group consisted of 19 participants who wrote their modification with C++ as well as 16 participants who wrote their modification in C. Treating C and C++ as the same language may conceal insights about their differences, but has two advantages. First, a greater group increases the sample size and therefore the probability to find statistically significant differences if they exist. Second, C

is a subset of C++ and therefore a clear distinction between the two is sometimes difficult or impossible. They also use a mostly identical development tool chain.

Obviously, programming languages may influence only those parts of the contribution barrier that have some relation to the programming language. The items Setup and Build of the closed question on contribution barriers for code modification served as a measure of the complexity of the tool chain. The sum of the answers for these two items measured each respondent's difficulty with the tool chain. A t-test tested whether there was a difference in arithmetic mean between the group of C/C++ programmers and the group of JavaScript programmers. As an interpreted language, JavaScript needs no compilation and therefore has a simpler tool chain and especially no difficulties with a build. Seo et al. found that source code in C++ leads to more compiler errors than Java [Seo+14], which further supports that C++ builds can be difficult. Hence, the t-test was one-sided and a lower contribution barrier for JavaScript was hypothesized. Surprisingly, the p-value of 0.4318 indicated no significant differences.

One factor could be that only 1 JavaScript contributor submitted to GNOME and 46 submitted to Mozilla. Differences between the two projects may have canceled out the differences between JavaScript and C. Respondents' comments suggest that a majority of the JavaScript contributors submitted to Mozilla's new Firefox OS, which has many JavaScript components. As opposed to the common JavaScript web applications, developing for Firefox OS may require building Firefox OS and setting up a development tool chain to work with mobile devices. Therefore, the results are probably not representative for the programming language JavaScript in general.

Another factor that influences results of tests depending on the programming language may be correlations of other factors with the programming language. As C and C++ are older programming languages than JavaScript, maybe contributors of C and C++ modifications are also more experienced with software development and therefore have less problems with technical contribution barriers? This is not the case for the respondents of this survey. The JavaScript developers assessed their expertise with their programming language (Development Environment of PROJ) equally high as C/C++ developers do: The arithmetic mean is 2.57 as opposed to 2.55, respectively. However, a one-sided t-test reveals that C/C++ may have a higher expertise with Software Development in general than JavaScript developers: The means are 3.15 and 2.85 and this difference is slightly significant with $p=0.077$. Maybe, therefore, the C/C++ developers' higher expertise reduced the difficulties that the C/C++ development environment imposed. Nevertheless, the results provide no convincing evidence that the programming language has an impact on the contribution barrier and hence no support for Hypothesis 1.

2.6.2 Experience and Contribution Barriers

Hypothesis 2 predicts that experience lowers contribution barriers. This has multiple facets that need to be analyzed independently: Experience with a specific aspect of the contribution obviously cannot ease contribution barriers for other aspects. Specifically, experience with the technology and the organizational procedures of the FLOSS project are independent and therefore are treated separately.

For technological experience, this analysis tests whether experience with the technologies used in the FLOSS project reduces the contribution barriers of the Setup of the development environment and the Build of the application. For this analysis, the participants were divided into

2 Contribution Barriers to FLOSS Projects

two groups: A group of 45 unexperienced developers who had answered experience levels 1 or 2 with the Development Environment of PROJ, and a group of 51 experienced developers who had answered experience levels 3 or 4 with the Development Environment of PROJ. A one-sided t-test on their contribution barriers for the Setup of the development environment revealed no significant differences ($p=0.4811$), and a test for the differences of their contribution barriers for the Build of the FLOSS application was unnecessary, as the unexperienced developers had a lower arithmetic mean for this contribution barrier of 1.79 than the experienced developers with 1.90.

For organizational experience, this analysis tests whether either experience with other FLOSS projects or experience with the organization of Mozilla or GNOME helps to reduce barriers of submission in Mozilla or GNOME, respectively. Experience with other FLOSS projects may include experience of submitting a modification to these other FLOSS projects, but their processes may be different to the processes of Mozilla or GNOME. Thus, the experience may actually not apply fully. Experience with Mozilla or GNOME is specific to the processes for these projects, but generally does not include the experience of submitting a modification, because the survey asks about the first submission. The onion model assumes that an “active user” phase precedes a code contribution, and in this phase the subsequent contributor files issues or participates with comments [Cro+05]. Hence, the experience about the submission of a modification stems only from observation, but not own effort. Thus, there may be hidden aspects of Mozilla’s or GNOME’s submission procedures that become visible only when attempting to submit a modification oneself.

Both organizational tests used the arithmetic mean of all contribution barriers for submitting a modification rated in the closed question as a measurement for the general difficulty of submitting a modification.

For the first of the two organizational tests, the participants were divided into two groups based on their answer to their experience with FLOSS Processes of other projects: The unexperienced group comprises those 26 participants answering 1 and 2, and the experienced group comprised those 71 participants answering 3 and 4. A formal statistical test was unnecessary, because the unexperienced group had the lower arithmetic mean of 1.488 compared to the experienced group’s 1.544, showing no indication that experience with other FLOSS projects could help submitting a modification to Mozilla or GNOME. This substantiates Sethanandha et al.’s observation that differences in processes and tools between FLOSS projects can be a contribution barrier [SMJ10a].

For the other organizational test, the participants were divided into an unexperienced group of those 60 participants who had answered 1 to Processes of PROJ and those 34 participants who had answered 2 or 3 to Processes of PROJ. This threshold is different to the others, because no participant claimed experience of level 4 with Mozilla or GNOME and only 3 assessed their experience with Mozilla or GNOME on level 3. For such a small group, a t-test on data that slightly violate the condition of a normal distribution would lead to invalid results. With the shifted threshold, the arithmetic means of the two groups’ contribution barriers for submission are 1.504 and 1.507 for the unexperienced and the experienced group, respectively. With so close arithmetic means and even a slightly lower contribution barrier for the unexperienced group, a statistical test is again unnecessary: The data show no evidence that experience as active user makes it easier to contribute code to the FLOSS project.

The analysis found no support for Hypothesis 2, neither in the case of technical experience and

technical contribution barriers nor for organizational experience and organizational contribution barriers – neither showed any association. This indicates that at least Mozilla and GNOME, maybe FLOSS projects in general, employ procedures for submission of modifications and technical environments that differ from other FLOSS project’s procedures and environments. Therefore, experience with other projects do not help when joining Mozilla and GNOME. Furthermore, an “active user” cannot see the intricacies of the submission of modifications and therefore has no advantage in doing this over others whose first involvement with the FLOSS project is the submission of a modification.

2.6.3 Projects and Contribution Barriers

According to Hypothesis 3, the contribution barriers of one FLOSS project are different to those of another FLOSS project. To test this hypothesis, a metric for the overall contribution barrier of a respondent was calculated as the sum of all answers to the closed question on contribution barriers of this respondent. A t-test tested whether this overall contribution barrier had a different arithmetic mean for Mozilla contributors than for GNOME contributors. With $p=0.5441$, there were no statistically significant differences between Mozilla and GNOME. However, the addition might have cancelled out differences in the individual contribution barriers.

Therefore, further t-tests compared all contribution barriers individually. Table 2.4 displays the results: The columns Mozilla and GNOME shows the arithmetic means of all answers that only contributors to Mozilla and GNOME, respectively, gave. The column p-value contains the p-values for each t-test to test whether there are differences in the arithmetic means of the Mozilla contributors compared to the GNOME contributors. Only one difference is statistically significant at the $\alpha = 0.05$ level, but this may easily be a coincidence given that there are 13 t-tests. Another difference was slightly statistically significant ($p < 0.1$). These results will not be analyzed further due to the high chance of statistical error.

However, Wallis’s method to combine multiple experiments [Wal42] results in the more meaningful compound p-values of 0.333, 0.040, and 0.891 for all contribution barriers, the submission barriers, and the modification barriers, respectively. This suggests that Mozilla and GNOME may indeed create different sets of submission barriers with their respective submission procedures. The modification barriers, on the other hand, are surprisingly similar for both projects.

While this difference between submission barriers suggests a partial confirmation of Hypothesis 3, the modification barriers are very similar for Mozilla and GNOME with no statistically significant differences. This suggests that the values found for the modification barriers may be constant for a larger set of FLOSS projects, of which Mozilla and GNOME are just two examples. Without further studies of other projects, it is unclear to how many projects this set of FLOSS projects with similar modification barriers extends.

2.6.4 Own Need, Motivation, and Occupation⁴

Hypothesis 4 says that most newcomers modify the FLOSS project’s source code primarily because they need the modification for themselves. This means that more than half of the respondents should have ranked Own Need as their most important modification motivation. As

⁴A preliminary version of this section was published previously [HG16b].

2 Contribution Barriers to FLOSS Projects

Table 2.4: Arithmetic means of contribution barriers by project and p-values of t-tests for inter-project differences and compound p-values

Contribution Barrier	Mozilla	GNOME	p-value
Submission Procedure	2.038	1.762	0.267
Documentation	1.720	1.667	0.522
Issue Tracker	1.551	1.300	0.201
Bureaucracy	1.485	1.143	0.066 [†]
Member Attitude	1.190	1.350	0.040 [*]
Combined Submission Barriers			0.040[*]
Download	1.390	1.524	0.453
Setup	1.701	1.952	0.471
Build	1.813	1.850	0.918
Bug Reproduction	1.639	1.367	0.147
Redundant Work	1.458	1.600	0.810
Find the Code	2.114	2.050	0.950
Solution	1.731	1.650	0.958
Community Support	1.387	1.500	0.406
Combined Modification Barriers			0.891
Combined Contribution Barriers			0.333

[†] significant at $\alpha = 0.1$

^{*} significant at $\alpha = 0.05$

only 25.8 % of the respondents ranked Own Need first as a modification motivation, Hypothesis 4 must be rejected – a binomial test as post hoc analysis using Hypothesis 4 as null hypothesis gives a p-value < 0.0001.

Without Hypothesis 4, there must be another explanation for the high number of respondents ranking Own Need as their primary modification motivation in the exploratory survey and in Shah's survey [Sha06]. As the phrasing and user interface in the main and exploratory surveys were identical, the questionnaire implementation cannot cause bias. The differences between the participants of the exploratory survey and those of the main survey must be this explanation. All participants in the exploratory survey are professional software developers, as explained in Section 2.1. Thus, the high importance of the modification motivation Own Need may be due to the participants' occupation. This first possibility will be discussed next.

Modification Motivation and Occupation

Previous research did indeed identify associations between occupation and contributor motivation for FLOSS developers. The results are ambiguous whether Own Need depends on occupation. The next two paragraphs discuss the state of research and the paragraphs afterwards presents statistical clues from this survey.

In their survey, Hars and Ou [HO01] differentiate between contributors paid for their contribution, employed developers whose primary assignment is not their contribution, and contributors who are non-professional programmers. Although the three groups vary in some contributor motivations, they do not vary measurably in others. Interestingly, Own Need has of all included contributor motivations the least variance between the three groups: 38.5 % of the contributors who were paid for their contribution, again 38.5 % of the employed developers, and 36.4 % of the non-professional programmers rate Own Need “high”. This indicates that while occupation does influence the contributor motivation, it does not influence Own Need in particular. This seemingly indicates that the difference between the exploratory survey and the main survey supposedly has another reason.

Lakhani and Wolf [LW03] also collected contributor motivation and occupation data. They found statistically significant differences in contributor motivations between contributors who were paid for their contribution and those who were not paid. These results are difficult to compare to the results of the main survey in this chapter: First, the main survey explicitly excludes employees of the Mozilla Foundation, the main employer of developers working on Mozilla, but does not distinguish explicitly between paid and unpaid contributions for the remaining participants. Second, Lakhani and Wolf partition Own Need in “Work need only” and “Non-work need” and so the overall number for Own Need cannot be derived from the published data for paid and unpaid contributors, as the data do not contain the overlap between those two types of needs. They tested both subtypes “Work need only” and “Non-work need” of Own Need successfully for statistically significant differences between paid and non-paid contributors, but they did not test Own Need as an aggregated motivation. The individual differences might cancel each other out and therefore still be in concordance with Hars and Ou’s results. Therefore, a direct comparison with Lakhani and Wolf’s results is not possible, but casts doubt on the independence between contributor motivation and Own Need.

Next, Hypothesis 4 is restricted to employees and tested again. With this restriction the respondent group better resembles the respondents of the exploratory survey. Of the 29 employed respondents who answered to the closed question of modification motivation, 14 ranked Own Need first, so this is still slightly less than 50 %. Contrary to the original unrestricted form of Hypothesis 4, the restricted hypothesis may still be true and the too low number of employed respondents in the main survey may have introduced a random error: Neither the restricted hypothesis nor its opposite can be rejected with statistical significance.

In continuation of the literature discussion above, the next research question is: Do employees rank Own Need as a more important modification motivation than non-employed contributors? As the motivations scales are rankings and not continuous numerical data, a t-test is not possible. Instead, a Mann-Whitney U test fits this research question, especially as it allows to consider those respondents who did not consider Own Need as a modification motivation at all – these respondents implicitly ranked Own Need less important than any explicit rank. In contrast to a t-test, the Mann-Whitney U test does not need a concrete numeric value for these implicit rankings, as it suffices to know that they are ranked with least importance. Low ranks designate a high importance, so the null hypothesis is: The rank of Own Need for employed respondents is at least as high as the rank for non-employed respondents. The Mann-Whitney U test yields a p-value of 0.0781, so the difference is not significant at a significance level of $\alpha = 0.05$, but only at $\alpha = 0.1$. Occupation may have an influence on Own Need, but the results are not conclusive.

Groups of Contributors Distinguished Through Own Need

As noted in Section 2.3.3 and visible in Figure 2.11, Own Need is more polarizing than the other modification motivations: A comparatively low number of respondents ranked Own Need second or third, although Own Need is the second-most popular option for the first rank. Specifically, 25.8 % of the respondents ranked Own Need as their primary modification motivation. 41.9 % of the respondents saw Own Need as a modification motivation but not as their primary motivation: most of them, 25.8 % of all respondents, ranked Own Need fourth. 32.3 % did not see Own Need as modification motivation at all.

Thus, using this motivation, three main groups of contributors can be distinguished: First, those who missed a feature or suffered from a defect in the FLOSS application, and modified the FLOSS application to solve their problem. They ranked Own Need first. Second, those who also missed a feature or suffered from a defect, but there was a workaround or the problem did not impact them very much – fun of programming, the project community, and learning about technology were more important motivations for them to modify the FLOSS application, but at least Own Need was a modification motivation at all for them. Third, those who actively searched for a task in the FLOSS project and who did not find a problem through their own usage of the FLOSS application. For them, Own Need was no modification motivation at all.

2.6.5 Motivation and Contribution Barriers

Hypothesis 5 predicts a specific relation between motivations and contribution barriers: Motivations are indicators for the contributors' mental model and those contribution barriers expected in the mental model have less impact than those contribution barriers which are not expected. This is a consequence of the Theory of Cognitive Dissonance [Fes57].

For tests of Hypothesis 5, participants are assigned roles derived from their motivation. *Technology Learners* is an example of such a motivation role, which comprises those participants for whom learning about the technologies used in the FLOSS project was an important motivation to contribute. Motivation roles are independent, so a participant either is or is not in a specific motivation role, independently of the other motivation roles the participant is assigned to. As discussed in the presentation of Hypothesis 5 in Section 2.2.2, motivation roles presuppose that the participants had a specific model about the contribution and that they should have expected or not expected a particular hurdle in the contribution procedure.

The Technology Learners, for example, should have expected that the FLOSS project's technical environment has some intricacies that a newcomer has to overcome – by learning about the FLOSS project's technology. These hurdles manifest in a contribution barrier that the survey measured in its closed questions for contribution barriers. Statistical tests therefore reveal whether or not there is a difference in perception of these contribution barriers between members and non-members of a motivation role. Differences support Hypothesis 5, while a lack of differences opposes Hypothesis 5.

All motivation roles are listed in Table 2.5. For each motivation role, the table also lists the number of members and non-members of the role and which contribution barriers should be lower or greater for members of the motivation role according to Hypothesis 5. The last column p-value lists p-values that result from one-sided t-tests that test whether the expected difference between

members and non-members of the motivation role exist. For most motivation roles, multiple different contribution barriers are tested for differences resulting in multiple p-values. Using Wallis' method of combining p-values from multiple experiments [Wal42], these p-values are combined into a single p-value for each motivation role and eventually for the whole comparison. The following paragraphs discuss the construction and rationale for the individual motivation roles in detail.

The analysis will first take up the introductory example of *Technology Learners*. Defined precisely, Technology Learners are those participants who ranked Learning as the most or second-most important motivation to modify the FLOSS application. Their complementary role comprises the respondents who rejected Learning as a motivation to modify the application or ranked it only third-most important or less. There are 44 Technology Learners and 50 respondents in the complementary role. The null hypotheses are: Technology Learners rank the contribution barriers Setup and Build at least as high as their complementary role. As the result of t-tests, there are statistically significant differences between Technology Learners and their complementary role for Setup ($p=0.04907$) and no statistical significance for Build ($p=0.2319$).

The next motivation role is *Joy Programmers*, which includes those respondents who ranked Joy as their primary reason to modify the application. Those who ranked Joy as the second-most reason were assigned to the complementary role, as otherwise the Joy Programmers would be more than 70 % of all respondents and therefore the complementary role would be too small for a meaningful statistical comparison. There are 35 Joy Programmers, their complementary role counts 59 respondents. Those who want to experience the joy of programming know that programming is a challenge and specifically want to face it. Hence, the null hypotheses are: Joy Programmers rank the contribution barriers Find the Code and Solution at least as high as their complementary role. The corresponding t-tests show no statistically significant differences for Find the Code ($p=0.1728$) and slightly statistically significant differences for Solution ($p=0.05176$).

Pragmatic Patchers are those respondents who ranked Own Need as their primary or secondary motivation to modify the application. There are 29 Pragmatic Patchers and 65 respondents in their complementary role. Pragmatic Patchers supposedly consider patching the application as less effort than a workaround like using another application, as that would also satisfy their motivation. Thus, their effort estimate of the modification has an upper limit that those respondents in the complementary role do not have. If the modification effort is nevertheless high, Pragmatic Patchers with their lower effort estimation will therefore less likely expect this effort than their complementary role. Hypothesis 5 suggests that this unexpected effort can be measured as higher contribution barriers. Core contribution barriers like Solution are obviously necessary for the modification and therefore less likely to be underestimated; thus, the unexpected effort is operationalized via the secondary contribution barriers Download, Setup, and Build. The null hypotheses say that Pragmatic Patchers experience the contribution barriers Download, Setup, and Build at most as high as their complementary role. However, none of the three null hypotheses can be rejected with t-tests: There are no statistically significant differences for these contribution barriers. The results are $p=0.2478$, $p=0.7426$, and $p=0.4598$ for Download, Setup, and Build, respectively.

I tried to shed light on the lack of differences between Pragmatic Patchers and their complementary role with a post hoc analysis. Instead of the original criterion for Pragmatic Patchers,

2 Contribution Barriers to FLOSS Projects

Table 2.5: Respondent motivation roles and whether they perceive contribution barriers differently than their complementary roles, as p-value of a *one-sided* t-test

Motivation role	Role sizes (respondent/ complementary)	Contribution barrier	Expected Difference	p-value
Technology Learners	44/50	Setup	<	0.0490*
		Build	<	0.2319
		Combined		0.0623 [†]
Joy Programmers	35/59	Find the Code	<	0.1728
		Solution	<	0.0518 [†]
		Combined		0.0511 [†]
Pragmatic Patchers	29/65	Download	>	0.2478
		Setup	>	0.7426
		Build	>	0.4598
		Combined		0.5517
Strictly Pragmatic Patchers (post hoc)	23/30	Download	>	0.0888 [†]
		Setup	>	0.2412
		Build	>	0.2215
		Find the Code	>	0.1586
		Solution	>	0.1035
		Combined		0.0412*
Community Joiners	36/58	Community Support	<	0.0526 [†]
		Combined		0.0526 [†]
		Submission Pro- cedure	<	0.2052
FLOSS Learners	42/49	Issue Tracker	<	0.9671
		Bureaucracy	<	0.3687
		Combined		0.5146
		Submission Pro- cedure	>	N/A
Economic Submitters	14/77	Issue Tracker	>	N/A
		Bureaucracy	>	N/A
		Combined		N/A
		Groups combined		0.0301*

[†] significant at $\alpha = 0.1$

* significant at $\alpha = 0.05$

the *Strictly Pragmatic Patchers* were limited to only those respondents who ranked Own Need as a primary motivation. The contribution barriers for these Strictly Pragmatic Patchers were compared with those of the respondents who saw Own Need as no motivation at all. This corresponds to two of the three groups identified in the discussion in Section 2.6.4. Additionally, the post hoc analysis includes the contribution barriers Find The Code and Solution, as these are among the highest modification barriers according to the closed question for modification barriers, and thus are more likely to be the source of an unexpectedly high modification effort. Just like the three contribution barriers already tested, Find the Code and Solution were not significantly different between Pragmatic Patchers and their complementary role ($p=0.7197$ and $p=0.2015$). The differences between the Strictly Pragmatic Patchers and those who had no Own Need at all were only slightly significant for Download ($p=0.08887$) and not significant for Setup ($p=0.2412$), Build ($p=0.2215$), Find the Code ($p=0.1586$), and Solution ($p=0.1035$). However, combining these p-values using Wallis' method [Wal42] yields a significant difference ($p=0.04129$) between Strictly Pragmatic Patchers and the respondents without Own Need. This result is even more outstanding when considering that the roles of Strictly Pragmatic Patchers and respondents without Own Need were smaller than Pragmatic Patchers and their complementary role: Although statistical tests generally yield lower p-values for larger groups if an effect exists, Strictly Pragmatic Patchers still show statistically detectable differences to respondents without Own Need. This indicates that the a priori differentiation between Pragmatic Patchers and their complementary role did not very well match real differentiations between motivation roles of contributors. Strictly Pragmatic Patchers and respondents without Own Need better characterize motivation roles than Pragmatic Patchers and their complementary role.

Those respondents who ranked Community as a modification motivation on first or second place are called *Community Joiners*. There are 36 Community Joiners and 58 respondents in the complementary role. As a consequence of their motivation, Community Joiners expect social interaction with the community of the FLOSS project. They want to get in contact not purely because of an immediate need, instead social contacts have value for them in their own right. Their complementary role on the other hand may find it unexpected if it is necessary to get in touch with multiple members of the community in order to solve the issue. Thus, the null hypothesis is that Community Joiners rank the modification barrier Community numerically at least as high as their complementary role. The resulting t-test shows that the null hypothesis can almost be rejected with statistical significance ($p=0.05256$).

The motivation for modification determined who belonged to the previously defined motivation roles and who does not. In contrast, *FLOSS Learners* are the respondents whose first or second motivation for their submission to the FLOSS project was Learning. Analogously to Technology Learners, FLOSS Learners know that each FLOSS project has its own processes and its own culture, because their motivation is to learn about these specifics. Their complementary role may focus on technical aspects of the contribution and forget about the submission procedure. If the submission requires effort, the complementary role may not expect it and experience a high submission barrier. Hence the null hypotheses: FLOSS Learners rank the submission barriers Submission Procedure, Issue Tracker, and Bureaucracy at most as high as their complementary role. Contrary to Technology Learners, t-tests could not statistically significantly reject the three null hypotheses for FLOSS Learners, neither for Submission Procedure ($p=0.2052$), Issue Tracker ($p=0.9671$), nor Bureaucracy ($p=0.3687$). Possibly, both FLOSS Learners and their

complementary role knew about and expect the efforts of the submission, but only the former saw it as a motivation for submission, while the latter just found it to be a necessary task.

The motivation role *Economic Submitters* consists of those respondents whose first-ranked or second-ranked motivation for submission is avoiding the Stupid Tax. Analogously to Pragmatic Patchers, Economic Submitters assume that less effort is necessary for the submission of their modification than for the possibly repeated reintegration of their modification into newer versions of the FLOSS application. Thus, they should rank submission barriers higher than their complementary role. However, only 14 respondents are Economic Submitters. This is a small size for a statistical test, and therefore statistical significance is unlikely even if an effect exists. Consequently, no differences were tested and the analysis does not take Economic Submitters into account.

In summary, this subsection analyzed five motivation roles. Statistical tests revealed slightly significant differences for three motivation roles. The remaining two motivation roles Pragmatic Patchers and FLOSS Learners showed no statistical significant differences. However, a post hoc analysis revealed that adapting the motivation role of Pragmatic Patchers to Strictly Pragmatic Patchers yields a significant difference. One motivation role, Economic Submitters, has not been analysed as the sample was too small. The results for the five valid motivation roles, excluding the post hoc analysis of Strictly Pragmatic Patchers and omitting the missing test for Economic Submitters, combine into a p-value of 0.0301 using Wallis' method [Wal42]. Thus, the result supports Hypothesis 5: Contribution barriers depend on the contributors' motivation.

2.7 Threats to Validity

This section discusses different types of threats to the survey and how the survey set-up ensured validity in spite of these threats. First, construct validity describes how the survey measured the data that it was supposed to measure. Next, the survey questions shall adequately cover contribution barriers to ensure content validity. Third, the conclusions drawn in the discussions and Section 2.6 must have internal validity. Lastly, threats to external validity endanger the application of this survey's conclusions to FLOSS projects other than Mozilla and GNOME.

2.7.1 Construct Validity

Incomprehensible or ambiguous questions endanger construct validity, as the participants' answers would not fit to the questions. However, both the exploratory and the main survey received thorough pretests and the first participants were invited only after the pretesters had not misunderstood any question. The participants' answer also showed few signs of misunderstanding and those were discussed in the description of results.

One metric for the importance of contribution barriers in the survey were arithmetic means of scale scores, which are similar to Likert scores. Likert scores are ordinal and there is criticism that arithmetic means and parametric tests like the t-test may not be used on ordinal scales [Gai80]. On the other hand, other researchers point out that this criticism founds on a confusion between measurement theory and statistical theory and that the type of scale does not influence the statistical method used [Lor53; Gai80]. Therefore the arithmetic means of ordinal scaled contribution

barriers must not be interpreted as means of actual importance of contribution barriers, but as means of measured and ordinal scaled importance.

As described in Section 2.2.4, the survey measured experience through self-assessment. Participants might over- or underestimate their abilities, or simply lie about their own ability. This would induce high measurement errors for this question. However, empirical evaluation showed that self-assessment is superior to other types of measurements of programming skills like university marks [KH11].

2.7.2 Content Validity

Answers to closed questions in surveys strongly depend on which answers the participants can choose from. For example, survey participants tend to choose an average option, even if the average is very high or very low in absolute numbers [Dil99, Chapter 2]. The survey's central questions, those about contribution barriers and motivation, therefore had an open and a closed form. For each question, the survey presented the open question first and afterwards the closed question. This ensured that the participants had no preconceptions when answering to the open question. The closed question let participants think about answers that they did not consider in the open question. For example, an item might have influenced their decisions only subconsciously, so they did not answer it in the open question but agreed with it in the closed question as they had to think about it. This was supposedly the case for the Joy of programming in the question for modification motivation. Furthermore, answers to the closed question were less ambiguous and are better suited for statistical evaluation.

The closed questions for motivation explicitly asked for the motivation at the time of the participants' first modification and submission. This is different to the participants' motivation at the time of the survey, as they had learned more about the FLOSS project in between and found out which motivations were reasonable and which were not. Nevertheless, participants might have chosen motivations that sounded plausible for them, although they may not have initially thought of them and therefore they did not influence their decision to contribute. This is an alternative explanation for answers occurring only in the closed but not in the open questions. This threat cannot be ruled out completely and its possibility has to be evaluated for every case where the number of closed answers exceeds the number of equivalent open answers. As described in Section 2.2.3, the participants were invited shortly after their first contribution, so their memories about the contribution were still fresh, which further limits this threat.

2.7.3 Internal Validity

The analysis in Section 2.6 frequently used t-tests on scores from ordinal scales similar to Likert scales to prove or disprove statistical hypotheses. t-tests require normally distributed variables, which is never the case for Likert scores. Nevertheless, the t-test is robust to minor violations to its requirements [Nor10] and works well especially with Likert scores [WD10]. Nevertheless, the data analyzed in Section 2.6.4 deviated too much from the requirements of a t-test and a Whitney-Mann U test was used as alternative. Consequently, the statistical methods were in all cases appropriate for the data.

2 Contribution Barriers to FLOSS Projects

There may be a self-selection bias, as the set of survey invitees and survey participants are not identical. However, as explained in Section 2.2.3, the response rate of 65 % is high for a survey [Dil99, p. 3f]. Therefore the identified contribution barriers and motivations really exist for a major number of FLOSS contributors.

During the Second World War, the Allies collected which zones of their returning bombers were hit by the enemy. They wanted to know where they should reinforce their bombers' armor. Because planes must be lightweight, there is only limited capacity for armor. Wald [Wal80, Part V] showed with statistical arguments the surprising result that the armor should be reinforced in those zones that received low number of hits – because bombers hit in these zones had not returned from combat and were therefore vulnerable in these zones. This type of selection bias is called survivorship bias. The bomber situation can be seen as an analogy to contributors in FLOSS projects who try to drop their contribution on the FLOSS project, but contribution barriers of different kinds may hit them and therefore prevent the contribution. The survey only invited contributors with successful patches, those who had overcome the contribution barriers. Continuing the analogy would lead to the conclusion that the contribution barriers identified in the survey are not important, because obviously the contributors were able to overcome them, and that the really important contribution barriers are those that none of the participants experienced. However, Wald formulated two assumptions which do not hold for contribution barriers.

First, Wald assumes that the probability that a hit downs a bomber is independent from previous hits on the same bomber. For FLOSS projects, it is more plausible that contribution barriers accumulate in some way and that potential contributors cancel their contribution if all experienced contribution barriers together exceed their level of tolerance induced by their motivation to contribute. In this case, the distribution of contribution barriers among successful contributors approximates the distribution of contribution barriers among all potential contributors. Nevertheless, to a lesser degree Wald's survivorship bias still applies to contribution barriers, and it is stronger for contribution barriers that have a greater individual impact, as contributors experiencing these contribution barriers are underrepresented in the set of successful contributors.

Second, Wald assumes that the probability for a bullet to hit a specific zone of the bomber is known. He approximates this in an example with the size of each zone. In the case of FLOSS projects, it is not only relevant whether a specific contribution barrier has a high or low probability to cancel a contribution, but also for how many contributions the contribution barrier arises. A frequent contribution barrier with low individual impact on the contribution may have the same overall effect as an infrequent contribution barrier with high individual impact on the contribution. The distribution of contribution barriers among successful contributors not only shows each contribution barrier's individual impact, but also its occurrence frequency in the base set, which is interesting in the case of contribution barriers. For bombers, armor in small zones is less costly than armor in large zones, because more armor is required and it will be heavier for large zones. Therefore, the vulnerability of a zone is more interesting than its probability to get hit. In contrast, lowering a contribution barrier does not necessarily depend on its occurrence frequency and therefore a mixed measure of impact and occurrence frequency is more useful than one of them alone. This further lessens the survivorship bias.

2.7.4 External Validity

As shown in Section 2.6.3, Mozilla's and GNOME's modification barriers are surprisingly similar. This indicates that the modification barriers identified in this survey apply to a larger class of FLOSS projects, of which Mozilla and GNOME are two examples. Nevertheless, there may be FLOSS projects of different classes that have a different composition of modification barriers. For example, small FLOSS projects are potentially different to Mozilla and GNOME. Mozilla's and GNOME's submission barriers were different and therefore it is unclear to which degree they apply to other FLOSS projects.

Furthermore, both Mozilla and GNOME are very successful FLOSS projects. As the success of a FLOSS project depends on the project's ability to gain new developers [Xu+05; CM07], both GNOME and Mozilla supposedly have a relatively low level of contribution barriers. The results in Section 2.4 support this assumption. Thus, other FLOSS projects may have a generally higher level of contribution barriers.

Some further results, like the lack of relationship between programming language and contribution barriers as predicted by Hypothesis 1, may be specific to Mozilla and GNOME. Possible reasons were discussed in the appropriate sections. Other further results like the confirmation of Hypothesis 5 found on general theories which do not depend on the choice of the FLOSS project. These results can be assumed to hold also for other FLOSS projects.

2.8 A New Model for Joining FLOSS Projects

This section describes a model for new contributors joining a FLOSS project. It is based on the previous findings, especially the relationship between contribution barriers and contributor motivations, and on previous research. The overall impact of all contribution barriers on a newcomer is determined by which contribution barriers occur and the strength of their individual impact. Both factors can be assessed individually, as will be shown in the following.

Von Krogh et al. argued that new contributors have to adhere a specific process called the joining script to become co-developers of a FLOSS project [KSL03]. Jensen and Scacchi noted that there are different paths towards the core roles of FLOSS project [JS07]. Herraiz et al. [Her+06] specifically showed that GNOME employs at least two joining scripts. Moreover, Herraiz et al. showed that quite discriminably volunteer developers use one joining script while employed developers use another. Since every joining script comes with its own steps and tasks, the joining script defines which contribution barriers are in effect.

All newcomers have their own mental model of the FLOSS project. The mental model determines the impact of each individual contribution barrier. If a task needs more time to be carried out, this increases related contribution barriers; however, it is more important how "frustrating" the contribution barriers are. More specifically, Hypothesis 5 implies that unexpected tasks have higher contribution barriers than expected tasks. Additionally, tasks that are perceived to create value induce lower contribution barriers than tasks that are considered unnecessary.

The mental model of the FLOSS project also determines the newcomers' motivations. The proof of Hypothesis 5 shows that this relationship can be reversed: Specific motivations imply specific ideas of the FLOSS project and therefore whether each specific task is expected or not. In particular, the survey identified in Section 2.6.5 four roles of newcomers that experience

2 Contribution Barriers to FLOSS Projects

especially low or especially high contribution barriers of some kinds. Future research should try to identify more of these particular groups.

Demographic properties influence which joining script newcomers use and they also determine their motivations, their mental model of the FLOSS project, and consequently which contribution barriers have high impact and which have only low impact. For example, learning is an important motivation for students. For members of the role Technology Learners, the modification barriers for setting up a development environment have less impact. Students are volunteer developers and therefore advance slowly through supporting roles in the FLOSS project before they contribute code [Her+06]. Barriers that hinder non-code contributions would therefore indirectly hinder code contributions from students. The consequences of this insight on the management of each FLOSS project depends on the project's goals.

Each FLOSS project attracts contributors with different demographics. Thus, it could be reasonable to lower particularly those contribution barriers that strongly affect the contributors who are the most attracted by the FLOSS project. This would maximize developer influx. Inversely, a FLOSS project might increase attractiveness for contributors of demographics that the FLOSS project has low contribution barriers already. However, the latter option is not in the scope of this thesis, as the mechanisms of attractiveness are a field of research on their own.

Section 2.2.4 discusses evidence that contributors of some demographics stay longer with a FLOSS project than others. Additionally, the first activities of a person in a FLOSS project can be used as an indicator whether the person will contribute over a long term [ZM12]. Hence, lowering contribution barriers for contributors of these demographics may be desirable.

Another approach would be lowering contribution barriers for contributors of underrepresented demographics. This would diversify the FLOSS project's group of developers and thereby help to implement requirements of users belonging to the previously underrepresented demographics.

2.9 Conclusion

This chapter addressed the first of three core research question of this thesis: Which contribution barriers exist and what is their individual effect? An exploratory survey gave qualitative insights about contribution barriers and helped to create a succeeding, more comprehensive survey. 117 useful responses to this main survey delivered enough data for a statistical analysis of five hypotheses that were formulated as a result of the exploratory survey. The following summarizes the insights gained from the survey and its analysis.

Existing research does not differentiate between modification and submission in regard to contributor motivations and contribution barriers. Existing research focuses more on submission motivations than modification motivations, although this survey's results in Section 2.4 show that modification barriers are more important than submission barriers. Future research on motivation should pay additional attention to modification motivations.

Section 2.4 identified that five contribution barriers are of high importance. These contribution barriers, roughly in order of importance, are

- difficulties to understand the structure of the FLOSS project and its architecture, and consequently difficulties to find the part of the source code that needs to be changed. This can be a consequence of a large project size.

- Setting up and configuring the development environment can be frustrating for newcomers.
- Newcomers have trouble to grasp the submission process and consequently sometimes get lost in the submission of their patch.
- Programming the actual solution can be challenging, but it is usually not frustrating.
- Getting the reviews required for acceptance of the patch can be troublesome.

Generally, accidental tasks, which newcomers feel unnecessary, are more frustrating than essential tasks, which are considered at the core of the contribution. Accordingly, issues with the code style were an important contribution barrier for 17.6 % of the respondents. FLOSS projects should take care not to introduce these types of contribution barriers.

The five hypotheses formulated in Section 2.2.2 were tested in Section 2.6. The results delivered no statistical evidence for a dependency between contribution barriers and programming languages, neither was there evidence that contribution barriers could be lowered by experience of different kinds.

FLOSS projects can be classified such that all FLOSS projects in one class have similar contribution barriers. Mozilla and GNOME belong to one class regarding their modification barriers, but they differ in terms of submission barriers.

Contributors can be split into three groups of very roughly equal size: The first modifies the application because they need the modification for themselves. The second also uses the modification, but this is only a subordinate modification motivation for them. The third group does not even use the modification, but have completely different reasons for their modification. Contributor occupation may be a factor determining to which group a contributor belongs. These groups differ in what they perceive as contribution barriers.

The contributors' motivations are indicators for their mental model about the FLOSS project. As a consequence, more groups than those discussed above can be identified using their motivation. These groups are different in their perception of contribution barriers. Expected contribution barrier have less impact than unexpected contribution barriers.

These findings lead to a new model for joining FLOSS projects. The demography of newcomers determine on the one hand which joining path these newcomers take and thereby which contribution barriers they encounter. The demography also determines which contribution barriers the newcomers expect and thereby the impact of each individual contribution barrier. FLOSS projects may adapt their joining scripts to the demographies they want as contributors.

3 FLOSS Pattern Languages

As pointed out in Section 1.1, the definitions of Open Source Software [P*Ope04] and Free Software [P*Fre15] depend only on the license used for the FLOSS project and not on the development style. Researchers have observed and described different characteristics of software development in FLOSS projects [Ray00; GA04] and their communities [Kri02; YK03]. Development methods differ strongly between FLOSS projects. However, there are not only two or three types of FLOSS development methods, instead there are several interrelated aspects of the development method that occur in different variants.

The pattern approach introduced by Alexander allows the description of individual parts of a method, independently from other parts and still showing their interrelations [Ale79]. Kelly suggested that this pattern approach may be applicable to FLOSS development, and that the best practices of running FLOSS projects may constitute its own FLOSS Pattern language [Kel06]. In fact, several authors picked up the idea and authored FLOSS patterns that they had observed in practice and drawn from literature, e.g. [Wei09; Lin10; HWG10].

Each pattern describes at its core a solution to a problem occurring in a context [Ale+77, p. x]. Accordingly, each FLOSS pattern describes the context of a FLOSS project, a problem that this FLOSS project faces, and a solution to the problem based on experience. Patterns describe existing and not new solutions. Thus, pattern authors usually are not the original creators of solutions but only their observers. This is similar to natural sciences, which do not create new laws of nature, but observe *patterns* in natural behavior and describe the invariants of the observed behavior. This is the same for patterns, which describe invariants of problems and solutions in other disciplines [Ale79].

Contribution barriers are one type of problem for FLOSS projects, and consequently, it is possible to write FLOSS patterns describing how to lower these contribution barriers. The contribution barriers itself were identified and described in Chapter 2. Section 3.2 categorizes all FLOSS patterns published until 2015-12 and describe their relationships as a pattern language. Sections 3.1 reviews the pattern literature and introduces some conventions for this categorization. Section 3.3 contains seven complete FLOSS patterns that lower contribution barriers identified in Chapter 2.

Patterns include examples of their application. Therefore, FLOSS patterns are empirically justified and they are known to have been applied in practice. As such, they are reliable observations of FLOSS development methods. However, Open HUB listed 670 563 FLOSS projects already as of 2015-10-27 [P*Ope15b], and so a few examples in each pattern do not prove that the FLOSS pattern is common. In fact, it could still be the very rare exception and in the theoretical extreme, the given examples could be the only applications of the FLOSS pattern at all. Therefore, the chapter contains an evaluation of application frequency of the FLOSS patterns.

Section 3.4 presents the execution of this evaluation and Section 3.5 discusses its findings. The evaluation takes the reversed approach compared to examples given in patterns: first, five

FLOSS projects were selected and then it was evaluated which FLOSS patterns they use. Very common FLOSS patterns will occur in this selection with high probability, while uncommon patterns do not occur or do not occur often in the selection. This evaluation also puts the proposed relationships to the test.

3.1 Introduction

This section presents a review of literature on patterns and some definitions required for the survey of all FLOSS patterns in Section 3.2.

Currently, there are 40 FLOSS patterns described in 13 monographs [HWG10; HWG11; HLG14; Lin10; Lin11; Lin12; Wei09; Wei10; Wei11; Wei15; WN13; WHG13; ZHG16]. Although patterns are supposed to facilitate the reader's understanding of a domain, it is difficult to gain an overview of this large number of patterns distributed in several publications, with no apparent order in these FLOSS patterns. The first goal of the survey is to give a concise overview of all FLOSS patterns and thereby serve as a guideline that helps readers to find the FLOSS patterns relevant for their situation.

As part of this endeavor, the FLOSS patterns have been categorized into eight categories. Additionally, some FLOSS patterns are similar and it turned out that they describe different aspects of the same problem and solution. These FLOSS patterns have been combined. Resulting FLOSS patterns, whether unchanged or combined, are described as thumbnails according to the PROBLEM/SOLUTION SUMMARY PATTERN of the pattern language for pattern writing [MD98] in Section 3.2.

FLOSS patterns currently stand mostly on their own, as current research did not put much emphasis on the relationship between patterns, especially between patterns of different authors. Partly, this is due to the publication order of patterns, as obviously a work can only reference previously published patterns. Sometimes earlier FLOSS patterns rest upon later FLOSS patterns, possibly without the authors' knowledge, and these relations have yet to be revealed. Thus, the second goal of the survey is the identification of relationships between FLOSS patterns.

3.1.1 Classification of FLOSS Patterns

Each of the eight FLOSS pattern categories has its own subsection in Section 3.2. A category subsection starts with a summary of the patterns in the category as described in the PATTERN LANGUAGE SUMMARY PATTERN [MD98]. This summary introduces the patterns as a whole and describes their relations. A diagram known as pattern map visualizes the patterns in the category. Section 3.1.3 describes the pattern maps used in this thesis in more detail. The remainder of the category subsection contains thumbnails of all patterns in the category.

According to the PROBLEM/SOLUTION SUMMARY PATTERN [MD98], thumbnails of patterns present patterns in a strongly abbreviated form, limited to the name of the pattern, the problem that the pattern tackles, and the main idea of the solution. The thumbnail of a pattern gives a basic idea of the pattern, but it is insufficient for its implementation. In this study, thumbnails of patterns additionally include an icon symbolizing each pattern for easier recognition. Furthermore, a verification section follows each thumbnail of a pattern in preparation of the evaluation. This

verification section was not part of the original pattern and is instead derived from it. The verification section describes objective criteria that allow to check whether a given FLOSS project applies the FLOSS pattern or not.

Some FLOSS patterns described in this study have been published in multiple papers by different names, sometimes by different authors. The original publications do not explicitly state that these patterns are refinements of previously published patterns. In fact, at least in some cases the authors of refined patterns have not been aware that the refined patterns resemble previously published patterns. Indeed, these different versions of the same pattern differ in emphasis and highlight slightly different aspects of the pattern. A hard proof that these patterns are the same is impossible and therefore up to discussion. Consequentially, thumbnails of these deduplicated patterns additionally contain a deduplication section with arguments showing why the different variants represent the same pattern.

3.1.2 Related Work on Pattern Languages and Pattern Maps

Alexander explained that patterns cannot stand on their own, but have relations constituting a pattern language [Ale79]. Alexander et al. [Ale+77] presented their architectural pattern language in order of application size, starting with the structure of countries and regions and ending with specific arrangements of interiors in a house. In Alexander et al.'s pattern language, each pattern usually relies on more specific patterns that come later in the pattern language.

A diagram that visualizes a pattern language, especially the classification of patterns and their relationships, is called a pattern map. Alexander et al. [Ale+77] did not use a pattern map. Other authors have no common let alone standardized notation for pattern maps. Hence, authors of pattern languages focusing on software development teams and organizations varied in their use of pattern maps and similar visualizations of pattern languages.

The following paragraphs present a survey of related work about pattern languages and the different kinds of pattern maps. This lays the groundwork for structuring and visualizing the FLOSS pattern language.

Pattern languages can be seen as graphs, with patterns being nodes that have edges to the patterns required or useful for their implementation. This graph structure is a common technique used for pattern maps.

For example, Fowler [Fow96] showed pattern names in a figure with arrows between two patterns when the first influences the second. Furthermore, the patterns' x-axis positions designate their complexity. Their positions on the y-axis show to which of two major categories they belong. In contrast, Cunningham [Cun96] used the position of the pattern name in the diagram to indicate who (first dimension) implements the pattern on which type of work (second dimension). Again, related patterns are connected, although only with a line instead of an arrow. Cunningham explained that a relationship means that after application of a pattern, the related pattern may be applicable. These connections seem similar to Alexander's interpretation of pattern relationships [Ale79]. Coplien and Harrison [CH05a] described this notion in more detail: Every project applies pattern after pattern in any order that the pattern relations allow. Each such series of pattern applications is called a sequence. They used example sequences to illustrate the use of their pattern languages.

Notably, Weir [Wei98] and Roberts and Johnson [RJ98] used a different approach to visualize

patterns: Each pattern spans a time frame on the chronological x-axis, indicating in which phases of the design process it is applicable. Hannebauer et al. also applied this to a set of FLOSS patterns [HWG11].

Another type of pattern map can be seen in different works from Kelly [Kel06; Kel07]. These pattern maps categorize the patterns into families and lead the readers with questions to applicable patterns.

Previous work on FLOSS patterns also frequently visualized FLOSS patterns as nodes in directed graphs, e.g. [Wei09; Lin10]. This often implies a chronological order or order of dependency, but also includes more complex relationships. In some cases, edges include labels to denote their meanings. However, as explained before, previous work have referenced FLOSS patterns from other works only rarely. Consequently, it is unclear how patterns of different works relate to each other. This work addresses this gap and give an overview over the FLOSS pattern language as a whole.

There is a pattern language about writing patterns which includes a section specifically about pattern languages [MD98]. The pattern COMMON PROBLEMS HIGHLIGHTED PATTERN [MD98] of this pattern language describes the case where multiple different patterns solve the same problem. Thus, these patterns are alternatives and at most one of them should be used at once. Ratzka connected alternative patterns in his pattern map with an arrow with two heads [Rat13, p. 123].

Where applicable, the pattern language on writing patterns has been used as a guide for the FLOSS pattern language developed in this chapter. As discussed above, two types of relationships stand out in the related work: First, a pattern may depend on another pattern. Second, implementation of a pattern may exclude implementation of another pattern.

3.1.3 Pattern Maps for the FLOSS Pattern Language

Figure 3.1 shows an example of a pattern map as used in this thesis. A pattern is represented as a rectangle containing a pattern symbol and a pattern name in small caps. In the example, PATTERN A is a pattern with a symbol shaped like an A.

The pattern maps in this thesis were limited to the two types of relationship identified in the related work to keep them simple and understandable. The first type of relationship are alternatives that exclude each other. PATTERN A and PATTERN B are alternatives, indicated by a line with a circled $\bar{\wedge}$. At most one of the two or more alternatives can be implemented.

The second type of relationship is indicated by arrows, like the ones from SUPER PATTERN to PATTERN A and from SUPER PATTERN to PATTERN B. Arrows indicate that the starting pattern uses or requires the ending pattern. In this example, when choosing to implement SUPER PATTERN, it is also advisable or required to implement PATTERN A and/or PATTERN B. Whether it is strictly required or only sensible depends on the specific pattern combination and is not visible in this type of pattern map.

In some cases, one of multiple alternatives must be implemented, for example because another pattern requires this. The example in Figure 3.1 might mean that SUPER PATTERN requires either PATTERN A or PATTERN B, but this is not clear from the pattern map alone and readers would have to look at the patterns in detail.

All FLOSS patterns were unambiguously categorized to one of eight categories. In the pattern maps, a box for each category surrounds all patterns of that category and the box contains the

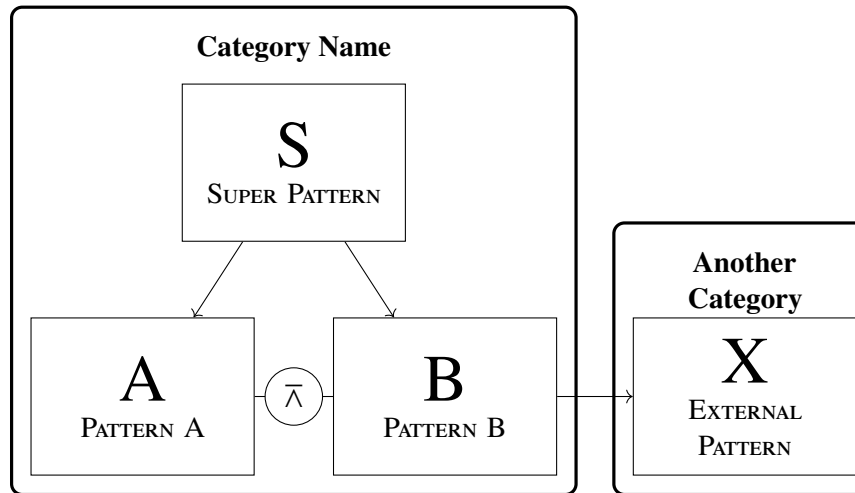


Figure 3.1: Example Pattern Map

category name in boldface. Each category has its own color that fills the category box (not shown in Figure 3.1). Besides the categorization, pattern positions have no further meaning, although more specialized patterns tend to be placed lower or more on the right in the figures.

The pattern map for each category section shows all FLOSS patterns that belong to this category and also all other FLOSS patterns that they depend on. EXTERNAL PATTERN is an example of such a pattern belonging to Another Category. However, for the sake of conciseness, dependencies are included only in one direction: The pattern maps do not show patterns that depend on patterns of the current category, but only patterns with arrows *from* the current category. In the example, there might be ANOTHER EXTERNAL PATTERN depending on PATTERN A, but it is not shown because no pattern of the current category depends on this ANOTHER EXTERNAL PATTERN. Patterns in other categories also have their respective category boxes. These category boxes are open on one side to indicate that there are other patterns not shown in the pattern map belonging to the category.

3.2 Categories of FLOSS Patterns

Figure 3.2 and Figure 3.3 are the two parts of a pattern map including all FLOSS patterns collected in this study.

Figure 3.2 contains organizational FLOSS patterns. Organizational FLOSS patterns are especially interesting for commercial companies, as the category Publish Closed Sources comprises FLOSS patterns to transform a closed-source project into a FLOSS project, and the depicted FLOSS patterns in the category Architecture hint at methods to earn money with a FLOSS project. FLOSS patterns in the category Licensing help to choose a licensing model for a FLOSS project that fits the purposes of its maintainer. While this covers the power over the source code from a copyright perspective, FLOSS patterns in the category Empower the Most Suitable focus on social aspects and help to channel the power over the FLOSS project’s community.

Figure 3.3 contains performance FLOSS patterns: how can a project improve its software

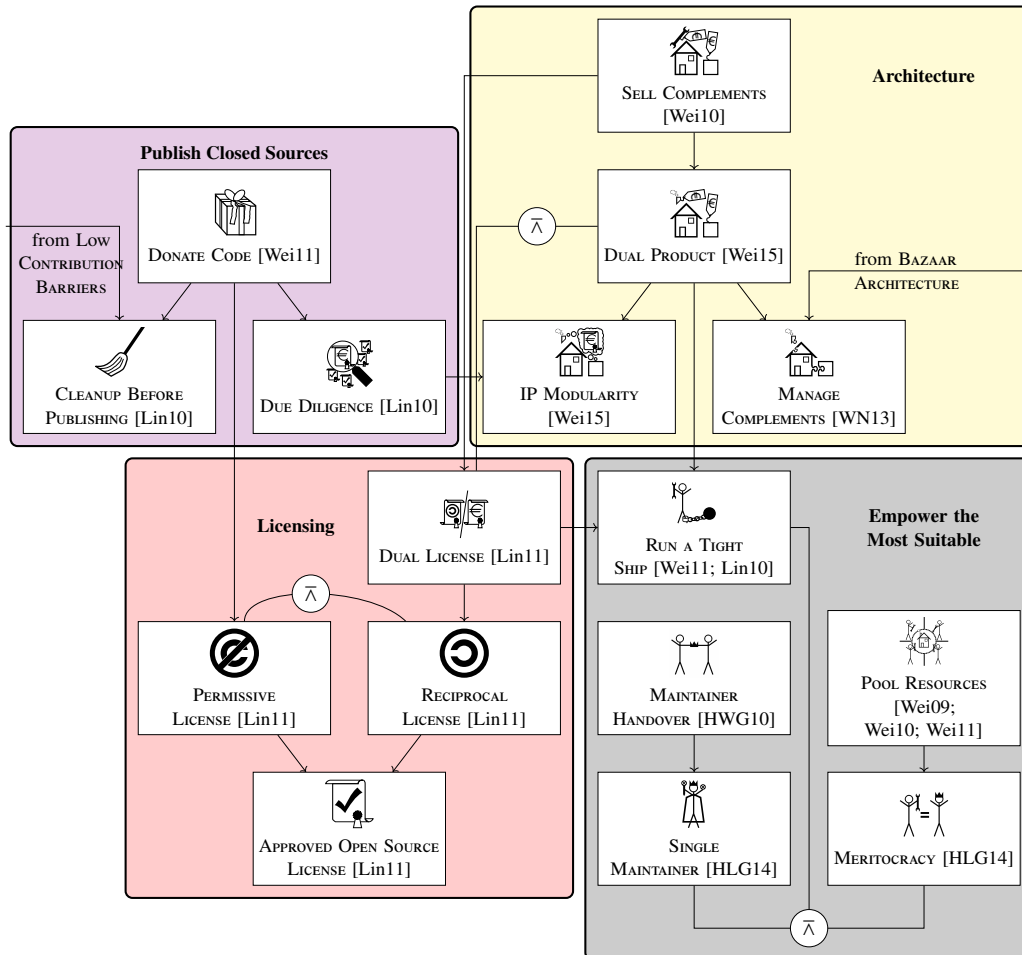


Figure 3.2: Pattern map for organizational FLOSS patterns

3 FLOSS Pattern Languages

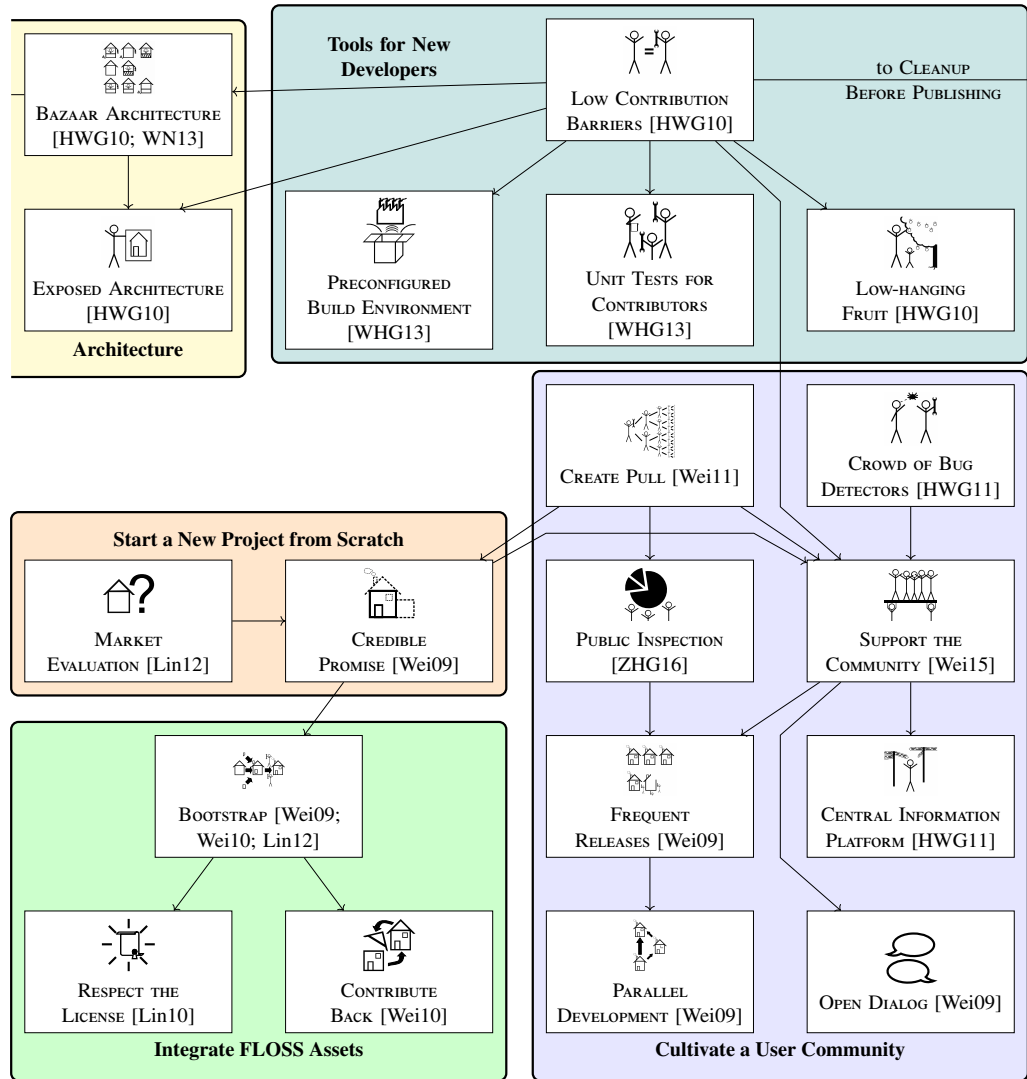


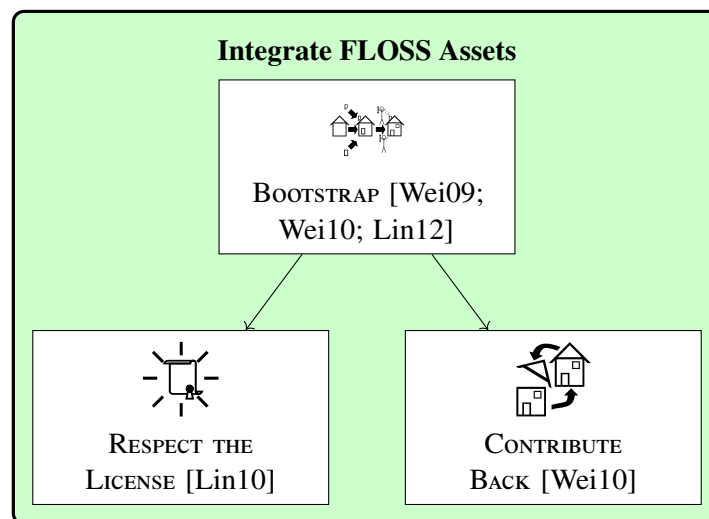
Figure 3.3: Pattern map for performance FLOSS patterns

development using FLOSS? The category Integrate FLOSS Assets contains the only FLOSS patterns that can also be used in closed-source software development projects. All other FLOSS patterns target maintainers of FLOSS projects. Start a New Project from Scratch describes the situation where a FLOSS project starts without a closed-source predecessor. The patterns in Cultivate a User Community describe how to attract new users and how to activate them for contributions. The special case of source code contributions is covered by the FLOSS patterns in the category Tools for New Developers and partly by the two depicted FLOSS patterns in the category Architecture.

The order of categories in this section corresponds to the order in which readers likely apply them. The first category, Integrate FLOSS Assets, targets the broadest audience, as both FLOSS maintainers and developers of closed software can apply its FLOSS patterns. The following two categories deal with the beginnings of FLOSS projects. This goes on to the last category, Empower the Most Suitable, which provides FLOSS patterns for large-scale FLOSS projects that require formal organization to keep their structure.

3.2.1 Integrate FLOSS Assets

While the other patterns in the FLOSS pattern language target maintainers of FLOSS projects, the patterns in this section are not specific to maintainers. Instead, they target general software development projects that want to use FLOSS components maintained by other parties. Figure 3.4 shows the three patterns and their dependencies: BOOTSTRAP describes how to quickly build an application based on FLOSS components. When using FLOSS components, RESPECT THE LICENSE helps to avoid legal problems, and CONTRIBUTE BACK shows the advantages of returning modifications on the FLOSS components back to their projects.



Bootstrap [Wei10]

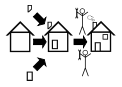
(also known as BUILD ON THE SHOULDERS OF OTHERS [Wei09], BUILDING ON OPEN SOURCE [Lin12])

Problem: How do you keep the costs for developing your product low?

Solution: Integrate assets from FLOSS projects.

Verification: BOOTSTRAP is considered fulfilled for a FLOSS project A if it depends on at least one separately maintained FLOSS project B. This is the case if one of the following criteria is fulfilled: First, A builds only if a static library of B is present. Second, A runs only if a dynamic library of B is present. Third, the source code of A incorporates a considerable amount of source code from B, where considerable shall be defined as 300 Lines of Code (LOC) in this case.

De-duplication: The patterns BUILD ON THE SHOULDERS OF OTHERS [Wei09], BOOTSTRAP [Wei10], and BUILDING ON OPEN SOURCE [Lin12] describe how FLOSS components can be used to quickly develop an initial version of a software product. Only BOOTSTRAP explicitly mentions the lower costs of this approach, although BUILD ON THE SHOULDERS OF OTHERS also talks about cost savings in its opening text. Related to this, BUILDING ON OPEN SOURCE argues for the reduced cost of a failure when using the pattern, and adds the additional aspect that FLOSS components may be replaced by commercial components once the product turns out to be a success. BOOTSTRAP refers to BUILD ON THE SHOULDERS OF OTHERS, so a difference seems to be intended, but it does not become clear from the pattern texts.



Respect the License [Lin10]

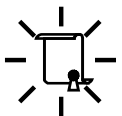
Problem: How to prevent problems with licensing after your product is released?

Solution: Check Licenses of FLOSS components you use starting early in the development process.

Verification: This pattern targets users of FLOSS libraries. Therefore, there are two possibilities to verify the usage of this pattern: First, it may be verified whether users of the FLOSS project under research RESPECT THE LICENSE. Second, it may be verified whether the FLOSS project under research RESPECTS THE LICENSE when using libraries of other FLOSS projects.

The first option theoretically includes checking every software project that exists, as any project might use libraries of the FLOSS project under research. Thus, this is not possible to validate.

For the second option, a verification needs to check whether the FLOSS project under research publishes its component under a license that is compatible with one of the licenses of each FLOSS library used. If this is not the case, this pattern was probably not applied. However, this conclusion does not work the other way around: If all licenses are compatible, there is no objective way to judge from the outside whether they are compatible by chance or because of careful checks. Thus, this pattern will not be used in the verification.



Contribute Back [Wei10]

Problem: How do you keep aligned with the FLOSS project?

Solution: Contribute non-core modifications back to the FLOSS project.



Verification: As with RESPECT THE LICENSE, there are two possibilities to verify the usage of this pattern: First, other projects may CONTRIBUTE BACK to the FLOSS project under research. Second, the FLOSS project under research may CONTRIBUTE BACK to the FLOSS project it depends on. The first case is more a statement about other projects than about the FLOSS project under research, although the FLOSS project under research may influence its contributions via other patterns, for example those in Section 3.2.7. The second case involves a definition problem: As FLOSS projects do not necessarily formally employ their developers, a developer might contribute to two (or more) FLOSS projects A and B, where A depends on B. If the developer is seen as a member of FLOSS project A, then A CONTRIBUTES BACK to B. If the developer is seen both as a member of A and of B, then the contributions to B are merely normal work and A does not apply the pattern. Thus, this pattern will not be used in the verification.

3.2.2 Start a New Project from Scratch

The FLOSS patterns in this section apply only to new greenfield FLOSS projects. Figure 3.5 shows the two FLOSS patterns and their relation. MARKET EVALUATION explains how FLOSS facilitates the production of what Ries calls a Minimum Viable Product (MVP) [Rie11], a prototype for testing whether there is demand for a real product. This first software version should show a CREDIBLE PROMISE to attract early adopters.

BOOTSTRAP helps to build the core features for CREDIBLE PROMISE. SUPPORT THE COMMUNITY attracts a user base from which to recruit developers with the CREDIBLE PROMISE.

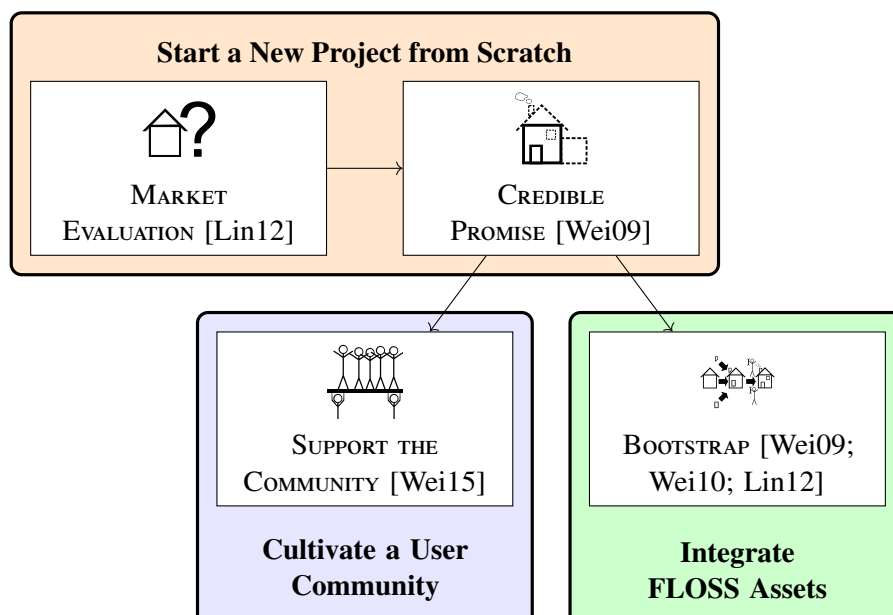


Figure 3.5: Patterns in the category Start a New Project from Scratch

Market Evaluation [Lin12]

Problem: How to facilitate interest in your software, determine if it is the right product and attract the early adopters, and cross the chasm and get the market majority to adopt your product?

Solution: Establish a beachhead by releasing your component from the start as FLOSS.

Verification: With the first public announcement of the project, the source code must be available under a FLOSS license.



Credible Promise [Wei09]

Problem: How do you mobilize developers to contribute to your project?

Solution: Build a critical mass of functionality early in your project that demonstrates that the project is doable and has merit.

Verification: With the first public announcement of the FLOSS project, a compilable or pre-compiled version of the component must be available. For applications, this version must run on at least one platform and support at least one core feature. For libraries, a test or demo application must also be available that fulfills the criterion for applications.



3.2.3 Publish Closed Sources

This section incorporates FLOSS patterns targeted at organizations developing closed source software. The pattern map in Figure 3.6 depicts these FLOSS patterns and their dependencies. These FLOSS patterns help to open up the closed development process. As visible in Figure 3.6, the FLOSS patterns circle around the core FLOSS pattern DONATE CODE that describes the main step of publishing the closed source code. CLEANUP BEFORE PUBLISHING and DUE DILIGENCE describe preparation steps before publishing that help being successful and prevent legal problems in the project, respectively.

A PERMISSIVE LICENSE allows everyone to use components published with DONATE CODE. IP MODULARITY helps to deal with those parts of the code that shall not be published after DUE DILIGENCE.

Donate Code [Wei11]

Problem: How do you extend the lifespan of your proprietary code?

Solution: Relinquish control over the code by releasing it under a flexible FLOSS license that allows others to build on it easily.

Verification: An organization must have developed the component for internal use or to sell it without releasing its source code. Its first release under a FLOSS license must be a working component already. This study considers this requirement fulfilled if there have been binary releases before the source code release or if the source code consists of at least 10 000 LOC. The last condition takes into account components developed internally or sold on marketing channels hidden from analysis of public data. The specific value of 10 000 LOC is an approximation of the



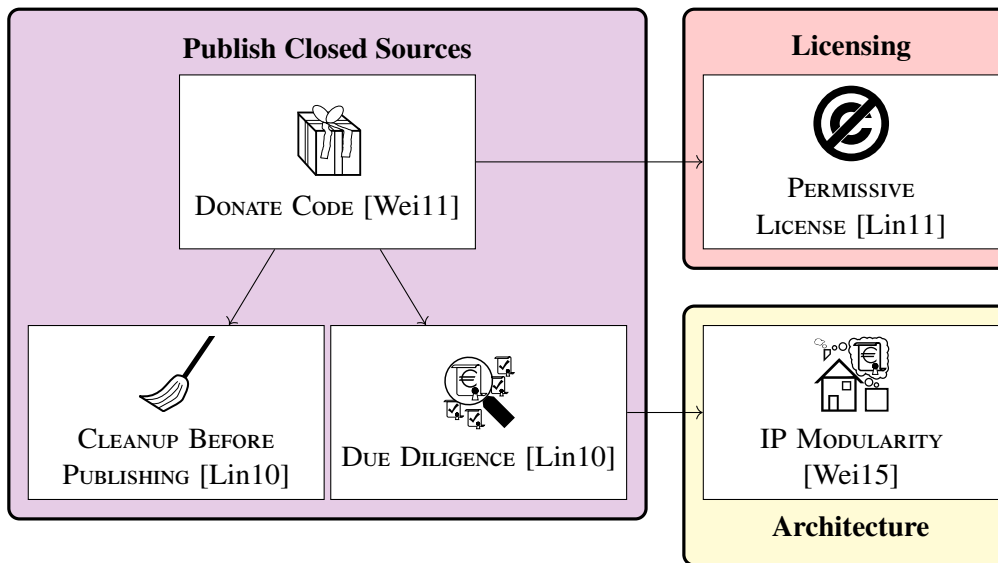


Figure 3.6: Pattern Map for Publish Closed Sources

average code size of a component on its first release. This threshold is based on expert opinion only and therefore subjective, but the number of LOC in actual FLOSS projects on their first release will presumably be much lower or greater than 10 000 LOC and therefore the specific value is not important for most verifications.

Cleanup Before Publishing [Lin10]

Problem: How to prepare the publication of a component as FLOSS in a way that attracts and not discourages acceptance and contributions and prevents problems with intellectual property?

Solution: Cleanup the structure, source, documentation, and supporting material of your project material before you publish it.

Verification: As the state of the component before its publication is usually hidden, analyses from outside cannot decide whether the quality of the component has been increased in a cleanup process before publishing or whether it has already been this way. Although counterexamples of projects not using this pattern might be identified in some cases, this pattern is not verified in the analysis.



Due Diligence [Lin10]

Problem: How to reduce the risk of being attacked because of claimed intellectual property violations like copyright, patent, trade secrets, or license violations?

Solution: Check for copied or otherwise integrated code, intellectual property, or patented technology. Ensure that the reuse complies with licenses and regulations.



Verification: Similarly to CLEANUP BEFORE PUBLISHING, there are cases where the existence of checks through the application of DUE DILIGENCE can neither be confirmed nor rejected. Thus, this pattern is also not verified in the analysis.

3.2.4 Licensing

The FSF as well as the OSI define Free Software and Open Source by their licenses [P*Fre15; P*Ope04]. Figure 3.7 shows FLOSS patterns that help to choose the right license for a FLOSS project. While an APPROVED OPEN SOURCE LICENSE is generally recommended, maintainers can still choose between a PERMISSIVE LICENSE and a RECIPROCAL LICENSE. The latter allows the combination with a commercial license in the DUAL LICENSE model. Notably, the FLOSS pattern DUAL LICENSE does not cover any combination of two licenses, but of two specific kinds. In fact, PERMISSIVE LICENSE and RECIPROCAL LICENSE are exclusive alternatives and therefore cannot be combined.

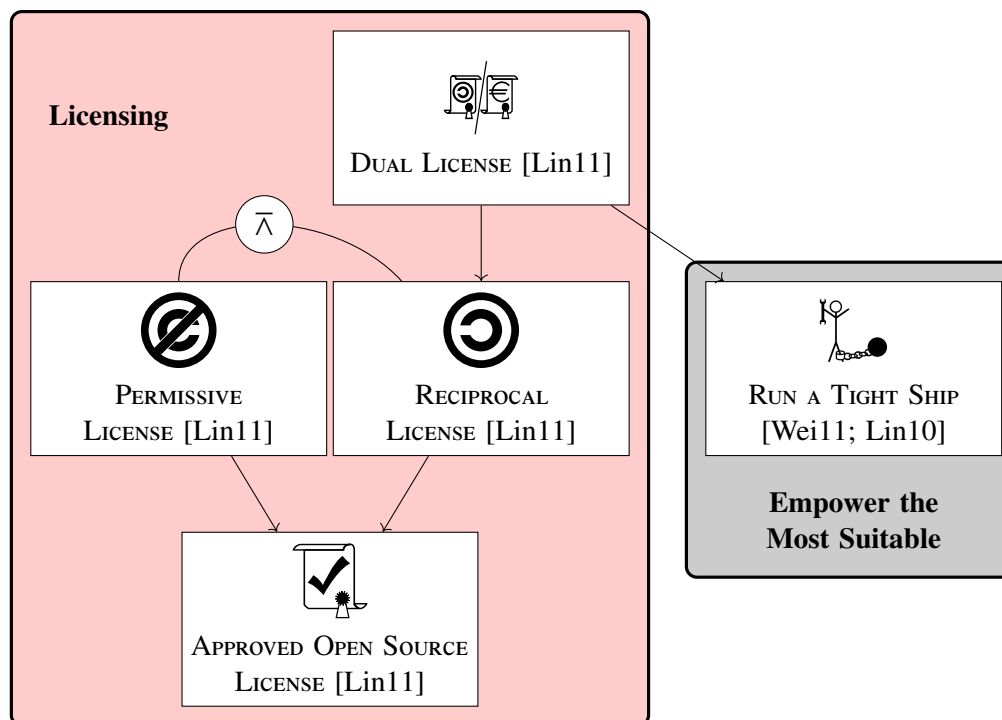


Figure 3.7: Pattern Map for Licensing

External contributors must assign copyright to the maintainer for the DUAL LICENSE model. Therefore, the maintainer must RUN A TIGHT SHIP in the FLOSS project or not accept external contributions at all.

Not depicted in Figure 3.7 is the relationship to RESPECT THE LICENSE, as it most commonly has no influence on the choice of the license for one's own software development project. However, if one of the incorporated FLOSS components uses a RECIPROCAL LICENSE, then the own software development project must also distribute its product under a RECIPROCAL LICENSE.

Dual License [Lin11]

Problem: How to encourage people to pay for your FLOSS component?

Solution: Release your software component with two (or more) licenses.

Verification: Users of a FLOSS project applying this pattern can choose between at least two licenses. At least one of the licenses requires users of the component to pay royalties. At least one other license is considered a RECIPROCAL LICENSE.



Reciprocal License [Lin11]

(also known as COPYLEFT LICENSE [Lin11])

Problem: How do you choose the type of FLOSS license so that it supports your business model?

Solution: Choose a license which requires derived or combined works to be released under the identical license.

Verification: The project must distribute its component and source code under at least one license that either the FSF lists as copyleft license [P*Fre16] or that is not on the FSF's list but its conditions use the copyleft method [P*Fre13].



Permissive License [Lin11]

Problem: How do you choose the type of FLOSS license so that it supports your business model?

Solution: Publish your component under a license which permits distribution with modifications without enforcing the publication of the source.

Verification: The project must distribute its component and source code under at least one license that either

- the FSF lists as permissive free software license [P*Fre16] or
- that is not on the FSF's list but it meets the requirements for an Open Source license [P*Ope04] and does not use the copyleft method [P*Fre13].



Approved Open Source License [Lin11]

Problem: How to choose the FLOSS license for your component?

Solution: Use an existing FLOSS license which best fits your needs.

Verification: The project must distribute its component and source code under at least one license that

- the FSF lists as compatible to the GNU General Public License (GPL) [P*Fre16] or
- the OSI lists in the categories “Licenses that are popular and widely used or with strong communities” or “Special purpose licenses” [P*Ope14].



3.2.5 Architecture

The FLOSS patterns in this section describe how to divide the software into modules and how to manage the resulting architecture. Figure 3.8 shows the FLOSS patterns and their relationships.

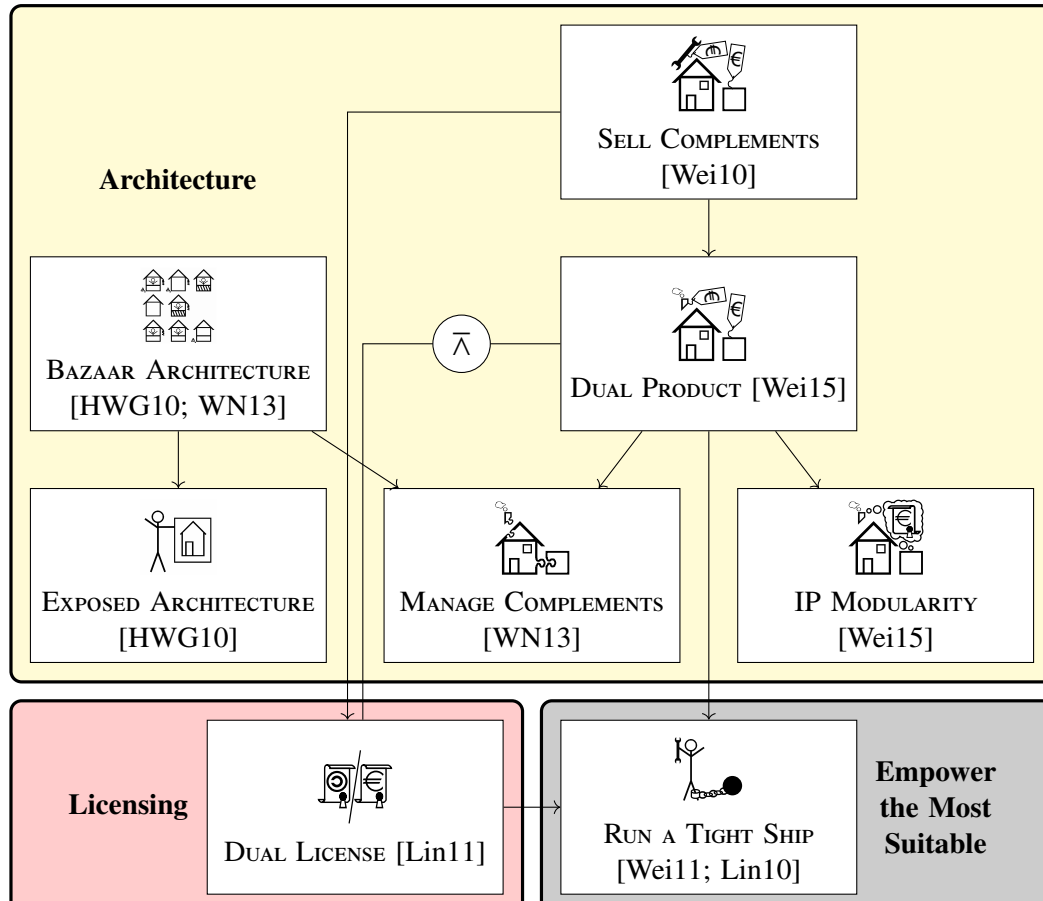


Figure 3.8: Pattern Map for Architecture

A **BAZAAR ARCHITECTURE** allows easy integration of contributions from the community, but it requires to carefully **MANAGE COMPLEMENTS** submitted to the FLOSS project. Contributors see in the **EXPOSED ARCHITECTURE** where to integrate their contributions.

For commercial maintainers of FLOSS projects, the business model has strong influence on the architecture. There are multiple ways to **SELL COMPLEMENTS** for the FLOSS application, one of which is a **DUAL PRODUCT**: One basic version is free and another extended version must be paid for. In this case, maintainers must have **IP MODULARITY** in their architecture to separate free from paid modules. Another possibility is a **DUAL LICENSE**. In both cases, maintainers **RUN A TIGHT SHIP** to use external contributions for their commercial products.

Bazaar Architecture [HWG11]

(also known as MODULAR ARCHITECTURE [WN13])

Problem: It is difficult for newcomers to add innovative features that have not been anticipated and require modifications of the architecture.

Solution: Plan the architecture of your software as a flat hierarchy of optional modules.

Verification: One requirement for this pattern is the existence of multiple modules in the project. At least 75 % of its modules must have less than 50 source code files, excluding build scripts and tests. At least 75 % of commits must modify the files of only one module.

De-duplication: Both BAZAAR ARCHITECTURE [HWG11] and MODULAR ARCHITECTURE [WN13] explain that changes to the software may require knowledge not only about the changed part of the source code, but also about related parts of the source code. With more dependencies, contributors have to acquire more knowledge about the code before they can change something. For both FLOSS patterns, the solution is an improved modularization. BAZAAR ARCHITECTURE argues for a flat dependency tree while MODULAR ARCHITECTURE describes techniques to refactor modules with many dependencies. Thus, the two FLOSS patterns differ in their specific implementation, but the problem and the general method of creating an “architecture for participation” [MRB06; WN13] is the same. Notably, both FLOSS patterns include UNIX and Mozilla as examples, but this is no evidence for their equivalence: The specific subsystems and events in Mozilla and UNIX used as examples are disjoint.



Exposed Architecture [HWG10]

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

Solution: Publish the component’s architecture.

Verification: The project’s documentation must describe the application’s architecture to realize this pattern. Multiple architecture diagrams or extensive textual descriptions both satisfy this condition.



Sell Complements [Wei10]

Problem: How do you monetize a FLOSS product?

Solution: Sell products or services (such as hardware or support) that complement the FLOSS product.

Verification: The FLOSS project’s maintainer is a commercial company and sells complimentary services for the FLOSS component.



Dual Product [Wei15]

Problem: You want to entice commercial users to pay for a FLOSS product.

Solution: Keep parts of the FLOSS project closed, and sell a commercial version of the product



3 FLOSS Pattern Languages

with exclusive features.

Verification: The FLOSS project's maintainer is a commercial company and sells another version of the same product that includes the FLOSS application and offers additional features or offers the FLOSS application as a service. If the FLOSS project's maintainer is unclear, every organization counts as maintainer that is responsible for at least one fifth of the commits to the FLOSS project's VCS within the study period.

Manage Complements [WN13]

Problem: How to extend the code while maintaining the architectural integrity of the overall system?

Solution: Establish a contribution process, set guidelines and provide a development toolkit to manage complement development.

Verification: The project must provide an API for its main application that allows others to develop complements. A complement is either an add-on or a complementary application. An add-on is an executable package that the main application loads at runtime. A complementary application is a stand-alone application that accesses data or functions of the main application through its API while it is running.



IP Modularity [Wei15]

Problem: You need to manage open and closed components of a dual product.

Solution: Align intellectual property (IP) with product architecture.

Verification: This pattern guides companies on how to partition the architecture into closed and open parts. Since the closed parts and especially their usage of IP are invisible from the outside, this pattern cannot be verified without internal knowledge. The analysis in this study will therefore skip verification of this pattern.



3.2.6 Cultivate a User Community

The success of a FLOSS project depends on its users [Xu+05], but FLOSS projects must also be able to draw on their capital of users [CM07]. Figure 3.9 shows patterns that help with this goal.

Attracting users via both PUBLIC INSPECTION and SUPPORT THE COMMUNITY helps to CREATE PULL for the FLOSS project and thereby grow a large and active community. New FLOSS project may also use CREDIBLE PROMISE to CREATE PULL. Because of Linus's Law [Ray00], a large community can work as a CROWD OF BUG DETECTORS.

Possible ways to SUPPORT THE COMMUNITY are an OPEN DIALOG, a CENTRAL INFORMATION PLATFORM, and FREQUENT RELEASES. An EXPOSED ARCHITECTURE can be one element of an OPEN DIALOG. For FREQUENT RELEASES, it is advisable to use PARALLEL DEVELOPMENT to separate stable from unstable releases.

Another way to attract users is to demonstrate the FLOSS project's quality via PUBLIC INSPECTION. These PUBLIC INSPECTIONS must always refer to recent releases to be useful, so a requirement is FREQUENT RELEASES.

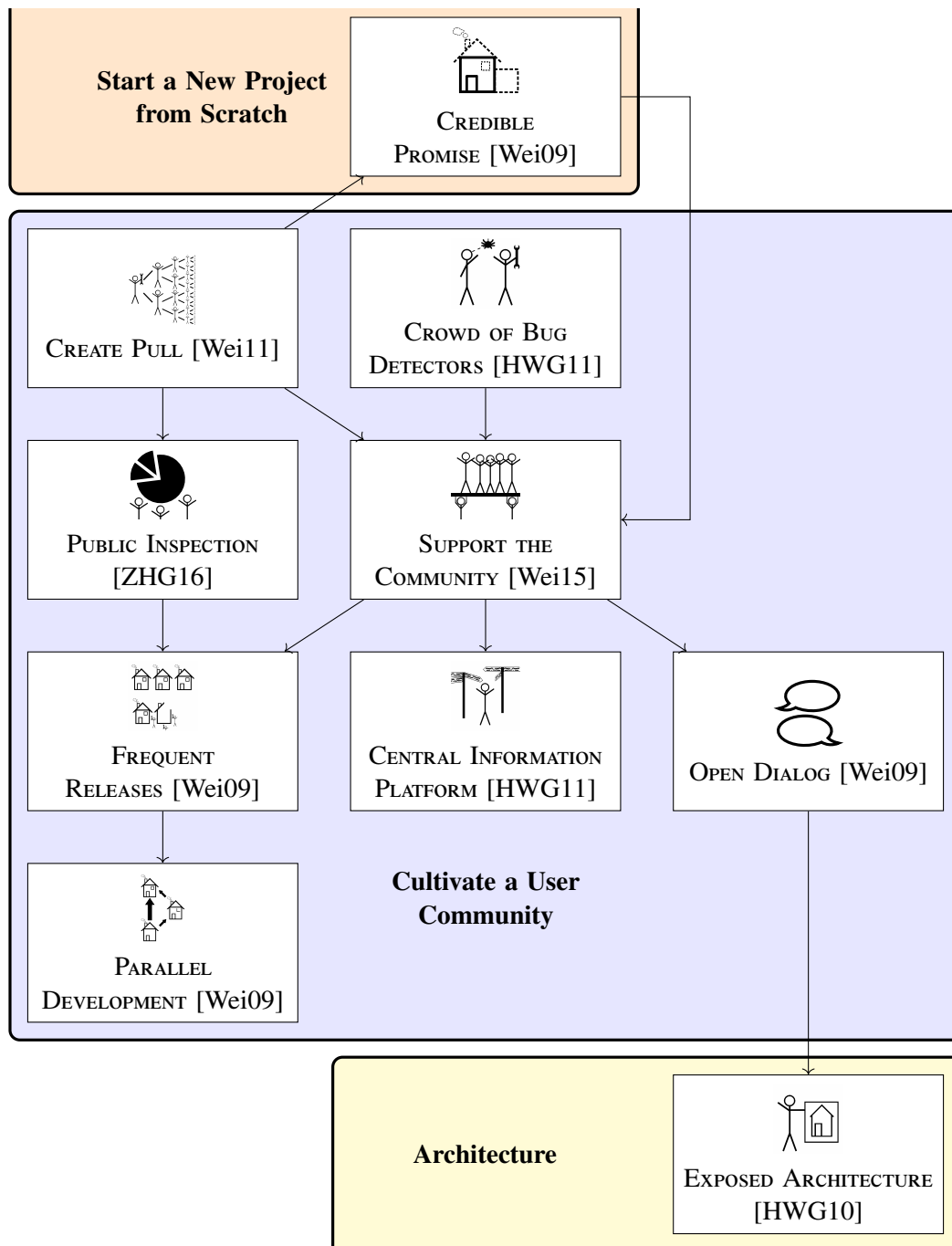


Figure 3.9: Pattern Map for Cultivate a User Community

3 FLOSS Pattern Languages

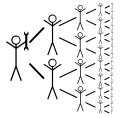
FLOSS patterns that distinctively support source code contributions are excluded from this category because of their distinguished importance for the FLOSS project. This category of FLOSS patterns is treated separately in Section 3.2.7.

Create Pull [Wei11]

Problem: How do you market your FLOSS product?

Solution: Make it easy to access your product and leverage distributors, partners, as well as community members to market your product.

Verification: Commercial software companies use this FLOSS pattern to market their products. It describes an intention to use FLOSS and the effects that help market services around a FLOSS product. It is therefore only used in FLOSS projects with a commercial software company as maintainer. For FLOSS projects with commercial maintainers, the positive effects described in the FLOSS pattern are natural consequences of being FLOSS and no result of deliberate actions of the project's maintainers. CREATE PULL mentions no distinctive features of projects using this FLOSS pattern, except a forum for the community. The analysis will therefore not verify this FLOSS pattern.



Crowd of Bug Detectors [HWG11]

Problem: Developers do not fix some of the bugs because they occur only seldom or never in their environments.

Solution: Take bug reports seriously and encourage your users to file bug reports.

Verification: For projects where the developers may commit changes without corresponding issue report in the issue tracker, CROWD OF BUG DETECTORS is considered applied if reactions to reported issues take less than 24 hours in average.

This is different for projects with the policy that every commit must have a matching issue report in the issue tracker. In this case, issue reports filed by users will be compared to issue reports filed by developers. The average duration till the first answer to users' issue reports must be lower than the average duration till the first answer in developers' issue reports.

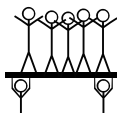
Issue reports may be bug reports and feature requests. The former describe faults, i.e. differences between program behavior and specification, while the latter describe changes to the specification. The FLOSS pattern focuses on bug reports, but these are often hard to distinguish from feature requests, because FLOSS projects seldom use formal specification documents. Therefore, the verification does not distinguish between bug reports and feature requests.



Support the Community [Wei15]

Problem: You need to build a healthy community around your project.

Solution: Support the community without expecting an immediate return. It's all about respect and building legitimacy, one step at a time.



Verification: The pattern describes five aspects in its solution part to implement the pattern. One aspect is automatically fulfilled for an active FLOSS project (i). Three of the aspects correspond to other FLOSS patterns, specifically APPROVED OPEN SOURCE LICENSE [Lin11] (ii), CENTRAL INFORMATION PLATFORM [HWG11] (iii), and OPEN DIALOG [Wei09] (iv). The fifth aspect directs the maintainer to support community members without charge instead of only offering paid support. This fifth aspect is considered fulfilled if at least 75 % of the support inquiries receive answers within 72 hours, and at most 10 % of support inquiries receive answers with offers for premium support.

Public Inspection [ZHG16]

Problem: Demanding users abstain from the FLOSS project, because they are afraid that the project's quality is too low.

Solution: Let an independent third party measure your software quality and visualize current results publicly in real-time.

Verification: An independent third party must measure a non-obvious quality property of the FLOSS project's software and the FLOSS project must visualize the result on its web-site. Visualizations include badges, diagrams, and textual markup. The visualization must update automatically when the measurement changes. The third party must measure multiple FLOSS projects, as this allows comparisons of the results.



Frequent Releases [Wei09]

Problem: How do you move a FLOSS project along quickly?

Solution: Release code in small, quick increments.

Verification: A release is a downloadable package with an executable version of the component. In this context, executable usually means binary data. There are two exceptions where source code also counts as executable: First, if the component is interpreted instead of compiled. Second, if the project never releases compiled versions, for example because it is a library that needs to be compiled along with the application that uses it. This FLOSS pattern is applied if a FLOSS project publishes new releases of its component at least once a month.

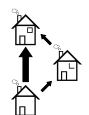


Parallel Development [Wei09]

Problem: How do you balance the need of users for stability with the need to explore new directions for your project?

Solution: Maintain separate release streams, those with the official stable releases and others for experimental development.

Verification: The project's VCS must contain a main branch and multiple additional branches. Only merges from other branches add new features to the main branch. Fixes of defects may go directly into the main branch.



Central Information Platform [HWG11]

Problem: The project provides different kinds of development and support resources to its users, but a user has to find the right system for every task.

Solution: Provide a central information platform that describes and links to all tools and resources that the project uses.

Verification: The existence of a web site that holds or links to all important resources of the project needs to be tested. Furthermore, it has to be verified that the web site is up to date and no additional alternative web sites with apparently similar purposes exist. If for some reason the location of the web site changed, the old web site is often kept for several reasons. Reasons for keeping the old website includes preventing broken links and conserving information which is not transferred to the new web site. However, such an old site needs to be marked as outdated and point to the new site. Finally, all of the FLOSS project's resources are expected to link back to the main web site.



Open Dialog [Wei09]

Problem: How do you engage others in your project?

Solution: Conduct the project in the open, maintaining a two-way dialog with project participants (users and external developers).

Verification: The FLOSS project must discuss its plans and technical decisions on a publicly visible platform like a wiki, a forum, or a mailing list. There must be a documented way how outsiders may gain write access to this platform, e.g. send emails to the mailing list.



3.2.7 Tools for New Developers

Figure 3.10 shows the patterns of this section. The success of a FLOSS project depends on its ability to attract new developers [BGD07; CM07; ACB09], which is only possible with **LOW CONTRIBUTION BARRIERS**. There are three other, more specific FLOSS patterns in this category, which help to achieve **LOW CONTRIBUTION BARRIERS**. Each tackles a specific modification barrier and presents a technique to lower it. Specifically, **PRECONFIGURED BUILD ENVIRONMENT** lets newcomers skip the sometimes cumbersome setup of the development environment. **UNIT TESTS FOR CONTRIBUTORS** allow newcomers to test their modifications before they publish them. **LOW-HANGING FRUIT** particularly reduce the essential difficulty of a contribution, allowing newcomers to choose a problem suiting their abilities.

Both a **BAZAAR ARCHITECTURE** and an **EXPOSED ARCHITECTURE** help developers to integrate new features into the software and therefore foster **LOW CONTRIBUTION BARRIERS**. Similarly, **CLEANUP BEFORE PUBLISHING** reduces the work required for a modification. Some developers go through a phase of being active users before they become developers [YK03; Her+06]. **SUPPORT THE COMMUNITY** helps to grow an active user base and provides a joining script with **LOW CONTRIBUTION BARRIERS** for these active users.

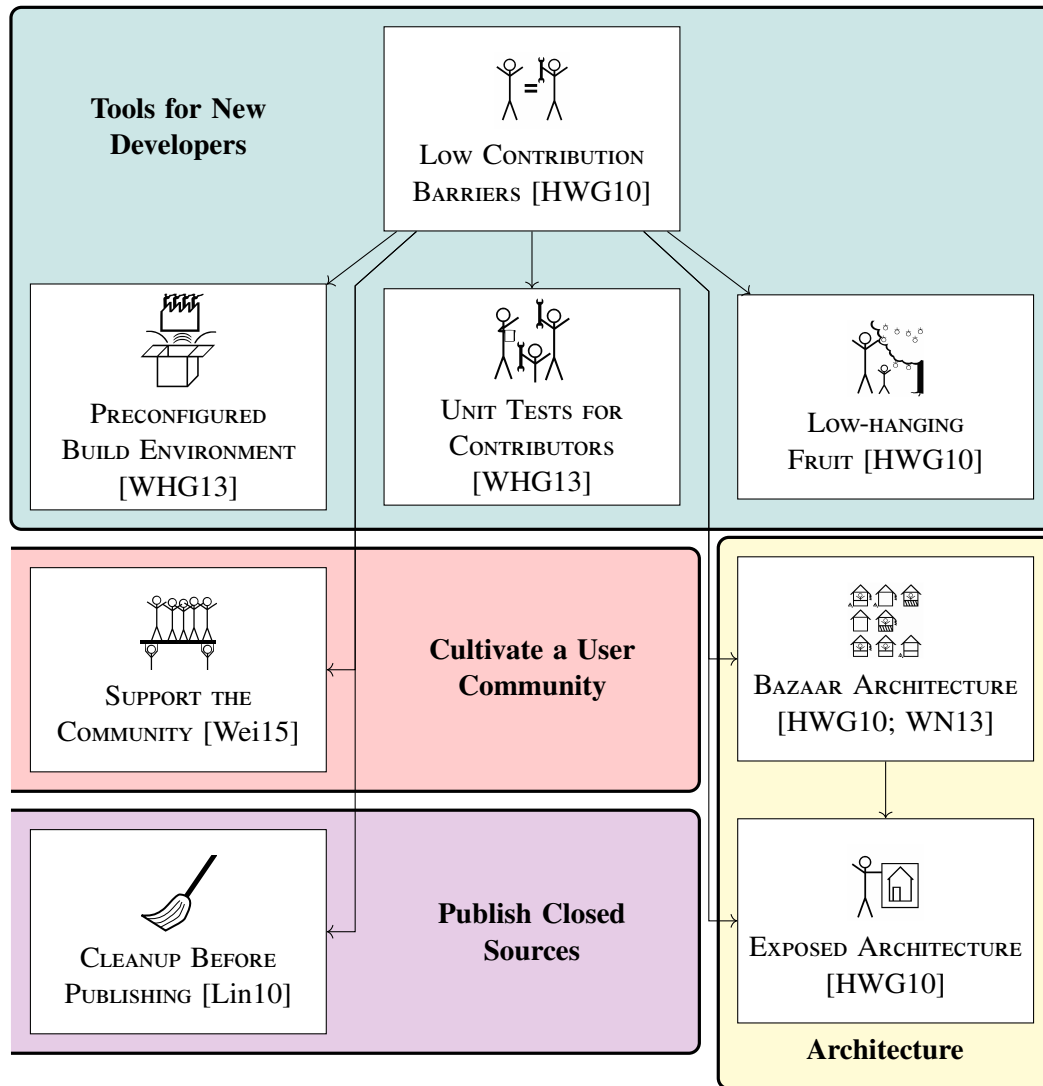


Figure 3.10: Pattern Map for Tools for New Developers

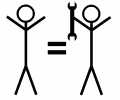
Low Contribution Barriers [HWG10]

(also known as USER CONTRIBUTIONS [HWG10])

Problem: The existing developers cannot fulfill the expectations of the grown user base.

Solution: Lower the contribution barriers to your FLOSS project, so your users can satisfy their demand themselves.

Verification: A FLOSS project implementing this FLOSS pattern regularly receives source code contributions from developers that had not contributed source code to the FLOSS project before. Developers stayed with the Linux kernel projects for 17 months on average [HNH03], so in order to at least hold a steady number of developers, the Linux kernel projects have to recruit $\frac{1}{17}$ of all current developers every month. More generally, this FLOSS pattern is considered fulfilled if at least 5 % of all active developers in any month were newcomers. Developers are considered newcomers in a given month if they contributed source code to the FLOSS project for their first time in this month.



Preconfigured Build Environment [WHG13]

Problem: Newcomers get stuck when setting up the build environment.

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

Verification: A FLOSS project must provide a download package supporting the build to realize this FLOSS pattern. The download package contains all build tools and dependencies or must at least automatically download all missing dependencies. Except from starting a Virtual Machine (VM) or installation script, the first build must not require more steps than later builds.



Unit Tests for Contributors [WHG13]

Problem: Newcomers avoid modifications as they are afraid to introduce new bugs.

Solution: Provide easy access to unit tests even for newcomers.

Verification: To successfully apply this FLOSS pattern, the source code must contain unit tests. The unit tests do not need configuration except for the configuration of the application under test itself. Alternatively, a CI system builds and tests all commits. In this case, it must build and test even pending commits from newcomers.



Low-hanging Fruit [HWG10]

Problem: Newcomers cannot work on the FLOSS project's issues because they have too much to learn before they could be helpful.

Solution: Mark some issues as easy and leave them open for newcomers.

Verification: The FLOSS pattern is applied if some of the issues in the issue tracker are tagged "good first bug" or with an equivalent phrase. A separate list of tasks for newcomers also fulfills this FLOSS pattern.



3.2.8 Empower the Most Suitable

FLOSS licenses ensure that there is no *legal* owner of the software who has the exclusive rights to decide how to develop the FLOSS component. Legal power over a component is the primary subject of the FLOSS patterns in the category Licensing and also of DONATE CODE.

Nevertheless, there is usually a community around any FLOSS application with a *social* hierarchy that determines who has social power in the FLOSS project. The core developers and maintainers rank highest in this hierarchy and decide about the direction of the project [YK03]. The FLOSS project's power model determines who these maintainers are. Figure 3.11 shows FLOSS patterns describing these power models. SINGLE MAINTAINER and MERITOCRACY are very common basic models that exclude each other, with only special exceptions. When organizations RUN A TIGHT SHIP on FLOSS projects, the locus of power is outside the FLOSS project's volunteer community.

A SINGLE MAINTAINER may transfer power over a FLOSS project to a successor with a MAINTAINER HANDOVER. When multiple companies want to POOL RESOURCES, they typically use a MERITOCRACY to direct the corresponding FLOSS project.

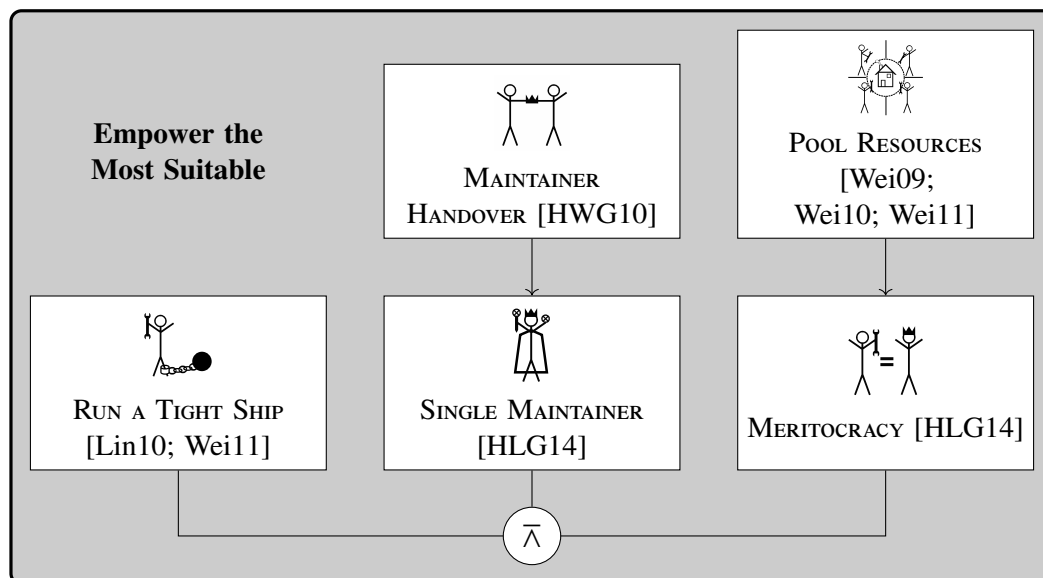


Figure 3.11: Pattern Map for Empower the Most Suitable

Run a Tight Ship [Wei11]

(also known as CONTRIBUTOR AGREEMENT [Lin10])

Problem: How do you keep control of the project's direction?

Solution: Maintain full ownership of the code.

Verification: Power over the FLOSS project does not stem from inside, i.e. the development team, but from an organization not originating in the development team. This is the case if a commercial organization holds the copyright over the code and requires non-employed contributors to transfer



copyright non-exclusively to the commercial organization. It is also the case if the commercial organization pays the FLOSS project's maintainers for maintaining the FLOSS project and holds copyright over name and logo.

De-duplication: The solution of RUN A TIGHT SHIP [Wei11] describes two alternatives for its realization: Either write all source code yourself or let all external contributors transfer the rights to you. The latter possibility specifically refers to CONTRIBUTOR AGREEMENT [Lin10] by its previous name ASSIGN COPYRIGHT. Writing all source code yourself can be seen as a special case of CONTRIBUTOR AGREEMENT in which no external contributors exist. This special case is not very interesting for a pattern on FLOSS, so it does not deserve a FLOSS pattern on its own. Consequences, examples, as well as related patterns for the two FLOSS patterns intersect to a high degree, further substantiating their equivalence. However, CONTRIBUTOR AGREEMENT generally puts more emphasis on legal aspects of the pattern, while RUN A TIGHT SHIP emphasizes how to keep the power over the FLOSS project. Ultimately, each FLOSS pattern cannot exist without the other, making them two aspects of the same FLOSS pattern.

Maintainer Handover [HWG10]

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

Solution: Appoint a new maintainer if you do not want to continue maintaining the FLOSS project.

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



no change, when the current maintainer is still the original maintainer,

smooth transition, if no or only few problems with the new maintainer have occurred,

abandoned, when nobody maintains the FLOSS project anymore,

branched, if the maintainer left the FLOSS project, but instead of a handover, a new development branch was initiated, and

complicated transition, when the FLOSS project has been taken over, but the transition was not smooth and ended in or lead to conflicts. A significant share of users or developers must have left the project because of the transition.

Pool Resources [Wei10]

(also known as ONE SOLUTION [Wei09] and GOOD ENOUGH [Wei09], NO SINGLE VENDOR IN CHARGE [Wei10], and FOUNDATION [Wei11])

Problem: How do you attract other companies to contribute to a FLOSS project that you have created?

Solution: Pool resources with other companies to jointly develop a common stack of FLOSS assets that the companies can all build on to develop their individual products.



Verification: This FLOSS pattern applies to all FLOSS projects maintained by a foundation.

For other FLOSS projects, the FLOSS pattern may still be fulfilled if the FLOSS project has no legal form, for example, participating companies may cooperate via a GitHub organization [P*Nea10]. In this case, the FLOSS pattern is also considered applied when external organizations with commercial interest in the FLOSS project contribute at least 20 % of all commits together. All organizations are considered external except for the company with the most contributions, so this main company does not count to the 20 % of commits.

De-duplication: In its earliest description, the FLOSS pattern already has two names, ONE SOLUTION and GOOD ENOUGH [Wei09]. This FLOSS pattern helps to decide which parts of a product should be kept closed and which parts should be offered as FLOSS. This has the advantage that externals contribute to the FLOSS parts. POOL RESOURCES [Wei10] goes one step further and specifies that likely candidates for these externals are competitors, as they most likely have the same use cases and therefore need the same components for their software. The same paper shortly mentions NO SINGLE VENDOR IN CHARGE [Wei10], which was published later as FOUNDATION [Wei11]. The pattern FOUNDATION is specific about the legal framework when implementing POOL RESOURCES and the two papers depict them as two separate patterns, with POOL RESOURCES being applicable after FOUNDATION.

The strongest argument to treat them as a single FLOSS pattern is that neither can exist without the other: It is useless to have a FOUNDATION maintain a FLOSS project in the sense of the FLOSS pattern FOUNDATION if there are no external contributors who POOL RESOURCES with the FOUNDATION. As described above, the papers themselves describe the dependency in the other direction already: It is unlikely that competitors POOL RESOURCES with you if you remain in control over the source code and there is no FOUNDATION. All three versions of the FLOSS pattern include Eclipse as an implementation example, further supporting that the three versions describe different aspects of the same FLOSS pattern instead of separate FLOSS patterns.

Single Maintainer [HLG14]

(also known as BENEVOLENT DICTATOR [HLG14])

Problem: How can a FLOSS project quickly create an organizational structure to enable community contributions?

Solution: Concentrate all power in the FLOSS project in the hands of one capable individual.

Verification: The FLOSS project's maintainer is a single natural person. If the community discusses problems, for example on the mailing list, the maintainer still has the last word on every discussion. There can be community decision processes, but they are only recommendations for the maintainer and the maintainer sometimes decides differently.



Meritocracy [HLG14]

Problem: How should key decisions of the FLOSS project be made such that the maximum number of developers identify with the decisions?

Solution: The more a person contributes to the project, the more influence is granted to that person.



Verification: The FLOSS project has formal statutes that specify one or more committees. The committees elect each other or themselves and steer the project.

3.3 Patterns Lowering Contribution Barriers

This section contains six full FLOSS patterns that lower contribution barriers. The first four of them belong to the category Tools for New Developers. They depend on four other FLOSS patterns in other categories that help to lower contribution barriers. This section includes two of them, BAZAAR ARCHITECTURE and EXPOSED ARCHITECTURE, both of which belong to the section Architecture. The two FLOSS patterns CLEANUP BEFORE PUBLISHING [Lin10] and SUPPORT THE COMMUNITY [Wei15] referred to in Section 3.2.7 also lower contribution barriers and thematically fit into this section. However, I am not the author of these two FLOSS patterns, so they are not part of this section.

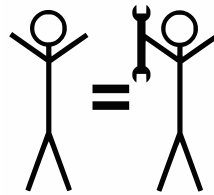
There are multiple formats for patterns. This paper uses a format similar to Kelly’s pattern format [Kel06]. Each FLOSS pattern comprises

- the *name* of the FLOSS pattern,
- an *icon* symbolizing the FLOSS pattern,
- the *context* in which the FLOSS pattern may be applied,
- a short description of the *problem* that the FLOSS pattern solves,
- *forces* that implementors of the FLOSS 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 ✓ and ✗, respectively. Each consequence describes the effect of applying the solution on a force,
- the *known uses and sources* serving as examples for the application of the FLOSS pattern,
- *related patterns* that are influenced by or influence the FLOSS pattern at hand, and
- objective *verification* criteria to check whether a FLOSS project applies the FLOSS pattern. These are the same criteria as those in Section 3.2.

The target audience of the FLOSS patterns presented in this section are maintainers of FLOSS projects. They may implement the solution described in the FLOSS pattern and therewith solve the problem described in the FLOSS pattern. Thus, FLOSS patterns directly address this target audience, especially in the solution part and use “you” to refer to FLOSS project maintainers.

3.3.1 Low Contribution Barriers¹

also known as USER CONTRIBUTIONS



Context:

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

Problem:

The existing developers cannot fulfill the expectations of the grown user base.

Forces:

- **90 % suitability.** The FLOSS project can solve a problem for its potential users, but not perfectly. Each user has additional requirements that the component not yet fulfills.
- **High demand of many users.** The success of a FLOSS project depends on the number of users it can attract [CAH03; YK03; SSN09], but more users also have more different feature requirements. The benefits of a larger user base like more and more detailed bug reports and more code contributions from those users may be lower than the additional effort required to meet their growing demand.
- **Dichotomy of domain and implementation expertise.** Users have inherent knowledge of what the project should deliver, but the developers cannot access this knowledge directly [Bro10, Chapter 15].
- **Project-specific development expertise.** Some of the users are experienced software developers and know the programming language and technologies used in the FLOSS project from other projects already. They are willing and able to develop patches that would satisfy the requirements of themselves and other users, but they have little or no knowledge about the architecture and coding style of the FLOSS project.
- **Goal differences.** The core developers may have their own ideas where the project should be going, but these ideas might be different from the users' requirements.

¹A preliminary version of this section was published previously [HWG10].

Solution:

Lower the contribution barriers to your FLOSS project, so your users can satisfy their demand themselves.

As described in Section 2.8, each FLOSS project offers different joining scripts to recruit new developers. The demographic background of a newcomer is an important factor to determine which joining script the newcomer uses. The joining script determines which contribution barriers apply. The demographic background also determines the newcomer's perspective on the FLOSS project and thereby the extent by which each contribution barrier hinders the newcomer.

It is effective to recruit developers from the same audience that also creates the demand, because these developers know the requirements already. Thus, if for example private users are not yet satisfied with the FLOSS project and have additional requirements, then the contribution barriers relevant for private users should be lowered. As private users usually become active users before they become developers [Her+06], you should take care of your issue tracker, so your users can report errors and request new features. You should also create and maintain support forums and answer to support requests of your users.

If, on the other hand, organizations use your component and adapt it to their needs, then the newcomers are Strictly Pragmatic Patchers as defined in Section 2.6.5. As shown, these newcomers are especially sensitive to modification barriers. These newcomers must be able to access the source code of your component easily and they must be given the right to change it. Describe how to build the software from the source code and provide tools that help with this task. Explain your own requirements for code patches like coding style and design considerations.

If your FLOSS project is a library or framework, your users have expertise with the programming language and IDE. You should focus on submission barriers in order to have them submit their modifications back to your FLOSS project. In all cases, you should clearly point out how to contribute.

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 [Ray00].

Consequences:

The consequences are discussed in more detail in Section 1.3.7.

- ✓ **Users are developers.** Users can adapt the component to their own need if the component is easy to modify. They contribute these modifications back to the FLOSS project if patches are easy to submit.
- ✓ **The long tail.** When the number of users is large enough, then 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.
- ✓ **Implicit requirements engineering.** Having users and developers in a double role allows projects to exist without "precise specifications or requirements documents" [GA04]. 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 FLOSS project can be used for the actual development.

- ✗ **Brooks's Law.** The core developers now need to split time between reacting to input from newcomers and pursuing their own scheme. They need to help newcomers adapt code contributions to the FLOSS project's coding style and filter out the code that does not work together with the FLOSS project's architecture. This mentoring may cost more work time than it provides [Bro95, Chapter 2].
- ✗ **Lost gatekeeper.** Newcomers 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. If the newcomers are less perspicacious than the core developers due to lowered contribution barriers, their influence may have a negative impact on the FLOSS project.

Known Uses and Sources:

This pattern is one of the key elements of what Raymond [Ray00] described as the bazaar development style. Accordingly, Raymond's own project "fetchmail" employed the FLOSS pattern and also the Linux kernel [P*Lin15], Raymond's primary object of observation. Other examples include Mozilla Firefox [P*Moz15m].

Gacek and Arief identified a characteristic "Developers are always users" and found it to be common among all FLOSS projects [GA04]. Raymond also advises that "treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging." [Ray00, advice six].

Other research suggests that in fact only part of the successful FLOSS projects employ this pattern: On the one hand, Krishnamurthy [Kri02] has shown that the number of developers in successful FLOSS projects increases over time: The age of a FLOSS project in months correlates with the number of its developer with a coefficient of 0.228 in his data set. 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. This suggests that these projects have recruited none or only few developers and thus, they have not successfully used the pattern **LOW CONTRIBUTION BARRIERS**, at least not to the extent that users became developers. Capiluppi and Michlmayr [CM07] showed that FLOSS projects may have two different statuses. If they are in a "cathedral" status, a more or less constant group of developers creates the source code. They reach a "bazaar" status if more developers join than leave the FLOSS project, and then the number of developers as well as the amount of their output keeps growing. They also present Wine [P*cod15] as a FLOSS project with "bazaar" status, so Wine also implements this FLOSS pattern.

Related Patterns:

Section 2.9 summarizes important contribution barriers. The following FLOSS patterns lower contribution barriers, especially the two most important ones listed in Section 2.9:

3 FLOSS Pattern Languages

While EXPOSED ARCHITECTURE helps newcomers to understand the overall structure of the FLOSS project, a BAZAAR ARCHITECTURE reduces the architectural complexity in the first place. Both let newcomers more quickly find the right location in the source code that they shall modify and therefore lower the most important contribution barrier listed in Section 2.9, difficulties to understand the structure of the FLOSS project.

The FLOSS pattern PRECONFIGURED BUILD ENVIRONMENT lowers modification barriers especially for those newcomers who do not want to hassle around with the technical details of the FLOSS project. It reduces the amount of work necessary for the often frustrating task of setting up and configuring a development environment.

LOW-HANGING FRUIT reduces the essential difficulty to modify the software, but only for those newcomers who do not already have a specific modification in mind. This lowers a modification barrier and lets newcomers more easily Find the Code (see Section 2.4) to be modified if it is used appropriately.

UNIT TESTS FOR CONTRIBUTORS lets newcomers more easily test their modifications before submission. This can reduce submission barriers, as it reduces the probability that reviewers request improvements on the patch from newcomers before the patch is accepted.

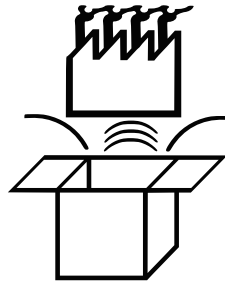
If a closed source project is about to be published as a FLOSS project, CLEANUP BEFORE PUBLISHING [Lin10] helps to lower the modification barrier induced by incomprehensible source code.

The pattern SUPPORT THE COMMUNITY [Wei15] lowers barriers to become an active user. These are not contribution barriers as defined in Section 1.3, but as a consequence, eventually more volunteer developers will join the FLOSS project.

Verification:

A FLOSS project implementing this FLOSS pattern regularly receives source code contributions from developers that had not contributed source code to the FLOSS project before. Developers stayed with the Linux kernel projects for 17 months on average [HNH03], so in order to at least hold a steady number of developers, the Linux kernel projects have to recruit $\frac{1}{17}$ of all current developers every month. More generally, this FLOSS pattern is considered fulfilled if at least 5 % of all active developers in any month were newcomers. Developers are considered newcomers in a given month if they contributed source code to the FLOSS project for their first time in this month.

3.3.2 Preconfigured Build Environment²



Context:

Your FLOSS project is growing already and wants to benefit from Low CONTRIBUTION BARRIERS. Maybe, you BOOTSTRAPPED [Wei10] your development by building on other FLOSS components. The technical structure and the number of dependencies of the FLOSS project require a sophisticated build process that needs substantial configuration effort.

Problem:

Newcomers get stuck when setting up the build environment.

Forces:

- **Disproportionate configuration effort.** As shown in Chapter 2, setting up the development environment is an important modification barrier. The build environment is an essential part of the development environment. Setting up the build environment is not easier if the proposed modification is small. Therefore, the smaller a proposed modification is, the worse is the ratio between required effort to build the modification and the benefit of the modification. Thus, this encumbers especially small patches, but these are especially welcome to FLOSS projects [WND08].
- **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 BOOTSTRAP [Wei10] using other FLOSS projects, 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

²A preliminary version of this section was published previously [WHG13].

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 these changes also changes the build process. The FLOSS project flourishes, but developers that once have mastered the build process and have gotten their computers to build the main component constantly have to keep pace with 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 similar software development projects. They profit from this experience and maybe they can even partly reuse their existing build systems to build the component 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 components 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.** Newcomers 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 component. 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 component your FLOSS project develops.

As the first alternative, you may in fact set up a build environment on a VM and ensure that this build environment builds your component 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 [P*Red15] and MinGW/MSYS [P*Min08]. 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 your FLOSS project's component 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 [P*Fre14], a Microsoft Visual Studio project file [P*Mic16a], or a similar 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.

Consequences:

- ✓ **Lowered contribution barrier for newcomers.** Build VMs ease the build of the component. Newcomers experience lower barriers for their first 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 newcomers.
- ✓ **One package for everything.** The build VM contains all dependencies of the FLOSS project along with the other build tools. Newcomers can immediately build the component without searching for and building dependencies first.
- ✓ **Cross-platform support.** Build VMs provide an additional layer between the build scripts and the developers' 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 cannot build the latest versions of the component [P*Moz09].
- ✗ **VM proliferation [Pri08].** 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 components.** Build VMs compile even complex components

3 FLOSS Pattern Languages

without much configuration effort if they are preconfigured correctly. Newcomers have little trouble to modify these complex components.

- ✓ **Barrier to become a core developers.** While build VMs suffice for smaller source code modifications, they may lack the flexibility needed for larger source code modifications. Newcomers still need to get in touch with the details of the build processes if they want to become developers. Still, build VMs are an intermediate step on the way to a regular developer and make the slope more gentle, 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 [P*Moz151].

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 [P*The16].

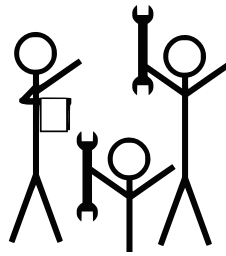
Related Patterns:

A successful build is only the first step to LOW CONTRIBUTION BARRIERS. Newcomers may fail to fix complicated defects or add advanced features at the beginning. These newcomers need LOW-HANGING FRUIT to get started with the development on the FLOSS project. If the FLOSS project has a BAZAAR ARCHITECTURE, joining developers will integrate their modifications into the existing source code more easily – and if the FLOSS project also has an EXPOSED ARCHITECTURE, they will even know where to integrate these modifications.

Verification:

A FLOSS project must provide a download package supporting the build to realize this FLOSS pattern. The download package contains all build tools and dependencies or must at least automatically download all missing dependencies. Except from starting a VM or installation script, the first build must not require more steps than later builds.

3.3.3 Unit Tests for Contributors³



Context:

You maintain a FLOSS project with Low CONTRIBUTION BARRIERS that has grown to a complexity that makes it difficult to foresee all side effects of changes. This applies especially to newcomers, who have not yet acquired insight in the interdependencies between the FLOSS project's modules.

Problem:

Newcomers avoid modifications as they are afraid to introduce new bugs.

Forces:

- **Fixes with hidden defects.** After developers have modified the source code, they want to check whether their modification behaves as intended. For the fix of an easily reproducible defect, this check guarantees that the modification really fixes the defect as it was supposed to. But even in this simple case, the modification may still cause unpredicted side effects that induce new defects in the component.
- **Embarrassing mistakes.** Submitting patches back to the FLOSS project serves as a defect check, because the core developers review submitted patches. Newcomers might be willing to invest their resources for the submission of their patches, but they are afraid to publish erroneous patches, as it is embarrassing for them when others can see their mistakes. This was mentioned as a submission barrier in Section 2.4.3, although only infrequently.
- **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 component, 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 component design quickly in any direction – even if the developers had only been users before they started to work on the new feature.

³A preliminary version of this section was published previously [WHG13].

- **Platform specific bugs.** Developers may intensively test a patched component on their platforms, but this does not ensure that the patches are free of defects. 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.
- **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 experience the malfunction need to put effort in reproducing the malfunction and finding its cause. Even if they can track down the cause to a single patch, they still have to understand the rationale of the patch before they can fix the defect [Tao+12].

Solution:

Provide easy access to unit tests even for newcomers.

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.

Regular tests are best achieved with a CI systems like Jenkins [P*Jen15], Hudson [P*Ora15a], or CruiseControl [P*Cru15]. These systems support different test frameworks, so later addition of automated tests is easy. CI is also available as a service which is often free for small FLOSS projects, for example Travis CI [P*Tra15] and CircleCI [P*Cir15a].

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 [Fow06]. Therefore, newcomers shall also be able to run unit tests. Include the code for unit tests into your PRECONFIGURED BUILD ENVIRONMENT, so newcomers 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:

- ✓ **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.
- ✓ **Confident developers.** As all patches are tested before they are published, their developers are confident that the patches do not contain obvious defects. The patch developers know that even if the patches introduce new defects, 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 component 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 these main components. Thus, interface or architectural changes now also require modifications 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 architectural changes, because the 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 as 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.
- ✓ **Fixes for oneself's defects.** Some defects in the patches are now detected early, so the original developers of the patches may fix these defects before any other developers come into contact with the defect. As the original developers of the patches know the rationale and the structure of the patch, they need much less effort to fix the defects than other developers would have needed.

Known Uses and Sources:

Mozilla tests all changes in their code base automatically [P*Moz15i]. Travis-CI [P*Tra15] and CircleCI [P*Cir15a] provide CI with tests for FLOSS projects hosted on GitHub. FLOSS projects like The Legion of the Bouncy Castle [P*Leg15] include unit tests in their source code packages. Facebook provided unit tests for developers of facebook apps and plugins, like tests for iOS apps [P*Fac15] and for PHP plugins [P*Fac14].

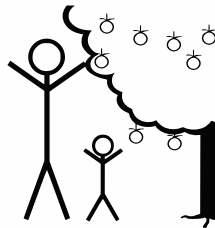
Related Patterns:

Unit tests allows fast and automated tests of the application. This is a precondition for FREQUENT RELEASES [Wei09], as every release requires tests. FREQUENT RELEASES would require much manual effort if no automated tests are in place. Consider more intensive tests for the less frequent, official releases in a PARALLEL DEVELOPMENT [Wei09] strand.

Verification:

To successfully apply this FLOSS pattern, the source code must contain unit tests. The unit tests do not need configuration except for the configuration of the application under test itself. Alternatively, a CI system builds and tests all commits. In this case, it must build and test even pending commits from newcomers.

3.3.4 Low-hanging Fruit⁴



Context:

The FLOSS project grows and gets more and more complex. The core developers grew up with the project's code and know how to handle it. Because of **Low CONTRIBUTION BARRIERS**, newcomers approach the FLOSS project who are yet missing the overview over the FLOSS project's structure.

Problem:

Newcomers cannot work on the FLOSS project's issues because they have too much to learn before they could be helpful.

Forces:

- **No Own Need.** As shown in Section 2.3 and discussed in more detail in Section 2.6.4, about one third of the newcomers approach a FLOSS project without a specific modification for their own in mind. Instead they hope that the FLOSS project offers programming opportunities. They search the FLOSS project for open issues, but they join only if they find issues appropriate for them.
- **Opportunity for beginners.** As evaluated in Section 2.2.4, about one third of all newcomers are students, and Learning is the primary modification motivation for about one fifth of all newcomers, as shown in Sections 2.3. Thus, there is a considerable fraction of newcomers who are not yet professional software developers and use FLOSS projects to improve their programming skills. They want to work on interesting, real issues, but will become frustrated if the solution is too difficult for them.
- **Lack of project-specific knowledge.** The FLOSS project's user base may include experienced developers who would be valuable for the FLOSS project if they joined, but even these experienced developers lack project-specific knowledge. As shown in Section 2.6.2, general SE knowledge or even experience with other FLOSS projects is of little help with contribution barriers.

⁴A preliminary version of this section was published previously [HWG10].

- **Survival of the hardest.** When developers work on the open issues of a project, solving easy issues gives a more immediate sense of achievement and leads 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 newcomers.

Solution:

Mark some issues as easy and leave them open for newcomers.

Experienced developers of the project shall focus on the hard tasks that only they can solve. Leave some of the easy tasks open. Newcomers are more motivated to work on these problems and therefore pass the Onboarding phase [SGR14] more smoothly. Choose the tasks to be left open with care: They should not be so critical that the software quality suffers much until they are fixed, and they should not be so urgent that the core developers are forced to fix them because none of the newcomers have worked on them in time.

You should announce easy tasks left open for newcomers on the FLOSS project's web site. This could be a list of tasks for newcomers or there could be a tag for tasks in the issue tracker that indicates which tasks are suitable for newcomers. Name an experienced developer as mentor for each newcomer task. The mentor will provide help for the newcomer who wants to work on the announced task.

A first code contribution is only the first step in the Onboarding phase. You should encourage newcomers to contribute further. If the source code has insufficient quality, core developers should not take over the issue 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 newcomers the abilities of a developer of the FLOSS project and resolving the issue is only secondary.

Consequences:

- ✓ **Joining script for issue searchers.** Those who try to join a FLOSS project by searching for an issue to work on now experience a lower modification barrier. They can select issues specifically suited for newcomers. Over time, newcomers learn more and more about the FLOSS project and contribute more regularly. In the long term, there will be a higher number of developers with project knowledge and the FLOSS project improves more quickly.
- ✓ **Easier solution.** Highlighting easy issues reduces the modification barrier for creating the Solution as described in Section 2.4.2. Methods to reduce these essential modification barriers are sparse [Bro87].
- ✓ **Less overview required.** It is especially interesting to highlight those issues as appropriate for newcomers that require little knowledge about the general architecture of the FLOSS project and its code structure. This lowers the modification barriers Find the Code and

3 FLOSS Pattern Languages

Understanding the General Structure of the code, which are among the most important contribution barriers, as shown in Sections 2.4.1 and 2.4.2.

- ✗ **Short-term losses.** In the short term, core developers spend more time instructing newcomers than they save by not working on the easy tasks. The component will gain less features and bug fixes temporarily, since there are issues that are easy to implement, but which are left open on purpose. Additionally, some issues seem to be easy, but they are difficult. These issues will be unfinished for a long time, because newcomers fail to solve them.
- ✗ **Lower retention.** Core developers now spend more time on difficult tasks. While this is a better use of their valuable time, they finish tasks only seldom. This can be discouraging.

Known Uses and Sources:

The Google Summer of Code [P*Goo15a] is a program that encourages students to work on FLOSS projects. Every participating FLOSS project clearly defines a work package that can be solved in a three month 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 about the FLOSS project and may stay as developers with the project.

One research thread in End User Programming (EUP) tries to reduce the big leap from a user to a developer into a “gentle slope” [MSH92, p. 382]. Narnmore suggested to “Save some low-hanging fruit for beginners” [Nar10], so new developers of FLOSS projects incrementally learn what is needed to participate in the project. This incremental learning in a gentle slope is a joining script with low contribution barriers for those about one third of newcomers who actively seek issues to be solved, as described in Section 2.6.4.

Related Patterns:

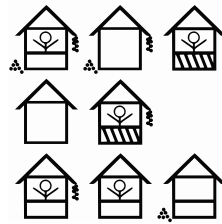
This pattern is one method to reduce modification barriers and therefore achieve Low CONTRIBUTION BARRIERS. BAZAAR ARCHITECTURE can help to encapsulate issues such that they are easier for newcomers. Alternatively or additionally to this FLOSS pattern, you might document an EXPOSED ARCHITECTURE, so your users can figure out how to fix the difficult tasks on their own.

Verification:

The FLOSS pattern is applied if some of the issues in the issue tracker are tagged “good first bug” or with an equivalent phrase. A separate list of tasks for newcomers also fulfills this FLOSS pattern.

3.3.5 Bazaar Architecture⁵

also known as MODULAR ARCHITECTURE [WN13]



Context:

The FLOSS project strives for LOW CONTRIBUTION BARRIERS and more and more newcomers try to extend the component developed by the FLOSS project.

Problem:

It is difficult for newcomers to add innovative features that have not been anticipated and require modifications of the architecture.

Forces:

- **Unsupported code.** Features require support and maintenance that the original programmer of the feature can provide best. But this costs time and effort and some developers do not want to work on the project beyond their initial contribution.
- **Knowledge requirements for newcomers.** Adding features can require changes in many parts of the FLOSS project, but newcomers may not want to dig this deep into the architecture.
- **API for features.** Providing interfaces for feature additions relieves newcomers from learning the details of the whole architecture. They only need to understand the respective interfaces, but the core developers have to invest their time into the development of these interfaces.
- **Modularity.** When the software grows, architectural complexity increases and modularization becomes necessary. A modularized architecture allows having one development team for each module. The effort of communication between the development teams is reduced if the modules are coupled loosely. Small development teams may integrate newcomers more easily than large development teams, but newcomers nevertheless need to understand the FLOSS project's structure for finding the right development team to join. Joining this small development team can still be difficult if these particular core developers happen to be unavailable.

⁵A preliminary version of this section was published previously [HWG11].

- **Module dependencies.** Modules break a complex system into less complex parts. Changes to the system may require only changes to one of the modules. Thus, developers may add features without understanding the whole system, instead they only have to understand and change the module to which the new features belong. This is obviously easier, but other modules may depend on the behavior of the changed module. In this case, new features in the changed module can cause defects in other modules that the developers could not predict just by looking on the changed module alone. Raymond observed that defects occur at the interfaces between modules of different teams [Ray00, chapter How Many Eyeballs Tame Complexity]. New features therefore introduce new defects.

Solution:

Plan the architecture of your software as a flat hierarchy of optional modules.

Modularity is an agreed-upon aim in software development. However, you should strive for more than just modularity. Parnas pointed out that the decomposition of an application and its hierarchical structure are two independent properties [Par72]. It would be contrary to the goals of this pattern to build highly modularized software with a strict hierarchy, where all modules in the top layers of the hierarchy depend on all modules in the lower layers. Such a cathedral architecture [Ray00] is a top-down structure that divides the component into modules, modules into submodules, and so on. A cathedral architecture prevents developers from changing or removing a module in the middle or at the base of a hierarchy without changing the modules situated higher in the hierarchy. Dependencies can prevent the reuse of individual modules of the component in other projects.

In contrast, use a bazaar architecture that strives for a flat hierarchy with all or most modules on the same level. Design your component to have many interfaces, but keep them optional. This way, your component can be extended in all directions, but only a small number of modules are required to run the component. Design the interfaces such that newcomers can add modules with new features. Try to encapsulate the required core functionality in modules that newcomers will not change, so new features do not collide with changes made by the core developers or other newcomers.

One method to allow easy extension of an application with optional modules are add-on and plug-in APIs. Users need only the core modules to use the application and choose the functionality they need by installing additional plug-ins. Developers have a clearly defined interface to extend the application. This is especially useful when the FLOSS project provides a central market for developers to offer their add-ons and for users to search for these add-ons and rate them. A market is not necessarily a bazaar, though, and the full potential of the bazaar development style is only achieved if also the core application is modularized and loosely coupled.

The architecture of a component mirrors the structure of the organization that produced it [Con68]. Changing the component's architecture to a bazaar therefore requires the organization to become a bazaar. However, a bazaar project structure does not need to be and cannot be enforced, because freedom to leave or join the project is the key element of the bazaar project structure. Instead, you can only foster the bazaar project structure with low contribution barriers like a bazaar architecture.

Consequences:

- ✓ **Dynamic organization.** The FLOSS project will now be a bazaar organization. A bazaar organization can quickly adapt to changes in the staff, in other words the project contributors. Even if the original developers of a module leave the project, the software as a whole will still be maintainable: The program requires only a few core modules, all others are optional. Furthermore, a new module replaces a neglected module without changes to other modules if the new module adheres to the same interfaces as the neglected module.
- ✓ **Newcomer contributions.** Developers have an easy time adding features and joining the FLOSS project, since they only have to learn about the interfaces they are going to use for their new features. If any changes to the existing modules are necessary, they are limited to the few modules whose interfaces the new feature uses.
- ✗ **Development costs for interfaces.** Every change may be implemented directly and some changes may be encapsulated by an interface. Defining a new interface requires more effort than changing it directly, but using an existing interface requires less effort. Modularity therefore comes at the price of increased development time in the short term. Modularity may not pay off its overhead.
- ✓ **Dynamic architecture.** A newly developed module can reuse the modules that already exist. Reusing existing modules speeds up development. If the modules are reused often, modularity can lead to saving more development time than its initial architectural overhead costs.
- ✓ **Exchangeable modules.** The flat hierarchy reduces the number of dependencies, so it is easier to replace or change a module.

Known Uses and Sources:

Plug-in APIs are common among commercial and even more in FLOSS projects. Microsoft Outlook [P*Mic16b], Mozilla Firefox [P*Moz15c], and MediaWiki [P*Med16] are some of the many examples.

Some projects put even the basic functions into plug-ins and reduce the main program to the smallest core possible. This allows other plug-ins to have the same degree of functionality that the components of the standard program have. Examples are Trac [P*tra15] with its component architecture and the Jenkins CI system with its build steps and source control interfaces implemented as plug-ins [P*Jen16].

In the UNIX operating system and its derivatives, there is one tool for each task and these tools are different from each other. The tools use text streams to communicate with each other. It is therefore easy to replace, add, or remove tools from the system while the system stays operable. [MPT78]

Raymond [Ray00] described a development method he coined as bazaar development style as opposed to the classical cathedral style. He indicated that the basic principles of the bazaar development style are the “UNIX gospel” of small tools with defined purpose and the additional

principles Torvalds introduced with the Linux kernel project. Only these additional principles enable FLOSS projects to use the bazaar style even if the software size exceeds a certain threshold. BAZAAR ARCHITECTURE describes the modularity aspect, the “UNIX gospel”, and not the bazaar development style as a whole.

Conway’s Law also implies that a large and scattered developer community is only possible with a highly modular approach [Con68]. Fink [Fin03, Chapter 9] describes an organizational structure to support the bazaar development style. On the basis of Fink’s description, Figure 3.12 sketches a possible realization of such an organizational structure. The Primary Architect keeps the FLOSS project’s vision and takes the technical lead. The Release Architect is on standby to take over the role of the Primary Architect when the Primary Architect is not available. The Subsystem Architects constitute a hierarchy of which the uppermost layer is appointed by Primary and Release Architect. They have clear responsibilities, each for a module or submodule of the software. The Engineering Talents are the contributors of the FLOSS project. Newcomers should optimally interact with only one Subsystem Architect, as their contributions needs modifications to only one module.

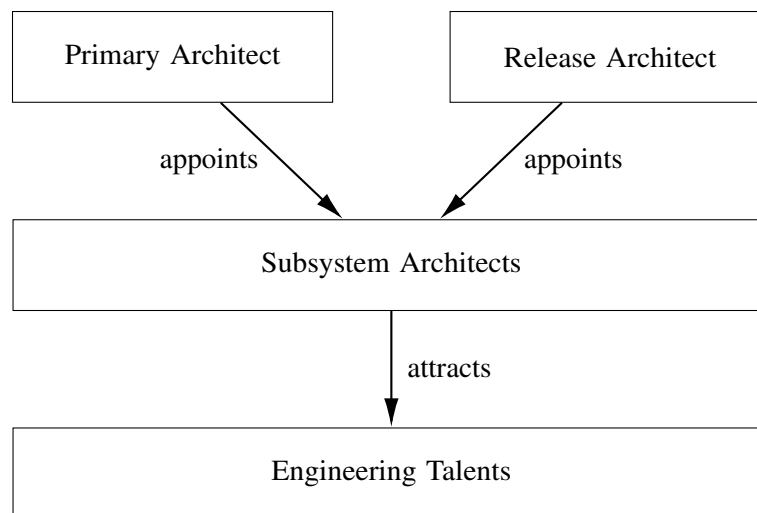


Figure 3.12: Project structure of a development team using a BAZAAR ARCHITECTURE, see [Fin03, Chapter 9]

Respondents to the question in Section 2.4.2 rated Find the Code as the most severe contribution barrier. In some cases, respondents may not have found the right location to modify the code because too many modules had to be modified. This can happen if the code was improperly structured, a problem that the BAZAAR ARCHITECTURE solves.

Related Patterns:

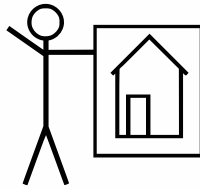
Very small projects in terms of lines of code do not need to be modularized, as there are not so many places to modify the code when adding features. Thus, the FLOSS project must have been growing and therefore already have achieved LOW CONTRIBUTION BARRIERS through other FLOSS patterns before the problem characterized in this BAZAAR ARCHITECTURE arises.

The developers might then have two problems when adding new features: First, there must be a place to add the new code. Second, the developers must find it. Only the first problem is tackled by the *Bazaar Architecture*, while an *Exposed Architecture* resolves the second one. Depending on the type of project and type of feature, developers may actually have only one of the two problems. If the software has a plug-in API, developers of a plug-in do not need a deep understanding of the overall architecture, instead they use only the plug-in API, defined in a *Plug-In Contract* [Mar99].

Verification:

One requirement for this pattern is the existence of multiple modules in the project. At least 75 % of its modules must have less than 50 source code files, excluding build scripts and tests. At least 75 % of commits must modify the files of only one module.

3.3.6 Exposed Architecture⁶



Context:

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

Problem:

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

Forces:

- **Understanding the General Structure.** Newcomers need some starting points to add features, but they have trouble finding the module where this feature 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 feature. Sections 2.4.1 and 2.4.2 show that Understanding the General Structure and Find the Code are important modification barriers.
- **Mentoring.** Core developers can guide newcomers and explain the code structure to them, but they need time for this mentoring. If they are not available or if newcomers do not request their help, for example because they are too shy, the joining process gets stuck and may fail eventually.
- **Implicit architecture.** 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.

⁶A preliminary version of this section was published previously [HWG10].

- **Code decay.** Even if newcomers find a way to add a feature, it may not be at the optimal place in the code because of their lack of overview. The software may be working, but software quality decreases over time, as extensions and fixes will be inserted in suboptimal places [Eic+01].
- **Conway’s Law.** The organizational structure of the project team is a representation of their software’s architecture [Con68]. Small project teams reorganize themselves and their software’s architecture quickly when the need arises. Occasional contributors may understand the architecture at some point, but this knowledge may quickly become obsolete.

Solution:

Publish the component’s architecture.

You can publish the architecture in different places, illustrated by Figure 3.13: In a dedicated document, in a well documented source code, or in the FLOSS project’s structure. Choose one of those places and publish this decision. Of course, combinations are also possible and sometimes advisable.

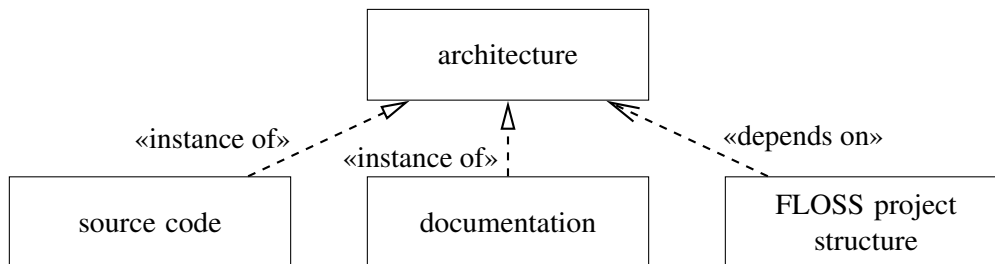


Figure 3.13: Relation between architecture, source code, documentation, and FLOSS project structure

Dedicated documents for the software’s architecture is the best way for outsiders to understand the structure and dependencies of the project. 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 code.

First, literate programming advocates source code with extensive explanation, such that it can be read like a book and is easy to understand by itself [Knu84]. Tools like JavaDoc [P*Ora15b] and Sandcastle [P*EWS15] create code documentation from source code and its comments. This also requires extensive use of sensible comments, as there is no use to method comments that do not say more than the method’s and parameters’ names already did.

Second, another approach known as Clean Code vouches to avoid comments at all and still have self-describing code [Mar08]. An important aspect of this are variable and function names that accurately explain the purpose of the named variable or function.

3 FLOSS Pattern Languages

Lay your FLOSS project structure open. Show who is responsible for which part of the software. Publish the information required to allow newcomers to overview the open tasks for each module and contact the module maintainers.

Consequences:

- ✓ **Lowered modification barrier.** Newcomers will find the place where to change code more easily. This reduces an important modification barrier and so the number of developers increases.
- ✓ **Less mentoring effort.** These new developers use less time searching the code and discussing with the core developers 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.
- ✓ **Reflections on architecture.** Developers are aware of the architecture and can potentially improve the architecture.
- ✓ **Ongoing code decay.** Understanding the architecture is key to find the best place to add new code. Good architecture documentation helps with this task and therefore code decay is reduced. However, the current architecture may just be unsuitable for the intended modification. In this case, a newcomer may use the documentation to find the optimal place to add new code, but this may still not be a good place. Newcomers may dismiss their contributions although they have understood the project and software architecture. This risk exists whether or not the documentation has been published.
- ✗ **Documentation maintenance.** Time spent on architecture cannot be used to develop new features and a formal structure might also suppress creativity. Nevertheless, it is crucial that the architecture documentation is up-to-date, as all benefits are gone if the documentation does not match the actual architecture of the components.

Known Uses and Sources:

The UML modeling tool ArgoUML [P*Arg09a] documents its own architecture as UML diagrams [P*Arg09b]. Also, they document their processes like issue handling in UML diagrams [P*Arg09c].

Gacek and Arief [GA04] have found out that some FLOSS projects have exposed architectures while others have closed architectures. They call this property of a FLOSS project “Visibility of software architecture”. An architecture can be kept secret intentionally, which may be another FLOSS pattern. In this case, there are some forces that want to discourage others from joining the project. More likely, the architecture is hidden unintentionally 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 can be written down in some formal way like UML. The source code also 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 [Con68].

Related Patterns:

An EXPOSED ARCHITECTURE reduces an important modification barrier and, thus, helps to achieve LOW CONTRIBUTION BARRIERS. As discussed in the consequences, the component of a FLOSS project with an EXPOSED ARCHITECTURE may lack interfaces to encapsulate newcomers' modifications, so finding a good place to add a modification may still be a modification barrier. A BAZAAR ARCHITECTURE is a complementing FLOSS pattern to solve this problem.

It is not only sufficient to have an EXPOSED ARCHITECTURE, it must also be clear to all newcomers that it exists. Therefore, it is a good idea to provide a CENTRAL INFORMATION PLATFORM [HWG11] and promote the EXPOSED ARCHITECTURE there.

In some cases, a PLUG-IN CONTRACT [Mar99] can be an alternative to an EXPOSED ARCHITECTURE. It does not help with changes to the core modules, but also allows addition of new features.

Verification:

The project's documentation must describe the application's architecture to realize this pattern. Multiple architecture diagrams or extensive textual descriptions both satisfy this condition.

3.4 Evaluation

In its section Known Uses and Sources, each FLOSS pattern lists examples of FLOSS projects that apply the FLOSS pattern. The examples prove that the FLOSS patterns are not just theoretical constructs, but that they can be applied successfully in practice. Furthermore, examples help readers to gain an understanding of how to implement the FLOSS pattern in practice. Although examples are an important part of a FLOSS pattern, they do not show whether a FLOSS pattern is a universal common practice in FLOSS projects or whether it is a rare special case. Much less, there is no empirical data showing which FLOSS patterns are crucial for the success of a FLOSS project and which FLOSS patterns are just nice-to-haves or only applicable in special situations.

This section contributes to close this research gap with an empirical study on five FLOSS projects. The verification part of the pattern thumbnails in the previous sections described objective criteria with which the application of each FLOSS pattern in each FLOSS project will be evaluated in this analysis. This allows reproduction of the results on the selected and other FLOSS projects.

This section documents the verifications of FLOSS patterns on the five selected FLOSS projects. The documentation serves three purposes. First, readers may reproduce the study on the selected FLOSS projects and evaluate the methodology. Second, the section is also an example for the application of the FLOSS patterns' verification sections and help illustrate their usage. Third, patterns generally help to understand the complex structures, so this analysis based on FLOSS patterns gives insight into the inner workings of the five selected FLOSS projects and serves as case studies for the mechanics of FLOSS projects.

The five selected FLOSS projects are Firefox, Rack, Ruby on Rails, Spree, and Rake. The analysis deliberately used FLOSS projects of different kinds to get a more representative picture of the FLOSS ecosystem. Firefox is an application for end users and is written in various programming languages, while the other four target developers and use Ruby. Although they target developers, Rake is an application, while Rack, Ruby on Rails, and Spree are libraries or frameworks. In terms of the total number of commits, Rack and Rake are one order of magnitude smaller than Spree and Ruby on Rails, while Firefox is another order of magnitude larger. Spree is backed by a commercial company that employs most of its developers, while Ruby on Rails is more a community project with less direct involvement by a single company. However, all selected FLOSS projects have in common that they are successful. Success of FLOSS projects may have many meanings [CAH03], but no measure is equally applicable to all FLOSS projects, especially if they are so different as the ones selected for this analysis. Therefore, the analysis omits a precise and complex, but also unnecessary definition of what success means and instead substantiates each FLOSS project's success individually with evidence in the introduction of the respective FLOSS project's subsection. For example, the FLOSS projects all serve a considerable fraction of users in their application domain.

Successful FLOSS projects more likely chose their applied FLOSS patterns well. For example, if the selected FLOSS projects all disregarded a specific FLOSS pattern, this FLOSS pattern is obviously not necessary for the success of FLOSS projects. Conclusions in the other direction have to be drawn with more caution: A universally applied FLOSS pattern may have helped these FLOSS projects to succeed, but maybe it just did not do any harm.

As a secondary goal, the evaluation shows which patterns are commonly used together and

which patterns exclude each other. Thus, the evaluation also puts the relationships between FLOSS patterns described in the previous sections to the test.

The evaluation has to be taken with a grain of salt, though: Due to the effort necessary to evaluate all FLOSS patterns on a FLOSS project, the data do not suffice for a proper statistical treatment of all previously described goals. The results are therefore merely pointers to the importance of and relationships between FLOSS patterns and no irrevocable facts. Nevertheless, this study provides a first empirical foundation for the FLOSS pattern language.

The analysis of some of the FLOSS projects and FLOSS patterns took data from the projects' VCSs into account. I developed the tool `GitHistoryAnalyzer` [Han15a] to analyze the VCS history of some of the FLOSS projects and measure data used for the evaluation of the FLOSS patterns' verification criteria where applicable. `GitHistoryAnalyzer` is published as FLOSS itself and can be downloaded to understand or reproduce the parts of the verification where it was applied.

3.4.1 Mozilla Firefox

The company Netscape published their commercial product Communicator as the FLOSS project Mozilla in 1998 [P*Net98]. Mozilla runs a suite of different software products, one of which is the browser Firefox [P*Moz15m]. Most products, including Firefox, support different development and target platforms like Linux, Mac OS, Windows, and platforms for mobile devices [P*Moz15d].

Firefox and Mozilla have played an important role in the development of browsers. Firefox itself [Mas+09; Nur+09; AM11; Bay+12] as well as Mozilla [MFH02; RM02; Yin+04; CC06; JS07; ZM12] have been the subject of research of different kinds. For example their development processes and tools have been in the focus of research, and they served as data sources to answer general software engineering questions.

Integrate FLOSS Assets

Firefox depends on the compression library `zlib` [P*RA15] as a module [P*Moz14b], among other third-party modules. Firefox directly includes the source code, as it supports multiple platforms. `zlib` as used in Firefox's source code as of 2014-07-07 [P*Fir14] comprises 12 998 LOC, which proves that Firefox fulfills the criterion for `BOOTSTRAP`.

Mozilla requires its developers to follow the Mozilla Foundation License Policy [P*Moz15p] to ensure all of its projects `RESPECT THE LICENSE` of third-party components. However, for the reasons explained in the Verification section of `RESPECT THE LICENSE`, this result will not be used in further analysis.

Start a New Project from Scratch

On its initial publication under the name Phoenix 0.1, Firefox already included a browser's core and additional functionality like a Plug-in Manager. Firefox was split from the main Mozilla Suite and therefore the developers could provide this functionality from the start. Thus, Firefox showed a `CREDIBLE PROMISE`. The first version included the source code already, so `MARKET EVALUATION` was also fulfilled. [P*Moz02]

Publish Closed Sources

Although the Mozilla Suite is based on the proprietary Netscape browser product and applies DONATE CODE, Firefox itself or a similar product was not released using a closed source license before its release under a FLOSS license. Thus, DONATE CODE does not directly apply to the Firefox project.

Licensing

Firefox comprises code using different licenses. The Firefox binaries are distributed using the APPROVED OPEN SOURCE LICENSE Mozilla Public License (MPL) [P*Moz15s]. The MPL requires any modifications to be distributed as source code instead of only binaries [P*Moz15s, §3.1f]. Therefore, the MPL is a RECIPROCAL LICENSE and not a PERMISSIVE LICENSE, as sometimes falsely claimed [GA04]. The MPL enforces only a weak copyleft, though, as MPL code can be distributed together with closed source code. Some components of Firefox use other licenses than the MPL. There are two reasons why this does not suffice for the DUAL LICENSE pattern: First, common secondary licenses in the code base are the GPL, the X11 License [P*Fre12], and code snippets in public domain. None of these is a commercial license and requires users to pay royalties as demanded in the DUAL LICENSE verification. Secondly, the product as a whole is only distributed using the MPL.

Architecture

Firefox used to have submodules, but this submodule structure was discarded to foster self-organized work over submodule boundaries [P*Sha11a; P*Sha11b]. Hence, Firefox has no BAZAAR ARCHITECTURE.

Firefox does not have an EXPOSED ARCHITECTURE: While there is documentation about program internals in the Mozilla Developer Network (MDN) [P*Moz15h], it lacks a clear description of the software architecture. The search terms “architecture”, “firefox architecture”, and “firefox overview” in the MDN search reveal no architectural documents about Firefox in the first 30 hits. Mozilla’s wiki contains a list of modules, but with only a few keywords describing each module and no explanation of the relations between the modules [P*Moz15x]. The Firefox developer introduction [P*Moz15e] also does not hint at the structure of the code. Grosskurth and Godfrey attempted to extract at least the architecture of the closely related Mozilla Browser project [GG05]. A common suggestion is to accumulate information about interesting parts of the code from diverse sources like the code commits, IRC channels, and the issue tracker [P*Mat10].

Firefox implements MANAGE COMPLEMENTS with a JavaScript API for add-ons [P*Moz15c]. Additionally, Firefox provides a C API for plugins, which have a different application scope than add-ons [P*Moz15j].

Mozilla Foundation as the host of the Firefox project is a non-profit organization and as such does not aim at maximizing its profit [P*Moz15v]. Although Firefox does MANAGE COMPLEMENTS with its add-on API as shown earlier, the management mechanism does not facilitate paid add-ons. Thus, Mozilla does not SELL COMPLEMENTS for Firefox. Consequently, Firefox also does not have a DUAL PRODUCT.

Cultivate a User Community

In order to find out whether the Firefox project treats their users as CROWD OF BUG DETECTORS, the response time to issue reports must be measured. Even core developers need to file issue reports for changes [P*Moz15f], so the response time to issue reports from core developers must be compared to the response time for other issue reports. The study sample includes all issue reports opened in January and February 2014. In this period, the owner and peers of the Firefox module had not changed [P*Moz14e]. Owner and peers were considered as core developers and all others as users. The sample data stem from a database snapshot of the Bugzilla database from 2014-05-04 [P*Moz14c] to measure response times. The query in Listing 3.1 shows response times to issues open by core developers, while the query in Listing 3.2 shows response times to issues open by users. The table `ff_coredevelopers` was created before manually and contains the database ids of the core developers previously identified.

Out of 250 issues that core developers had reported in the sample time frame, 52 issue reports weren't answered at all. That results in an answer rate of 79.2 %. Others had reported 2267 issues, out of which 1825 were answered, so 442 were left unanswered. This is an answer rate of 80.5 % and therefore slightly higher than the answer rate to core developers. The arithmetic mean of time till the first answer for issues reported by core developer is 6.77 days in the sample. In the same sample, the arithmetic mean of the time till the first answer for issues reported by others is 6.74 days. Issues reported by users are therefore answered very slightly earlier, about half an hour earlier. A Wilcoxon rank sum test reveals that this difference is only slightly statistically significant ($p = 0.096$). Although based on weak evidence, this analysis considers Firefox to fulfill the formal requirements of a CROWD OF BUG DETECTORS.

Neither Mozilla [P*Moz15t] nor Firefox [P*Moz15m] visualize information measured by third parties on their website. However, GitHub measures commit data on the gecko repository that is mirrored to GitHub [P*Moz15u]. GitHub is independent from Mozilla and commit data is non-obvious, but it is only indirectly related to quality and most importantly, Mozilla and Firefox do not advertise these measurements. Thus, Firefox does not implement PUBLIC INSPECTION.

The “Nightly Builds” of Firefox contain all changes that have been merged into the main branch mozilla-central until the time of build [P*Moz13, Section Release timeline]. Thus, Firefox has FREQUENT RELEASES.

Listing 3.1: SQL query for response times to core developer issues

```
SELECT bugs.bug_id, MIN(bugs.creation_ts) AS datBugOpened,
      MIN(longdescs.bug_when) AS datFirstComment
FROM ff_coredevelopers /* for core developers */
LEFT JOIN bugs ON ff_coredevelopers.id_profile = bugs.reporter
LEFT JOIN bugs.longdescs ON bugs.bug_id=longdescs.bug_id
      AND bugs.reporter <> longdescs.who /* skip self-answers */
WHERE product_id=21 /* 21 is Firefox */
      AND creation_ts >= '2014-01-01' AND creation_ts < '2014-03-01'
GROUP BY bugs.bug_id;
```

Listing 3.2: SQL query for response times to issues from users

```
SELECT bugs.bug_id , MIN(bugs.creation_ts) AS datBugOpened ,  
      MIN(longdescs.bug_when) AS datFirstComment  
FROM bugs  
LEFT JOIN bugs.longdescs ON bugs.bug_id=longdescs.bug_id  
      AND bugs.reporter <> longdescs.who /* skip self-answers */  
WHERE product_id=21 /* 21 is Firefox */  
      AND creation_ts >= '2014-01-01' AND creation_ts < '2014-03-01'  
      AND bugs.reporter NOT IN /* all except core developers */  
      (SELECT id_profile FROM ff_coredevelopers)  
GROUP BY bugs.bug_id;
```

Beside the main branch for Firefox development, mozilla-central [P*Fir15], there are multiple branches called integration repositories that merge into mozilla-central if they are stable [P*Moz14g]. There are rare exceptions for direct commits to mozilla-central [P*Moz15aa, Section mozilla-central], but these are within the scope of a PARALLEL DEVELOPMENT.

Mozilla uses a CENTRAL INFORMATION PLATFORM: The main Mozilla web page [P*Moz15t] links to the Firefox web pages for users, the MDN portal, and other subsites. Within each subsite, a portal like the MDN [P*Moz15h] serves as CENTRAL INFORMATION PLATFORM for the subsite. Each page links back to the main Mozilla web page and its corresponding subsite portal.

Discussions on development topics are open to all interested parties and use public media like Usenet groups [P*Moz06]. Their issue tracker Bugzilla contains lower level details regarding the software implementation [P*Moz01]. Firefox therefore conducts an OPEN DIALOG with its community.

SUPPORT THE COMMUNITY has five requirements, four of which have already been checked: Firefox is a FLOSS project and it implements the three patterns APPROVED OPEN SOURCE LICENSE, CENTRAL INFORMATION PLATFORM, and OPEN DIALOG. The data for CROWD OF BUG DETECTORS described above showed that about 80 % of issue reports were answered, which suffices the last requirement, especially as Mozilla does not offer premium support at all and therefore is below the allowed marketing threshold.

Tools for New Developers

Data from February 2013 serve as the sample to test the pattern LOW CONTRIBUTION BARRIERS. In February 2013, 333 developers with distinct names authored commits to Firefox's main hg repository mozilla-central [P*Fir15]. This number is an upper limit, as some developers authored multiple patches using different names. A manual check of the names revealed 6 cases of duplicate names, but there may be more. 41 of these developers, excluding 3 duplicates, had never authored a commit to mozilla-central before February 2013 and can be considered newcomers. 41 is more than 5 % of 327, therefore Firefox implements LOW CONTRIBUTION BARRIERS. Notably, 14 of these newcomers had mail addresses from Mozilla Foundation and have been hired instead of acquired as volunteers, but $41 - 14 = 27$ is still more than 5 % of 327.

Firefox can be build for different platforms and on different platforms. Supported platforms include like Linux, Windows, and Max OS X. Furthermore, Firefox shares a build system with other Mozilla products like Thunderbird and SeaMonkey. These factors complicate the build of Firefox. Mozilla tries to counter these build problems through measures like extensive documentation of the build process. [P*Moz15d]

Another measure is a set of PRECONFIGURED BUILD ENVIRONMENTS: For Linux and Max OS X, a python script downloads all build tools and dependencies [P*Moz15g]. For Windows, Mozilla provides the MozillaBuild package [P*Moz15l]. MozillaBuild contains the required tools of the build chain in a VM. Builds on other, less common platforms like Solaris and OS/2 are more difficult, and no PRECONFIGURED BUILD ENVIRONMENT is available, but this is negligible for most contributors. Despite the PRECONFIGURED BUILD ENVIRONMENTS, builds can still be complicated, which is illustrated by the help requests on build errors in the newsgroup for builds [P*Moz08a].

There are multiple ways to test a patch for Firefox. Mozilla offers CI systems for building and testing patches submitted to so-called Try Servers, but this requires commit access to Mozilla's VCS [P*Moz15z]. This method is therefore restricted to core developers. The publicly available source code includes test code [P*Moz15i], though, and the PRECONFIGURED BUILD ENVIRONMENTS help execute those tests. Due to the high number of different test suites, running all tests requires effort and computing time [P*Moz15k]. However, this effort is justified, given the large size of the Firefox project and its dependent projects in Mozilla. Thus, these tests can be considered UNIT TESTS FOR CONTRIBUTORS.

All Mozilla projects like Firefox use the same Bugzilla instance [P*Moz01] as their issue tracker. Some tasks that are suitable for beginners are tagged as “[good first bug]” [P*Moz15w]. This verifies the application of the LOW-HANGING FRUIT pattern.

Empower the Most Suitable

The Mozilla Foundation maintains Firefox. This is a foundation, so POOL RESOURCES was implemented. However, search engines fund the Mozilla Foundation and, for example in 2012, they account for almost 98 % of revenues of the Mozilla Foundation [P*Hoo13]. The Mozilla Foundation is nevertheless independent from specific search engine providers.

The Mozilla project is divided into modules, one of which is Firefox. Modules usually have exactly one owner. This is also the case for Firefox, whose owner was Gavin Sharp as of 2015-05-30 [P*Moz15y]. Module owners have the authority to decide about acceptance of patches, but they do not have absolute authority: Their own patches must be reviewed by a peer and in case of disagreements with parts of the community, a committee may enforce a decision against the module owner or even install a new module owner. The Firefox module owner therefore is no SINGLE MAINTAINER. [P*Moz15q]

As explained, the module owner of Firefox relies on approval from the Mozilla project. From a Firefox perspective, Mozilla RUNS A TIGHT SHIP on Firefox. However, Mozilla itself is a MERITOCRACY [P*Moz15n]. Since Firefox contributors bring merit also to Mozilla and therefore gain power over Mozilla, Firefox itself can also be seen as a MERITOCRACY.

There have been multiple MAINTAINER HANDOVERS in the history of the Firefox project, which can be considered smooth transitions. This analysis only looks at the last two of these occurrences before 2015-05-30. Mike Shaver took over maintenance from Mike Connor in a smooth

3 FLOSS Pattern Languages

transition [P*Con10]. Later, when Gavin Sharp took over module ownership over Firefox from Mike Shaver, this was a MAINTAINER HANDOVER with a smooth transition again [P*Nig11].

3.4.2 Rack

Rack is a middleware between web servers and Ruby frameworks like Ruby on Rails [P*Neu13]. Ruby on Rails [P*Rub15a] indeed depends on Rack and so does its competitor Sinatra [P*Sin15]. Thus, Rack has served websites for hundreds of millions users.

Integrate FLOSS Assets

Rack has no runtime dependencies, but two build dependencies, Bacon and Rake [P*Rub15m]. However, both Bacon's and Rack's main author is Christian Neukirchen [P*Rub12; P*Rub15m]. Thus, Neukirchen could have used Bacon for Rack even if it was no FLOSS component. Rake [P*Rak15b] is part of the common Ruby development toolchain and therefore not to be considered as a separate FLOSS component to build on. Thus, Rack does not BOOTSTRAP its development.

Start a New Project from Scratch

Rack was published with an initial git commit [P*Neu07] on 2007-02-15, fulfilling the criterion of MARKET EVALUATION. This commit of 445 LOC included tests for a basic Hypertext Transfer Protocol (HTTP) request. As this is a core feature, Rack showed a CREDIBLE PROMISE.

Publish Closed Sources

Since Rack was not used in a commercial setting before its initial publication, it does not use the pattern DONATE CODE.

Licensing

The license of Rack [P*Rac15c] is the X11 License [P*Fre12]. This is an APPROVED OPEN SOURCE LICENSE and a PERMISSIVE LICENSE. Vice versa, Rack does not implement DUAL LICENSE and RECIPROCAL LICENSE.

Architecture

The main component of Rack [P*Rac15g] has no submodules. However, Rack MANAGES COMPLEMENTS with a repository on its own for add-ons [P*Rac15a]. The repository contains 30 add-ons as of 2015-11-17, all of which together have 37 files of source code. Thus, Rack has a BAZAAR ARCHITECTURE. Add-ons are available for free under the X11 License and no paid complements are advertised, so Rack neither uses the pattern SELL COMPLEMENTS nor DUAL PRODUCT.

The documentation describes the Rack interface [P*Rac15f] and Rack's classes [P*Rac15d]. There is also a wiki [P*Rac15b]. However, there is no high-level description of Rack's architecture. Thus, Rack has no EXPOSED ARCHITECTURE.

Cultivate a User Community

Rack's issue tracker [P*Git15d] contains 974 issues including pull requests as of 2015-11-17, but there are 2109 commits to the repository [P*Rac15g]. Thus, some developers may commit without corresponding issue. This means that the criterion for CROWD OF BUG DETECTORS is whether answers to issues take less than 24 hours.

The sample for this criterion includes all issues excluding pull requests created between 2015-06 and 2015-10. Of these 26 issues, 6 were closed by their authors within 24 hours and therefore it is unclear whether Rack's developers would have responded quickly enough. Of the remaining 20 issues, 14 received no answer within 24 hours and in the majority of cases, they received no answer for weeks. The other 6 received a reply within 24 hours. This is not enough for a CROWD OF BUG DETECTORS.

GitHub publishes code metrics of Rack's source code on Rack's GitHub site [P*Rac15g]. Furthermore, Rack uses Travis CI to build and test the component and publishes the current build status on its main page [P*Rac15g]. This fulfills the conditions for PUBLIC INSPECTION.

A RubyGem is a packaged Ruby component. Thus, published RubyGems count as release. There have been 61 RubyGem releases of Rack between 2007-03 and 2015-10 [P*Rub15h]. Only 11 RubyGems were released before 2011 and 50 of the RubyGems were released after 2011-01. However, often multiple RubyGems for Rack were released on the same day, because multiple branches had received the same patch or because the release introduced a problem that had to be fixed immediately. The intention of FREQUENT RELEASES is that users may always use a very recent version of the component and quickly see their changes in the released software, which is not the case if releases come in infrequent bulks. Thus, only releases on different days count for this analysis, which reduces the number of releases even in the more frequent, second phase to 37 releases between 2011-01 and 2015-10. This corresponds to an arithmetic mean of 47.7 days between two releases and this is too long for FREQUENT RELEASES.

Rack has multiple git branches [P*Git15a]. The main branch contains the latest development version, while there is one other branch for each older release of Rack. Anybody can fork Rack's git repository for modifications and then create a pull request to merge these modifications back to the official Rack repository. Thus, Rack uses PARALLEL DEVELOPMENT.

Rack's website [P*Neu13] links to the documentation, the communication channels, and the GitHub repository. The GitHub repository's main page [P*Rac15g] links to the same resources and also back to the Rack Website, although links to the documentation are less visible and less complete. Both are candidates for a CENTRAL INFORMATION PLATFORM. However, depending on the method to view the documentation, there are no links back to any of the two sites [P*Rac15f] or only to the GitHub repository [P*Rac15e]. Thus, none of the candidates is a real CENTRAL INFORMATION PLATFORM.

There is explicit documentation on how to contribute add-ons [P*Pal15]. The website of the Rack project [P*Neu13] and Rack's GitHub page [P*Rac15g] both refer to the developer mailing list [P*Goo15b], which publicly discusses technical decisions. The public issue tracker [P*Git15d] also contains discussions regarding decisions for Rack. Nevertheless, the GitHub page explains that there is also a mailing list for the core developers and this mailing list is not public. Also on the GitHub page, they request to send reports about security defects to this mailing list, so attackers may not exploit the defect before there is a fix available. This is

an acceptable exception and Rack can still be considered to conduct an OPEN DIALOG with their community.

There is no user mailing list and so support requests are posted to the developer mailing list [P*Goo15b] or the issue tracker [P*Git15d]. The previous analysis showed that issues in the issue tracker often receive no timely response. For the developer mailing list, this evaluation used all nine mail threads created between 2015-01 and 2015-10. Four mail threads were responded to within 24 hours, but one was a patch and not a support request. The remaining five received answers only after more than 72 hours if at all. For some of these, the first answer took weeks or months. Only 37.5 % of the support requests received an answer within 72 hours. The responsiveness of neither the issue tracker nor the mailing list suffices for SUPPORT THE COMMUNITY. Additionally, the required pattern CENTRAL INFORMATION PLATFORM is not implemented for Rack.

Tools for New Developers

Table 3.1 contains for each month from 2014-01 to 2015-10 in the lower data row the number of newcomers to Rack and in the upper data row the number of developers who have been active in the respective month. The data came from Rack’s main git repository [P*Rac15g] and have been processed with GitHistoryAnalyzer [Han15a]. More than 5 % of the active developers have been newcomers in all months, thus Rack has LOW CONTRIBUTION BARRIERS.

Table 3.1: Number of active developers per month versus number of newcomers per month for Rack

	2014-												2015-											
Number of	01	02	03	04	05	06	07	08	09	10	11	12	01	02	03	04	05	06	07	08	09	10		
developers	11	6	3	4	3	9	13	9	3	5	11	4	7	13	15	8	11	17	4	9	7	5		
newcomers	5	4	2	2	2	7	2	3	1	3	4	1	5	9	6	5	4	6	1	4	5	2		

Rack’s source code [P*Rac15g] includes a gemspec file, so the Ruby tool gem installs Rack and all of its dependencies. The code also includes a simple web server and web application to run Rack without any configuration. This is described in the documentation on the main GitHub page [P*Rac15g, Quick start] and counts as a PRECONFIGURED BUILD ENVIRONMENT.

Rack also has UNIT TESTS FOR CONTRIBUTORS: First, the source code includes unit tests executed with a Rake [P*Rak15a] task. Second, Rack uses Travis CI [P*Tra15] to build and test commits to Rack.

The issue tracker [P*Git15d] uses neither tags nor titles to designate issues suitable for newcomers. Documentation on the contribution procedures [P*Pal15] and other documentation [P*Rac15e; P*Rac15d; P*Rac15b; P*Rac15g; P*Rac15a] do not hint at easy issues for newcomers. Thus, Rack does not offer LOW-HANGING FRUIT for newcomers.

Empower the Most Suitable

Christian Neukirchen was the original author of Rack in 2007 [P*Neu07] and its first SINGLE MAINTAINER. At the end of 2011, there was a MAINTAINER HANDOVER to James Tucker with

a troublesome start [P*Rac12] (complicated transition). On 2014-08-18, there was another MAINTAINER HANDOVER when Aaron Patterson took over as SINGLE MAINTAINER [P*Tuc14] (smooth transition).

Conversely, Rack is no MERITOCRACY and there is no organization to RUN A TIGHT SHIP. There is also no indication that organizations with commercial interest POOL RESOURCES to develop Rack.

3.4.3 Ruby on Rails

Ruby on Rails is a framework for web applications for the programming language Ruby. Among all FLOSS project on GitHub using Ruby, it has the highest number of watchers (27 628) and forks (11 097) as of 2016-01-13 [P*Ols16]. Ruby on Rails is extensively used, famous applications include Twitter, GitHub, and Shopify [P*Rub15a]. David Heinemeier Hansson wrote the first version of Ruby on Rails for the company Basecamp and released it as a FLOSS project in 2004 [P*Rub15b]. There are practical textbooks [RTH13; Har15] as well as research [BK07; SJW08] on Ruby on Rails.

Integrate FLOSS Assets

The gemspec file for the main part of Ruby on Rails lists the gem bundler as a dependency among other dependencies [P*Rub15e]. The gem bundler is developed by a FLOSS project on its own [P*Rub15k], and therefore serves as an example that Ruby on Rails implements BOOTSTRAP.

Start a New Project from Scratch

The first archived commit of Ruby on Rails in November 2004 comprises 28 880 LOC including executable tests [P*Han04]. Although this was shortly after the initial publication as FLOSS in July 2004 [P*Rub15b], it indicates that the first publication also showed a CREDIBLE PROMISE already. It also proves the successful application of the MARKET EVALUATION pattern.

Publish Closed Sources

Ruby on Rails was developed for the company Basecamp [P*Rub15b; P*Bas15] and the first release consisted of more than 10 000 LOC [P*Han04]. Therefore Ruby on Rails implements the pattern DONATE CODE.

Licensing

Ruby on Rails uses the X11 License [P*Fre12] as its only license [P*Rub15d]. This is an APPROVED OPEN SOURCE LICENSE considered as PERMISSIVE LICENSE by the FSF [P*Fre13]. This implies that it uses neither a RECIPROCAL LICENSE nor a DUAL LICENSE.

Architecture

The central repository for Ruby on Rails [P*Rub15f] contains the core module Railties and more central modules. The organization Ruby on Rails at GitHub hosts additional modules in their

3 FLOSS Pattern Languages

own repositories [P*Git15f]. The community has created further modules, but these will not go into analysis, as they are not part of the original FLOSS project Ruby on Rails. Of the modules that Ruby on Rails hosts, the following 20 modules comprise less than 50 source code files: Action Mailer, Strong Parameters, Active Record Session Store, Coffee-Rails, Commands, Active Job, Active Model, Active Resource, GlobalID, Active Model: GlobalID, Jbuilder, Sprockets Rails, Rails::Observers, sass-rails, jquery-rails, Protected Attributes, actionpack-action_caching, actionpack-page_caching, etagger, and Web Console. The following 5 modules comprise more than 50 source code files: Railties itself, Action Pack, Action View, Active Record, and Active Support. Thus, 80 % of the modules have less than 50 source code files and the first criterion for a BAZAAR ARCHITECTURE is fulfilled.

Commits to modules with their own repositories can affect only one module. The repository rails [P*Rub15f] is the only repository hosting multiple modules. Table 3.2 lists the number of commits in the week from 2015-11-02 to 2015-11-08 per day to the repository rails. Table 3.2 distinguishes between commits modifying only one module and commits modifying more than one module. Commits modifying code as well as documentation in the documentation module are regarded as commits modifying only one module. Merged pull requests are counted only on the date of the original commit, not on the later merge. As shown, more than 75 % of the commits modify only one module, so Ruby on Rails also fulfills the second criterion for a BAZAAR ARCHITECTURE and therefore implements this pattern.

Table 3.2: Number of commits to the repository rails [P*Rub15f] from 2015-11-02 to 2015-11-08 that modify only one (upper row) or multiple modules (lower row)

2015-11-	02	03	04	05	06	07	08	Sum
Number of Commits to one module	6	5	14	7	15	14	4	65
Number of Commits to multiple modules	1	1	1	2	0	0	0	5

The documentation for Ruby on Rails [P*Rai15c] describes the architecture, which is also necessary because Ruby on Rails targets software developers as its users, specifically web developers. Furthermore, there is documentation specifically for contributors that goes into more detail. Ruby on Rails uses a model view controller (MVC) architecture and the API documentation [P*Rub15g] details the components for the model Active Record, the view Action View, and the controller Action Pack. The API documentation also covers extensions of and interactions between these components and with dependencies, for example with Rack in the guide Rails on Rack [P*Rai15b]. Although I did not find diagrams of the architecture, a mind map visualizes changes to the architecture [P*Rai13]. Therefore, Ruby on Rails implements an EXPOSED ARCHITECTURE. This documentation also proves the implementation of MANAGE COMPLEMENTS.

Basecamp [P*Bas15] is the maintainer or at least main sponsor of Ruby on Rails, its logo is shown at the bottom of each page on Ruby on Rails's website [P*Rub15a]. Ruby on Rails uses DUAL PRODUCT, there is a product by the same name that complements Ruby on Rails. Consequentially, the pattern SELL COMPLEMENTS is also implemented.

Cultivate a User Community

Ruby on Rails uses the issue tracker integrated in GitHub [P*Git15e]. Core developers do not need to use the issue tracker before committing [P*Rai15a]. Within the sample period from 2015-11-02 to 2015-11-08, 19 issues were created for the main repository rails. 10 of these issues were answered within 24 hours, 2 were closed within 24 hours, and 7 took more than 24 hours to answer. The median answer time was less than 24 hours, so Ruby on Rails implements a CROWD OF BUG DETECTORS.

Ruby on Rails is hosted on GitHub and like all FLOSS projects hosted there, anyone can see metrics like the number of commits. As discussed for Mozilla Firefox, this is not enough for the pattern PUBLIC INSPECTION. However, the main GitHub page for Ruby on Rails [P*Rub15f] features a report of the current “Code Status”, as measured by the CI system Travis CI [P*Tra15]. Hence, Ruby on Rails implements PUBLIC INSPECTION.

Analogously to Rack, Ruby on Rails is released as RubyGem. In 2014, Ruby on Rails published 43 RubyGems [P*Rub15i] and therefore implements FREQUENT RELEASES.

The git repository for Ruby on Rails [P*Rub15f] has branches for each major release and for major changes. Additionally, some developers fork Ruby on Rails into their own repository and then merge their changes back into the master branch. The commit history shows that bug fixes sometimes go directly into the main branch. This is an example of PARALLEL DEVELOPMENT.

The project website of Ruby on Rails [P*Rub15a] is a CENTRAL INFORMATION PLATFORM: It links to the releases, documentation, the project on GitHub, and a list of all communication channels. The communication channels are mailing lists, blogs, and IRC. These platforms and the public GitHub threads host the technical discussions on Ruby on Rails, thus the project conducts an OPEN DIALOG with the community.

As shown before, Ruby on Rails uses an APPROVED OPEN SOURCE LICENSE, hosts a CENTRAL INFORMATION PLATFORM, and conducts an OPEN DIALOG with the community. In order to validate responsiveness to support inquiries, I analyzed help requests on the google group Ruby on Rails: Talk [P*Goo15c]. The threads whose first message was posted between 2015-11-02 and 2015-11-08 comprised 5 announcements and 14 support inquiries, out of which 11 were answered within 72 hours and 3 received no answer within 72 hours. None of the answers advertised paid support of any kind. Although based on a small sample, this indicates that the last requirement for SUPPORT THE COMMUNITY is fulfilled and Ruby on Rails implements the pattern.

Tools for New Developers

Table 3.3 shows statistics on the fraction of newcomers to Ruby on Rails as opposed to existing developers. The first row of contents contains the number of active committers on the master branch of the Ruby on Rails main repository [P*Rub15f] per each month as reported by Open Hub [P*Ope15c]. The second row of contents contains the number of developers who contributed their first commit to the Ruby on Rails main repository in the month designated by the column. The data for the second row were extracted from the Ruby on Rails contributor statistics [P*Nor15a]. The high number of commits from newcomers every month shows that Ruby on Rails has LOW CONTRIBUTION BARRIERS.

Ruby on Rails provides a VM with the IDE to work on Ruby on Rails itself [P*Rub15c]. This

Table 3.3: Number of active developers per month versus number of newcomers per month for the main Ruby on Rails repository

	2014-										2015-									
Number of	05	06	07	08	09	10	11	12	01	02	03	04	05	06	07	08	09	10		
developers	116	81	88	77	79	80	96	103	103	93	101	101	91	70	71	86	81	72		
newcomers	66	34	37	28	24	34	39	44	41	44	38	52	42	33	27	38	32	34		

is a **PRECONFIGURED BUILD ENVIRONMENT**. All modules of Ruby on Rails have a directory “test” with unit tests [P*Git15f]. They run without further configuration and are therefore **UNIT TESTS FOR CONTRIBUTORS**.

Ruby on Rails does not support **LOW-HANGING FRUIT**: The guide Contributing to Ruby on Rails [P*Rai15a] does not refer to any issues suited especially well for newcomers. The issue tracker [P*Git15e] neither contains tags nor do the issues’ titles indicate which issues are suited well for newcomers.

Empower the Most Suitable

David Heinemeier Hansson holds the rights on the Ruby on Rails logo and name. His placement and description on the Ruby on Rails website [P*Rub15b] indicates that he is a **SINGLE MAINTAINER**. Accordingly, there are no formal committees and Ruby on Rails is no **MERITOCRACY**. As Hansson was the initial creator of Ruby on Rails already [P*Rub15b; P*Han04], no **MAINTAINER HANDOVER** has taken place yet, although other contributors have overtaken Hansson in terms of commits to Ruby on Rails [P*Nor15a]. Hansson does not clearly **RUN A TIGHT SHIP**, as he is part of the core developer team and therefore power does not stem from outside of the FLOSS project. However, as a co-owner of the core maintainer Basecamp [P*Rub15b], strongly reduced involvement in development activities [P*Nor15b], ownership over Ruby on Rails name and logo [P*Rub15a], as well as claiming the copyright over the source code [P*Rub14a], Hansson is on the path to **RUN A TIGHT SHIP** over Ruby on Rails.

Nevertheless, different vendors **POOL RESOURCES** to develop Ruby on Rails. All 13 core developers have founded companies or work for companies with a commercial interest in Ruby on Rails, with only 2 of them working for the primary maintainer Basecamp [P*Rub15b]. The other 11 core contributors account for 23 763 commits out of the total 68 321 commits to Ruby on Rails as of 2015-11-10 [P*Nor15a], which is more than the required 20 %.

3.4.4 Spree

Spree is an e-commerce platform based on Ruby on Rails, it is used to create web shops. It was founded with the name RailsCart as a FLOSS project in 2007, and renamed to Spree in 2008 [P*Cas14]. The code is hosted on GitHub [P*Spr15k], the maintainer is the commercial company Spree Commerce [P*Spr15g], which was acquired by First Data Corp. on 2015-09-21 [P*Dal15]. Spree Commerce will reduce their support for Spree after the acquisition [P*Sch15] and so Spree is in a process of change at the time of this analysis. Thus, this

analysis will analyze Spree before the acquisition. Spree Commerce claims that 45 000 retailers use Spree [P*Spr15g].

Integrate FLOSS Assets

Spree uses other FLOSS components and therefore the FLOSS pattern **BOOTSTRAP**. An example of such a FLOSS component is Ruby on Rails used by the Spree Frontend [P*Spr15l].

Start a New Project from Scratch

The first commits to Spree's predecessor RailsCart show that RailsCart started as a FLOSS project from the very beginning [P*Sch07]. RailsCart and therefore Spree used the pattern **MARKET EVALUATION**, but since the first 10 commits did not even contain source code, RailsCart did not show a **CREDIBLE PROMISE** then.

Publish Closed Sources

Spree's predecessor RailsCart was developed as a FLOSS project from the beginning. Hence, it did not use the pattern **DONATE CODE**.

Licensing

Spree uses only the Modified BSD License [P*Spr15j]. This is an **APPROVED OPEN SOURCE LICENSE** and a **PERMISSIVE LICENSE** [P*Fre16]. Accordingly, Spree neither implements **RECIPROCAL LICENSE** nor **DUAL LICENSE**.

Architecture

Spree consists of the meta module Spree and six submodules. The six submodules are spree_api, spree_frontend, spree_backend, spree_cmd, spree_core, and spree_sample. All of these are RubyGems on their own. While they work together as a platform, they can also be used independently. [P*Spr15k]

Spree_core has more than 50 files of source code, the other five submodules have less than 50 files of source code. Additionally, Spree supports extensions [P*Spr15b] to **MANAGE COMPLEMENTS**. Spree hosts 60 extensions in the separate GitHub organization spree-contrib as of 2015-11-11 [P*Git15i]. In their main GitHub organization, Spree hosts about 10 extensions and other types of add-on components [P*Git15h].

The analysis for **BAZAAR ARCHITECTURE** takes only the six core submodules into account. As explained, $\frac{5}{6} = 83.\bar{3} \%$ of the submodules have less than 50 files of source code, so the first condition that at least 75 % of the modules have less than 50 files of source code is fulfilled. The analysis for the second condition took all commits to the master branch of Spree [P*Spr15k] in 2015-08 into account, excluding pure merge commits. Of these 29 commits, 27 commits modified only one module, 1 modified multiple modules, and 1 commit modified one module and also the documentation. This suffices the second condition that at least 75 % of commits must modify at most one module, so Spree implements a **BAZAAR ARCHITECTURE**.

Documentation of the Spree API features an interactive visualization of the data structures and its Representational State Transfer (REST) architecture [P*Spr15h]. The developer documentation includes a chapter specifically for the architecture of the core components [P*Spr15d]. Thus, Spree has an EXPOSED ARCHITECTURE.

Spree Commerce Inc. SELLS COMPLEMENTS for Spree, specifically Wombat, a software to integrate different parts of a web shop, like the web frontend with distributors and accounting solutions [P*Spr15i]. Although Wombat does not depend on Spree and also works with competing platforms for web shops [P*Spr15e], the Spree Commerce Inc.'s engagement with Spree suggests that they sell Wombat mostly as a DUAL PRODUCT.

Cultivate a User Community

Spree uses the issue tracker of GitHub [P*Git15j]. Although the documentation does not say it explicitly, the phrasing implies that core developers may commit directly without corresponding issue [P*Spr15a]. All issues in the issue tracker for the spree repository on GitHub in the week from 2014-07-18 to 2014-07-24 were included in a sample. Of these 42 issues, 34 received an answer within 24 hours, 6 did not receive an answer within 24 hours, 2 were closed by the creator. The median delay between issue report and answer was lower than 24 hours, so Spree uses a CROWD OF BUG DETECTORS.

Besides the basic VCS data that GitHub measures, Spree presents its build status, an evaluation of its code quality, and responsiveness to issue reports prominently on its main GitHub page [P*Spr15k]. CircleCI measures the build status [P*Cir15b], Code Climate measures the code quality [P*Cod15], and Issue Stats measures the responsiveness to issue reports [P*Sto15]. These providers are independent and measure multiple projects, the measurement results are drawn directly from the provider and therefore update immediately whenever a provider has new results. Thus, Spree implements PUBLIC INSPECTION.

From 2008-02-26 to 2015-08-31, there have been 182 releases of RubyGems for the Spree meta module [P*Rub15j]. The arithmetic mean time between two releases was 15.08 days and therefore less than a month. Thus, Spree provided FREQUENT RELEASES of its software.

Spree has multiple branches, one for each released version of the software and the master branch including new features. Additionally, the documentation recommends to create a branch for each isolated set of changes. Thus, Spree supports PARALLEL DEVELOPMENT. [P*Spr15a]

The website of Spree Commerce [P*Spr15g] links to the different types of documentation, complementary information, and most importantly to the development tools within the item Open Source in the menu under Resources. The main GitHub repository [P*Spr15k] also links to the development tools like the issue tracker, and it links back to the website of Spree Commerce. All documentation subpages link back to the Spree Commerce website, while the issue tracker links back only to the main GitHub repository. Access information for the Instant Messaging (IM) services of Spree, a Gitter chat room and an IRC channel, is buried deeper in the documentation [P*Spr15c]. Thus, information is distributed mainly between the website of Spree Commerce and the main GitHub repository and links between them are not so obvious that both could be treated as a single CENTRAL INFORMATION PLATFORM. Thus, Spree does not implement the pattern.

Spree encourages contributions and new feature ideas from the community, and discusses

them publicly on the mailing list or IRC [P*Spr15a]. Spree conducts an OPEN DIALOG with the community.

Spree does not SUPPORT THE COMMUNITY, as it does not run a CENTRAL INFORMATION PLATFORM. Additionally, user support requests are not answered often enough: 28 mail threads in the Spree user mailing list [P*Goo15d] were started in 2015-08. These served as a sample to analyze responsiveness to support requests. One of the mails was an announcement from Spree Commerce, another mail was a technical problem with the mail list system. The remaining 26 mail threads started with requests for support from users. 15 of these mail threads were answered within 72 hours after creation, 11 received no answer at all or it took longer than 72 hours. $\frac{15}{26} \approx 57.7\%$ is below the required threshold of 75 %. None of these requests advertised premium support.

Tools for New Developers

Table 3.4 shows the number of active contributors in every month from 2014-01 to 2015-08 versus the number of newcomers in each of these months. Data of the total number of contributors were retrieved from Open Hub [P*Ope15d]. Data for the second row were retrieved via git from the main Spree repository [P*Spr15k] and processed with GitHistoryAnalyzer [Han15a]. The data show that there is a high number of commits from newcomers every month, so Spree has Low CONTRIBUTION BARRIERS.

Table 3.4: Number of active developers per month versus number of newcomers per month for the main Spree repository

	2014-												2015-							
Number of	01	02	03	04	05	06	07	08	09	10	11	12	01	02	03	04	05	06	07	08
developers	25	26	30	31	24	28	49	36	32	28	30	26	20	19	29	21	18	14	18	11
newcomers	7	17	16	12	12	14	22	10	17	13	8	10	9	7	10	11	10	7	8	6

Spree is written in Ruby and therefore mostly platform independent. They provide the files Gemfile and Rakefile in their main repository [P*Spr15k]. Gemfile allows the tool Bundler [P*Bun15] to automatically install all dependencies for Spree. Rakefile includes a task “sandbox” that orders the Ruby tool Rake [P*Rak15a] to create a test installation of Spree to work with. Although this solution requires a preinstalled Ruby development environment, it counts as a PRECONFIGURED BUILD ENVIRONMENT.

All submodules of Spree have their own set of unit tests located in each submodule’s folder “spec” [P*Spr15k]. This fulfills the requirements for UNIT TESTS FOR CONTRIBUTORS. Spree also uses a CI server that tests all modifications [P*Cir15b].

Spree’s issue tracker uses the labels “easy”, “medium”, and “hard” to indicate the difficulty of solving an issue [P*Git15j]. This allows newcomers to first pick the LOW-HANGING FRUIT and then continue with more and more difficult issues as their experience with Spree grows.

Empower the Most Suitable

Spree Commerce Inc. hosts Spree's website [P*Spr15g] and pays the maintainers Sean Schofield and Brian Quinn for their work on Spree [P*Spr13]. It also holds the rights over the spreecommerce logo and name of Spree [P*Spr15g]. Thus, Spree Commerce Inc. RUNS A TIGHT SHIP over Spree, and there is neither a SINGLE MAINTAINER nor a MERITOCRACY. As Sean Schofield created the initial version of Spree's predecessor RailsCart [P*Sch07] and he is now Chief Executive Officer (CEO) of Spree Commerce Inc. [P*Spr15f], no clear MAINTAINER HANDOVER took place.

An analysis of all commits to the main git repository of Spree [P*Spr15k] until 2015-10 with the tool GitHistoryAnalyzer [Han15a] shows that there are 15 703 commits in total. Spree Commerce Inc. employs or employed the developers with the highest number of commits. Thus, the pattern POOL RESOURCES is not fulfilled, based on the assumption that less than 20 % of the commits, i.e. less than 3140 commits, came from organizations with commercial interest in Spree other than Spree Commerce Inc. itself. Specifically, I attributed 9088 commits to Spree Commerce employees and 1110 to developers who probably work for other organizations with commercial interest in Spree. Affiliations of authors were sometimes unclear only from investigation of publicly available data or it was unclear whether their work for Spree had a commercial interest. Accordingly, declining the implementation of POOL RESOURCES might be erroneous, but the margin of error is small when extrapolating the determined data.

3.4.5 Rake

Rake [P*Rak15a] is part of the standard Ruby development chain. Inspired by Make [P*Fre14], it automates build tasks in a Ruby development project. Developers create Rakefiles written in Ruby to define Rake Tasks executed when building, installing, or testing their component.

Rake's maintainer and original creator Jim Weirich died unexpectedly in 2014-02 [P*The14]. Rake moved its main repository from Weirich's personal GitHub account [P*Rak14] to the Ruby organization [P*Rub14c] as a consequence.

Integrate FLOSS Assets

Rubygems.org shows that Rake has no runtime dependencies but three development dependencies [P*Rub14c]: Hoe, Minitest, and Rdoc. These three components help to build Rake, but are not necessarily required, as the build runs without these tools with some additional effort. In fact, Hoe requires Rake to run [P*Rub15l]. Accordingly, Rake does not implement the pattern BOOTSTRAP.

Start a New Project from Scratch

The initial commit of Rake [P*Wei03] already shows a CREDIBLE PROMISE, as it has most functionality that Make has according to the description in the documentation in the commit. The first commit also contains a file `rake/doc/rational.rdoc` that explains the motivation to create Rake. Rake's initial author Jim Weirich described that he is uncertain whether Rake is interesting for others and indicates that this is the initial release of Rake. Thus, Rake also started as a MARKET EVALUATION.

Publish Closed Sources

Before the initial commit of Rake [P*Wei03], there have been no uses of the code in a closed source context. Therefore, Rake does not qualify for `DONATE CODE`.

Licensing

Rake uses the X11 License [P*Fre12] for its software [P*Rak15a]. The X11 License is an `APPROVED OPEN SOURCE LICENSE` and a `PERMISSIVE LICENSE`. Rake does not implement the patterns `RECIPROCAL LICENSE` and `DUAL LICENSE`.

Architecture

Rake's source code has no modular structure [P*Rak15b], so Rake does not implement the pattern `BAZAAR ARCHITECTURE`. Without subcomponents, Rake implements neither `DUAL PRODUCT` nor `MANAGE COMPLEMENTS`. There are no indications that maintainers `SELL COMPLEMENTS` for Rake.

Documentation exists on the main GitHub page [P*Rak15b] and on the Rake website [P*Rak15a], but this is only a subset of the documentation in the git repository [P*Rak15b, directory doc]. None of these documentations explain Rake's architecture, so Rake has no `EXPOSED ARCHITECTURE`.

Cultivate a User Community

Rake's commit history [P*Rak15b] shows commits with no corresponding pull request. Furthermore, it contains more commits than there are issues in Rake's issue tracker [P*Git15g]. Thus, developers may commit to the repository without opening an issue in the issue tracker. Therefore, the criterion for `CROWD OF BUG DETECTORS` is that answer delays for issues should average below 24 hours. All 30 issues created between 2015-01 and 2015-10 were used as a sample. 3 of these issues were closed by the creator within 24 hours, but of the remaining 27 issues, only 6 received an answer within 24 hours. The other 21 issues received no answer for weeks or months, with only one exception where it took less than a week. Thus, Rake has no `CROWD OF BUG DETECTORS`.

Rake uses the same metrics as Ruby on Rails: Default GitHub metrics [P*Rak15b] and status of the build as measured by Travis CI [P*Tra15]. Rake publishes these metrics on their GitHub page [P*Rak15b]. Therefore, Rake implements `PUBLIC INSPECTION`.

RubyGems.org lists 25 releases of Rake as RubyGem between its first release as a RubyGem in 2004-09-15 and 2010-09-15 [P*Rub14b], corresponding to one release in about every three month. After this, the release frequency increased, as also beta versions of Rake were released as RubyGem. There were 38 releases between 2011-01 and 2015-10, so the arithmetic mean is 46.4 days between two releases. This is still not considered `FREQUENT RELEASES`.

Although the Rake repository has multiple branches, only "master" was used between 2013-02 and 2015-10 [P*Git15b]. Developers may fork the main repository and merge their modifications back to it. However, since this is not common policy, Rake has no `PARALLEL DEVELOPMENT`.

There is an orphaned Rake website [P*Rak13] that can neither be administered nor deleted after the death of Rake's original maintainer Jim Weirich. The new website [P*Rak15a] contains no information that is not already present on the start page of the GitHub repository [P*Rak15b]. Rake uses only services from GitHub, like the issue tracker [P*Git15g], which all link back to the

3 FLOSS Pattern Languages

Rake GitHub repository. Thus, the Rake GitHub repository is a CENTRAL INFORMATION PLATFORM for Rake.

Rake has no mailing list or similar communication channels. The documentation still contains the email address of the late maintainer Jim Weirich as contact address [P*Rak15a]. By design, issues are discussed publicly in the issue tracker [P*Git15g], but core developers commit directly to the repository without explaining their reasons. As a consequence, Rake has no OPEN DIALOG with its community. Without OPEN DIALOG and with the high number of unanswered issues in the issue tracker as discussed above, Rake also does not SUPPORT THE COMMUNITY.

Tools for New Developers

Table 3.5 shows the number of newcomers per month for Rake in the time frame from 2014-01 to 2015-08. These numbers are opposed to the number of active developers in the same months. The data were extracted from Rake's git repository [P*Rak15b] with GitHistoryAnalyzer [Han15a]. The fraction of newcomers is above 5 % on average, median as well as arithmetic mean, so Rake has LOW CONTRIBUTION BARRIERS.

Table 3.5: Number of active developers per month versus number of newcomers per month for Rake

	2014-												2015-							
Number of	01	02	03	04	05	06	07	08	09	10	11	12	01	02	03	04	05	06	07	08
developers	6	3	4	4	4	4	4	3	2	1	3	7	1	4	4	1	4	2	4	2
newcomers	5	2	2	1	2	2	2	1	1	0	0	5	1	3	3	0	2	1	3	0

Rake includes a `Rakefile` for its own build [P*Rak15b]. It can be difficult to build Rake from source code [P*MB15], as the repository is missing a `rake.gemspec`. Rake therefore has no PRECONFIGURED BUILD ENVIRONMENT.

The Rake repository includes unit tests [P*Rak15b, directory test] and the project uses Travis CI [P*Rak15b]. Thus, Rake implements UNIT TESTS FOR CONTRIBUTORS.

Issues in Rake's issue tracker [P*Git15g] neither have tags nor titles that indicate whether they are well-suited for newcomers. Rake does not implement LOW-HANGING FRUIT.

Empower the Most Suitable

No external organization is involved with Rake, so the pattern RUN A TIGHT SHIP is not implemented. Instead, Jim Weirich had been a SINGLE MAINTAINER for Rake until his death on 2014-02-19. The subsequent MAINTAINER HANDOVER classifies as *branched*, as the original repositories [P*Rak14] were not continued. Accordingly, Rake is no MERITOCRACY. It also does not fulfill the requirements for POOL RESOURCES.

3.5 Discussion

Table 3.6 summarizes the results of the previous evaluation. Each column shows the result for one FLOSS project. The first rows show results for the individual FLOSS patterns, and the last rows show other metrics of the FLOSS projects. For each combination of FLOSS pattern and FLOSS project, a ✓ indicates that the FLOSS project implemented the FLOSS pattern, while a ✗ indicates that it did not implement the FLOSS pattern. If the pattern MAINTAINER HANDOVER was applied, the table does not contain a ✓ but the specific outcome using the first letter of the outcome's title as described in the MAINTAINER HANDOVER's verification section: S stands for a smooth transition, C for a complicated transition, and B for a branched FLOSS project. A plus indicates additional MAINTAINER HANDOVERS that have not been analyzed. Number of patterns counts all implemented FLOSS patterns. MAINTAINER HANDOVER counts at most once. The number of commits refers only to the main branch of the main repository of each FLOSS project [P*Rac15g; P*Rub15f; P*Spr15k; P*Rak15b] until 2015-10-31 as reported by GitHub. For Mozilla Firefox, the number of commits comes from Open Hub [P*Ope15a] and represents all commits until about 2015-09-19. The number of commits is included as a metric for the size of a FLOSS project.

3.5.1 FLOSS Patterns and the Success of FLOSS Projects

Results of the analysis suggest that there is a positive relationship between the number of implemented FLOSS patterns and the size of a FLOSS project. The two smaller FLOSS projects Rack and Rake have implemented considerably less FLOSS patterns than the three larger FLOSS projects. This relationship seems to be saturated already in a FLOSS project of the size of Spree: Firefox implements about the same number of patterns as Spree, although it is an order of magnitude larger in terms of number of commits. The data itself also does not show causality in one or the other direction: Do FLOSS projects receive many commits if they implement many FLOSS patterns? Or do some FLOSS patterns emerge when a FLOSS project reaches some critical size? The FLOSS patterns itself suggest that both effects exist: LOW CONTRIBUTION BARRIERS and related FLOSS patterns specifically try to increase the number of contributions, while laborious FLOSS patterns like PRECONFIGURED BUILD ENVIRONMENT are easier to implement for large FLOSS projects with more contributors. For a FLOSS pattern like OPEN DIALOG, it may become more and more difficult to resist its implementation and seal the developers off from the public when the FLOSS project grows.

The three larger FLOSS projects implemented BOOTSTRAP, and not implementing it is only feasible for small FLOSS projects like RACK and RAKE. All analyzed FLOSS projects started with a MARKET EVALUATION and all except Spree with a CREDIBLE PROMISE. These may be important patterns for successful FLOSS projects. The results show that successful FLOSS projects may start from scratch and do not need DONATE CODE to be successful.

The analyzed FLOSS projects with commercial intentions, Rails and Spree, favored DUAL PRODUCT OVER DUAL LICENSE TO SELL COMPLEMENTS. However, these are just two data points and conclusions must be drawn with caution. Another licensing aspect is the choice between a RECIPROCAL LICENSE and a PERMISSIVE LICENSE. Whether one or the other helps with a FLOSS project to succeed has been the subject of research based on much larger data sets than only five

Table 3.6: Results of the evaluation for all FLOSS patterns and projects

FLOSS Pattern	Firefox	Rack	Rails	Spree	Rake
Integrate FLOSS Assets					
BOOTSTRAP	✓	✗	✓	✓	✗
Start a New Project from Scratch					
MARKET EVALUATION	✓	✓	✓	✓	✓
CREDIBLE PROMISE	✓	✓	✓	✗	✓
Publish Closed Sources					
DONATE CODE	✗	✗	✓	✗	✗
Licensing					
DUAL LICENSE	✗	✗	✗	✗	✗
RECIPROCAL LICENSE	✓	✗	✗	✗	✗
PERMISSIVE LICENSE	✗	✓	✓	✓	✓
APPROVED OPEN SOURCE LICENSE	✓	✓	✓	✓	✓
Architecture					
BAZAAR ARCHITECTURE	✗	✓	✓	✓	✗
EXPOSED ARCHITECTURE	✗	✗	✓	✓	✗
SELL COMPLEMENTS	✗	✗	✓	✓	✗
DUAL PRODUCT	✗	✗	✓	✓	✗
MANAGE COMPLEMENTS	✓	✓	✓	✓	✗
Cultivate a User Community					
CROWD OF BUG DETECTORS	✓	✗	✓	✓	✗
SUPPORT THE COMMUNITY	✓	✗	✓	✗	✗
PUBLIC INSPECTION	✗	✓	✓	✓	✓
FREQUENT RELEASES	✓	✗	✓	✓	✗
PARALLEL DEVELOPMENT	✓	✓	✓	✓	✗
CENTRAL INFORMATION PLATFORM	✓	✗	✓	✗	✓
OPEN DIALOG	✓	✓	✓	✓	✗
Tools for New Developers					
LOW CONTRIBUTION BARRIERS	✓	✓	✓	✓	✓
PRECONFIGURED BUILD ENVIRONMENT	✓	✓	✓	✓	✗
UNIT TESTS FOR CONTRIBUTORS	✓	✓	✓	✓	✓
LOW-HANGING FRUIT	✓	✗	✗	✓	✗
Empower the Most Suitable					
RUN A TIGHT SHIP	✓	✗	✗	✓	✗
MAINTAINER HANDOVER	SS+	CS	✗	✗	B
POOL RESOURCES	✓	✗	✓	✗	✗
SINGLE MAINTAINER	✗	✓	✓	✗	✓
MERITOCRACY	✓	✗	✗	✗	✗
Number of implemented patterns	20	14	23	19	10
Number of commits	271 370	2108	54 125	15 691	1911

FLOSS projects [SAM06; CMP07; CF09; SSN09], and their results still contradict each other. No matter which of the two licenses a FLOSS project chooses, an APPROVED OPEN SOURCE LICENSE seems to be an important success factor.

The results for the FLOSS patterns in the category Architecture do not give a clear picture. BAZAAR ARCHITECTURE and SELL COMPLEMENTS do not seem necessary for the success of the FLOSS projects, but are still common enough to be observed in the selected sample of FLOSS projects. An EXPOSED ARCHITECTURE is presumably not necessary for smaller FLOSS projects like Rack and Rake, but helpful for larger FLOSS projects. For very large FLOSS projects like Firefox, implementing an EXPOSED ARCHITECTURE is possibly difficult. MANAGE COMPLEMENTS encourages the contribution of add-ons. Add-ons may serve as an intermediate step for core contributions, lessening the difficulties to contribute to the core product. Add-on developers need to consider less dependencies to other components than developers of the core component. They still learn the core component's API and therefore gain knowledge about its internal structure. Therefore, FLOSS projects gain a double benefit from MANAGE COMPLEMENTS.

Remarkably, implementation of the FLOSS patterns in the category Cultivate a User Community seems to depend on the size of the FLOSS project: For each FLOSS pattern except PUBLIC INSPECTION and CENTRAL INFORMATION PLATFORM, it is possible to define a threshold for the number of commits such that all analyzed FLOSS projects beyond this threshold implement the pattern. Table 3.7 is a rearranged extract of Table 3.6 to illustrate this relationship: Is is observable that the ✓ symbols have the arrangement of a stair.

Table 3.7: Results for the FLOSS patterns of Cultivate a User Community ordered by number of commits and supposed pattern threshold

FLOSS Pattern	Rake	Rack	Spree	Rails	Firefox
SUPPORT THE COMMUNITY	✗	✗	✗	✓	✓
CROWD OF BUG DETECTORS	✗	✗	✓	✓	✓
FREQUENT RELEASES	✗	✗	✓	✓	✓
PARALLEL DEVELOPMENT	✗	✓	✓	✓	✓
OPEN DIALOG	✗	✓	✓	✓	✓
Number of commits	1911	2108	15 691	54 125	271 370

The FLOSS patterns shown in Table 3.7 seem to be key ingredients to grow the community of a FLOSS project and benefit from this community. The threshold for SUPPORT THE COMMUNITY seems to be higher than for CROWD OF BUG DETECTORS and FREQUENT RELEASES, while it seems lower for PARALLEL DEVELOPMENT and OPEN DIALOG. However, while these data give reliable support for the existence of a relationship between the FLOSS patterns in the class CULTIVATE A USER COMMUNITY and project size, the specific order suggested by Table 3.7 may be only a peculiarity of the sampled FLOSS projects and is not reliable.

All analyzed FLOSS projects except Firefox implement PUBLIC INSPECTION. Firefox has end users as target audience, while the other analyzed FLOSS projects target software developers, specifically Ruby developers. There are two reasons why FLOSS projects for end user applications

do not use `PUBLIC INSPECTION` while FLOSS projects for software developers do. First, only software developers have the expertise to understand the metrics of a `PUBLIC INSPECTION`, so most users of Firefox would not benefit from it. Second, the Ruby FLOSS projects except Rake are frameworks and libraries, so their users interface with the code of the components they use. Rake users also have to write Ruby code. Thus, the code must have a high quality. `PUBLIC INSPECTION` lets the users evaluate the code quality and so software developers know whether they are using a component with high code quality. Users of applications like Firefox interface only with the GUI or command line and therefore need usable applications instead of applications with high code quality. Of course, usability and code quality may be connected, but high code quality does not ensure usability. Thus, `PUBLIC INSPECTION` seems to be important for FLOSS projects developing libraries and frameworks, but not for those developing end user applications.

`CENTRAL INFORMATION PLATFORM` seems not to have a strong influence on the success of a FLOSS project. However, the two FLOSS projects without `CENTRAL INFORMATION PLATFORM` still implemented parts of it, so maybe the current form of `CENTRAL INFORMATION PLATFORM` or its verification criteria do not yet capture the invariant of the FLOSS pattern.

The FLOSS patterns in Tools for New Developers and especially `LOW CONTRIBUTION BARRIERS` seem to be crucial for the success of FLOSS projects, as they are universally implemented by the analyzed FLOSS projects. Rake may not need a `PRECONFIGURED BUILD ENVIRONMENT`, as it is small and the build is not so difficult even without the `PRECONFIGURED BUILD ENVIRONMENT`. `LOW-HANGING FRUIT` is an exception in this category, as only two FLOSS projects in the sample implemented it.

The analyzed FLOSS projects employ a diverse mixture of power models and all of them have been shown to work. Rails and Spree `SELL COMPLEMENTS` and use it to finance development. Running a FLOSS project with volunteers or with the support of a foundation seems to work as well. Remarkably, only the three FLOSS projects without commercial support, Firefox, Rack, and Rake, had multiple `MAINTAINER HANDOVERS`, some of which were complicated. Firefox as the only FLOSS project backed by a Foundation also had an especially high number of `MAINTAINER HANDOVERS`, but they all went smoothly. However, the evaluation comprises too few FLOSS projects to decide whether these observations are general rules or only coincidences in the selected sample of FLOSS projects.

3.5.2 Validation of Relationships

This chapter uses two types of relationships to model the dependencies between FLOSS patterns. The first type of relationship are alternatives that exclude each other. If these relationships were identified correctly, at most one FLOSS pattern standing in such a relationship should be implemented by any FLOSS project. The other type of relationship shows that one FLOSS pattern uses or requires a second FLOSS pattern. If these relationships were identified correctly, the chance that a FLOSS project implements a specific FLOSS pattern is greater if other related FLOSS patterns are also implemented. Thus, results should contain clusters of related and implemented FLOSS patterns.

All analyzed FLOSS projects either implemented `PERMISSIVE LICENSE` or `RECIPROCAL LICENSE`. This is not surprising, as they exclude each other by definition. Two analyzed FLOSS projects implemented `DUAL PRODUCT` and none `DUAL LICENSE`. Thus, the results do not conflict with the excluding relationship between the two FLOSS patterns. All FLOSS projects except Firefox

implement exactly one of the three alternative FLOSS patterns RUN A TIGHT SHIP, SINGLE MAINTAINER, and MERITOCRACY. Firefox implements both RUN A TIGHT SHIP as well as MERITOCRACY, but as a subproject of the MERITOCRACY Mozilla, it can be considered a special case. The alternative between the three FLOSS patterns seems to be mostly correct.

The next hypothesis to test is that use relations between patterns increase the chance of concurrent implementation of the two related patterns. The corresponding null hypothesis states that use relations have no influence on which FLOSS patterns a FLOSS project implements. Thus, the null hypothesis can be rejected if it turns out that actual FLOSS projects satisfy relations, i.e. implement both FLOSS patterns of a relation, more often than expected if the null hypothesis were true. Each FLOSS project is tested individually whether it provides evidence for or against rejection of the null hypothesis.

More specifically, first, for each FLOSS pattern the fraction of its implementations within the set of analyzed FLOSS projects is used as approximation for the probability that any FLOSS project implements this FLOSS pattern. For example, BOOTSTRAP is assumed to have a probability for implementation of $\frac{3}{5}$, because three of the five analyzed FLOSS projects have implemented it. This step is independent from the specific FLOSS project.

Described formally, let \mathbb{F} be the set of all analyzed FLOSS patterns, and let $\mathbb{P} := \mathcal{P}(\mathbb{F})$ be the set of all possible FLOSS pattern combinations, which we interpret as the set of possible different FLOSS projects. Every selection of FLOSS patterns, i.e. every FLOSS project, has a probability to occur in practice, modeled by the probability function $P : \mathcal{P}(\mathbb{P}) \rightarrow [0; 1]$. P maps a set of FLOSS projects to the probability that a randomly chosen FLOSS project lies within this set. The probability $P(\{A \in \mathbb{P} | X \in A\})$ that a FLOSS project implements the FLOSS pattern $X \in \mathbb{F}$ shall be $p_X \in [0; 1]$. p_X is known for all FLOSS patterns in \mathbb{F} , as we use the fraction of implementations within the analyzed FLOSS projects as approximation, as described in the last paragraph.

Second, the tool PatternStatistics [Han15b] written for this analysis calculates the probability distribution for the number of satisfied relations under the precondition that a FLOSS project implements n FLOSS patterns. n equals the number of implemented FLOSS patterns of the tested FLOSS project. Figure 3.14 shows the distribution for $n = 20$, which is used for the test of Firefox.

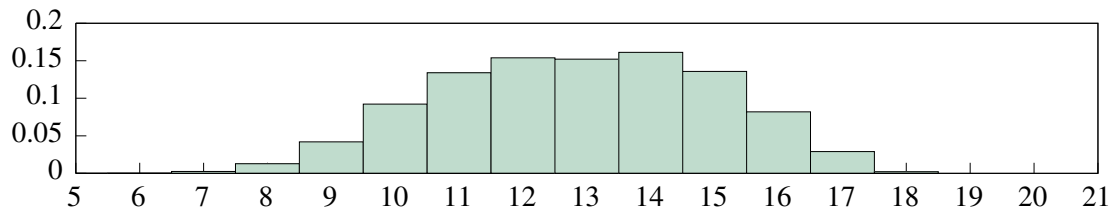


Figure 3.14: Probability distribution of satisfied relations under the precondition of $n = 20$ implemented FLOSS patterns

The set of all relations between FLOSS patterns shall be $\mathbb{R} \subset \mathbb{F} \times \mathbb{F}$. A FLOSS project $A \in \mathbb{P}$ satisfies a relation $X \times Y \in \mathbb{R}$ if $X, Y \in A$. Let $R_A := (A \times A) \cap \mathbb{R}$ denote the set of relations that the FLOSS project $A \in \mathbb{P}$ satisfies. Using these definitions, the probabilities shown in Figure 3.14 are the conditional probabilities $p_n^i := P(\{A \in \mathbb{P} | |R_A| = i\} | \{A \in \mathbb{P} | |A| = n\})$ for $i = 5 \dots 21$ and

3 FLOSS Pattern Languages

$n = 20$. PatternStatistics calculates the values p_n^i for all relevant n and all $i = 0 \dots |R|$ by summing the probabilities of all FLOSS projects with the requested number of satisfied relations i and implemented FLOSS patterns n , using the following equation. The first equality is based on the definition of conditional probabilities, the last uses the pairwise independence of FLOSS pattern occurrences implied by the null hypothesis:

$$\begin{aligned}
 p_n^i &= \frac{P(\{A \in \mathbb{P} \mid |R_A| = i \wedge |A| = n\})}{P(\{A \in \mathbb{P} \mid |A| = n\})} \\
 &= P(\{A \in \mathbb{P} \mid |A| = n\})^{-1} \sum_{\substack{A \in \mathbb{P} \\ |A| = n \\ |R_A| = i}} P(\{A\}) \\
 &= P(\{A \in \mathbb{P} \mid |A| = n\})^{-1} \sum_{\substack{A \in \mathbb{P} \\ |A| = n \\ |R_A| = i}} \left(\prod_{X \in A} p_X \prod_{Y \in \mathbb{F} \setminus A} (1 - p_Y) \right)
 \end{aligned}$$

Figure 3.15 shows a simplified pattern map showing only the FLOSS patterns used in the evaluation and additionally omitting DUAL LICENSE, which has not been implemented in any of the analyzed FLOSS projects. The map also does not show the alternate relationships. All FLOSS patterns that Firefox has implemented have a green ✓. FLOSS patterns not implemented by Firefox have a red ✗. Satisfied relations are thick and blue, other relations are thinner and black.

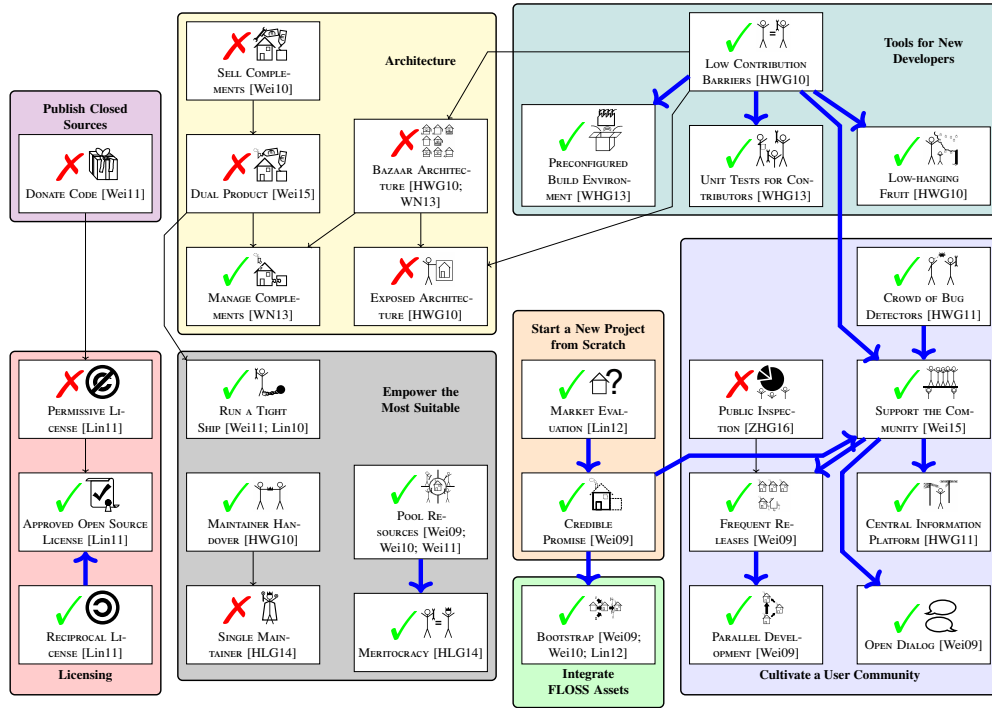


Figure 3.15: Implemented FLOSS patterns (green check mark) for Firefox and their 14 satisfied relations (thick and blue)

As shown in Figure 3.15, Firefox satisfies 14 relations. The distribution depicted in Figure 3.14 implies that 41.0 % of all randomly selected FLOSS projects that implement 20 FLOSS patterns, the same number as Firefox, satisfy at least 14 relations: $\sum_{i=14}^{24} p_{20}^i \approx 0.410$. This means that if the null hypothesis were true, a result at least as extreme as that of Firefox could be seen in $p = 0.410$ of all cases. This is clearly not enough to reject the null hypothesis.

The results for the five analyzed FLOSS projects are shown in Table 3.8. Although all analyzed FLOSS projects satisfy at least as many relations as an equivalent median FLOSS project, the differences to the random selection are quite small and therefore the p-values are high. Only the result for Ruby on Rails is significant at a significance level of $\alpha = 0.05$, but this may be by chance.

Table 3.8: Results for the statistical analysis of the relations between patterns

Number of ...	Firefox	Rack	Rails	Spree	Rake
... implemented patterns (i)	20	14	23	19	10
... satisfied relations	14	7	20	13	4
... median satisfied relations for the same i	13	7	17	12	4
p-value	0.410	0.513	0.048*	0.367	0.575

* significant at $\alpha = 0.05$

The overall results for all analyzed FLOSS projects are not enough to reject the null hypothesis, so the statistical analysis cannot rule out that FLOSS projects do not use the relations described in Section 3.2. However, this may be due to a too small sample size. Some relations may be so weak that they can only be demonstrated statistically in a large sample. As another explanation, the statistical model may not reflect reality well enough. The set \mathbb{P} contains combinations of FLOSS patterns that have a positive probability to occur but cannot happen in practical FLOSS projects: For example, combinations of FLOSS patterns excluded by alternate relationships cannot happen, but are considered in the distribution. As another example, SUPPORT THE COMMUNITY requires the implementation of other FLOSS patterns by definition and therefore FLOSS projects missing these required FLOSS patterns should not implement SUPPORT THE COMMUNITY in the statistical model. Another interpretation of the mostly negative outcome is that the relations describe combinations of FLOSS patterns that support each other and are sensible to combine – but real FLOSS projects use combinations that are suboptimal.

3.5.3 Newly Observed Relationships

What relations can be observed in the empirical data that had not been previously postulated? Some FLOSS pattern implementations seem to cluster more in families than along relations: Firefox, Rack, and Rake implement one, two, and zero Architecture patterns, respectively, while Rails and Spree implement all five. The data also show some FLOSS patterns that always come in pairs, which may or may not be a coincidence.

Specifically, Firefox uses a RECIPROCAL LICENSE and has a MERITOCRACY, while the other analyzed FLOSS projects implement neither. Both FLOSS patterns may be more likely in nonprofit FLOSS projects.

Firefox, Rails, and Spree implement both CROWD OF BUG DETECTORS as well as FREQUENT RELEASES, while Rack and Rails implement neither. There is already an indirect relationship via SUPPORT THE COMMUNITY, but maybe FREQUENT RELEASES influences CROWD OF BUG DETECTORS more directly. On the one hand, according to Raymond [Ray00], FREQUENT RELEASES stimulate users to search for defects, as they can always access new source code, and also reward finding defects as their fixes quickly appear in official releases. On the other hand, Firefox, Rails, and Spree are the three largest analyzed FLOSS projects and maybe the implementation of FREQUENT RELEASES and CROWD OF BUG DETECTORS only depends on project size as a common cause, but there is no direct relation.

While the analyzed data support these observations of previously unpredicted relationships, these are only post hoc observations. New data are needed to verify or reject these hypothesized relationships for a proper a priori analysis.

3.6 Conclusion

This chapter presented a pattern language for FLOSS development. In particular, six FLOSS patterns have been described in full that help to lower contribution barriers identified in Chapter 2. The relationships identified in the FLOSS pattern language provide the setting to implement and understand those six FLOSS patterns lowering contribution barriers. The chapter contributed to four goals.

First, a classification of all published FLOSS patterns into eight categories helps readers to get a high-level overview of FLOSS patterns. Duplicate FLOSS patterns have been merged and 35 unique FLOSS patterns have been identified.

Second, besides the association of FLOSS patterns in the same category, two additional types of relationships between FLOSS patterns have been introduced and identified between some of the FLOSS patterns. These relationships constitutes a pattern language among FLOSS patterns. These relationships as well as the classification help to identify the FLOSS patterns that directly or indirectly support specific goals of a FLOSS project, in the context of this thesis especially lowering contribution barriers.

Third, out of the whole FLOSS pattern language, eight FLOSS patterns explicitly try to lower contribution barriers, especially modification barriers. Section 3.3 contains six of these FLOSS patterns. The FLOSS pattern LOW CONTRIBUTION BARRIERS serves as a meta-pattern and an entry point to lower contribution barriers. The FLOSS patterns LOW-HANGING FRUIT, BAZAAR ARCHITECTURE, and EXPOSED ARCHITECTURE tackle the modification barrier that understanding the structure of a FLOSS project can be hard for newcomers, which was identified as the most important contribution barrier in Chapter 2. PRECONFIGURED BUILD ENVIRONMENT and UNIT TESTS FOR CONTRIBUTORS help to reduce problems setting up the development environment. This modification barrier was also identified as one of the most important contribution barriers in Chapter 2.

Fourth, an evaluation on five real FLOSS projects assessed which FLOSS patterns occur more often and which are less common in practice. Some FLOSS patterns are very common in successful FLOSS projects. This suggests that these FLOSS patterns may be crucial for a FLOSS project's success. Specifically, LOW CONTRIBUTION BARRIERS, APPROVED OPEN SOURCE LICENSE,

CREDIBLE PROMISE, and MARKET EVALUATION are candidates for very important FLOSS patterns. The factors influencing the implementation of other FLOSS patterns have been hypothesized based on the results.

The evaluation also put the proposed relationships to the test. The results mostly confirm the *alternative* relationships, but could not provide sufficient evidence for a statistical confirmation of the suggested *use* relations. However, the data showed a weak trend in favor of the proposed use relations, so possibly the data sample had been too small. Other possibilities for this negative result have also been discussed. The data also suggested two additional relations in the FLOSS pattern language and that FLOSS patterns in the category Architecture may often be implemented together.

Future research should try to identify FLOSS patterns that fill gaps in the proposed FLOSS pattern language. Newly identified FLOSS patterns can be classified into the proposed eight categories. Verification sections in FLOSS patterns can help readers to understand the FLOSS pattern and help implementers to verify whether their implementation had been successful. Researchers can use the verification sections for scientific evaluations of FLOSS patterns.

Evaluation on additional FLOSS projects can provide more evidence on the prevalence of FLOSS patterns. This article purposefully used different types of FLOSS projects to identify associations between the implementation probability of FLOSS patterns and characteristics of the FLOSS project. Still, more samples can cover additional characteristics of FLOSS projects. For example, further analysis could reveal which FLOSS patterns depend on the programming language or application domain of the FLOSS project.

Moreover, the data sampled from the five analyzed FLOSS projects was insufficient to show statistically whether the proposed *use* relations occur in practice. A larger sample size might help to accept or reject the proposed or other *use* relations.

Future research should observe FLOSS projects for new FLOSS patterns, especially those that ease the submission process and getting reviews for acceptance of submitted patches. Chapter 2 identified these as important submission barriers, but no published FLOSS patterns target them yet.

4 Wiki Development Environments

The last chapter showed established techniques for lowering contribution barriers in the pattern format. However, some contribution barriers still lack corresponding FLOSS patterns. For some contribution barriers, FLOSS patterns exist, but it is open whether there are other techniques to lower these contribution barriers even further.

This chapter presents the concept for a new type of system, a WikiDE. A WikiDE adapts the wiki concept to software development in order to lower contribution barriers. The chapter starts with a section of theoretical considerations about the WikiDE concept. The following two sections focus on two components of a WikiDE whose realization are especially challenging, but provide the opportunity to lower two important contribution barriers identified in Chapter 2. The chapter concludes with a documented PoC realization of a WikiDE.

4.1 WikiDE Concept

This section discusses the WikiDE concept and derives requirements that a WikiDE must fulfill. It also analyzes the current state of technology in regard to WikiDEs and how this technology fulfills the derived requirements. An adapted contribution process shows how a WikiDE may fulfill two seemingly contradicting requirements. The section closes with an architectural outline for the integration of the individual components.

4.1.1 Requirements¹

A WikiDE is a combination of two systems: A wiki system and a web-based IDE. A realization of a WikiDE must therefore fulfill the requirements for both types of systems.

Section 1.1.4 describes from a technical perspective how a newcomer joins a FLOSS project. This process starts with a process of programming, which consists of the three actions *Modify source code*, *Build the application*, and *Test application behavior*. The software developers repeat these three actions until the application works as expected. An IDE supports software developers in this process of programming. Each of the three actions involved in the process of programming results in a requirement for an IDE. An IDE therefore comprises at least three different subsystems to realize these requirements: A code editor enables software developers to edit the code, a build system builds the application executables, and a debugger allows the software developers to debug the application. [RC04; Har09, p. 5]

Wikis disperse constructive work over a high number of contributors. In this manner, each contributor needs to contribute only a little to still yield a considerable accumulated contribution. This works only if people can easily become users of the wiki project and users can easily become

¹A preliminary version of this section was published previously [GH12b].

contributors. Section 1.3.7 discusses the effects of lowering contribution barriers in FLOSS projects to wiki-like levels.

The authors of the Portland Pattern Repository's Wiki have also discussed requirements of a WikiDE [Aut12]. These requirements contain among other aspects extensions to a WikiDE that may lower contribution barriers but are not strictly necessary to realize a WikiDE. They also imply that the primary concern of a WikiDE would be to modify its own source code. This thesis does not impose general restrictions on the type of development project the WikiDE is used for. Consequently, their findings differ from the findings in this section.

Self-administration

If a central administration decides who can join a project as a contributor, the number of contributors can grow only with the speed the central administration can handle. For the exponential growth of the number of contributors seen in many wikis, the administration must grow with the contributor size. This is only possible if the contributors also handle the administration themselves.

Some wikis realize this with a policy that allows anonymous users to edit content. In this case, all users administer themselves. Users then decide on their own where to contribute. Other wikis require users to register before they can contribute. In this case, registration may be an automatic process that requires no administrative intervention. The system may check that new contributors have a valid email address. Corporate wikis may also integrate with a corporate user directory. In this case, the corporate wikis reuse the administration of the corporate user directory. These corporate wikis also require no extra administration.

A WikiDE may use any of these techniques to realize self-administration. The growth of an on-line programming site without self-administration is limited by the capacity of the administration.

No Installation

As pointed out in Section 2.9, setting up a development environment is currently one of the most important contribution barriers. Wiki contributors are not required to install any software on their computers. Instead, they modify the wiki content using their web browsers.

This requirement must hold true also for a WikiDE. The WikiDE must relieve newcomers from the various tasks required to prepare their computers for the software development project. Thus, a WikiDE must automate the first two actions in the contribution process described in Section 1.1.4: *Download the source code* and *Configure build environment*.

Code Editor

Writing the source code for software and writing text in a natural language are special cases of editing text. Editors for writing source code for software are referred to as *code editors*, while editors used to write natural language text are referred to as *word processors*. Word processors provide only basic meta information about the text like orthographic checks at the time of writing. Code editors on the other hand provide extensive syntax highlighting: Language keywords, types, numbers, identifier, and other types of terms are each displayed in different colors to improve

the software developers' understanding of the source code. Additionally, code editors highlight contextual information, from basic information like which opening and closing brackets match together, highlighting other occurrences of a currently edited variable, and API documentation of the method currently edited. [MP09]

Software and hence their source codes are written and maintained by humans. Additionally, a compiler or an interpreter must be able to automatically process the source code. Thus, source code needs to be human-readable and also machine-readable. Natural language only needs to be human-readable. It is therefore sensible for a code editor to show the human working on the source code how a machine understands this source code, i.e. provide features like syntax highlighting and suggestions for word completion. These features are necessary beyond the features of a word processor. Consequently, a WikiIDE needs a code editor that fulfills these additional requirements beyond the requirements of the text editors in other wikis.

Immediate Feedback

All wiki users are treated as contributors. Therefore, all users can contribute to any part of the wiki's artifact. This also implies that they can see the results of their edits immediately [Ebe+08, p. 11], without manual interaction with a group of wiki maintainers.

Most wikis offer a preview feature that allows the contributors to see the results of their edits before they are published to the other users of the wiki. With the help of the preview feature, contributors detect and correct errors that they have made while editing.

The process of programming with its three actions resembles this wiki editing process on a high level of abstraction. After the software developers have modified source code, the IDE creates the executables of the component. There are two purposes for the creation of the executables: First, the software developer tests whether the modifications in the source code actually cause the desired change in the component. Second, end users of the component need the executables to use the component. These two features of an IDE map to the preview feature of a wiki and the eventual publishing of the edited content.

For the first purpose, software developers run the component itself if it is an application or run a test application that uses the component if it is a library. Then they interact with the running application. While interacting with the running application, software developers may detect only those failures that crop up in the user interface. Software developers attach a debugger to the running application to detect failures that do not crop up immediately in the user interface. The position in the source code of the user interface at which a software developer first detects a failure is not necessarily the position of the causing defect. Instead, an underlying module may be defective. The underlying module brings the execution in an invalid state, but this becomes visible in the user interface only at a later point. Software developers also use a debugger to find the defective source code in the underlying module.

These two purposes to create executables also manifest in WikiIDEs: First, developers see whether their modifications do what they intended them to do only when they run an executable application that incorporates their modifications. Second, for other users to give feedback to modifications, the other users must be able to test these modifications. For these tests, they need to run an executable application that incorporates the modifications.

With only the executables but without a debugger, users can only test modifications with

sufficient impact on the user interface. Users need a debugger to find and fix defects deep within the component. If a WikiDE does not include a debugger, defects in the user interface may still be fixed. A build system is therefore a requirement, while an integrated debugger only increases a WikiDEs application range. A debugger is therefore an optional component of a WikiDE.

Version Control

Wikis usually store a list of historic versions for each web page. Each historic version stores the content, author, and date of its modifications. Users can access this information and revert changes back to a historic version. If contributors know that there are backups, they are more encouraged to change the content. They can rest assured that even if they make a fatal mistake, it will be easy to correct afterwards [LC01, p. 324].

In software development, an atomic modification to the source code may take hours of work and the modification may span days or weeks. This can happen when software developers restructure the architecture of the component or when a modification to one code file requires modifications to dependent code files. While the modification has started but is not yet finished, the behavior of the whole component may be different than intended or the component cannot even be built.

Historic versions of source codes are usually stored in a VCS. Because of the long time span an atomic modification may take and because a single syntactic error can prevent the whole component from building, there are two major differences between the way VCSs store historic versions of the source code and the way wikis store the historic versions of its pages.

First, wikis store historic versions per web page. A web page of a wiki corresponds to a code file, as both of these store exactly one uninterrupted text. However, VCSs aggregate code files into so called repositories. Every repository has a list of historic versions. Depending on the VCS in use, a repository may not only aggregate modifications to code file contents but also additions, deletions, or movements of whole code files. If a modification of the component requires modifications in multiple code files, this can be realized as one atomic modification to the repository. This avoids unstable intermediate versions, where one side of an interface has been changed but a resulting change to the other part is not yet stored in the VCS.

Second, VCSs store the historic versions of a repository in multiple dimensions: There can be multiple lists of historic versions of each repository. Each such list is called a branch. Each branch has a common ancestor with every other branch. Software developers can also merge multiple branches back to a single branch. Software developers may create a new branch if they start a major modification of the application, for example. The new branch enables the software developers to split up this major modification into several smaller modifications that are stored as historic versions of the new branch. The other software developers can still continue with their work since they are working with the original branch and ignore the new branch. Hence, the software developers working on the major modification do not need to care whether their smaller modifications render some of the interfaces temporarily unusable or even break the build. Thus, multiple dimensions of historic versions enable the software developers to use the advantages of historic versions on a more fine-grained level.

A WikiDE must provide historic versions of the source code in the way a VCS does. Without the two major features explained above that distinguish a VCS from a standard wiki version

history, a WikiDE does not scale to high numbers of contributors, as they will disrupt each other with their contributions.

However, this imposes an additional problem a WikiDE has to solve: As there is no single latest version for a wiki page, which version of the wiki page shall the WikiDE display to the user? A WikiDE must therefore also come up with a strategy to choose the version of a wiki page for displaying on every access to wiki content. Similarly, new versions cannot be just appended to the list of versions in all cases. Instead, the WikiDE must have a versioning strategy: Which modifications create a new branch and which branches shall merge back together?

Code Reviews

In software development projects and especially in FLOSS projects, code modifications may be subject to a code review before they are added to the primary development branch [CH06a]. Code modifications must (i) not be malevolent, (ii) meet the quality standards of the project, and (iii) solve the problems they have proposed to solve or add the feature they have proposed to add. Code reviews ensure that only those code modifications are added to the primary development branch that meet all of the three criteria.

Wiki contributions resemble code modifications in that they may also meet or not meet any of the three preceding criteria. Wikipedia calls contributions that validate the first criterion “vandalism” [P*Wik16c]. The second criterion is less important in wikis, as the quality standard can be improved by others later [P*Wik16a]. Contributions to wikis may also be checked for relevance, which is equivalent to the third criterion [P*Wik15c].

If a wiki contribution fails at one of the criteria, it is usually sufficient to revert the wiki page *after* the contribution that failed the criteria. Software modules however depend stronger on each other. For example, if a malevolent modification was temporarily accepted for inclusion in a component, builds of the whole component may fail until reversion. Even worse, the component may contain a virus that infects the computers of users of the component. A low quality code modification may decrease the whole application’s stability. Additions of the wrong features may decrease performance or usability.

It is therefore critical to review software modifications *before* they are included in the primary branch of the application’s VCS. This is also a requirement for WikiDEs, although it seemingly contradicts other requirements like *Immediate Feedback*.

IDE Compatibility

Desktop IDEs are sophisticated applications that support developers in a broad range of development activities. The Eclipse IDE without any programming language support, for example, has a code base of 2 521 174 lines of source code at 2016-03-09 and Open Hub estimates about USD 40 million of development cost for the Eclipse IDE [P*Ope16a]. A WikiDE must fulfill the requirements of a wiki in addition to the requirements of an IDE and may therefore be less sophisticated than a desktop IDE. There is no general restriction that keeps a WikiDE from becoming as sophisticated as a desktop IDE, but this may take years of development. In the meantime, desktop IDEs are generally more powerful tools for software development than WikiDEs.

However, a WikiDE has advantages over desktop IDEs: It allows software developers to join a software development project more easily. For senior project contributors, this is not an advantage for their personal development experience, as they have already joined the project. They will therefore likely prefer a desktop IDE over a WikiDE for the reasons mentioned in the previous paragraph. These senior project contributors are essential for ongoing growth of the project [KK10].

Therefore and at least until WikiDEs have matured to the level of desktop IDEs, software projects should not rely on WikiDEs exclusively. A WikiDE system must therefore be compatible to other IDEs. If desktop IDEs can access the source code through the VCS of the WikiDE, the requirement is fulfilled. Compatibility is further increased if the WikiDE shares a project file format with common desktop IDEs.

4.1.2 Existing Wiki Systems for Programming²

This section gives an overview of existing systems that fulfill some of the requirements described in Section 4.1.1 but lack support of others. Table 4.1 lists the systems analyzed in this section and shows which of the requirements they fulfill. Entries in the column *Code Editor* show *Partial* if the wiki system allows editing the source code but does not provide any extended features like syntax highlighting. Entries in the column *Immediate Feedback* show *Partial* if the wiki system builds application executables but provides no debugger. If the system provides a debugger, the column contains *Yes*.

Wikimatrix.org listed 140 different **wiki applications** as of 2016-03-09 [Cos16]. These wiki applications differ from each other among other things in their user interfaces, their distribution and editing licenses, the databases supported for storing their content, and in the programming languages used to develop the wiki application. Some of these wikis use a VCS as their backend and therefore fulfill the requirement *IDE Compatibility*. Even those using a VCS as their backend do not satisfy the requirement *Version Control*, though: Olelo [P*Men14] is an example for a wiki with a git [P*git16] backend. However, Olelo does not use the two additional features of a VCS required to fulfill the requirement *Version Control*. Olelo provides only a simplified view on the VCS version history, probably in order to maintain compatibility to standard wiki GUIs.

Some of the wiki applications allow their users to create not only human-readable content but also executable scripts within the wiki. Wiki applications with this feature are called **Structured Wikis**. Structured Wikis execute user-created scripts to process wiki content, for example in order to generate statistics. Because users create these scripts as they need them, these scripts are situational applications [Shi04]. Examples for Structured Wikis are XWiki [P*XWi16], TWiki [P*Tc16], and MediaWiki [P*Wik15a]. Situational applications do not require *Code Reviews*, as their primary users are also their creators. Also, the source code editing feature is restricted to the programming languages of the scripts. The scripts are executed only within the wiki, so common-purpose application development is not possible.

Wiki Business Query [AR08], **Programming Wiki** [HW09], and **adessowiki** [Lot+09] are wiki systems that allow source code as content within wiki pages. This source code executes on the server side and displays its results within the wiki page. Intended content comprises figures

²A preliminary version of this section was published previously [GH12b].

Table 4.1: Wiki systems that fulfill part of the WikiDE requirements

WikiDE candidate	<i>Self-administration</i>	<i>No Installation</i>	<i>Code Editor</i>	<i>Immediate Feedback</i>	<i>Version Control</i>	<i>Code Reviews</i>	<i>IDE Compatibility</i>
Wiki applications	Yes	Yes	No	No	No	No	Yes
Structured Wikis	Yes	Yes	Yes	Yes	No	No	No
Wiki Business Query	Yes	Yes	Partial	Partial	No	No	No
Programming Wiki	Yes	Yes	Partial	Partial	No	No	No
Adessowiki	Yes	Yes	Partial	Partial	No	No	No
mHub	No	Yes	Partial	No	No	No	No
Cloud IDEs	No	Yes	Yes	Yes	Yes	No	No
XSDoc	Yes	Yes	No	No	No	No	Yes
Galaxy Wiki	Yes	Yes	Yes	Yes	No	No	Yes
XYLUS	No	Yes	Partial	Yes	No	No	Yes
CrowdCode	Yes	Yes	Yes	No	No	No	No
Software Forges	Yes	Yes	Yes	No	Yes	Yes	Yes
Desktop toolset	No	No	Yes	Yes	Yes	Yes	Yes

and tables. The code can access data from databases or from other wiki pages. The wikis are intended to teach programming concepts to its users or for EUP. The wiki systems therefore do not support generic programming languages and are no WikiDEs in that they do not fulfill the requirements of *Version Control*, *Code Reviews*, and *IDE Compatibility*.

Sauer [Sau11] described the system **mHub** whose concept extends the concept of a Structured Wiki. In contrast to Structured Wikis, the applications developed with mHub are not situational applications. Instead, these applications work as a middleware between an Enterprise Resource Planning (ERP) and the production machines of a factory. The concept does not explain how to authorize users for modification, therefore mHub does not fulfill the requirements *Self-administration* and *Code Reviews*. Sauer does not describe how code editing works in detail. Thus, the requirement *Code Editor* is considered partially fulfilled. The concept of mHub does not provide solutions for the requirements *Immediate Feedback*, *Version Control*, and *IDE Compatibility*.

Cloud IDEs are code editors that run in the browser. Examples for cloud IDEs are Cloud9 IDE [P*Clo16], Koding [P*Kod16], and CloudPebble [P*Peb15]. The existing cloud IDEs are restricted to certain programming languages in order to provide the three core features of an IDE explained in Section 4.1.1, namely a code editor, a build system, and a debugger. For example, Cloud9 uses the browser's ability to execute JavaScript in order to provide a debugging environment for JavaScript. Other programming languages can be edited, but they cannot be executed or debugged. CloudPebble specializes on apps for the smartwatch Pebble. The debugger emulates a Pebble. A Pebble is simpler than a smart phone or a desktop computer, considering for example that Pebble apps are only allowed 256 kB of data [P*Peb16]. Thus, emulating other target platforms than a smartwatch may bring additional challenges that CloudPebble did not have to face. Hence, the concept of cloud IDEs does not solve the problem of how to integrate the build of application executables into a web-based IDE. The concept cannot be generalized to most other types of application development. Some existing cloud IDEs satisfy the requirement for a *Code Editor* in a WikiIDE for all kinds of application development, though.

Leaving the restrictions discussed above aside, the existing cloud IDEs are not wikis. They use multi tenancy, so the software developers using the system see only their own files. The project owners must actively invite other software developers before they can contribute to the project. Once a software developer joined the project, the software developer can change everything without preceding review. Thus, cloud IDEs do not fulfill the requirement of *Code Reviews*. They also do not fulfill *Self-administration* for the same reasons.

Aguiar and David [AD05] developed **XSDoc**, a wiki system that weaves source code and documentation together. Using a web service interface, desktop IDEs can access this content and thereby edit the source code in the wiki. XSDoc focuses on the integration of source code and documentation and not on lowering contribution barriers. For source code edits, the whole technical environment like an IDE and a build system still have to be installed.

Xiao et al. [XCY07] developed **Galaxy Wiki**, a wiki that integrates source code and documentation with an IDE. With Galaxy Wiki, software developers can edit Java source code in the wiki with a web browser, they can also compile the application and execute it. Galaxy Wiki also integrates a debugger, which executes as a Java applet within the browser. Thus, Galaxy Wiki shows that a WikiIDE can realize the requirements of a *Code Editor* and *Immediate Feedback*. The debugger concept of Galaxy Wiki cannot be used for most other languages, though, since most

languages do not execute within a standard web browser. Additionally, only Java applications for which the restricted Java environment within the browser context suffices can be debugged reasonably. Galaxy Wiki is based on the wiki system MoinMoin [P*Moi15], so the history of each source code file is only a single list, which does not fulfill the requirement of *Version Control*. Also, pre-commit code reviews are not possible, so Galaxy Wiki does not solve the problem how to fulfill the *Code Reviews* requirement and at the same time fulfill the apparently contradictory requirements of *Self-administration* and *Immediate Feedback*. By providing the source code via a web service interface, Galaxy Wiki maintains desktop *IDE Compatibility*.

Abbasi et al. [ASP10] designed the web application **XYLUS** as a cloud IDE for C# desktop applications. XYLUS is realized with Asynchronous JavaScript and XML (AJAX). Therefore, XYLUS can be used with standard web browsers and is platform independent. After a developer using XYLUS has modified the source code of an application, XYLUS builds the application on the server and starts a VM. The modified application is then loaded within the VM. The web application opens a Remote Desktop Protocol (RDP) session and displays the VM's screen in the web browser. Keyboard and mouse input is redirected from the web browser to the VM through RDP, so the developer can debug the modified application running in the VM. In contrast to the debugger of Galaxy Wiki, this debugger concept applies also for languages that do not run within the browser itself. In their paper, Abbasi et al. do not explain where XYLUS stores the source code of the component and how changes to the source code are integrated into the store of the source code. Therefore, their concept provides no solution for the requirements of *Version Control* and *Code Reviews*. There is also no explanation on how the system's users are administered. The concept also does not explain if and how XYLUS supports syntax highlighting.

LaToza et al. [LaT+14] also approached the goal of having many contributors to a software development project with **CrowdCode**. The method is quite different to those previously discussed, though, and its row in Table 4.1 reflects the features of CrowdCode only superficially. The system automatically generates tasks from the existing set of tasks and dispatches them to the developers participating in the system. For example, new code automatically spawns a task to specify and eventually write a test case. Developers may call functions that do not exist yet, in which case the system spawns a task to create the function. The system works with *Self-administration* and includes a *Code Editor*. It is no wiki in the sense that a user consumes content and may edit the same content with *Immediate Feedback*.

Software forges provide a set of typical software development tools, so new projects do not have to set up all of these tools individually. Furthermore, software forge services like GitHub [P*Git15c], GitLab [P*Git16c], and SourceForge [P*Sla15] also host these software development tools, so a new FLOSS project may use all of these tools without configuration effort. This standard set of software development tools reduce the effort to work on the FLOSS project. Riehle et al. [Rie+09] go even further and state that “software forges are geared toward making it as easy as possible to find a project, understand it, and contribute to it.” – this also summarizes the goal of wikis.

The development tools of a software forge include a VCS repository, typically with access via the web site. GitHub and GitLab include a web-based editor for source code. Per default, users cannot modify code in software repositories other than their own. This does not create a contribution barrier, though: If users try to modify source code in a software repository other than their own, the whole repository is branched and re-created as their own software repository. The

maintainers of the original project can merge modifications in branches back into the original project. This enables *Code Reviews* before merging the modifications back. They do not fulfill the requirement *Immediate Feedback*, though.

Table 4.1 also lists a **desktop toolset** for comparison, referring to a traditional development toolset consisting of a desktop IDE, a VCS, and an issue tracker. The desktop IDE fulfills the requirements of a *Code Editor* and with their integrated debuggers, also the requirement of *Immediate Feedback*. The VCS satisfies the requirement of *Version Control* per definition. The issue tracker fulfills the requirement of *Code Reviews*. Depending on the language and the IDE, compatibility to other IDEs is also possible, although this may require an additional overhead to store the application metadata in multiple formats for multiple IDEs. A desktop IDE is not a wiki, though, and requires installation of tools and downloading the application's source code.

As shown in Table 4.1, no system currently provides a solution on how to integrate *Code Reviews* into a wiki system. Using the method for *Code Reviews* from the desktop toolset or via GitHub prevents *Immediate Feedback* in a wiki and is therefore no solution.

4.1.3 WikiDE Contribution Process³

This section shows how a WikiDE can realize the requirements *Self-administration*, *Immediate Feedback*, and *Code Reviews* at the same time. A modification of the joining process described in Section 1.1.4 is used to fulfill the seemingly contradictory requirements.

Wikis most often rely on post-commit reviews. Examples are the English, French, and Italian Wikipedias [P*Wik16d]: Even anonymous visitors to these wikis can change the texts and everybody else sees the changes immediately. All users can undo this change. This way, everybody can review any change after it was committed. Sometimes two parties alternately revert each other's changes. This is called an edit war [P*Wik16b]. Edit wars need to be resolved by administrators, who are wiki users with additional rights. Most often, administrators are not necessary in the review process: There are around three to five percent double reverts per revert in the English Wikipedia [Bur+06].

Wikipedias in some other languages, for example the German and the Russian Wikipedia, use a concept called flagged versions [P*Wik15b]. In the flagged versions concept, every change may or may not be flagged. Users with a special reviewer right can flag or un-flag any change. Anonymous visitors of the wikipedia web site see only flagged changes of articles, while authenticated users can configure whether they want to see unflagged changes. Users of these Wikipedias are automatically given the special reviewer right if they fulfill certain criteria like being authenticated and having made at least 300 edits in the past. Changes made by these users are automatically flagged and visible to all others. If users without the special review right, for example anonymous users, change an article, their changes are not immediately visible to most other users, as their changes are unflagged. Only after a user with the special reviewer right flags the change, it will become visible to everybody else. The concept of flagged versions is intended to reduce vandalism [P*Wik16c]. Flagged versions remove the incentive of vandalism, as the change will not be visible to the general public. Flagged versions therefore provide a method to realize *Immediate Feedback* while at the same time establishing a process for *Code Reviews*.

³A preliminary version of this section was published previously [GH12b].

Using the concept of flagged versions for a WikiDE allows *Immediate Feedback* for newcomers while the core developers can still perform *Code Reviews* before integrating newcomers' modifications. On the other hand, the flagged versions concept must be adapted to VCSs before it can be used in software development.

GitHub [P*Git15c] and GitLab [P*Git16c] already use a branching mechanism that realizes some aspects of the flagged versions concept. As described in Section 4.1.2, newcomers can branch a public project into their own repository with a single click. They can thereby modify the source code without prior review by the core developers of the original component. This is isomorphic to the flagged versions concept, if modifications in the original repository are considered flagged and modifications in other repositories are considered unflagged. Most users see only the modifications in the original repository. All users can create unflagged modifications in their own repositories. It requires a special review right to flag a modification or create a flagged change from scratch: The user needs modification rights on the original repository.

However, in order to test modifications, software developers still have to set up a local git client, download the source code to their machines, install and configure a build environment, and build the executable. Therefore, the FLOSS contribution process depicted in Figure 1.3 is still valid when using software forges with a code editor and review system like GitHub. The web-based editor can only be used if the modification is not to be tested, for example if the modification only affects the documentation.

Thus, further adaption of the flagged versions concept is required. The UML activity diagram in Figure 4.1 shows how to adapt the original FLOSS contribution process described in Section 1.1.4 with the flagged versions concept of wikis. The diagram shows how a newcomer modifies source code and how the source code is merged into the primary branch of the VCS. Newcomers modify the source code and test the modified application in their own context within the WikiDE, so they can prepare an atomic modification package. When they are finished, the modifications are stored as private modification package, yet invisible to other software developers using the WikiDE. When newcomers decide to publish their private modification packages, they become public modification packages. Other software developers can review these public modification packages and the core developers can decide to include the modification into the main branch of the application, which is equivalent to flagging the change in the Wikipedia concept of flagged versions.

This contribution process solves the problem of how to fulfill the requirements of *Self-administration*, *Immediate Feedback* and *Code Reviews* at the same time. With this new process, WikiDEs relieve newcomers from the major activities *Build the application* and *Submit a patch* of the process in Section 1.1.4. As concluded in Section 2.9, these activities have been shown to be especially frustrating and important contribution barriers, so WikiDEs using the adapted process lower these contribution barriers.

4.1.4 Architecture⁴

This section describes the components necessary to realize a WikiDE that fulfills the requirements listed in Section 4.1.1. Each of the following subsections describes one component along with

⁴A preliminary version of this section was published previously [GH12a].

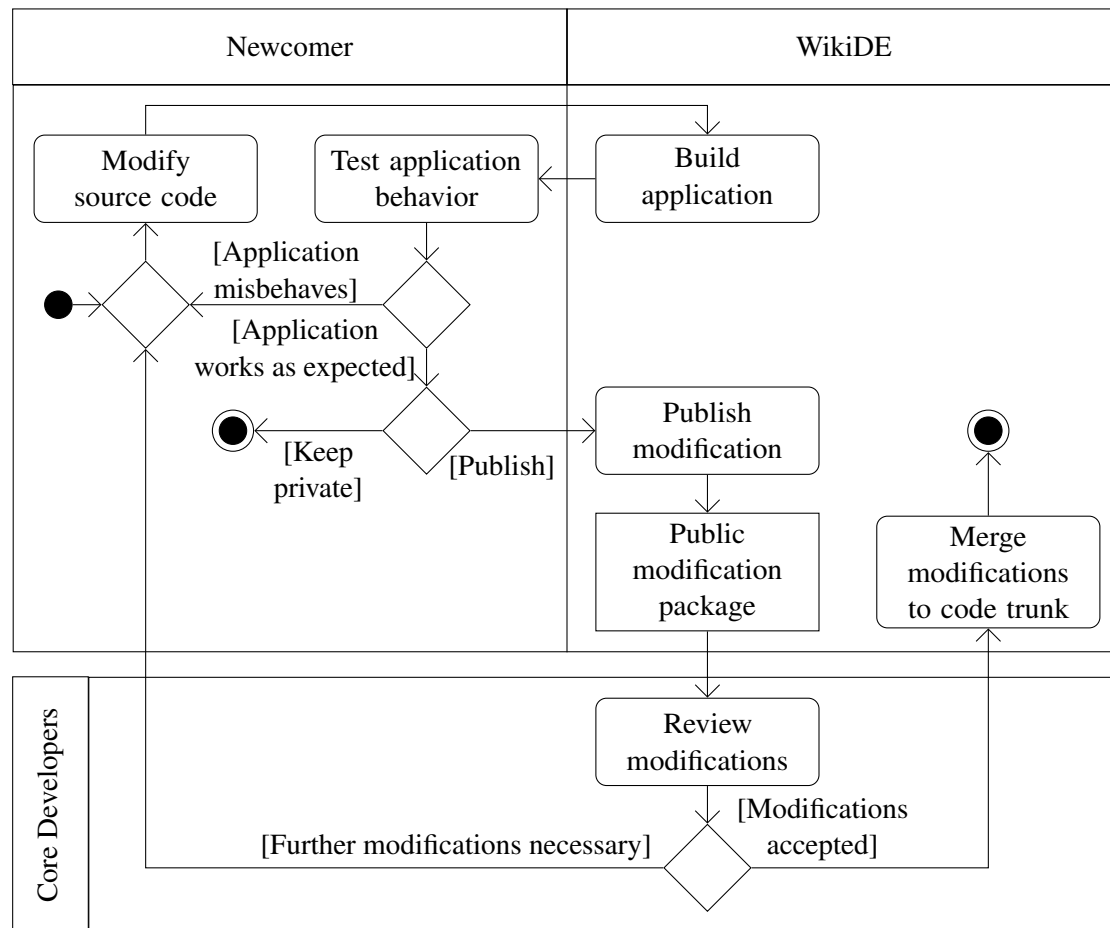


Figure 4.1: Activity diagram of a code modification by a newcomer with the help of a WikiDE

references to existing software for its realization.

There are different Wiki systems with textual content that differ in purpose and features. Accordingly, different WikiDEs may differ in purpose and features. Thus, for most of the WikiDE components, there is not just one possible realization. Therefore, each component can be realized with the help of multiple different software solutions, but this analysis does not evaluate which software is the best to realize a particular WikiDE component.

Figure 4.2 shows a UML component diagram of a possible WikiDE architecture. The WikiDE system is a wiki and therefore a web application. The WikiDE system interacts with the software developers via a standard web browser. This standard web browser runs a web-based code editor. The web-based code editor communicates with the web server via a web service to load and save source code as well as load application executables for the software developers to test. On the server side, the web server uses a VCS as the backend to load and save the source code. An additional database stores information about the software developers who have created an account in the WikiDE. This information consists of login data, personal preferences for the user interface, and a configuration of the target environment the software developers want to develop their applications for. The final component is a build system that builds the application executables from the given source code. The software developers can download these application executables to test their changes.

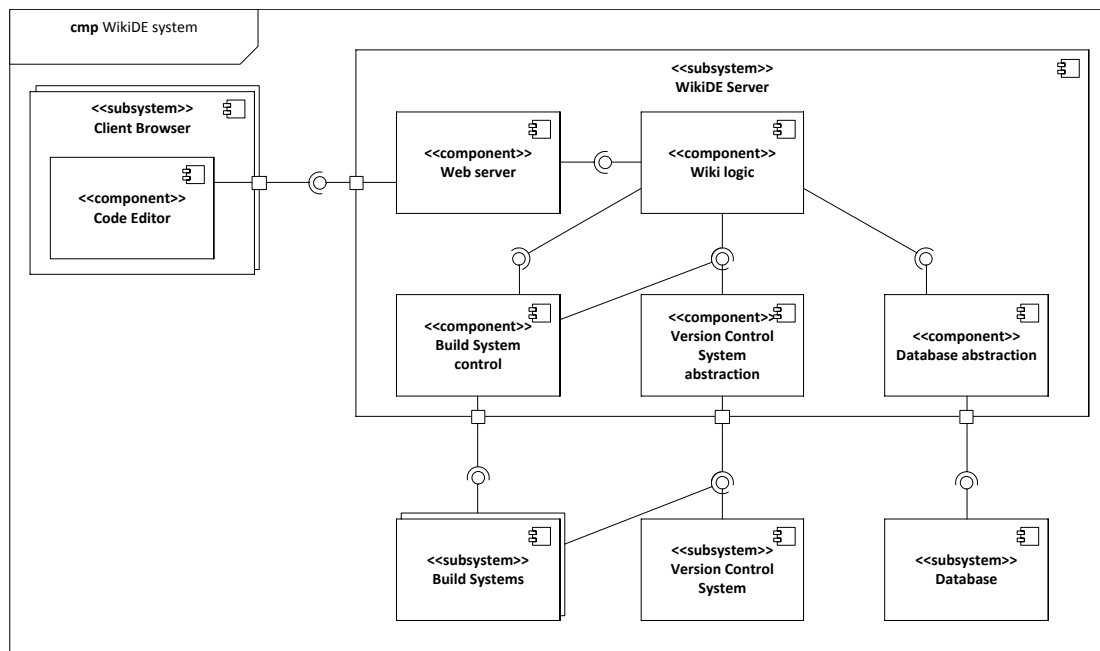


Figure 4.2: Proposed Architecture of a WikiDE

Wiki Systems

Section 4.1.2 discussed the features of wiki applications. As shown, both wiki applications and Structured Wikis lack essential WikiDE requirements. Wiki systems may serve as the

foundation for a WikiDE. Other components have to be integrated in the wiki system to satisfy the requirements for a WikiDE.

Web-based Code Editors

Section 4.1.2 described that cloud IDEs let users write code in a web browser and store or execute it on a server. As shown in Figure 4.2, a web-based code editor and therefore a cloud IDE is an essential part of the WikiDE architecture.

Build Systems

A desktop IDE can create debug builds as well as release builds. When debugging the application, the IDE creates a debug build and attaches its debugger to the application executables. Even in a traditional development setting, often dedicated build servers create the release builds. Typically, these dedicated build servers are part of a CI system.

A CI system creates a build for every change in the source code. This uncovers incompatibilities between different software modules quickly, as the CI system shows an error message immediately after the first change that causes the build to fail. Examples for CI systems are Jenkins [P*Jen15] and CruiseControl [P*Cru15].

From the perspective of a cloud IDE, build systems and especially CI systems provide an abstraction for the build process as opposed to executing the build steps manually, e.g. calling the compiler and linker directly from the cloud IDE. Building the application executables with the help of a CI system has two advantages over executing the build steps directly: First, existing CI systems support various build configurations like programming languages and target platforms already. If a cloud IDE executed the build steps manually, the software developers of the cloud IDE would have to invest time for each supported build configuration. Second, CI systems like Jenkins and CruiseControl already provide a web interface to control the build process. Software developers using the cloud IDE use the browser to access both the cloud IDE as well as the CI system.

The proposed WikiDE therefore uses a CI to build the application executables, but the CI is controlled by the WikiDE logic instead of the users directly. When WikiDE users modify the source code, the CI provides executable versions of the modified component immediately. This satisfies the *Immediate Feedback* requirement.

Debuggers

There are two machines involved in the debugging process. The first is the *development machine*, which hosts the source code and displays the current state of the application. The other is the *test machine* that actually executes the application. A developer using a desktop IDE usually runs the application on the same machine that runs the IDE, so the two machines are identical. The WikiDE web interface displays the source code, allows editing, and consequently should display the application's state when debugging the application. Thus, the web server running the WikiDE web application is the development machine. This web server might run on a platform different to the intended target platform of the application. At least in this case, the development machine must be different to the test machine.

The architecture of the WikiIDE and the debugging scenario differs depending on where the test machine is placed. There are two possibilities for the placement of the test machine: First, the developers provide test machines themselves, possibly the same machines they are running their web browsers on. In this case, the developers must execute a remote debugger on the test machines in addition to the application itself. The remote debugger transfers the application's state to the development machine, where it is displayed via the WikiIDE web interface. The developers interact with the application directly. Thus, the test machine is placed on the client side, the developer is responsible for its operation. Second, the WikiIDE provides a test machine on the server side. The test machine's input and output are redirected to the WikiIDE web interface. The developer interacts with the test machine via the WikiIDE web interface. If the application has a GUI, the test machine's display output can be shown using a web-based remote desktop application like TightVNC Java Viewer [P*Tig09] or even an AJAX-based RDP client like AJAX Remote Desktop Viewer [P*Dam06]. This realizes a debugger for the *Immediate Feedback* requirement. Abbasi et al. [ASP10] have shown the feasibility of this approach with their tool XYLUS.

Version Control Systems

Distributed VCSs like git [P*git16] and hg [P*Mer16] create a new branch of versions every time a new development machine downloads the source code. They merge two branches of versions every time a change set is uploaded to a server hosting a distributed VCS [OSu09]. Therefore, multi-dimensional version lists become more common with the advent of distributed VCSs.

Using the one dimensional list of historic versions provided by most wiki systems for a WikiIDE contradicts the requirement *Version Control System*. Using the version history of a wiki system is also not compatible with most existing VCSs, so it is more difficult to use a WikiIDE and a desktop IDE in parallel. As described in Section 4.1.2, some wiki applications use a VCS as backend and therefore fulfill the *IDE Compatibility* requirement. They still have a one-dimensional version history, so additional changes to such a wiki application are required before it fulfills the requirement *Version Control*.

The software forge services GitHub [P*Git15c] and GitLab [P*Git16c] provide web-based access to its hosted git VCSs. They automatically fork repositories into a user's space on modifications. This can be seen as a step towards the realization of the contribution process described in Section 4.1.3.

Issue Trackers

Code reviews can be realized with an issue tracker or with a specialized review system. One possibility to realize a review process with an issue tracker is the following: Users and software developers of an application can create tickets in the issue tracker. These tickets describe defects or feature requests. Software developers can then submit patches containing source code modifications and attach these patch files to the corresponding tickets. If core developers positively review the patch files, they will be committed to the main VCS branch. Examples for issue trackers are Mozilla Bugzilla [P*Moz01] and Atlassian JIRA [P*Atl16].

Examples of systems specialized for reviews are Gerrit Code Review [P*Goo16] and Review

Board [P*Bea16]. These systems automatically create tickets and attach patch files and are therefore more comfortable to use than a standard issue tracker, but the server side review process is the same.

4.2 CI Services⁵

Over the course of the last 15 years, CI has become an important practice in software development. Developers applying this practice commit their code modifications to the main code repository at least every day. Every modification that compiles and passes the tests should be committed. A dedicated integration machine continuously compiles and tests the current versions of the source code. This way, integration problems are detected early. Thus, integration problems are easier resolved as they do not have time to grow. [Fow06; DMG07]

Continuous Delivery is an extension to CI. In the Continuous Delivery approach, the CI system builds the release versions of the software products that will be shipped to the customers [HF10]. Current trends push this practice even further and propose to integrate even uncommitted code modifications [GS12].

However, in the case of FLOSS, the advent of CI did not have a measurable impact on development practices [DR08b]. Antoniadou et al. did not list CI systems as common tools used in FLOSS development [Ant+08]. This is one of the indications that CI was not a common practice in FLOSS development. This seems surprising at first, since both CI and FLOSS development are commonly associated with agile software development [MWA12].

Hundreds of thousands of FLOSS projects do not maintain their own infrastructure and rely on software forge services like SourceForge [P*Sla15] and GitHub [P*Git15c]. These software forge services provide a broad range of software development tools like VCSs, issue trackers, and web site and download hosting. They do not usually provide CI systems, though. Therefore, maintainers of FLOSS projects have to maintain CI systems on their own servers if they want to use CI. Small FLOSS projects may rely completely on software forge services for their server-based tools. Thus, setting up and maintaining a CI system on their own servers is a major effort. This is presumably a major fact restraining the spread of CI among FLOSS projects.

Consequently, large FLOSS projects that maintain their own infrastructure anyway hesitate less to use CI: For example, Mozilla [P*Moz15t] uses CI techniques like automated testing [P*Moz15i]. These CI systems integrate and test modifications committed to the FLOSS project's VCS repositories automatically. However, even for those large FLOSS projects, CI is restricted to core developers with access to the project's VCS repositories.

After Deshpande's and Riehle's study about lacking impact of CI on FLOSS [DR08b], a couple of organizations started offering CI as a service in the cloud. Examples include Travis CI [P*Tra15] and Circle CI [P*Cir15a]. Bugayenko [Bug15] sampled a more comprehensive list of current CI services. Some of these services offer free usage plans to FLOSS projects and even integrate into software forge services like GitHub. Similarly to the CI systems of large FLOSS projects like Mozilla, only the core developers of each FLOSS project may use its CI systems. Tools like Homu [P*Lee15] allow builds of newcomer's modifications, but only after they have been submitted, so the target audience are still core developers.

⁵A preliminary version of this section was published previously [GHJ13].

A *multitenant CI system* is a CI system that (i) hosts multiple projects and (ii) has multiple users each of whom shall access only some of the projects. These users are called tenants of the multitenant CI system. Such a multitenant CI system is the core of every public CI service. As argued in Section 4.1.4, a multitenant CI system is also an integral part of a WikiDE architecture and is required to realize the WikiDE contribution process described in Section 4.1.3. Furthermore, a lack of public CI services makes it harder to realize the FLOSS pattern `UNIT TESTS FOR CONTRIBUTORS` described in Section 3.3.3. Its widespread adoption would lower one of the most important contribution barriers identified in Chapter 2, setting up the development environment.

This section starts with a discussion of what currently limits the widespread adoption of CI practices in FLOSS projects. Two major reasons are identified: Resource cost and security vulnerabilities specific to multitenant CI systems. The remaining section focuses on the second problem of public CI services, the security vulnerabilities in multitenant CI system and how to counter them.

4.2.1 Current Limits of FLOSS CI

This subsection discusses two reasons that limit the widespread usage of CI systems among FLOSS projects, resource cost and security.

Resource Cost

First, CI systems need more computational resources than other software development tools. This is because CI systems do not only download every modification, but also have to compile all new versions of the source code and have to run a possibly large array of tests. These operations constitute a build job. Each committed source code modification of a project leads to a new build job in the CI system.

According to Moore's Law [Moo75], transistor count doubles every two years and therefore computing performance doubles every 18 months [Und04]. Assuming that computing performance is the dominating factor of cost for resource requirements of a build job, Moore's Law reduces the cost of each build exponentially over time.

However, the total number of FLOSS source code lines had doubled every 15 months in the past [DR08c]. The number of added FLOSS source code lines and similar measures double at the same rate, as they are derivations of the number of lines and derivations of exponential functions are exponential functions with the same base again. Therefore, no matter what specific measure accounts most for the cost of CI maintenance, the described measure may be used to estimate the growth of computing performance required to provide CI services to all FLOSS projects. Hence, this required computing performance also doubles every 15 months.

$C(t)$ shall be the cost to provide CI services to all FLOSS projects at a specific date, where t is the number of months lapsed since some fixed start date until that specific date. $C(t)$ is relative to the cost at the fixed start date and therefore has no unit. $\rho(t)$ is the computing performance required to provide CI services to all FLOSS projects at the specific date, and $\alpha(t)$ is the computing performance available for a constant amount of money. Taking the facts about the growth of computing performance and FLOSS growth together, $C(t)$ adheres to the following equation:

$$C(t) = \frac{\rho(t)}{\alpha(t)} = \frac{2^{\frac{t}{15}}}{2^{\frac{t}{18}}} = 2^{\frac{t}{15} - \frac{t}{18}} = 2^{\frac{t}{90}}$$

This corresponds to a yearly cost increase of 9.7 percent. It is therefore possible that free usage plans of CI services for FLOSS projects will decline for economical reasons. The same argument still applies if CI services shall be provided only to a constant fraction of FLOSS projects, as this would not change the rate of cost changes.

One might question the assumption that computing power is the dominating factor of cost for resource requirements of a build job. In this case, the general argument still applies, as variations of Moore's Law also apply to related resources. For example, hard disk space grows at a similar rate as computational power, with data density doubling about every two years [KK09]. This is still exponentially, but slower than computing power, so the yearly cost increase would be even higher.

Accordingly, the same reasoning also applies to other software development tools like VCS. Assuming that the cost of VCSs is largely determined by hard disk prices, the cost of VCSs for all FLOSS projects increases at a yearly rate of about 23 percent. Thus, free usage plans for these services on software forges may become economically unfeasible in the future.

However, it is unknown for how long the exponential growth of FLOSS sustains. The growth may become subexponential before the economical wall for software development described above is reached. Additionally, with the growth of FLOSS in terms of source code, public interest in FLOSS also grows. This eventually results in increased funding for FLOSS projects and their maintenance. This may even out the increased cost of maintenance.

Security of Public CI Systems

Second, CI systems are more vulnerable to attacks and misconfiguration than the software development tools currently offered by software forges: In contrast to the other software development tools, build jobs require the CI systems to execute code that developers of the FLOSS project provide. This custom code may cause errors starting from unnecessary resource consumption over CI system outages to espionage, denial-of-service-attacks, virus bridgeheads, and similar malicious purposes.

The situation of multitenant systems is even worse, as one malicious tenant may attack other tenants via the CI system. If future CI services want to provide CI also for newcomers instead of only the core developers of each FLOSS project, it would be even easier for a malicious attacker to gain access to the CI systems.

There is a particularly dangerous example of an attack on CI systems, a variation of the trusting trust attack on compilers [Tho84]: An attacker could modify the CI system in a way such that software artifacts produced on those system would be infected with a virus. For Continuous Delivery systems, infected versions of the software product would spread to end users. These infections are insidious, as even a paranoid end user who carefully reviews the software product's source code would miss the malicious code parts that the virus infection induces. Eventually, the infected CI system would build a new infected version of its own software that hides all traces that an infection has ever happened, while still continuing to infect its own and other software.

4.2.2 Security Analysis of CI Systems

This subsection systematically analyzes attack vectors that malicious tenants of a multitenant CI system may use to attack the CI system. Of course, all tenants may easily compromise their own build on the CI system and potentially build malicious software with the help of the CI system. However, this is not a security risk, as these malicious tenants can also build malicious software without a public CI system. Thus, behavior is considered an attack only if the behavior either damages the CI system as a whole or compromises builds of other tenants.

The security analysis describes general attack vectors of CI systems, in contrast to actual flaws of specific implementations of CI systems. Thus, the attacks described cannot directly be used to attack a concrete CI system without modification. An attack on a concrete CI system must use one of the attack vectors determined in this subsection, though, but the attack still has to exploit a flaw in the implementation or configuration of the CI system.

As explained, the analysis applies to CI systems in general. For reasons of clarity, the CI system Jenkins [P*Jen15] serves as an example and guides the line of argumentation. The tool `jenkins-stats` counts 113 185 Jenkins installations as of 2016-02 [Bar16]. Hence, Jenkins is a common representative of a CI system. The concepts explained are common among implementations of CI systems, although the wording may be different for other specific implementations.

Langweg and Sneekenes [LS04] provide a taxonomy of attacks on software applications. Earlier taxonomies of attacks [Lan+94] focus on whole OSs. They argue that every attack on a software application has a location, a cause, and an impact: The attack needs an attack vector through which the data are transmitted to the target software application, which they call location. Further, the attack must exploit a fault in the software or configuration, which they call the cause of the attack. Finally, the attack allows unauthorized operations that defy one or more of the security attributes confidentiality, integrity, and availability [Avi+04]. They call these unauthorized operations the impact of the attack.

The analysis in this subsection systematically examines the data inputs to CI systems. The analysis checks which data inputs can serve as attack vectors. In Langweg's and Sneekenes's taxonomy, these attack vectors are locations of attacks. For each attack vector, the possible impact of the attack is discussed. As described above, Langweg and Sneekenes assign attacks a third property, the cause. However, causes are specific to the implementation of a CI system and therefore, the analysis in this subsection looks at causes only as far as they are relevant for CI systems in general.

Attacks Using the Web Interface

Even in a CI system with only a single tenant, there may be several projects configured. These projects define among other things which source code should be loaded and built, which events trigger a new build job, and what artifacts shall be archived after each build job. Different projects may depend on each other and form a hierarchy.

Users of multitenant CI systems can access their own projects but not the projects of others. CI systems may allow a much more fine-grained configuration of access rights on a sub-project level, for example Hudson [P*Pra10b] and Jenkins [Sma11].

Security flaws in CI web interfaces have proven to be usable attack vectors [Sec12b; Sec12a].

Web services may also open additional attack vectors. However, these are problems of web applications in general and are not specific for CI systems. Therefore, this class of problems will not be analyzed more deeply in this thesis.

Attacks Using the Build Process

Executing a build job in a CI system comprises the four steps

1. **VCS Checkout**, the build server downloads the source code from the VCS,
2. **Build Preparations**, the build server sets up the build environment,
3. **Builder Runs**, the build server compiles the source code to produce build artifacts, and
4. **Notification**, the build server notifies the master server about the results of the build.

The concept of these steps borrows from Prakash's presentation of build steps in the CI system Hudson [P*Pra10a]. There are differences in Prakash's build steps and the build steps shown here, as Prakash's build steps describe a build job from a software architect's perspective instead of a software security analyst's.

The build server does not execute these steps in strict order. Instead, some steps overlap. For example, *Notification* starts with the *VCS Checkout* already. Each step introduces possible attack vector on the CI system. The following paragraphs discuss the attack vectors for each of these steps.

VCS Checkout In this step, the build server downloads the latest version of the source code from the configured VCS. The downloaded source code usually also contains build scripts like Makefiles [P*Fre14], shell scripts, or Visual Studio project files [P*Mic16a]. These files are not actually compiled into the produced build artifact to be delivered to the end users. Instead, they support the build process with tasks like pre-processing the source code to target a specific execution platform or distinguishing between wanted and unwanted optional modules for a specific build. This step involves two possible attacks already.

First, a VCS may link to external resources. The VCS Apache Subversion (SVN) [P*The15b] is an example: A directory under source control may contain the property `svn:externals` to specify additional repositories to include in the checkout [CFP11, Chapter 3]. As another example, the VCS git [P*git16] contains a similar feature called submodules [Cha09, Chapter 6.6].

These VCS features may instruct the build server to execute HTTP requests to other servers. Therefore, these requests may

- access data on other web servers with host-based access restrictions and disclose them to the attacker,
- execute web service commands with host-based access restrictions on other web servers, or
- consume network bandwidth on the build server and target other servers in a denial-of-service attack.

Second, the downloaded source code may contain symbolic links. Attackers may forge symbolic links to files on the build server outside of the directory intended for the project. Subroutines of the CI system with high privileges may thereby be tricked to access files on the build server with elevated access rights. In other applications, there have already been attacks that exploit symbolic links [KFL94].

Build Preparations In the *Build Preparations* step, the build server configures the build environment for the build job. For example, a configure script may prepare files for a specific target platform or a batch process may copy dependencies to locations required for the builder runs. The *Build Preparation* step may require execution of arbitrary code, as requirements varies among projects and it is therefore not possible to offer only a closed set of preparation options.

After the *Builder Runs*, packaging the artifacts may also require the execution of arbitrary code within some post-build steps. From a security perspective, these post-build steps are similar to the *Build Preparation* step. Therefore, they are also discussed here.

Build Preparation steps can be configured directly in the CI system. For example, Jenkins allows shell commands to be configured directly via the web interface. An example of a malicious command is shown in Figure 4.3.

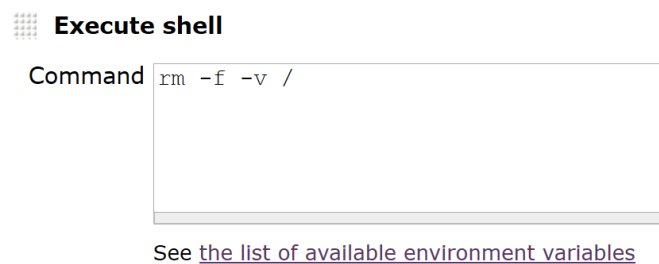


Figure 4.3: Screenshot of Jenkins’s web interface with a build step that deletes all system files

There are obvious countermeasures against this attack: The process executing the build may have only restricted user rights which suppresses malicious parts of the code. Alternatively, the CI system may completely forbid its tenants to configure *Build Preparation* steps. Automated analysis of build steps may also prevent malicious build steps to be executed.

It is not sufficient, however, to restrict build steps configured directly in the CI system only. The build scripts loaded from the VCS with the source code can also contain arbitrary code that is executed during the build. FLOSS projects often use Makefiles [P*Fre14] which allow execution of arbitrary commands. Other types of build environments also provide this possibility. Figure 4.4 shows how to configure a malicious deletion command in a Visual Studio project file. The CI system Jenkins can be configured to call the application MSBuild to execute these Visual Studio project files and then the build server also executes the commands configured there.

Builder Runs The build server compiles the source code and links all libraries in this step. The output are the software artifacts of the project. The output also comprises executable tests

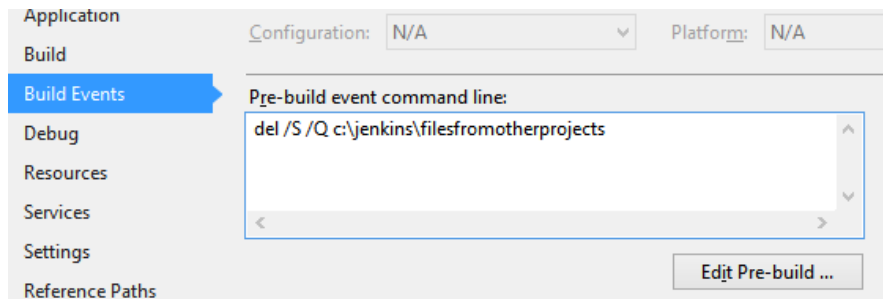


Figure 4.4: Screenshot of Microsoft Visual Studio 2012 with a build step that deletes files from other projects

that the developers have programmed. The build server executes these tests and adds the test results to the output. Plugins may allow the tests to be distributed to special test servers. This allows tests to be run on multiple platforms.

An attacker may include malicious behavior in the source code. Of course, the build server does not execute this malicious code directly, as it is only compiled and linked first. However, the build server executes tests and therefore indirectly also the source code. This way, the attacker can use the build server to start denial-of-service attacks or access restricted web resources. An attack has additional impact if the user account running the builder run was given too much rights or if the attackers use a security flaw to elevate their rights. In these cases, the attackers could shut down the build server. As a more sophisticated attack, they could modify the build server to compile or link virus code into all software artifacts build on the server. On a multitenant system, the software artifacts of other tenants would then get infected with malicious code.

As explained in the security discussion in Section 4.2.1, this may lead to a variation of Thompson's trusting trust attack [Tho84]: When the CI system builds a new version of its own software, the compromised build server may inject malicious code in the CI system software. If this is thoroughly executed, this attack is very difficult to detect, as the CI system may remove almost all traces that the attack is happening.

In a standard installation of Jenkins with default settings on a Linux Debian server, a *Builder Run* may execute commands on the build server with administrator rights. Thus, the attacks mentioned above would apply to such a system.

There are multiple countermeasures against the attacks listed above. Obviously, the account running the *Builder Runs* should be granted only minimal rights. Additionally, the build server might scan the source code for known malicious code fragments. However, it is a common practice to store library dependencies in binary form next to the dependent source code. An attacker may compile the malicious code into one of these libraries, so that the source code compiled on the build server is free of malicious parts, although the tests indirectly execute the malicious code in the library. Thus, the build server would need to take the extra step and also scan the binary libraries. A wide range of commercial programs exists for the task of detecting malicious binary files. However, malware detectors are unreliable especially when detecting new malware or new variations of existing malware [CJ04].

Section 4.2.3 presents another approach to counter the attacks emanating from this step of

the build. This approach works even if all of the countermeasures above are useless or not even applied. Of course, it is still better to have multiple lines of defense and also implement the measures mentioned above where possible.

Notification The build server reports the current status and eventually the results of the build back to the master server. In Jenkins, this includes every line of output to the console. After the build, the build server may transmit test results and build artifacts to the master server.

If attackers take control over the build server in one of the preceding steps, they may modify the data transmitted back to the master server. There is a weaker and a stronger form of an attack resulting from these modifications.

Some CI systems display test results or other result data in a web browser. Jenkins can also do this if it is extended with appropriate plugins. In the weaker form of the attack, the attackers add executable code like JavaScript code to the results. In this case, the attackers have to find a flaw in the master server's web interface to circumvent the security mechanisms that suppress execution of such code. If other tenants open the result data with their browser, the malicious executable code may act within the security context of these other tenants from within the web browser. Thus, the malicious executable code may change the configuration of other tenants' projects in the CI system. This may enable the malicious code to reproduce itself to other projects on the CI system.

The stronger form of the attack uses the communication channel from a compromised build server back to the master server. The build server uses a flaw in the communication protocol to take over the whole master server on behalf of the attackers. Obviously, this would compromise the whole CI system with all projects from all tenants.

4.2.3 Secure Build Servers

As shown in Section 4.2.2, users of multitenant CI systems may attack each other through the resources they share in the CI system. More specifically, attackers may modify the behavior of build servers to affect succeeding build jobs of other projects on the same build server. Attackers have to use security flaws in the source code or configuration of the CI system to accomplish their attack. While the maintainers may fix each of these security flaws when they become aware of them, the CI system is always susceptible to zero-day exploits and the maintainers' inattentiveness. This subsection describes an approach that inhibits attacks from one project to other projects built on the same build server. At the same time, the approach does not require a secure configuration of the build servers itself, for example it also works if the step *Builder Runs* uses an account with administrative privileges.

Through the attacks described in the steps *Build Preparations* or *Builder Runs* in Section 4.2.2, attackers can prepare a malicious build job that compromises a build server of the CI system. The attackers may then use the compromised build server for their purposes. On a multitenant CI system, build jobs of projects from other tenants will be scheduled to the same build server. Attackers may compile malicious code into the artifacts produced in the build jobs of those projects. These infected software artifacts may act as Trojan horses and compromise the computers of end users of the other projects. The build processes and infections are depicted as a UML state machine diagram in Figure 4.5. Note that the CI system maintainers and tenants of the CI usually

cannot distinguish between the states *Clean system* and *Compromised system*. In this thesis, a Default Build Server is a build server as described here.

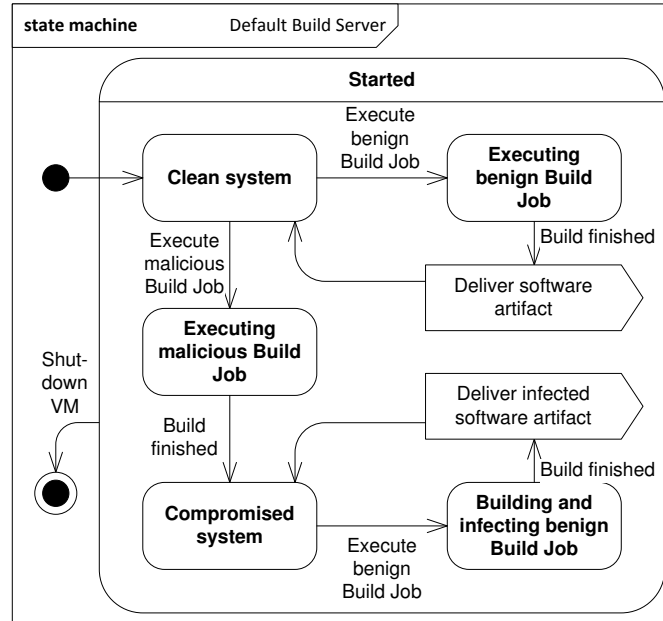


Figure 4.5: UML state machine diagram accounting for attacks that infect a Default Build Server

As explained, there is no generic method to prevent attackers from compromising the build server that executes build jobs of a malicious project. However, a build server can be considered in the state *Clean system* if it has not previously executed any build jobs. Since build servers executing build jobs for projects owned by attackers do not need to be in the *Clean system* state, this criterion is met if every project has its own dedicated build server. As the master server controls all build servers and therefore must not be compromised, the master server must not be a build server. Dedicated build servers are infeasible for CI systems responsible for too many projects to have one build server for each of these projects. Public CI systems that offer CI services for FLOSS projects therefore cannot use this approach.

However, the source code of most FLOSS projects changes seldomly and only in a few FLOSS projects, the source code changes often: The FLOSS project meta-repository Open Hub measures activity on FLOSS projects. As of 2016-03-15, Open Hub has 672 123 FLOSS projects in its database. With the FLOSS projects sorted by Open Hub's activity measure [P*Ope16b], the projects at the 0.1, 1, and 10 percentile are *mist.io*, *MediaArea website*, and *Wikitty*, respectively. These projects had 1891, 197, and 33 commits within the last 12 months, respectively.

Thus, most of the dedicated build servers would be idle most of the time, even for projects with very high activity. For example, *mist.io* would cause only 5.17 builds per day. Accordingly, the following approach uses virtualization to prevent builds from different projects on the same machine. At the same time, it does not require dedicated servers for every project.

The concept of a Secure Build Server extends the Default Build Server as described above. A Secure Build Server is a VM. After every build, a plugin on the master server restores the VM of

the build server to an original clean state. To improve performance, the VM uses a virtualization environment that supports snapshots, for example VirtualBox [P*Ora16b, Section 1.10.]. Restoring a snapshot allows the virtualization environment to quickly discard all changes made since the start of the VM. Similarly, Amazon’s cloud services allow using a so called local instance store to start machines from a specific snapshot [P*Ama13]. If an attacker schedules a malicious build job, the build server will build it and will be compromised, but the build server will be reset before it executes any other build jobs. Build jobs of other projects therefore cannot be infected anymore. Figure 4.6 shows a UML state machine diagram of such a Secure Build Server.

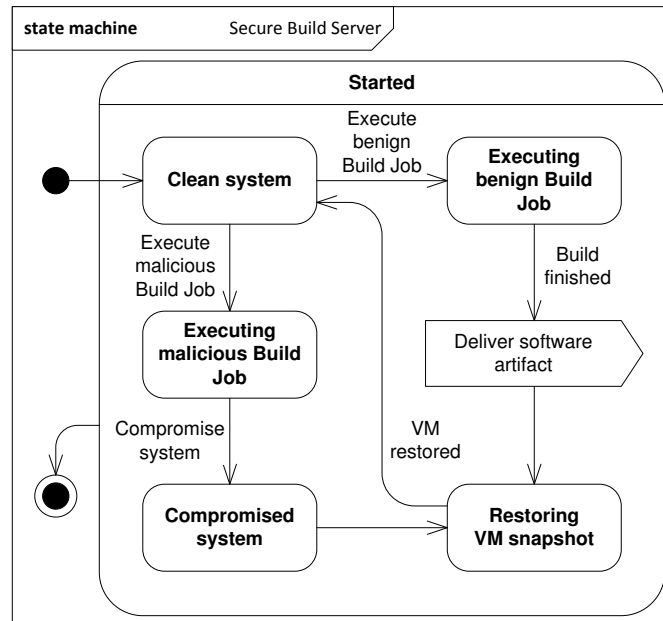


Figure 4.6: UML state machine diagram of an attack on a CI system using the proposed solution

Proof of Concept of a Secure Build Server

This subsection describes a PoC realization of a Secure Build Server. The realization uses the CI system Jenkins. The virtualization environment VirtualBox [P*Ora16a] hosts the VMs of the build servers. A modified version of the Jenkins plugin VirtualBox Plugin [P*MGG15] controls these VMs.

The VirtualBox Plugin has a beta version branch “snap” [P*Gre12] that supports VirtualBox snapshots. This branch serves as the basis for the realization of the Secure Build Server: An additional modification ensures that each build server VM restarts after every build job and thereby restores a specific snapshot. The modified version developed for this study can be downloaded from GitHub as FLOSS [P*Joh13].

Resource Considerations

As explained in Section 4.2.1, computational resources and security are two major problems limiting widespread CI use for FLOSS projects. The approach presented in this section increases security, as it prevents all attacks that use the build server to attack other tenants' builds. However, this is a trade-off, because the CI system has to restore a VM for every build. This subsection measures the impact on performance and therefore computational resources to take this trade-off into consideration.

The time needed to execute a build job depends on properties of the project: The build tools, the number of LOC, and the programming language, for example. Another important factor is the performance of the hardware the build server runs on. The project and the hardware are both implementation specific and therefore will not be regarded here. Hence, this analysis disregards the time of the actual build job's execution. Instead, the analysis compares the start up times of a Default Build Server and a Secure Build Server as described above.

There are two physical servers, one is a Windows system and the other runs a Debian Linux [P*Sof16]. The windows system runs the Jenkins master server on a Debian Linux VM. The other physical server hosts two VMs with Jenkins slaves. One represents a Default Build Server and the other represents a Secure Build Server. Both physical systems have a four core Intel Atom 330 Central Processing Unit (CPU) with 1.6 GHz per core, 2 GB physical memory, and a 250 GB SATA hard drive with 7200 RPM.

In the performance test, two build jobs are issued simultaneously on the Jenkins master server. There is a delay of about 3 seconds between the starts of the two build jobs, as they were issued manually via the web interface. Figure 4.7 shows a UML timing diagram where this issuance on the Jenkins master server is at time 0 seconds. After between 20 to 28 seconds, the VirtualBox host starts resuming or restoring the VMs. The Default Build Server needs about 58 seconds after the build job was issued to resume and start executing the build job. The Secure Build Server needs about 140 seconds to restore the original snapshot and then start executing the build job.

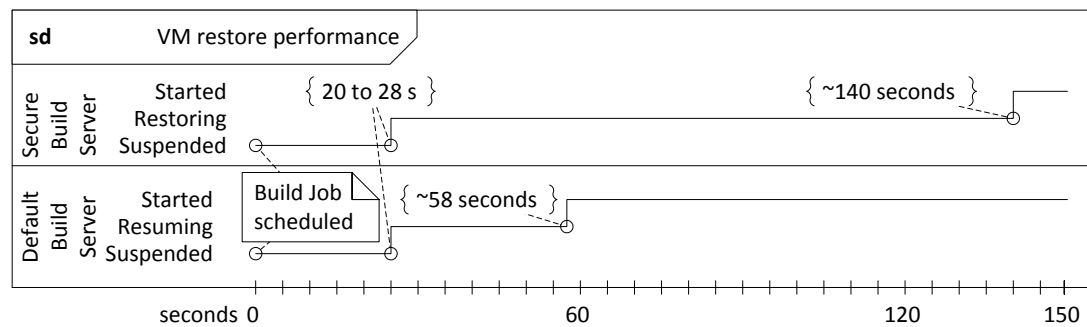


Figure 4.7: UML timing diagram of the VM start up latency in the test environment

The concept of a Secure Build Server described in this section has no influence on the time needed to execute a build job for the first time. However, a Secure Build Server needs more time for subsequent build jobs, as the Build Server cannot store intermediate data of the build job. For example, while a Default Build Server only loads the modifications of the source code since

the last build job from the VCS, a Secure Build Server has to download the whole source code for every build job. Additionally, a Default Build Server may reuse intermediate files from the last build job. Examples are object files for unmodified source code files in the programming language C. The performance decrease of a Secure Build Server in practice may therefore be greater than the pure start up times of the Build Server as analyzed in this subsection.

4.2.4 Validation

This subsection validates the security of the Secure Build Server concept. First, an attack is devised that modifies a build server such that all later builds on this server produce modified versions of software artifacts. The attack is sufficient to inject malicious code into these modified versions. However, an actual infection is not necessary for a PoC and will not be part of the attack. Second, the devised attack executes on a Default Build Server in the test environment described in Section 4.2.3. This shows that the Default Build Server is vulnerable to the attack. Third, the devised attack fails on a Secure Build Server. This shows that the approach successfully prevents these kinds of attacks.

Attack Scheme

As an example scenario, the CI system consists of a standard Jenkins installation with one master server and two managed slave servers as described for the performance test in Section 4.2.3. All systems run on a minimal Debian Linux [P*Sof16] environment. The master server connects to the slave servers via Secure Shell (SSH) tunnels. One slave server is a Default Build Server, the other slave server is a Secure Build Server. The master server only dispatches build jobs but is not a build server itself.

This section demonstrates the attack on a fictive FLOSS project ABC set up for this validation in the environment described above. Any real FLOSS project also works, as the weakness relies on the configuration of the CI system alone. ABC's SVN repository resides on servers with the domain name `svn.example.com`. The CI system also hosts a second project named Mallet. Unknowingly to the CI system and the ABC maintainers, Mallet tries to inject malicious code into the build artifacts of project ABC.

Mallet's strategy exploits the fact that the build server performs all build jobs. Mallet redirects access to ABC's SVN repository to its own SVN repository that contains modified code. Mallet achieves this with a modification of the `hosts` file on the build server. A post-build step of Mallet's build adds an entry for `svn.example.com` to the `hosts` file with an Internet Protocol (IP) address that Mallet controls, in this example `192.168.21.59`. This post-build step consists of the following line of shell script:

```
echo '192.168.21.59 svn.example.com' >> /etc/hosts
```

Attack on a Default Build Server

After Mallet's build, the `hosts` file contains an entry claiming that the name `svn.example.com` resolves to the IP address `192.168.21.59`, a server that Mallet controls. When the build server builds ABC the next time, the build server tries to fetch the freshest source code for project ABC.

ABC has an SVN repository configured on the server `svn.example.com`, so the build server resolves this host name. As the `hosts` file has a higher priority than Domain Name System (DNS), the build server does not resolve the correct IP address. Instead, the Default Build Server resolves the IP address `192.168.21.59` that Mallet configured in the `hosts` file.

With this modified `hosts` file present, the Default Build Server does not download the source code of ABC from `svn.example.com`, but instead some modified version from Mallet's server. In the actual implementation of this PoC, Mallet's server just delivers different source codes instead of actual malicious source code, but it is obvious that delivering malicious source code is easily possible.

In order to disguise the attack, Mallet might have configured the server on `192.168.21.59` to forward SVN requests to the actual SVN repository on `svn.example.com`. In this case, changes on `svn.example.com` still propagate to the software artifacts build on the CI system. Mallet's server might add malicious source code while forwarding the original source code. Thus, the attack would be difficult to detect for the ABC developers, as it does not show in the logs.

Attack on a Secure Build Server

Mallet now launches the same attack on a CI system that implements the approach described in Section 4.2.3, a Secure Build Server. Contrary to the previous course of events, the Secure Build Server resets the VM of the slave server after Mallet's build job. This restores the original `hosts` file and reverts Mallet's changes. When building ABC afterwards, the build server resolves `svn.example.com` to the correct IP address from DNS and loads the unmodified ABC source code. Thus, Mallet's attack was unsuccessful.

There are more obvious and direct countermeasures to prevent the devised attack. Running the build jobs on the build server in a restricted user account that has no access to the `hosts` file suffices already. However, the attack may be refined to outdo these countermeasures again, like combining the devised attack with a rights elevation attack. Of course, there are also countermeasures against these rights elevation attacks. Additionally, different but similar attacks to the shown attack also have obvious and direct countermeasures, like fixing a security flaw that allowed attackers to take over the build server. However, specific countermeasures only work against specific attacks. The devised attack is therefore only a trivial representative of a larger class of attacks, each of which has a specific countermeasure. The Secure Build Server approach prevents all attacks in this larger class.

4.2.5 Limitations

The Secure Build Server concept described in Section 4.2.3 has limitations. This section will discuss those limitations.

The build jobs are encapsulated and cannot influence each other, as the VM acting as Secure Build Server is restored to a clean snapshot after every build job. There are two methods that attackers may use to bypass the encapsulation.

First, attackers can use a flaw in the VM code to escape the VM [Reu07]. This way, they can compromise the computers hosting the VMs. Subsequently, they can compromise other VMs or other parts of the computer infrastructure.

Yet, VM escapes are not a problem specific to Secure Build Servers. In fact, if cloud services like Amazon EC2 [P*Ama16] host the Secure Build Servers, attackers might escape any VM that Amazon EC2 hosts to achieve the same goals as escaping a Secure Build Server. It would even be easier for the attackers to escape a VM that they themselves own, as they would have full control over these VMs without the need to find and exploit flaws in the CI system.

Second, the analysis of the attack vector *Notification* in Section 4.2.2 explained that a compromised build server could attack the master server of the CI system. Restoring a snapshot of the Secure Build Server would remove malicious code from the Secure Build Server, but the malicious code would survive on the master server. The compromised master server could easily compromise the Secure Build Server again after it restored the snapshot. The master server could compromise other build servers as well.

As explained in Section 4.2.1, the cost of computational resources is another problem of public CI services. The Resource Considerations in Section 4.2.3 showed that Secure Build Servers perform worse than Default Build Servers. In practical applications, the security benefit might not be worth the added cost of computational resources.

4.2.6 Related Work

This subsection gives a summary of current research on security solutions based on virtualization.

Rosenblum and Garfinkel [RG05] describe advantages and challenges in virtualization research. One section is also dedicated to security properties of VMs. Monitors running outside the VMs can analyze attacked system even down to the hardware level. Attacks are therefore more difficult to conceal. They also suggest to use VMs to separate environments or applications from each other. The solution in Section 4.2.3 may be seen as a special case of this general idea.

Garfinkel and Rosenblum [GR05] further elaborate that VMs with a dedicated purpose should only be allowed network access as far as necessary. This Guest OS Independence, as they call it, allows more fine-grained access control than solutions without VMs. This concept allows to prevent attackers from using Secure Build Servers for denial of service attacks.

Price [Pri08] compares security advantages and disadvantages of virtualization. The list of advantages contains encapsulation as the possibility to run different applications without influencing each other. Disadvantages that implementors of the Secure Build Server solution should consider include rollback vulnerabilities: Administrators may install security patches on VMs like the Secure Build Servers. If the VMs restore snapshots without the security patches afterwards, the VMs are vulnerable again. This may happen unknown to the administrators, who consider the VMs patched.

4.2.7 Summary

A multitenant CI system is an integral part of a WikiDE. Small FLOSS projects rely on software forge services instead of maintaining their own infrastructure for development tools. Therefore, these small FLOSS projects would also profit from CI techniques if software forge services offered CI services. Large FLOSS projects with their own infrastructure may speed up development if external developers could use their CI systems. The realization of these use cases requires public CI systems with multiple independent tenants, but they suffer from two problems: First,

CI systems come with a high cost because of the required computation, bandwidth, and memory resources. Second, as all tenants execute their own code on the CI systems, tenants may cause malicious or non-malicious failures that affect other tenants.

A security analysis showed which attack vectors enable failures to spread from one tenant's part of the CI system to the others'. One class of attack vectors starts on the build server that executes the tenant's own code. Section 4.2.3 detailed a concept that encapsulates build job via virtualization and snapshots, creating a Secure Build Server. The CI system resets the Secure Build Servers back to an original and uncompromised state after every build job. Thereby, Secure Build Servers prevent attacks that rely on multiple build jobs on the same build server, even if the attacks are not detected. A PoC implementation on the CI system Jenkins shows that the concept of a Secure Build Server can be realized.

4.3 Reviewer Recommendations⁶

Delayed reviews for submitted patches was identified as an important contribution barrier in Chapter 2. As highlighted in Section 3.6, none of the identified FLOSS patterns lower this contribution barrier. One reason for delayed reviews are problems with reviewer assignment. In FLOSS projects, problems with reviewer assignment defer the acceptance of patches by an arithmetic mean time of 6 to 18 days [Tho+15]. This problem is also not specific for FLOSS, finding reviewers is generally a problem in software development projects [LVD06; Wie02, p. 182]. Thus, a tool that speeds up reviewer assignment would lower the corresponding contribution barrier. As visible from the WikiDE contribution process described in Section 4.1.3, a WikiDE could use such a tool to speed up the action Review modifications.

There is extensive research on bug triaging, i.e. assigning developers to a specified issue and reducing the human effort involved, for example [CC06; Sho+13]. This is an important problem in practice, and FLOSS projects use tools to support bug triaging [TM12]. There is also research on the review practices of FLOSS projects in general, for example [Kop06; Bay+12]. More recently, research emerged on tools to support reviewer assignments [Bal13; Tho+14; Tho+15]. They suggest that algorithms evaluate the project's review history to recommend reviewers with the greatest *review expertise* for submitted patches.

In contrast, Sethanandha et al. propose that the ideal reviewer would be the person with the greatest *modification expertise* for the code that the submitted patch modifies [SMJ10b]. There is research based on practical observations and empirical data for algorithms to measure the modification expertise of individual developers for a given part of the source code, for example [AM07; Fri+14]. However, these algorithms have not yet been evaluated for their suitability to recommend reviewers.

Reviewer recommendation tools are only useful if they recommend competent reviewers for submitted patches. This depends on the employed algorithms. This section compares the prediction performance of the algorithms File Path Similarity (FPS) [Tho+14] and Weighted Review Count (WRC), both of which use review expertise for their decision, against six algorithms based on modification expertise. The evaluation uses historical data from the same three FLOSS

⁶A preliminary version of this section was published previously [Han+16].

projects as in a previous study [Tho+14] to ensure internal validity and added Firefox as a fourth FLOSS project for external validity.

4.3.1 Related Work

This section describes existing algorithms to measure expertise. A reviewer acquires *review expertise* when reviewing code. A developer acquires *modification expertise* when modifying code. Each algorithm may be used to recommend reviewers for submitted modifications. *Usage expertise* is a third type of expertise and refers to expertise gained by using a module, for example when using it as a library, but not modifying it. Research on usage expertise always came in conjunction with modification expertise. Thus, usage expertise is not discussed separately.

Review Expertise

Jeong et al. [Jeo+09] first suggested how reviews of submitted patches could be predicted automatically. They proposed Bayesian Networks to predict the outcome of the review and the actual reviewer of patches.

Balachandran [Bal13] developed the two reviewer recommendation algorithms Review Bot and RevHistRECO. If a patch for some files is about to be reviewed, Review Bot assigns scores to all previous authors and reviewers of the same lines of code that the patch modifies. RevHistRECO is similar but operates on file history instead of line history. The algorithms use a time prioritization parameter $\delta \in]0; 1]$ to decrease the impact of older modifications and patches. The developers with the highest scores are recommended as reviewers. Balachandran compares the two algorithms on two projects with 7035 and 1676 reviews. Review Bot had a better prediction performance than RevHistRECO. Both Review Bot and RevHistRECO use both review expertise as well as modification expertise.

Another approach was presented by Thongtanunam et al. [Tho+14]. They described the recommendation algorithm FPS. The main idea of FPS is that files with similar file paths are closely related by their function. It assumes that in most large projects the directory structure is well organized. The algorithm assigns scores to reviewers of files that have similar file paths as the files of a given review request. Reviewers with the highest accumulated score are recommended as reviewers for the given request. Like Review Bot, the algorithm contains a time prioritization factor $\delta \in]0; 1]$ to prefer more current over past reviews.

The following is a mathematical definition of FPS equivalent to the original definition in Thongtanunam et al.'s paper. Let F , R , and D denote the set of all files, all reviews, and all developers, respectively. Let $\text{Files} : R \rightarrow \mathcal{P}(F)$ be the function that maps a review to the set of its reviewed files. FPS uses a function $\text{CommonPath} : F \times F \rightarrow \mathbb{N}$ that counts the number of common directory levels from left to right in the paths of two given files. The file name counts as a directory level on its own. Thus, if two files are identical, the count is one higher than for two files in the same directory because of the identical file name. Similarly, $\text{Length} : F \rightarrow \mathbb{N}$ counts the number of directory levels including the file name in a file's path, or in a different formulation $\text{Length}(f) = \text{CommonPath}(f, f)$. The definition of the functions $\text{Similarity} : F \times F \rightarrow \mathbb{N}$ and $\text{ReviewSimilarity} : R \times R \rightarrow \mathbb{R}$ shall be the following:

$$\text{Similarity} : f_1, f_2 \mapsto \frac{\text{CommonPath}(f_1, f_2)}{\max\{\text{Length}(f_1), \text{Length}(f_2)\}}$$

$$\text{ReviewSimilarity} : r, s \mapsto \frac{\sum_{f_1 \in \text{Files}(r)} \sum_{f_2 \in \text{Files}(s)} \text{Similarity}(f_1, f_2)}{|\text{Files}(r)| \cdot |\text{Files}(s)|}$$

The function $\text{Reviewers} : R \rightarrow \mathcal{P}(D)$ shall map each review to the set of developers that participated in the review. Let r_1, \dots, r_n denote all reviews in R in chronological order. For a developer d and a review r_x , let $R_{x,d} := \{r_i \in R \mid 0 < i < x \wedge d \in \text{Reviewers}(r_i)\}$, the set of all reviews that happened before r_x in which the developer d participated as a reviewer. Then $\text{FPSScore}_\delta : R \times D \rightarrow \mathbb{R}$ gives a prediction of each developer's expertise for each review according to the following definition:

$$\text{FPSScore}_\delta : r_x, d \mapsto \sum_{r_i \in R_{x,d}} \text{ReviewSimilarity}(r_x, r_i) \cdot \delta^{x-i-1} \quad (4.1)$$

The time prioritization factor $\delta \in]0; 1]$ ensures that older reviews have less influence on the resulting score than newer reviews. For $\delta = 1$, there is no time prioritization and older reviews have the same weight as newer reviews.

Thongtanunam et al. compared the prediction performance of FPS to Balachandran's Review Bot. Both algorithms calculated recommendations for historical reviews of the three FLOSS projects Android Open Source Project (AOSP), OpenStack, and Qt. The algorithms used five different time prioritizations $\delta = 1$, $\delta = 0.8$, $\delta = 0.6$, $\delta = 0.4$, and $\delta = 0.2$. The number of recommended reviewers was limited to $k = 1, 3$, and 5 for each review. Then they compared each set of recommended reviewers with the actual historical reviewers in each review. A recommendation was considered correct if at least one of the k recommended reviewers was one of the actual reviewers.

The results answer two research questions: First, for all considered combinations of projects and values of k and δ , FPS performed better for greater values of δ . Consequently, $\delta = 1$ outperformed all other tested values of δ . For Review Bot, δ had little influence (less than 0.5 percentage points) with the exception of AOSP, where $\delta = 0.2$ and $k = 1$ is an extreme outlier. Since this outlier is not discussed in the text, and even contradicted with, even this outlier is likely just a misprint in the table. Second, FPS significantly outperforms Review Bot, except for some configurations in Qt especially for low values of δ and number of recommended reviewers. For $k = 5$ and $\delta = 1$, FPS predicts 77.1 %, 78.0 %, and 36.9 % correctly for AOSP, OpenStack, and Qt, respectively. For $k = 5$, Review Bot predicts, with a project-specific optimal value for δ , 29.3 %, 38.9 %, and 27.2 % correctly for AOSP, OpenStack, and Qt, respectively.

In a more recent study, Thongtanunam et al. [Tho+15] further improved FPS and added the FLOSS project LibreOffice to their evaluation. They used a more sophisticated version of the CommonPath function. This more sophisticated version does not only compare the longest common prefix of two paths, but also three other types of path similarity metrics and combines them with Borda count [Bla76]. The improved algorithm correctly predicts 79 %, 77 %, 41 %, and 59 % with $k = 5$ for AOSP, OpenStack, Qt, and LibreOffice, respectively. This is a change of +2, -1, and +4 percentage points for AOSP, OpenStack, and Qt, respectively, in contrast to their previous evaluation.

GitHub [P*Git15c] offers features to support “social coding” [Dab+12]. Jiang et al. [JHC15] analyzed reviewer recommendation algorithms that use social relationships between developers and reviewers among other properties for recommendations. The best algorithm in their evaluation uses support vector machines (SVMs) as machine learning algorithm for recommendations. The results cannot be directly compared to the other approaches, including the one in this section, because of their specialization on GitHub features that are not available in other FLOSS projects.

Yu et al. [Yu+16] also used machine learning to recommend reviewers for patches in GitHub. For their approach and data, Information Retrieval performed better than SVM.

Zanjani et al. [ZKB15] combine three metrics to calculate review expertise. For each combination of reviewer and file, each metric yields a value between 0 and 1. These three values are added to a composed review expertise.

Modification Expertise

There is various research on algorithms calculating a software developer’s modification expertise. The algorithms were not originally designed for reviewer recommendation, instead usual goals are

- finding a mentor for new team members, and
- finding an expert for specific questions on a part of the component.

Each unique algorithm evaluated for reviewer recommendation in this section is highlighted in boldface.

A simple algorithm to calculate modification expertise is **Line 10 Rule**: The last editor of a module is assumed to have the most expertise with it. The name of this algorithm derives from a VCS that stored the author of a commit in line 10 of the commit message. Because of its simplicity, the Line 10 Rule is a metric or the basis of a metric in various research. [MA00; SZ08]

In early research on expertise in the software development domain, McDonald and Ackerman [MA00] used three types of sources to identify experts, who may be software developers or support representatives: First, they asked all users to create profiles about themselves. Second, they counted the number of changes each developer made on a module using VCS logs. Third, they indexed the descriptions of issues for which support representatives were responsible using data from an issue tracker. This allowed their tool **Expertise Recommender** to recommend a set of experts when given a problem description. The recommendation list is ordered by the date of the last change to the module, and the last editor appears first in the list, hence this is a straightforward extension to the Line 10 Rule.

The **Number of Changes** to a module is only a very rough metric of modification expertise. For example, if the development team switches to the CI development method, the number of logged changes to the modules will increase due to the regular commits to the VCS [Fow06]. This introduces a bias to the metric, as early, seldom committing developers are considered less experienced than later developers with a high commit frequency.

Nevertheless, other research also uses the number of changes to an artifact as a metric to quantify the developers’ expertises with the artifact. Balachandran’s Review Bot [Bal13] described previously also falls into this category. As other early research of this type, Mockus and

Herbsleb [MH02] defined Experience Atoms (EAs) to be “elementary units of experience”. They distinguished between several categories of experience, where expertise with a specific module is only one example category. They mined VCS logs to calculate the EA a developer has acquired. Developers acquire one EA for each commit in each relevant category. Examples of categories are the module that the commit changes and the technology that the commit makes use of. They also developed a tool called Expertise Browser to visualize the EAs. Expertise Browser allows its user to find experts, for example the developers with the highest number of EAs for a module.

As another example, Robles et al. [RGH09] devised a method to visualize changes to the core group of developers in a FLOSS project at each point in time. Their calculation also counts the number of commits for each developer to the VCS. For each time period, a fraction of developers with the most commits are considered core developers. Qualitative analyses have shown 10 % to 20 % to be reasonable fractions of commits to be considered core developers. Thus, the calculation’s output is a project-wide decision whether a specific developer was a core developer at a specific point in time. The visualization techniques itself are not relevant for this study and will not be discussed here.

Minto’s and Murphy’s Emergent Expertise Locator [MM07] also calculates the number of times each developer has edited each file and stores this information in a File Authorship Matrix. This matrix is used as part of a calculation to find developers that share expertise and therefore are assumed to be part of an emerging team. The Emergent Expertise Locator can then propose prospective colleagues to a developer to help in the formation of such a team. However, the calculation step used to find a developer’s expertise with a file still only counts the number of changes.

Schuler and Zimmermann [SZ08] propose to count not only the Number of Changes directly to a module as modification expertise, but also the number of commits with code that call the module. This is no modification expertise, but usage expertise. However, they provide no algorithm that aggregates the measured modification and usage expertises into a single metric. As modification expertise appears more adequate than usage expertise to recommend reviewers for *modifications* of a module, their approach does not contribute an additional algorithm for reviewer recommendation.

Gîrba et al. [Gîr+05] approximated **Code Ownership** using VCS logs, which is specifically a list of file revisions with an author name and the number of removed and added lines for each file. They developed two algorithms that execute consecutively for the Code Ownership approximation. The first algorithm approximates the size of each code file, as the file size is not directly available in their type of VCS log. For each author, the second algorithm approximates the fraction of lines in the file which this author “owns”, i.e. that this author has edited last. Ownership Maps visualize this ownership data. They also showed how to identify patterns in Ownership Maps.

Alonso et al. [ADG08] assigned each source code file to a category based on its file path. Their tool **Expertise Cloud** measures a developer’s expertise in a category as the number of changes to files in the category. Thus, this algorithm is a variant of counting the number of changes on a module instead of the file level. It stands out in that it also specifies how to find out the changed modules from VCS logs. However, their example project, the Apache HTTP Server Project [P*The15a], clearly documented its source code layout which implies a classification scheme for the project’s files. It is not immediately clear how to classify files in other projects, which may have less clear documentation on their source code layout, let alone an automatic

classification without manual configuration. This study uses a classification scheme similar to FPS for Expertise Cloud. Their approach also includes a visualization of expertise in tag clouds, hence the name Expertise Cloud.

The Degree-of-Knowledge (DOK) defined by Fritz et al. [Fri+10; Fri+14] aggregates two metrics on different data sources, the Degree-of-Interest (DOI) and the Degree-of-Authorship (DOA). First, the DOI measures how much a developer *uses* a module. The DOI extends Schuler's and Zimmermann's idea of usage expertise [SZ08], as it accounts not only calls to the module, but also other interactions such as opening a module's source code in the IDE [KM06]. Second, the DOA measures how much of the module's source code a developer has *created*. Similarly to Gîrba et al.'s Code Ownership approximation [Gîr+05], a developer's DOA decreases when other developers change the module. Fritz et al. determined concrete weightings for aggregating the DOA components and the DOI into a single DOK value via linear regression on experiment data. In multiple experiments, they found that the weightings depend on the project, but they also gave general weightings that best fit the aggregated experiment data. Acquiring the data to calculate the DOI requires a plugin on each developer's computer. This impedes its usage in this study in three ways. First, it disallows using data before the plugin was installed. Second, additional effort is necessary to let the developers in the evaluated FLOSS projects use the plugin, as the plugin needs support and FLOSS developers need to be convinced to cooperate. Third, ensuring the plugin's usage requires a closed experimental setting. Therefore, only the **DOA** is part of the comparison.

Anvik and Murphy [AM07] compared three different algorithms to determine experts. They do not distinguish between different degrees of expertise, and instead categorize developers either as experts for an issue in the issue tracker or not. The algorithms require that the issue reports have already been closed, and are therefore not suitable for reviewer prediction. However, two of the algorithms evaluate VCS logs to find experts, and one uses data from the issue tracker. In this regard, the comparison resembles the comparison in this paper, as it also compares recommendation systems based on modification expertise with recommendation systems based on expertise acquired when using an issue tracker. Both algorithms using VCS logs use a heuristic like that of Expertise Recommender [MA00]: The developers who changed a module last are experts for this module. These two algorithms differ in what they consider a module: The first considers the change set of the issue, i.e. all files that the issue's resolution changes, as one module, the second other considers Java packages as modules. The algorithm that uses data from the issue tracker assembles its list of experts in a more sophisticated way. It considers not only the issue report under analysis itself, but also all issue reports that relate to the issue report under analysis, for example because it has been marked as duplicate. From each related issue report, including the issue report under analysis itself, three types of sources add to the list of experts. The first source are the people in the issue report's Carbon Copy (CC) list, although some of the entries are filtered out. Second, those who commented on the issue report are added, again only after filtering. The third source is a heuristic of who fixed the issue described in the issue report.

Anvik and Murphy found the two algorithms based on VCS logs to have an average precision of 59 % and 39 %, and an average recall of 71 % and 91 % when considering change sets and packages as modules, respectively. The algorithm based on issue reports shows results between the two VCS based algorithms, with an average precision of 56 % and an average recall of 79 %. Thus, no algorithm is better for all use cases, instead there is a trade-off between precision

and recall. Furthermore, the number of recommended experts differs between algorithms and their published data show a strong association between the average number of recommended experts and average recall. This suggests that the choice between the two sources of data and the granularity of modularization in the analysis may have little or no influence at all on the measurement of expertise.

4.3.2 Expertise Explorer

The algorithms presented in Section 4.3.1 use data from development tools like the issue tracker and the VCS. Based on these data, they calculate the expertise for a given code artifact for a given developer. Some algorithms calculate review expertise, others calculate modification expertise. Algorithms like EA and DOI/DOA distinguish between different types of expertise, but they have an explicit variant to calculate modification expertise.

This modification expertise may or may not be different from review expertise. If they are different, these two types of expertise are not distinct: Review expertise implies an understanding of the modifications to a component, which is helpful or even necessary to create code [Tao+12]. Thus, when gathering review expertise, a developer also gathers modification expertise. The reverse is also true: An expert creator of a component's code may anticipate side effects of a modification and, more generally, knows the context of a piece of code, like its intended purpose, its callers and callees, and its technical constraints. These skills obviously also help when reviewing code.

This study uses the platform Expertise Explorer developed for expertise calculations and the more specific purposes of this study. Expertise Explorer is publicly available [SHP15] under the GPL and therefore anybody can reproduce the results of this study or use it for similar calculations or comparisons. There is a lab package with the source and result data used in this study [HPG15].

This section provides definitions for ambiguous terms like review and reviewer. A description of the technical aspects of Expertise Explorer relevant for the evaluation of the historical data follows. The section eventually provides an alternative calculation method for FPS that fits to Expertise Explorer's data structure.

Data Model for the Review Process

The evaluation prerequires an exact definition of what a review is and when the algorithms must calculate their recommendation of reviewers. If the calculation takes place later, then the algorithms may obviously use more data and may come to different recommendations. The review model includes the concepts of *issues*, *patch submissions*, *reviews*, and the main VCS. Figure 4.8 depicts the former three and their relationships as an Entity Relationship Diagram (ERD) [Che76]. An issue is a change request to the FLOSS project, either caused by a fault in the software or in case of a feature request by a change to the specification. In either case, the change request requires the modification of the source code. For this definition, documentation files and build scripts also count as source code if they also reside in the VCS, because they can be difficult to distinguish and the review process is the same. Thus, any issue in the FLOSS project's issue tracker counts as issue. Proposed modifications to the source code manifest as patch submissions. Each patch submission has an author, occurs at a specific point of time, and modifies one or more

files in the VCS. A patch submission belongs to one specific issue and each issue may have any number of patch submissions. Each patch submission may receive any number of reviews. Each review is performed by a reviewer at a specific point of time and belongs to one specific patch submission. The files that a patch submission modifies also belong to its corresponding reviews. Obviously, a review can occur only after its corresponding patch submission. Note that although a review is actually positive or negative, this is not relevant for the model. What exactly counts as a review depends on the issue tracker and will be discussed separately for each FLOSS project in the evaluation. Eventually, some patch submissions are merged into the main VCS at a specific point in time.

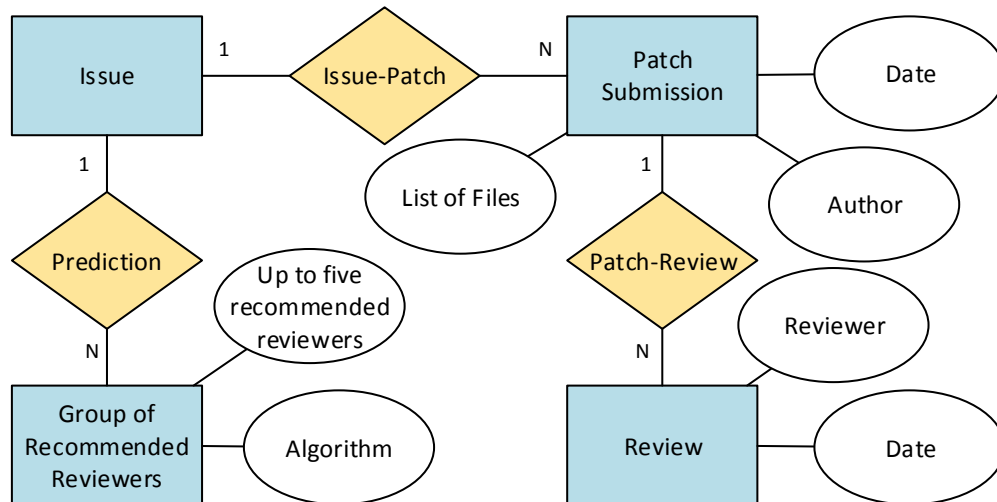


Figure 4.8: ERD of the core data structures used in the evaluation

Issues without patch submissions or reviews are discarded for the evaluation. For the remaining issues, all algorithms calculate up to five *recommended reviewers* for the issue when the earliest patch is submitted. If the algorithms were used in a recommendation tool in practice, this is when the FLOSS project has to decide who should review the patch. For later patch submissions, the algorithms will not calculate new recommendations: One reviewer often reviews all patch submissions of an issue and therefore a reviewer recommendation tool does not need to recommend new reviewers in these cases. Thus, there is exactly one group of up to five recommended reviewers for each algorithm and for each of the remaining issues. All reviewers who review any patch submission of an issue at any point in time are considered *actual reviewers* for the issue.

An algorithm may use data from all events that occurred before the time of calculation, i.e. when the earliest patch to an issue is submitted. For WRC and FPS, this includes all reviews on patch submissions in other issues. Reviews in the issue under analysis cannot have occurred before the first patch submission and therefore are not included. There can be multiple reviews for each issue, so the period between first and last review of two issues may overlap. When calculating a group of reviewers, it may happen that WRC and FPS take some early reviews of an issue into consideration while they do not take some later reviews of the same issue into consideration, because these reviews happen only after the patch submission that triggered the

calculation.

The six algorithms for modification expertise use data from the main VCS. That means that they can use patch submissions only after they have been merged into the main VCS. In contrast to WRC and FPS, they take the author of a patch submission into account.

Data Processing

Expertise Explorer needs two sets of data as input. These sets of data may or may not have the same origin, depending on the project under analysis. First, Expertise Explorer needs authorship data as stored in a VCS log. With these data, Expertise Explorer can execute algorithms for modification expertise. Second, Expertise Explorer needs the review history. With these data, Expertise Explorer can execute the algorithms for review expertise. The review history is also needed to compare the computed reviewer recommendations with actual reviewer recommendations, which will be used for the evaluation. The VCS history data may be substituted by the issue tracker history data if the latter contain enough authorship information, especially the number of added and removed lines and whether files are modified or newly created.

Expertise Explorer stores in a database for each project

- authorship data,
- expertise values for each combination of file, contributor, and algorithm,
- a group of recommended reviewers for each issue and each algorithm, and
- the actual reviewers of each issue.

In the first step, all authorship data are imported into the database. Next, Expertise Explorer iterates chronologically through a list of activities in the project's issue tracker, specifically reviews and patch submissions.

For each first patch submission, Expertise Explorer calculates all algorithms' expertise values for all combinations of contributors and submitted files using review and authorship data that occurred before the data of this first patch submission. Thus, the expertise values correspond to the values that a reviewer recommendation tool would have had access to at the date and time of the first patch submission to the issue. Afterwards, Expertise Explorer calculates for each algorithm the five contributors with the highest expertise for the patch. If the patch contains multiple files, this involves an algorithm-specific aggregation of expertise values. For most algorithms, expertise values are simply added. Note that adding and the arithmetic mean yield the same reviewer recommendations, as the order of recommended reviewers is the same for both types of aggregations. For Line 10 Rule and Expertise Recommender, the maximum expertise value is used instead of the sum, as this yields the latest editor of any of the reviewed files. For each review, the algorithms based on review expertise update their data to include the review. Furthermore, the reviewer is stored as an actual reviewer for the issue.

Afterwards, another module of Expertise Explorer iterates over the list of issues in the database. For each issue, it compares the up to five recommended reviewers for each algorithm with the set of actual reviewers for the issue. The results indicate the agreement between each algorithm and a manual reviewer selection.

Alternative Algorithm for FPS

Thongtanunam et al. [Tho+14] provided an algorithm in pseudo-code for FPS which directly uses Equation 4.1 presented in Section 4.3.1. Because of Expertise Explorer's database structure, Expertise Explorer cannot directly use this algorithm. However, this section will show that there is an equivalent definition for $FPScore_\delta$ that Expertise Explorer can use.

Using the same definitions as in Section 4.3.1, let $WRC_\delta : F \times D \times R \rightarrow \mathbb{R}$ be the function that calculates the value WRC, an intermediate value that specifies the review experience a developer has with a specific file at the time of a specific review. Remembering that r_1, \dots, r_n are all reviews in chronological order, let $\hat{R}_{x,d,f} := \{r_i \in R_{x,d} \mid f \in \text{Files}(r_i)\}$ be the set of reviews that occurred before r_x and involved developer d and file f . Then WRC_δ has the following definition:

$$WRC_\delta : f, d, r_x \mapsto \sum_{r_i \in \hat{R}_{x,d,f}} \delta^{x-i-1} |\text{Files}(r_i)|^{-1} \quad (4.2)$$

Expertise Explorer treats WRC like any other algorithm and calculates its value for all developers and all files. Using the pre-calculated values for WRC allows a different calculation of FPS using the following equivalence:

$$\begin{aligned} & FPScore_\delta(r_x, d) \\ &= \sum_{r_i \in R_{x,d}} \text{ReviewSimilarity}(r_x, r_i) \cdot \delta^{x-i-1} \\ &= \sum_{r_i \in R_{x,d}} \frac{\sum_{f_1 \in \text{Files}(r_x)} \sum_{f_2 \in \text{Files}(r_i)} \text{Similarity}(f_1, f_2)}{|\text{Files}(r_x)| \cdot |\text{Files}(r_i)|} \cdot \delta^{x-i-1} \\ &= \sum_{f_1 \in \text{Files}(r_x)} |\text{Files}(r_x)|^{-1} \sum_{\substack{r_i \in R_{x,d} \\ f_2 \in \text{Files}(r_i)}} \frac{\text{Similarity}(f_1, f_2)}{|\text{Files}(r_i)|} \cdot \delta^{x-i-1} \\ &= \sum_{f_1 \in \text{Files}(r_x)} |\text{Files}(r_x)|^{-1} \sum_{\substack{f_2 \in F \\ r_i \in \hat{R}_{x,d,f_2}}} \frac{\text{Similarity}(f_1, f_2)}{|\text{Files}(r_i)|} \cdot \delta^{x-i-1} \\ &= \sum_{\substack{f_1 \in \text{Files}(r_x) \\ f_2 \in F}} \frac{\text{Similarity}(f_1, f_2)}{|\text{Files}(r_x)|} \sum_{r_i \in \hat{R}_{x,d,f_2}} |\text{Files}(r_i)|^{-1} \delta^{x-i-1} \\ &= \sum_{\substack{f_1 \in \text{Files}(r_x) \\ f_2 \in F}} \frac{\text{Similarity}(f_1, f_2) WRC_\delta(f_2, d, r_x)}{|\text{Files}(r_x)|} \end{aligned}$$

This alternative algorithm to calculate FPS has the advantage over the original variant that the values for WRC can be calculated before knowing which files belong to r_x . Thus, Expertise Explorer pre-calculates all current values for WRC. When the FLOSS project needs a reviewer recommendation for a set of files $\text{Files}(r_x)$, the alternative algorithm can provide the recommendation more quickly.

4.3.3 Empirical Evaluation

The subject of evaluation are on the one hand the six algorithms based on modification expertise, in particular Line 10 Rule, Number of Changes, Expertise Recommender, Code Ownership, Expertise Cloud, and DOA and on the other hand two algorithms based on review expertise. The first is FPS developed by Thongtanunam et al. [Tho+14] as described in Section 4.3.1. The evaluated version of FPS uses the original definition of CommonPath and not the more sophisticated version developed in their more recent publication [Tho+15]. The more sophisticated version's results differ from the original version by only about +2, -1, and +4 percentage points for the three FLOSS projects AOSP, OpenStack, and Qt, and the newer version was published after the calculations in this study. The variants are restricted to $\delta = 1$ as parameter, as this was the optimum tested value for FPS as determined by Thongtanunam et al. [Tho+14]. In their more recent study, Thongtanunam et al. [Tho+15] also abstained from using other parameter values than $\delta = 1$. The second evaluated algorithm based on review expertise is WRC as defined in Equation 4.2 in Section 4.3.2. Expertise Explorer had to calculate the values of WRC for technical reasons anyway, so it could be included in the comparison without additional effort. Review Bot [Bal13] was not part of the comparison, as it had already been shown to be inferior to FPS [Tho+14; Tho+15]. Furthermore, Review Bot requires much more input data than all other considered algorithms, specifically a line-based modification history.

The evaluation used data from four different projects: Firefox, AOSP, OpenStack, and Qt. Thongtanunam et al. [Tho+14] also used AOSP, OpenStack, and Qt, which allows a comparison of the results. Firefox was added to increase external validity, as it uses a different issue tracker than the other three FLOSS projects.

Firefox uses Bugzilla to keep track of reported software bugs, AOSP, OpenStack, and Qt use Gerrit Code Review. Table 4.2 summarizes the characteristics of the study data. The study period includes a training phase of one year. Only prediction results after the training phase were part of the further analysis. This ensured that the algorithms could access historical data of at least one year for each recommendation. The numbers in parentheses refer to the properties excluding the data from the training phase.

Firefox

Section 3.4.1 described Mozilla Firefox in terms of its implemented FLOSS patterns. Section 2.2.1 treats its parent project Mozilla and especially its joining procedures.

The VCS log of Mozilla's hg repository for Firefox [P*Moz15o] is one of two data sources used for the comparison. It contains all information the recommendation algorithms based on modification expertise need to compute expertise values. In particular, hg saves not only the committer of a patch in its log, but also its author. Expertise Explorer reads hg logs to find this information. The VCS log used for the analysis contains all modifications committed between 2007-03-22 and 2013-03-07. On 2007-03-22, Firefox migrated from the VCS Concurrent Versioning System (CVS) to hg, therefore no older VCS logs were used.

The second data source used for the comparison is Mozilla's issue tracker Bugzilla [P*Moz01]. One component for the download and analysis of Bugzilla's history data of the Firefox project is Zhou's and Mockus's download and data transformation scripts [ZM12] for Mozilla. The

Table 4.2: Characteristics of study data, the values in parentheses exclude training data

	Firefox	AOSP	OpenStack	Qt
Study Period including Training Year	2007-03 to 2013-03	2008-10 to 2014-12	2011-07 to 2014-12	2011-05 to 2014-06
Number of Contributors	1762	2489	3717	1143
Number of Reviewers	781 (712)	1001 (997)	2813 (2803)	268 (244)
Number of Issues	60 329 (52 892)	39 090 (37 875)	119 287 (117 145)	70 358 (48 709)
Number of Reviews	73 355 (64 416)	55 292 (53 886)	376 494 (371 570)	75 907 (52 570)
Number of Files in VCS	123 939	704 112	103 486	215 890
Arithmetic Mean of Reviews per Issue	1.22	1.41	3.16	1.08

applicable output of the scripts is a Comma Separated Values (CSV) file with one “activity” in Bugzilla per line. Reviews are one of those activities. The CSV file contains the following information for each review:

- Who was the reviewer?
- When did the review take place?
- The CSV file does not say which files were reviewed, but only an identification number for an issue’s attachment containing the patch as diff file.

To retrieve this missing information about the reviewed files, Expertise Explorer has a component AttachmentCrawler that downloads the diff files of each reviewed patch to find out which files the patch changes. Expertise Explorer parses the CSV file and filters for relevant reviews such that each included activity fulfills the following constraints:

- Each relevant activity contains one of the flags “review+” or “review-”, as these indicate completed reviews [Bay+12]. The analysis discards architecture reviews indicated by “superreview”, as these are not in the focus of this work.
- The review took place between 2007-03-22 and 2013-03-07, as the VCS logs span this time frame.
- Reviews must be for files of the Firefox project. All Mozilla projects use Bugzilla and are therefore present in the CSV file. The reviewed file names must match a file in the VCS log to fulfill this condition.

In some cases, it is still ambiguous to which Mozilla subproject an issue belongs and it is up to subjective decisions. A clear distinction between Firefox issues and issues of other Mozilla projects is therefore not always possible.

AOSP, OpenStack, and Qt

Hamasaki et al. [Ham+13] provide review data for the Gerrit-based projects AOSP, OpenStack, and Qt in JavaScript Object Notation (JSON) format. Thongtanunam et al. [Tho+14] also use these data for their evaluation, but only the data collected until 2012, while this study uses newer data collected until 2014. The newer data cover the period up to the end of 2014 instead of only to the beginning and middle of 2012, which comprises about 14 times as many reviews in total.

In the latest data set, there are unfortunately no flags which indicate who performed the review and when. However, Gerrit generates messages for reviews. These messages can be identified by characteristic text patterns to find reviews. The date and author of each message are the date of the review and the name of the reviewer.

Expertise Explorer needs two sets of data, one for authorship and one for review history data. Both data sets can be generated from Hamasaki et al.'s JSON formatted files. One advantage of a single data source is that both data sets use consistent names for authors and reviewers.

4.3.4 Results

Analogously to the previous research on reviewer recommendation [Jeo+09; Bal13; Tho+14; Tho+15], accuracy is defined as the percentage of correct recommendations on the total number of recommendations. A recommendation is considered correct if the intersection between the set of recommended reviewers and the set of actual reviewers on the same issue is not empty. Again analogously to previous research, this study does not use the concepts of recall and precision, because all algorithms recommend five reviewers if possible and so there is no trade-off between recall and precision.

Figure 4.9 shows the aggregated top-5 accuracy of the eight recommendation algorithms in the four evaluated FLOSS projects over the course of the study period excluding the training phase. Table 4.3 shows the top-1 and top-5 accuracy for all combinations of algorithms and evaluated projects at the end of the study period.

Table 4.3: Top-1 and top-5 accuracy in % for all algorithms in the evaluated FLOSS projects, excluding data from the one-year training period

	Top-1				Top-5			
	Firefox	AOSP	Open-Stack	Qt	Firefox	AOSP	Open-Stack	Qt
Line 10 Rule	4.84	33.82	22.04	21.33	4.84	33.82	22.04	21.33
Expertise Recommender	4.85	33.83	22.04	21.36	24.85	54.75	51.52	53.33
Number of Changes	12.41	12.03	20.61	3.48	25.57	17.58	36.69	4.97
Code Ownership	4.95	10.37	16.23	2.88	18.70	16.50	31.14	4.70
Expertise Cloud	12.94	32.99	27.91	21.98	37.91	65.23	60.48	57.27
DOA	11.38	11.84	19.91	3.26	24.46	17.63	36.38	4.90
WRC	35.22	46.93	43.44	37.70	69.73	69.66	74.86	68.97
FPS	23.90	44.11	43.13	17.23	58.24	77.32	76.00	42.12

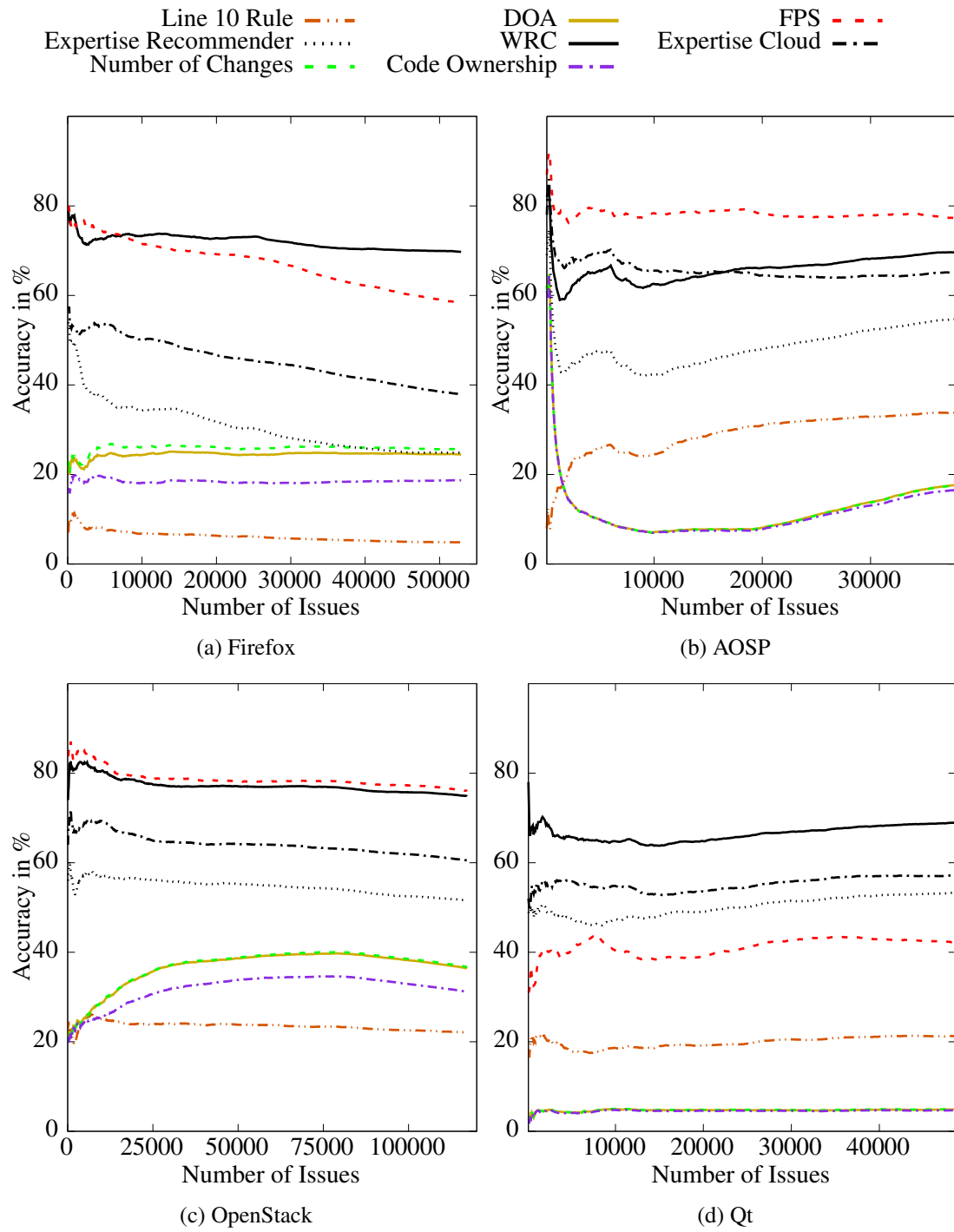


Figure 4.9: Cumulated accuracy over the course of the study period (after training period); the value differences between some curves are lower than the line width used for the drawing. Therefore some lines are on top of each other.

Discussion

The main research goal of this study was an empirical comparison of reviewer recommendation algorithms. Two of the compared algorithms used review expertise for their recommendations and six algorithms used modification expertise. For every evaluated FLOSS project, the algorithm with the highest accuracy was an algorithm based on review expertise. WRC achieved the highest accuracy for all top-1 recommendations and for two out of four top-5 recommendations. Furthermore, WRC performs close to the optimum even for AOSP and OpenStack, where it is not the best top-5 algorithm. Hence, WRC is a reliable algorithm. In contrast, the second-best algorithm FPS seems to be more sensitive to the characteristics of the FLOSS project, as its top-5 accuracy is only 42 % for Qt. This result is surprising, as WRC was merely a byproduct for the calculation of FPS.

In addition to its high accuracy, WRC is a simple algorithm, especially compared to other algorithms with high accuracy like FPS and Expertise Cloud: It requires less input data per recommendation, it is easier to implement, and its value can be computed more quickly. The definition in Section 4.3.2 immediately shows that WRC is simpler than FPS, as the value of WRC is used in the calculation of FPS. Like FPS, Expertise Cloud evaluates expertise not only directly on the reviewed files, but also on files with similar paths. As a consequence, calculating a value for Expertise Cloud or FPS relies on data for a large proportion of files in the FLOSS project. In large FLOSS projects like the ones evaluated in this study, this can involve querying hundreds of thousands of values. FPS and Expertise Cloud therefore have a worse computational performance than the other algorithms including WRC. During the evaluation in this study, FPS and Expertise Cloud needed at least one order of magnitude more time for the computation than the other algorithms.

Reviewer Base Set As described above, algorithms based on review expertise usually perform better in terms of accuracy than those based on modification expertise. However, it has to be taken into account that the former algorithms have an implicit advantage over the latter: they automatically exclude developers without review rights. As shown in Table 4.2, the number of contributors in a FLOSS project exceeds the number of reviewers by a factor of 1.3 to 4.3. By design, the algorithms based on review expertise can only recommend reviewers who had already performed a review in the past and therefore have a higher chance to currently have review rights. Algorithms based on modification expertise may therefore recommend developers who never perform reviews as they lack review rights. If the algorithms would filter out reviewers without review rights before recommendation, this might increase accuracy especially for algorithms based on modification expertise and thereby turn the tide.

The most Competent Reviewer Although it may seem paradoxical first, it may actually be undesirable in a FLOSS project that each patch receives its review from the most competent reviewer. This may be the case if some members of the FLOSS project are the most competent reviewers for so many patches that they cannot review all of them. It might be better if less competent but also less busy reviewers perform some of the reviews, as the disadvantages of long delays may at some point be greater than the disadvantages of slightly worse feedback in the reviews. As another reason, some very competent reviewers might be even more competent

developers. If this is the case in a FLOSS project, good developers might have higher value than good reviewers. These two reasons appear more likely when considering that a small group of about 3.1 % to 8.9 % of all contributors are the core developers of a FLOSS project who contribute 80 % of the code [Gel10]. Whether those trade-offs between reviewer competency and reviewer cost are necessary depends on the specific FLOSS project. Algorithms based on review expertise may have an advantage in this case, as their recommendations do not directly depend on competency but on who reviewed patches in the past – which already reflects current practice in the FLOSS project and the trade-offs that their members decided about.

Vicious Circles If a FLOSS project extensively uses a reviewer recommendation tool, algorithms based on review expertise base their decisions on data that they themselves influenced – they recommend reviewers, these reviewers review patches, and then they assume that these reviewers are well suited to review those types of patches because they did it in the past. This induces the danger of a vicious circle, where some unsuitable reviewers are recommended more and more often for some kind of patch because of prior unsuitable recommendations. When algorithms based on review expertise evaluate their historic review data, they should therefore take into account whether they themselves recommended the reviewer in the data or whether the decision for this reviewer had other reasons. Recommendation algorithms based on modification expertise do not suffer from this problem by design.

Variation Between Projects The accuracy varies stronger between projects for some algorithms than for others. Especially the Line 10 Rule has a top-5 accuracy of only 4.8 % in Firefox, but its accuracy for AOSP, OpenStack, and Qt is 22.0 % to 33.8 %. Similarly, its close relative Expertise Recommender achieves a top-5 accuracy of 24.9 % in Firefox, but 51.5 % to 54.8 % in the other FLOSS projects. AOSP, OpenStack, and Qt seem to follow a policy similar to the Line 10 Rule without a reviewer recommendation system. Simple metrics like the Line 10 Rule and Expertise Recommender do not need a specialized reviewer recommendation system, as the usual development environment already contains a VCS client that can display the VCS log, which immediately shows the experts according to Line 10 Rule and Expertise Recommender. This indicates that tool support is used already for selecting reviewers instead of just intuition, self-selection, or similar methods not using tools. Accordingly, acceptance of a reviewer recommendation tool in these FLOSS projects seems likely. Since the Line 10 Rule and Expertise Recommender are very simple metrics of modification expertise that have been shown to be inferior to metrics like DOA [Fri+14], a reviewer recommendation tool may supply useful additional information to select reviewers for a patch.

Variation Between Algorithms An algorithm either makes a correct or an incorrect prediction for each issue, so the results have a Binomial Distribution. As the algorithms worked on the same data, I used a paired test to check whether the differences in terms of correct top-5 recommendations are statistically significant. A McNemar test [McN47] with the usual continuity correction [Edw48] confirms highly significant ($p < 0.001$) differences for most pairs of algorithms, with exception of Expertise Recommender and DOA in Firefox and DOA and Number of Changes in AOSP, which are not significant at $\alpha = 0.05$, and between Expertise

Recommender and Number of Changes in Firefox ($p = 0.0016$). These results can be reproduced with the lab package for this study [HPG15].

As visible in the graphs in Figure 4.9, the algorithms Number of Changes and DOA generally have very similar accuracies. In Qt and AOSP, Code Ownership is also very similar, and in Firefox, Expertise Recommender has a similar accuracy. Having large data sets, the differences are still highly significant except for the mentioned cases. Even where the differences are statistically highly significant, they are often very small. For example, there are only 87 out of 48 709 recommendations for Qt for which DOA and Number of Changes have differences in correctness, i.e. one of them recommends a correct reviewer while the other does not. This implies that decreasing the expertise value when other authors change a file, which is the main difference between DOA and Number of Changes in regard to reviewer recommendation, has little influence on the overall results.

Time Sensitivity Figure 4.9 shows that some algorithms in some FLOSS projects gain accuracy over the months, while others lose accuracy. Yet others have a more or less stable accuracy. Possible causes are changes to the FLOSS projects' review policies. For example, if each patch requires more reviewers, it will be easier for the algorithms to recommend one of them correctly. If a FLOSS project grows, it will be harder to recommend a reviewer correctly, because the number of selectable reviewers grows. However, these changes can only affect all algorithms at once and do not explain differences between algorithms.

It is possible to distinguish three different categories of algorithms: First, Line 10 Rule and Expertise Recommender are *time-local*, they use only data for their recommendations that were generated shortly before their execution. Second, Code Ownership and DOA are *forgetful* algorithms. These algorithms consider all historical data for their recommendations, but older data have less influence and an author's expertise decreases during phases of inactivity. For DOA and Code Ownership, an author's expertise decreases when other authors modify files that the author has expertise with. For $\delta < 1$, WRC and FPS are also forgetful, but this is not the case in this study. Third are *time-global* algorithms like Expertise Cloud, Number of Changes, as well as WRC and FPS with $\delta = 1$. They use all historic data for their recommendation and all data have the same value, independently from when they were generated.

Forgetful and time-global algorithms may increase their accuracy over time, as they can base their recommendations on a larger data set. However, time-global algorithms may have trouble with changes in the project, because they always take old data into consideration that are based on obsolete and now invalid processes. For example, two reviewers that were active for two years will both be recommended with equal probability, even if one of them was active only five years ago and then left the project and the other is still active. This effect decreases their accuracy over time and may also affect forgetful algorithms that do not properly forget obsolete data. Conversely, if the accuracy of time-local algorithms changes, this must be an effect of the project as a whole.

For Firefox, both time-local algorithms stabilize on a slow loss of predictive performance. A possible cause for the loss is Firefox's growth. Consequently, other algorithms like Expertise Cloud and FPS also slowly lose predictive performance. WRC is more or less stable, though, as the benefits of its growing database cancels out the losses through Firefox's growth. Expertise

Cloud and FPS both use path similarities, which lets them access much more data for each prediction. These additional data possibly include obsolete data and therefore make them more vulnerable to changes in the project structure.

In this study, FPS and therefore also WRC uses the time prioritization factor $\delta = 1$, so effectively data do not age. This time prioritization turned out to be the best value for FPS in a previous study [Tho+14]. However, the next-lower tested value was $\delta = 0.8$, so the optimum may be between 0.8 and 1. In fact, $\delta = 0.8$ seems like a very low value for large FLOSS projects: Firefox, as an example, had 33.5 reviews per day during the study period. This means that expertise gained through a review has only $0.8^{33.5} \approx 0.000567$ times its original value after one day. Reasonable values for δ should therefore be much closer to 1 or the time prioritization factor should not have exponential impact. The exact optimum should be the subject of future research, as a forgetful FPS could have a higher long-term accuracy than a time-global.

Training Period The algorithms' accuracy becomes steady only after some time. In this study, the algorithms are granted a training period of one year. Considering the fluctuations of the graphs in Figure 4.9 during the first issues, a training period of about one and a half years may be more appropriate. However, random effects have a stronger impact in this area due to the low number of total issues, which partially also causes the fluctuations.

Additionally, the algorithms have different required training periods. Figure 4.10 shows aggregated accuracies for Firefox including the training period. Although WRC has the highest accuracy after the whole study period of six years, it takes about 25 000 issues of training to overtake FPS. Apparently, algorithms like FPS and Expertise Cloud have a shorter required training period than algorithms like WRC. WRC considers only expertise with the file itself, while FPS and Expertise Cloud can induce from other related files if the information about a requested file is insufficient.

4.3.5 Threats to Validity

This section discusses threats that endanger the validity of this study's findings. This includes threats to internal validity, including one threat to construct validity, and a threat to external validity.

Internal Validity

Each algorithm's accuracy is the fraction of issues in which the algorithm recommends reviewers that turned out to be actual reviewers in reality. However, actual reviewers may in some cases not be competent reviewers. This is a threat to construct validity. In an extreme case, an algorithm may recommend reviewers more competent than the actual reviewers and have a low accuracy because of the differences between recommended and actual reviewers. Nevertheless, the self-selection of reviewers ensures that actual reviewers are at least a good approximation to competent reviewers. Furthermore, even if a recommendation tool fails to recommend more competent reviewers than a manual selection does, it is still faster, which is a benefit by itself [SMJ10b]. Besides, the discussion in Section 4.3.4 pointed out that recommending the most competent

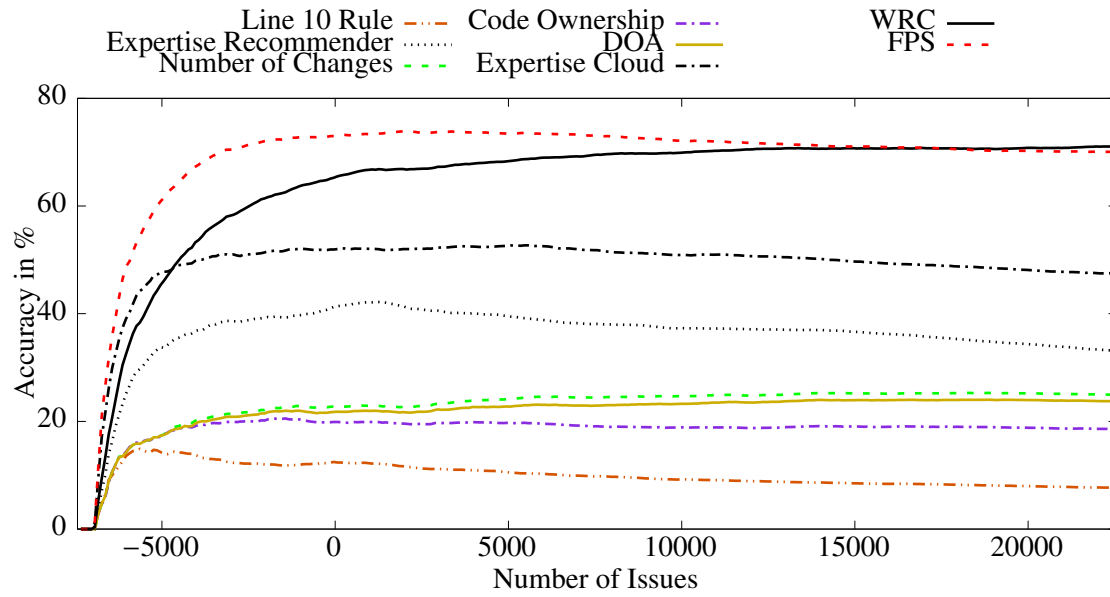


Figure 4.10: Partial Firefox results with focus on the training period

reviewer can be a pitfall. The same type of comparison to actual reviewers has also been used in all previous empirical research of this type, for example [Bal13; Tho+15].

As a considerable special case of the previous threat, the data indicate that AOSP uses the Line 10 Rule to select reviewers (see Variation Between Projects in Section 4.3.4). This and other selection strategies that resemble evaluated algorithms introduce a systematic bias to the comparison. In these cases, a high accuracy only indicates that the algorithm is currently already used, but not that it is necessarily better than other algorithms. However, some algorithms are consistently better than others among all evaluated projects with their diverse possible reviewer selection strategies, which mitigates this problem.

As discussed, if a FLOSS project grows over time, accuracy decreases. The observed results therefore depend on the length of observation. As shown in Table 4.2, each of the four evaluated FLOSS projects had a different study period. This threat impairs comparisons between the four FLOSS projects. Since all algorithms are evaluated within the same time period for each individual FLOSS project, this is no threat to comparisons of different algorithms within each FLOSS project.

The data used for the evaluation contain measurement artifacts. For example, someone might have accidentally flagged a patch in Firefox twice, in which case this reviewer would have been awarded review expertise twice. Expertise Explorer includes consistency checks to filter out corrupt data items. For example, it filters out reviews that a single reviewer performed within two seconds, which not only prevents errors like the initial example but also duplicated review entries due to failures of the FLOSS project's databases.

The exact dates of commits to Firefox's VCS are especially vulnerable to measurement errors. These dates stem from the developers' machines. The VCS respects time zones, but these may be incorrectly configured on the developers' machines. Indeed, the time and date on the developers'

machines may be completely wrong in some cases. Expertise Explorer filters out issues with reviews that happened before their corresponding patch was submitted. I also filtered issues with obviously wrong time stamps, like when some activities happened outside of the study period. Despite these efforts to eliminate measurement artifacts, a complete check of the data was not possible and therefore measurement artifacts may still remain.

When Firefox migrated their VCS from CVS to hg, they imported all existing source code in a single commit with the author “hg@mozilla.com”. The algorithms assumed an enormous modification expertise for hg@mozilla.com for some time after the migration. The training phase however allowed to recover from these types of problems. To test whether hg@mozilla.com biased the results, results were calculated again after filtering out all issues for which one of the algorithms had recommended hg@mozilla.com as a reviewer. This had only minor impact on the results and especially did not change the order of algorithms in terms of accuracy.

External Validity

The Paragraph Variation Between Projects in Section 4.3.4 discusses how the accuracy varies between the four evaluated FLOSS projects. While there is variation between projects, most results apply universally to all four evaluated FLOSS projects. This indicates that the results also apply to many other FLOSS projects and are, in fact, to a large extent independent from the specific FLOSS project. There may be exceptions though, especially for those algorithms that vary stronger already between the four evaluated projects, like the Line 10 Rule.

4.3.6 Conclusion and Future Work on Reviewer Recommendations

Submitted patches to FLOSS projects must usually undergo a code review before they are accepted. Problems with reviewer assignment cause delayed and forgotten patches. This has been identified as an important contribution barrier in Chapter 2. Chapter 3 shows that there are currently no FLOSS patterns lowering this contribution barrier. Reviewer recommendation systems may help with reviewer assignment problems, but only if they recommend competent reviewers with high accuracy. A reviewer recommendation system is a natural extension to a WikiDE, as it fits directly into the WikiDE contribution process described in Section 4.1.3.

The study in this section compared the accuracy of eight reviewer recommendation algorithms using historical data from four large FLOSS projects. Six algorithms are metrics of modification expertise that have been described in literature, but had not been used for reviewer recommendation before. The algorithm FPS was included as a reviewer recommendation algorithm based on review expertise. The eighth algorithm WRC is a pre-stage to FPS, but had not yet been considered as a recommendation algorithm. On the bottom line, WRC achieves the best results with the additional advantage of its simplicity.

These findings raise questions open for future research: Can a hybrid of existing algorithms combine the advantages of those algorithms? Can additional information like the current group of core reviewers improve the algorithms’ performance? Can a better time prioritization improve the accuracy of FPS? Do the tested algorithms really recommend competent reviewers or only common reviewers? When should an algorithm make a trade-off between competency and

workload to relieve competent but busy reviewers? Furthermore, when is a review necessary at all and which patches should undergo a more rigorous review [P*Fro14]?

With a top-5 accuracy between 68.97 % and 74.86 %, WRC can already be used in its current form in reviewer recommendation tools and thereby lower the contribution barrier of delayed reviews for submitted modifications.

4.4 Proof of Concept Realization⁷

This section describes a PoC realization of a WikiDE using the concepts introduced previously in this chapter. Its purpose is to show that these concepts can be realized together in a working environment and to explore the properties and problems of a real-world implementation with the technologies and components currently available.

4.4.1 Environment Overview

Figure 4.11 sketches the computers used in the implementation. GitLabMain and GitLabCIRunner are VMs running with Oracle VM VirtualBox 5.0 [P*Ora16a]. A Debian 8.3.0 Linux for x64 processors [P*Sof16] is installed on both machines. WRCRecommender is a physical machine running Microsoft Windows 8.1 x64 as OS.

4.4.2 Software Forge

GitLabMain hosts an installation of the software GitLab CE [P*Git16c], installed with the Omnibus package for Debian 8 [P*Git16d]. GitLab CE is a FLOSS project for a software forge. It integrates a git [P*git16] server as VCS and provides an issue tracker, a wiki, web-based access to the git repositories, and a CI system. The web site also features a text editor for files stored in git and can be used as code editor. GitLab's user administration includes a rights management system to control user's access to individual repositories.

When users access a code repository that they have read but not write access to, they can still use the integrated text editor to edit a text file. In this case, the whole repository is automatically forked into the editing user's space. Figure 4.12 shows how this looks like. First, a newcomer sees the file and directory structure of a project. This is illustrated in Figure 4.12a with a clone of the FLOSS project BusyBox [P*And15]. When a newcomer selects a file, the PoC realization presents the content. Source code is displayed with syntax highlighting. The example in Figure 4.12b shows the file `coreutils/cat.c` that prints file contents to `stdout`. An edit link leads to a text-editor for the viewed file. The newcomer has modified `coreutils/cat.c` in Figure 4.12c already. The modified version reads only the first file given as a command line parameter instead of any number of files. Newcomers are not allowed to edit the original repository directly, so the PoC realization automatically commits the modifications to a fork within the editing user's space. However, the wizard presents a form to allow the newcomer to submit the modification back to the FLOSS project afterwards. This part corresponds to the actions Modify source code and Publish modification of the WikiDE contribution process described in Section 4.1.3.

⁷A preliminary version of this section was published previously [HG16a].

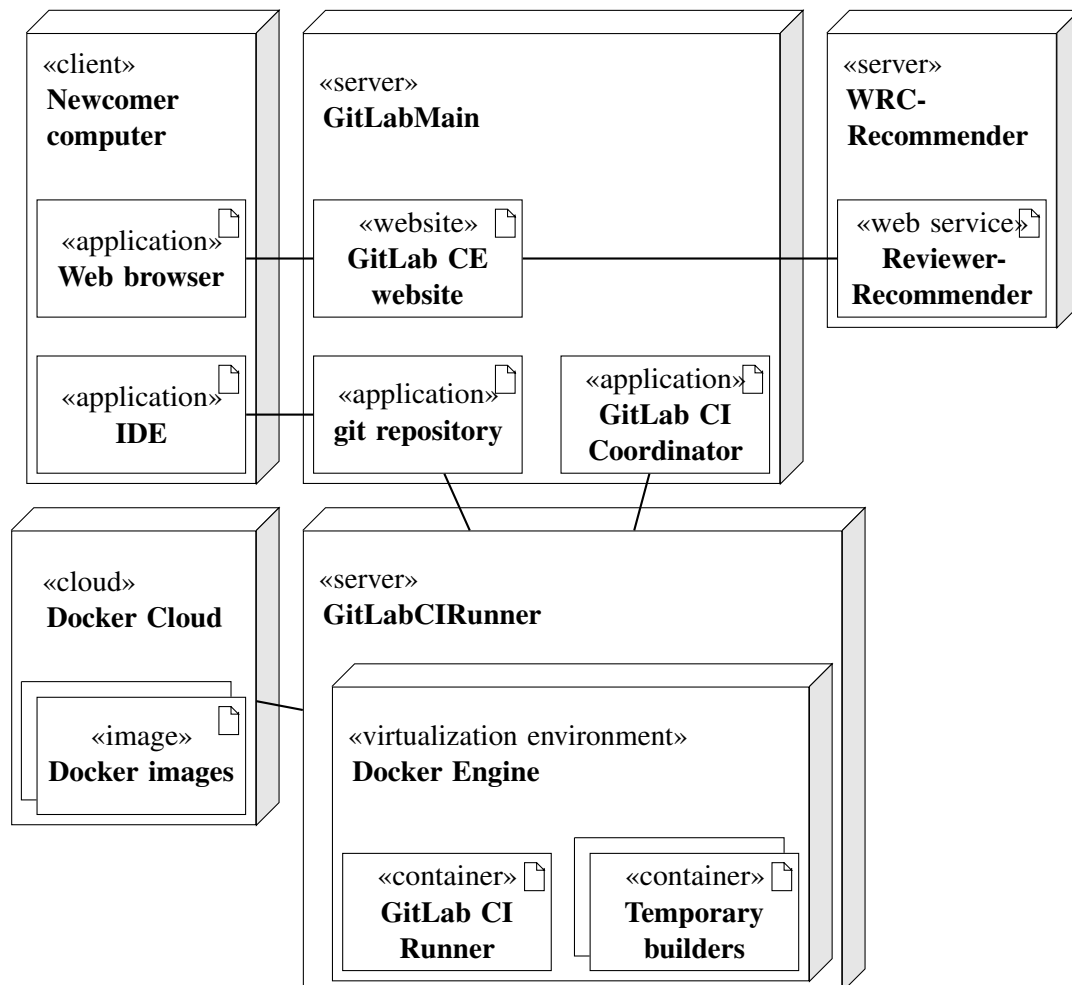


Figure 4.11: UML deployment diagram of the computers and software components in the PoC WikiDE realization

4.4 Proof of Concept Realization

Original Creator / busybox · Files

Search in this project

Find File Download zip

Name	Last Update	Last Commit
applets	5 months ago	Aboriginal linux/must build fixes
applets_sh	4 years ago	applets_sh: Add a few more examples of "shell applets"
arch/386	9 years ago	add comment why preferred stack boundary is 4 on 386
archival	4 months ago	[git] fix recent breakage.
configs	4 months ago	remove systemd support
console-tools	8 months ago	Removes stray empty line from code
coreutils	23 days ago	add support iflagskip_bytes
debianutils	4 months ago	libbb: make parse_chown_usergroup_or_die() set unspecified u...
docs	23 days ago	add support iflagskip_bytes
editargs	4 months ago	fix: do not use statics
editors	about a month ago	sed: make "/w FILE" actually write to FILE. Closes 8251
examples	4 months ago	taskcat: service example
findutils	4 months ago	xargs: make -l imply -r
include	23 days ago	Fix compiling with musl's utmp stubs
init	12 days ago	init: make the command-line rewrite optional
libbb	2 months ago	chpasswd: support -c argument and respect DEFAULT_PSSWD...

Original Creator / busybox · Files

Search in this project

cat.c.149 KB

Raw Blame History Permalink Edit Replace

```
1  /* cat: set save fd=0: */
2  /*
3  * cat implementation for busybox
4  * Copyright (C) 2005 Manuel Novoa III <njh3@codeport.org>
5  *
6  * Licensed under GPLv2, see file LICENSE in this source tree.
7  */
8
9
10 /* _RE_AUDET_SUDO3 compliant */
11 /* http://www.opengroup.org/onlinepubs/96969695/utlities/cat.html */
12
13 /* build: lib-$(COMP25_CAT) ++ cat.o
14 /* build: lib-$(COMP25_NAME) ++ cat.o # more users (2 if without 'm' a try
15 /* build: lib-$(COMP25_LSS) ++ cat.o # less too
16 /* build: lib-$(COMP25_MOUNTD) ++ cat.o # mountab -L
17
18 /* config: config CAT
19 /* config: build "cat"
20 /* config: default y
21 /* config: help
22 /* config: cat is used to concatenate files and print them to the standard
23 /* config: output. Enable this option if you wish to enable the 'cat' utility.
24
25 /* usage: #define cat_trivial_usage
26 /* usage: "FD[1]..."
27 /* usage: #define cat_full_usage "cat"
28 /* usage: "Concatenate files and print them to stdout"
29 /* usage:
30 /* usage: #define cat_example_usage
31 /* usage: "9 cat /dev/urandom"
32 /* usage: "138736.32 32.87"
33
34 #include "libbb.h"
35
36 /* this is a NOFORK applet, be very careful! */
37
38 int bb_cat(char **argv)
39 {
40     int fd;
```

(a) Repository view

(b) File viewer

Original Creator / busybox · Files

Search in this project

You're not allowed to make changes to this project directly. A fork of this project has been created that you can make changes in, so you can submit a merge request.

Edit File Preview Changes

master coreutils/cat.c

text

```
38
39 int bb_cat(char **argv)
40 {
41     int fd;
42     int retval = EXIT_SUCCESS;
43
44     if (!*argv)
45         argv = (char**) &bb_argv_dash;
46
47     // do {
48         fd = open_or_warn_stdin(*argv);
49         if (fd >= 0) {
50             /* This is not a xfunc - never exits */
51             off_t r = bb_copyfd_eof(fd, STDOUT_FILENO);
52             if (fd != STDIN_FILENO)
53                 close(fd);
54             if (r >= 0)
55                 return retval;
56         }
57         retval = EXIT_FAILURE;
58     // } while (*++argv);
59
60     return retval;
61 }
62
63 int cat_main(int argc, char **argv) MAIN_EXTERNALLY_VISIBLE.
```

Commit message

cat shall output only the first file given as argument

Commit Changes

A new branch will be created in your fork and a new merge request will be started.

Cancel

(c) Web-based text editor

Figure 4.12: A newcomers edit-forks a repository.

With these features, GitLab CE fulfills the requirements Self-administration, No Installation, Code Editor, Version Control, and IDE Compatibility of a WikiDE already. Additional configuration of the integrated CI system and a supplement to the issue tracker are required for the requirement *Immediate Feedback* and for the integration of a reviewer recommender as described in Section 4.3.

4.4.3 Immediate Feedback

The machine **GitLabCIRunner** is necessary for GitLab CI, the CI system integrated into GitLab CE. GitLabMain hosts only the GitLab CI Coordinator component that dispatches build jobs to so-called Runners. GitLabCIRunner hosts one such Runner. However, if the runner were installed directly on GitLabCIRunner, this would impose the security issues described in Section 4.2.

GitLab CE differentiates between Project Runners and Shared Runners. Project Runners are assigned to one project exclusively and execute builds only for this specific project. In particular, they do not execute builds for forks of the project. Thus, a Project Runner could not provide *Immediate Feedback* to newcomers who fork a repository using the integrated text editor. A Shared Runner executes builds of all projects.

A configuration of Docker Engine [P*Doc16] realizes the isolation of builds described in Section 4.2 as countermeasure against the security issues. After installation [P*Git16e], Docker created a container using the image `gitlab/gitlab-runner:alpine`. This container uses a token to register as a Shared Runner at GitLabMain.

Each time the Runner on GitLabCIRunner is requested to build a project, Docker spawns a new container for the builds and deletes it after usage. This isolates builds from each other, as requested in Section 4.2. The configuration file of the Runner achieving this behavior is shown in Listing 4.1.

Projects that want to provide *Immediate Feedback* for its newcomers must configure the CI system with a `.gitlab-ci.yml` file. If the file is present, GitLab CI will automatically dispatch builds to available Runners after modifications. Since the GitLabCIRunner hosts a Shared Runner, this also applies to the in-situ forks of newcomers editing a file with the integrated text editor.

However, access to the builds artifacts is only possible if they are configured for output. This is achieved with an `artifacts` directive as shown in Listing 4.2. After each build, the Runner uploads all files that match the configured paths to GitLabMain, which allows all users to download a ZIP package containing the uploaded files. Figure 4.13 shows how a newcomer accesses the modified binaries after forking, subsequent to the sequence in Figure 4.12. In Figure 4.13a, the tab Builds shows running builds of this fork, with and without the newcomer's modification. Once the build is finished, there is a download link to each build's artifacts, as displayed in the screenshot in Figure 4.13b. In the example case, the download is the modified `busybox` executable.

This part corresponds to the action Build application of the WikiDE contribution process described in Section 4.1.3. The newcomer may then execute the built artifact to perform the action Test application behavior.

Listing 4.1: config.toml for GitLab CI Runner

```

concurrent = 2

[[runners]]
  name = "dock-host"
  url = "http://10.0.2.5/ci"
  token = "ecabe8f2931eae195aa9e56806766f"
  tls-ca-file = ""
  executor = "docker"
  [runners.docker]
    image = "ubuntu:wikiderunner"
    privileged = false
    extra_hosts = ["gitlabmain.wikide.local:10.0.2.5"]
    volumes = ["/cache"]

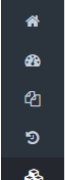
```

Listing 4.2: Example of a .gitlab-ci.yml

```

build:
  script:
    - make defconfig
    - make
  artifacts:
    paths:
      - busybox

```



Running builds from this project							
Status	Build ID	Commit	Ref	Stage	Name	Duration	Finished at
🔄 running	#10	1363b3c1	patch-1	test	build	1 minute 5 seconds	✖
🔄 running	#9	a35e9b0e	patch-1	test	build	1 minute 13 seconds	✖

(a) Build tab with running builds



Running builds from this project							
Status	Build ID	Commit	Ref	Stage	Name	Duration	Finished at
✔ success	#10	1363b3c1	patch-1	test	build	4 minutes 49 seconds	less than a minute ago 
✔ success	#9	a35e9b0e	patch-1	test	build	4 minutes 48 seconds	less than a minute ago 

(b) Finished builds with download link on the right

Figure 4.13: Download of the build artifacts

4.4.4 Reviewer Recommendations

WRCRecommender runs Internet Information Services (IIS) Express [P*Mic15a] as a web server, hosting the web service ReviewerRecommender. ReviewerRecommender was written for this PoC to realize reviewer recommendations as analyzed in Section 4.3 and is available for download under a FLOSS license [Han16].

A project must add ReviewerRecommender as a web hook for “Merge Request events” via the GitLab CE website [P*Git16b]. Every time a developer submits, accepts, or rejects a modification to the project, GitLab CE posts this via HTTP to ReviewerRecommender. The acceptances and rejections deliver the information ReviewerRecommender needs to calculate review expertise using the algorithm WRC described in Section 4.3. If a developer and especially a newcomer submits a modification to the project, ReviewerRecommender calculates the most suitable reviewers for this modification according to WRC. It then posts its recommendation to the submitted modification using GitLab CE’s API [P*Git16a]. Figure 4.14 shows a submitted modification with a recommendation from ReviewerRecommender. The names of the recommended reviewers are prefixed with an @ sign, so they are notified of the recommendation in the status bar on GitLab CE. With the precalculated WRC values, this takes less than five seconds in the PoC environment, so reviews are sped up and newcomers experience less contribution barriers because of long delays until they get feedback for their submission.

4.4.5 Limitations

The realization described in this section is a PoC, so maintainers of a real WikiDE have to take additional considerations into account. The following is a description of the major issues of the PoC that a practical implementation of a WikiDE has to address.

Builds run in their own Docker containers and are therefore isolated from each other. This is an alternative realization of the solution described in Section 4.2. However, attackers may circumvent Docker’s isolation using weaknesses in the Docker implementation. Security Enhanced Linux (SELinux) [P*SEL13] introduces an additional layer of security control to Linux and thereby mitigates these issues. SELinux requires some additional configuration and is not part of the PoC realization.

Docker supports only Linux containers. Thus, the PoC realization can execute builds only on Linux. The PoC realization therefore does not support projects that require other build platforms. This is not a problem inherent to the WikiDE design, though, as other virtualization solutions support all kinds of OSs. The approach presented in Section 4.2, for example, works with other build platforms.

Newcomers that fork a project can access the build output via the menu item Builds in the PoC realization. This possibility is not obvious and newcomers might miss the build. A more user-friendly version of a WikiDE should present the build artifacts more clearly or even integrate them into the commit wizard after a fork.

WRCRecommender calculates recommendations in their own threads within the process of the web service ReviewerRecommender. This is not a good practice, as web servers may kill or restart processes of their web applications if they consider it appropriate, for example if there are no connections from clients for some time. Thus, long running tasks should have their own process

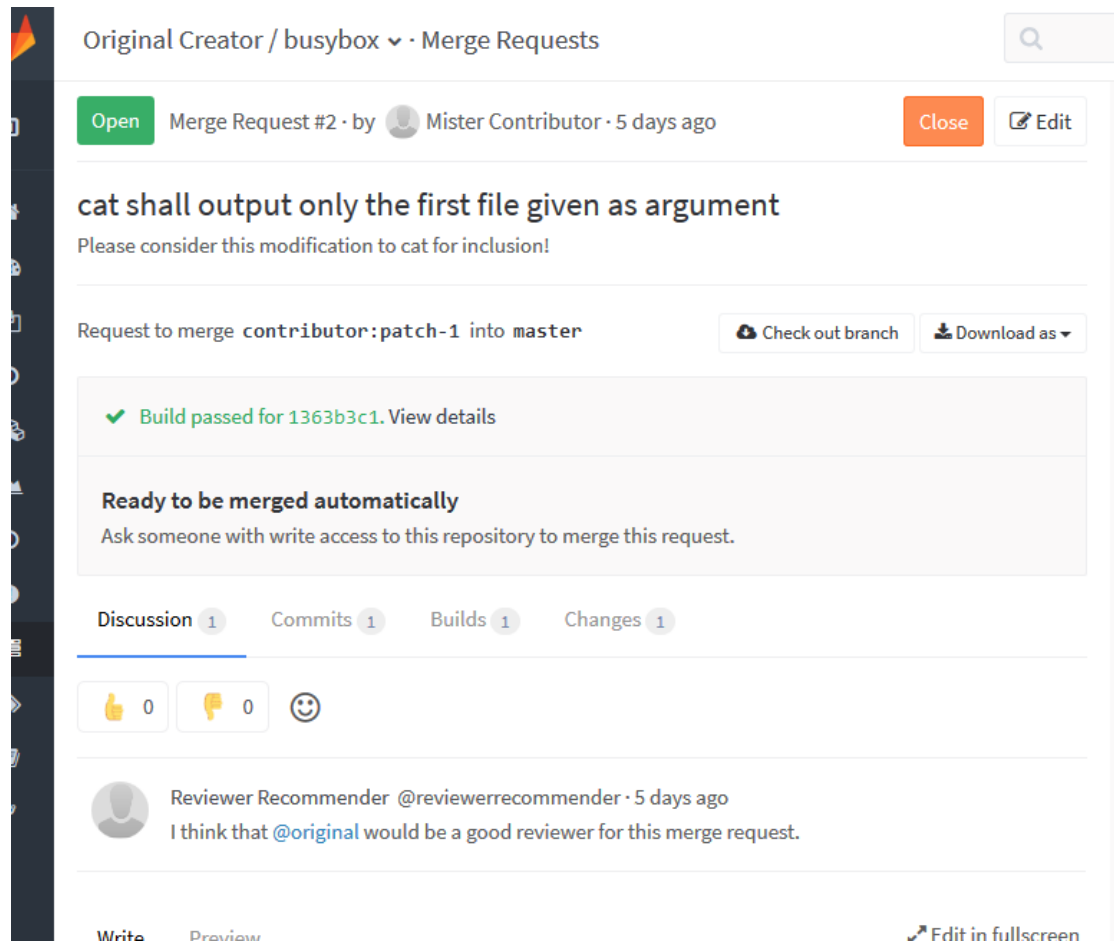


Figure 4.14: The newcomer submitted the modification back, ReviewerRecommender suggests a reviewer.

independent from the web server. For example, the web service could queue recommendation jobs in files and a different application could read the files and calculate the recommendations.

ReviewerRecommender runs on its own server with a different OS than the other servers. This complicates the architecture of the WikiDE realization and increases the effort necessary to install and maintain the system. It would be more convenient if ReviewerRecommender ran on GitLabMain and integrated directly into GitLab CE's architecture instead of using its API. However, such architectural changes require more effort than using the external API and are not necessary for a one-time PoC realization. Besides, the component to calculate WRC had been written in C# as a part of Expertise Explorer (see Section 4.3.2) already and would have had to be rewritten in Ruby for a use directly in GitLab CE.

4.5 Summary

This chapter described a system called WikiDE. A WikiDE uses wiki principles to lower important contribution barriers identified in Chapter 2 that the existing FLOSS patterns shown in Chapter 3 have not addressed sufficiently.

A WikiDE must fulfill the requirements of both an IDE and a wiki. Section 4.1 identified seven requirements and described conceptually how to fulfill these requirements, especially *Self-administration*, *Code Reviews*, and *Immediate Feedback* that seemingly contradict each other.

One important component of a WikiDE is a multitenant CI system. Section 4.2 analyzed what hampers the adoption of CI in FLOSS development: The cost of providing CI and security threats. The remainder of the section presented a technique based on virtualization to counter a large class of security threats endangering multitenant CI system.

Delayed reviewer assignments are an important contribution barrier identified in Chapter 2 that neither the FLOSS patterns nor a WikiDE as described so far remediate. Section 4.3 compares eight algorithms in a reviewer recommendation system to speed up reviewer assignments. A new algorithm called WRC achieves the highest accuracy in the comparison, between 68.97 % and 74.86 % for five recommended reviewers. With these results, WRC can be considered accurate enough for practical usage.

The chapter closes with the documentation of a PoC realization of a WikiDE using the concepts and techniques developed before. The PoC realization shows that a WikiDE can be realized in practice for lowering important contribution barriers.

5 Conclusion

This thesis started with the observation that FLOSS projects have properties that may prevent newcomers from contributing source code. These properties are called contribution barriers. The research goals of this thesis, as defined in Section 1.4, were

1. to identify which contribution barriers affect newcomers,
2. to discover which techniques help to lower contribution barriers, and
3. to analyze how wiki principles can minimize contribution barriers.

Each of these research questions was treated in its own chapter. The research presented in these chapters and thereby the main contributions of this thesis will be summarized in the following.

5.1 Identification of Contribution Barriers

Chapter 2 approached the first research question with a survey of 117 newcomers to the FLOSS projects Mozilla and GNOME. An additional exploratory prepared the ground for this main survey. The main survey is the first to analyze contribution barriers and also the first with newcomers as participants. Adapting the TDM to this type of web-based questionnaire, the survey achieved a response rate of 65.0 %, which is remarkably higher than the best response rate of FLOSS developer surveys so far, which was 38.1 % [BB10]. The survey identified five contribution barriers of high importance:

- Newcomers have difficulties to understand the architecture and find the location in the source code to be changed.
- Setting up the development environment can be troublesome.
- The submission process can be hard to understand.
- Programming the actual solution is sometimes challenging, yet not frustrating.
- Reviews to submitted patches are sometimes delayed or forgotten.

Additionally, the survey responses provided the data to test five hypotheses about contributor motivations, contribution barriers, and their relationship. These hypotheses were based on literature and results from the exploratory survey. With insights from literature and results from the hypothesis tests, a new model for joining FLOSS projects could be proposed.

This model describes that each newcomer takes an individual joining path that depends mostly on the newcomer's demographic properties. The joining path determines which contribution

5 Conclusion

barriers are in effect. The newcomer's mental model about the FLOSS project also depends on the newcomer's demography. The analysis in Section 2.6.4 showed that this mental model determines the newcomer's motivations. The mental model in turn determines the impact of contribution barriers. The model allows to estimate the overall effect of the contribution barriers for individual demographic groups.

FLOSS projects can use this model and the other results of the survey to assess which of their contribution barriers are the most important to be lowered. This requires that FLOSS projects have a toolbox of techniques to lower individual contribution barriers.

5.2 FLOSS Patterns

FLOSS projects use different measures to counter contribution barriers already. The effectiveness with which a FLOSS project may keep its contribution barriers low depends on whether its maintainers know these countermeasures and on the individual skill with which they apply them.

Chapter 3 explores which countermeasures have been successfully applied as FLOSS patterns and therewith tackles the second research question. The pattern format has its roots in architecture, but has been used for other domains and especially software engineering for over two decades. It allows the description of a problem in a context and its solution in a structured way. This allows the analysis of complex interrelated systems such as FLOSS projects and their description in an easily understandable way. Externalizing the knowledge about managing FLOSS projects reduces the problem that the success of a FLOSS project depends on the individual skills and experience of its maintainers.

Accordingly, different authors have written FLOSS patterns that help maintainers of FLOSS projects to optimize various aspect of the FLOSS project. Patterns are related to each other and their relations form a pattern language. Although patterns can be understood only as part of their pattern language, the relations between the existing FLOSS patterns had been widely neglected in research so far. Therefore, Section 3.2 classifies the 35 existing FLOSS patterns into eight categories, discusses their relations, and de-duplicates FLOSS patterns that describe the same problems and solutions from different perspectives.

Among other benefits, this overview of the FLOSS pattern language helps to identify which FLOSS patterns lower contribution barriers. Section 3.3 contains six full FLOSS patterns that lower important contribution barriers identified in Chapter 2. They especially help to understand the FLOSS project's architecture and help to set up the development environment.

Before the contributions of this thesis, research on FLOSS patterns was based mostly on anecdotal evidence. An evaluation of FLOSS patterns on five successful FLOSS projects with different characteristics starts to fill this research gap and shows which FLOSS patterns are common for successful FLOSS projects and which are not required to be successful. The statistical test developed for the evaluation shows how to analyze the representativeness of proposed pattern relationships statistically. The evaluation confirmed the proposed relationships between FLOSS patterns only partially, possibly due to a too small data set. Nevertheless, the developed test and data from the evaluation of FLOSS projects lay the groundwork for future empirical research on the FLOSS pattern language and other pattern languages.

The identified FLOSS patterns help to lower important contribution barriers. However, for

some contribution barriers, no appropriate FLOSS pattern could be found. In particular, there are no FLOSS patterns to ease the submission process or help with delayed reviews. Furthermore, some FLOSS patterns may require much effort for their implementation and FLOSS projects would benefit from a system that allows them to implement these FLOSS patterns with less effort.

5.3 Wiki Development Environments

Wikis are systems that allow its users to modify their content. They provide a technical infrastructure to minimize the barriers to contribution. The wiki principles have proven to be very effective in successful projects like Wikipedia. Chapter 4 explores the possibility to adapt these wiki principles to software development in a system that combines the properties of an IDE and a wiki. Such a system is called a WikiDE.

Section 4.1 develops the concept of a WikiDE. It derives the requirements of a WikiDE from theoretical considerations and based on literature. It proposes an architecture based on existing technology for the realization of a WikiDE and a WikiDE contribution process that resolves the seeming contradiction between three of the requirements. A WikiDE lowers the contribution barrier for setting up a development environment below the levels possible with FLOSS patterns alone. Additionally, it eases the submission of modifications. Both are important contribution barriers, as shown in Chapter 2.

A public CI service is an integral part of a WikiDE. Section 4.2 describes how such a public CI service can be realized securely. This technique is usable when hosting a public CI service even without a WikiDE.

The comparison in Section 4.3 shows which algorithms may be used to recommend reviewers for submitted modifications to closed-source or FLOSS projects. A reviewer recommendation system using such an algorithm is a natural extension to a WikiDE, as it fits into the WikiDE architecture concept and contribution process and it lowers an important contribution barrier: delayed and forgotten reviews.

The PoC realization of a WikiDE in Section 4.4 proves that the proposed concept is realizable. This includes a reviewer recommendation system based on WRC, the best algorithm found in the comparison.

5.4 Future Work

Although Chapter 2 presented the yet largest empirical study of contribution barriers, the participants were newcomers to only two FLOSS projects. Future research could validate the findings, especially the found contribution barriers, on other FLOSS projects. This would provide empirical support for those contribution barriers that apply universally to most FLOSS projects and distinguish them from those contribution barriers that are important only for the two surveyed FLOSS projects, but not to most other FLOSS projects.

The joining model described in Section 2.8 provides the theoretical foundation to assess the importance of individual contribution barriers depending on newcomer demographics. A few concrete examples of demographics with their joining paths and mental models are known, for example students who transition through supporting roles into developer roles and expect to learn

5 Conclusion

something from FLOSS. Future research may identify additional subgroups of newcomers with their individual joining paths and mental models. Additionally, it is not completely clear which specific contribution barriers occur in which joining path. Knowing the concrete demographics, their mapping to joining paths and mental models as well as the contribution barriers specific for each joining path would help to guide FLOSS maintainers how to optimize the technical infrastructure of their FLOSS project to support their contributor community.

The results of the evaluation of FLOSS pattern relationships were partly inconclusive, so a statistically significant proof for the proposed relationships is still missing. Section 3.2 provides objective verification criteria for the existing FLOSS patterns and Section 3.5 provides the statistical method to evaluate FLOSS pattern relationships. Future research can use this methodology to verify the proposed or different pattern relationships on additional FLOSS projects to find conclusive results for or against specific sets of FLOSS pattern relationships.

The FLOSS patterns described in Section 3.3 lower two important contribution barriers. However, the identified set of FLOSS patterns is not intended to be complete. Further analysis of FLOSS projects may yield additional FLOSS patterns to lower these and other contribution barriers. For example, none of the currently written FLOSS patterns help with the submission process, which was identified as an important contribution barrier in Chapter 2. Additionally, technological and social changes may form new techniques to lower contribution barriers in the future that can be identified, described as FLOSS pattern, and classified into the categories developed in Section 3.2.

LaToza et al. [LaT+14] presented the innovative software development system CrowdCode as reviewed in Section 4.1.2. This system might be combined with a WikiDE, so contributors have the choice whether they want to work on a problem that they need for themselves or whether they want to work on a micro-task. Micro-tasks as realized in CrowdCode lift the burden of searching for the right location for a modification in the code and understanding the context of a change, among other side-effects. This might open up a new kind of joining path for newcomers.

Reviewer recommendation is starting to gain research attention. Section 4.3 defined a new algorithm, WRC, that is already good enough to be used in practice. However, the evaluation indicates that there is still space for adjustments to the algorithms' parameters to increase their accuracy. There are also several related questions still open for research, like optimal training periods for the algorithms or how to make algorithms *forgetful*. Improving these recommendation algorithms leads to better recommendations and therefore quicker reviews that potentially need less time and find more errors in the submitted modifications, because the reviewers are better suited for their assigned tasks.

Furthermore, some modifications are more critical than others and therefore require more thorough reviews. For example, modifications to comments or documentation may not require pre-commit reviews as common in software development. Post-commit reviews as common in wikis with textual content might suffice in these cases. A WikiDE might automatically detect the type of modification and the type of required review. This would increase safety for critical modifications that should better receive multiple reviews, and lower contribution barriers for uncritical modifications, as these could use a fast lane and receive only a less rigorous post-commit review from community members with lower review rights.

Primary Sources

- [P*Ama13] Amazon Web Services, Inc. *Amazon EC2 FAQs*. <http://aws.amazon.com/en/ec2/faqs> [accessed 2016-04-26]. Mar. 2013.
- [P*Ama16] Amazon Web Services, Inc. *Amazon EC2 - Virtual Server Hosting*. <https://aws.amazon.com/ec2/> [accessed 2016-03-15]. 2016.
- [P*And15] Erik Andersen. *BusyBox Website*. <https://www.busybox.net/> [accessed 2016-02-23]. Oct. 2015.
- [P*Arg09a] ArgoUML Developers. *ArgoUML Website*. <http://argouml.tigris.org/> [accessed 2016-01-05]. 2009.
- [P*Arg09b] ArgoUML Developers. *ArgoUML Wiki: Design*. <http://argouml.tigris.org/wiki/Design> [accessed 2016-01-05]. Nov. 2009.
- [P*Arg09c] ArgoUML Developers. *ArgoUML Wiki: The big picture for Issues*. http://argouml.tigris.org/wiki/The_big_picture_for_Issues [accessed 2016-01-05]. Mar. 2009.
- [P*Atl16] Atlassian. *JIRA Software*. <https://www.atlassian.com/software/jira/> [accessed 2016-03-14]. 2016.
- [P*Bas15] Basecamp. *Basecamp Website*. <https://basecamp.com/> [accessed 2015-11-09]. 2015.
- [P*Bea16] Beanbag, Inc. *Review Board*. <https://www.reviewboard.org/> [accessed 2016-03-14]. 2016.
- [P*Bun15] Bundler developers. *Bundler Website*. <http://bundler.io/> [accessed 2015-11-12]. 2015.
- [P*Cas14] Tim Case. *A Brief Retrospective of Spree*. Blog. <http://blog.endpoint.com/2014/02/a-brief-retrospective-of-spree.html> [accessed 2015-11-11]. Feb. 2014.
- [P*Cir15a] CircleCI. *CircleCI Website*. <https://circleci.com/> [accessed 2015-12-16]. 2015.
- [P*Cir15b] CircleCI. *spree/spree*. <https://circleci.com/gh/spree/spree> [accessed 2015-11-11]. Nov. 2015.
- [P*Clo15] CloudBees. *BuildHive*. <https://buildhive.cloudbees.com/> [accessed 2015-09-04]. Sept. 2015.
- [P*Clo16] Cloud9 IDE, Inc. *Cloud IDE Website*. <https://c9.io/> [accessed 2016-03-10]. 2016.

Primary Sources

- [P*Cod15] Code Climate. *spree/spree*. <https://codeclimate.com/github/spree/spree> [accessed 2015-11-11]. Nov. 2015.
- [P*cod15] codeweavers. *WineHQ*. <https://www.winehq.org/> [accessed 2015-12-15]. Dec. 2015.
- [P*Con10] Mike Connor. *Stepping down as Firefox module owner*. [news://news.mozilla.org:119/mailman.4488.1265908064.4112.governance@lists.mozilla.org](https://news.mozilla.org:119/mailman.4488.1265908064.4112.governance@lists.mozilla.org) [accessed 2014-07-21]. Feb. 2010.
- [P*Cru15] CruiseControl developers. *CruiseControl Website*. <http://cruisecontrol.sourceforge.net/> [accessed 2015-12-16]. 2015.
- [P*Dal15] Jim Daly. *With Its Spree Commerce Buy, First Data Aims To Bolster Its Online Tech Foundations*. <http://digitaltransactions.net/news/story/With-Its-Spree-Commerce-Buy-First-Data-Aims-To-Bolster-Its-Online-Tech-Foundations> [accessed 2015-11-11]. Sept. 2015.
- [P*Dam06] Peter Damen. *AJAX Remote Desktop Viewer*. <http://www.peterdamen.com/ajaxrd/> [accessed 2016-03-14]. June 2006.
- [P*Doc16] Docker. *Docker Engine*. <https://www.docker.com/products/docker-engine> [accessed 2016-02-18]. 2016.
- [P*Ell10] Guy Ellis. *Open Source Software Stupid Tax*. Guy Ellis' Tech Blog. <http://www.guyellisrocks.com/2010/10/open-source-software-stupid-tax.html> [accessed 2015-05-19]. Oct. 2010.
- [P*EWS15] EWSSoftware. *Sandcastle Help File Builder*. <https://github.com/EWSSoftware/SHFB> [accessed 2016-01-06]. Dec. 2015.
- [P*Fac14] Facebook. *facebook/facebookarchive/facebook-php-sdk/tests GitHub repository*. <https://github.com/facebookarchive/facebook-php-sdk/tree/master/tests> [accessed 2015-12-16]. Apr. 2014.
- [P*Fac15] Facebook. *facebook/facebook-ios-sdk/FBSDKCoreKit/FBSDKCoreKitTests/ GitHub repository*. <https://github.com/facebook/facebook-ios-sdk/tree/master/FBSDKCoreKit/FBSDKCoreKitTests> [accessed 2015-12-16]. Sept. 2015.
- [P*Fir14] Firefox Developers. *modules/zlib in Firefox main development tree as of 2014-07-07T14:34Z*. hg. <https://hg.mozilla.org/mozilla-central/file/085eea991bb9/modules/zlib/src> [accessed 2015-12-10]. July 2014.
- [P*Fir15] Firefox Developers. *Firefox main development tree*. hg. <https://hg.mozilla.org/mozilla-central/> [accessed 2015-12-10]. Dec. 2015.
- [P*Fre12] Free Software Foundation. *License:X11*. <https://directory.fsf.org/wiki/License:X11> [accessed 2015-11-17]. Aug. 2012.
- [P*Fre13] Free Software Foundation. *What is Copyleft?* Electronic. <https://www.gnu.org/copyleft/copyleft.html> [accessed 2014-01-13]. Oct. 2013.

- [P*Fre14] Free Software Foundation, Inc. *GNU Make*. <https://www.gnu.org/software/make/> [accessed 2015-11-16]. Oct. 2014.
- [P*Fre15] Free Software Foundation. *The Free Software Definition*. <http://www.gnu.org/philosophy/free-sw.html> [accessed 2015-09-14]. 2015.
- [P*Fre16] Free Software Foundation. *GPL-Compatible Free Software Licenses*. Electronic. <https://www.gnu.org/licenses/license-list.html#GPLCompatibleLicenses> [accessed 20146-04-26]. Apr. 2016.
- [P*Fro14] Nathan Froyd. *On code review and commit policies*. Blog. <https://blog.mozilla.org/nfroyd/2014/06/23/on-code-review-and-commit-policies/> [accessed 2015-08-27]. June 2014.
- [P*Git15a] GitHub. *All branches of rack/rack*. <https://github.com/rack/rack/branches/all> [accessed 2015-11-18]. Nov. 2015.
- [P*Git15b] GitHub. *All branches of ruby/rake*. <https://github.com/ruby/rake/branches/all> [accessed 2015-11-17]. Nov. 2015.
- [P*Git15c] GitHub. *GitHub Website*. <https://github.com/> [accessed 2015-07-22]. July 2015.
- [P*Git15d] GitHub. *rack/rack issues*. <https://github.com/rack/rack/issues> [accessed 2015-11-17]. Nov. 2015.
- [P*Git15e] GitHub. *rails/rails Issues*. <https://github.com/rails/rails/issues> [accessed 2015-11-09]. Nov. 2015.
- [P*Git15f] GitHub. *Ruby on Rails*. <https://github.com/rails> [accessed 2015-11-09]. Nov. 2015.
- [P*Git15g] GitHub. *ruby/rake Issues*. <https://github.com/ruby/rake/issues> [accessed 2015-11-17]. Nov. 2015.
- [P*Git15h] GitHub. *Spree Commerce*. <https://github.com/spree> [accessed 2015-11-11]. Nov. 2015.
- [P*Git15i] GitHub. *spree-contrib*. <https://github.com/spree-contrib> [accessed 2015-11-11]. Nov. 2015.
- [P*Git15j] GitHub. *spree/spree Issues*. <https://github.com/spree/spree/issues> [accessed 2015-11-11]. Nov. 2015.
- [P*Git16a] GitLab. *GitLab Documentation / GitLab API*. <http://doc.gitlab.com/ce/api/README.html> [accessed 2016-03-03]. Feb. 2016.
- [P*Git16b] GitLab. *GitLab Documentation / Web hooks / Merge request events*. http://doc.gitlab.com/ce/web_hooks/web_hooks.html#merge-request-events [accessed 2016-03-03]. Feb. 2016.
- [P*Git16c] GitLab. *GitLab Website*. <https://about.gitlab.com/> [accessed 2016-02-17]. Feb. 2016.
- [P*Git16d] GitLab. *Install a GitLab CE Omnibus package on Debian 8*. <https://about.gitlab.com/downloads/#debian8> [accessed 2016-02-18]. Jan. 2016.

Primary Sources

- [P*Git16e] GitLab. *Run gitlab-runner in a container*. <https://gitlab.com/gitlab-org/gitlab-ci-multi-runner/blob/master/docs/install/docker.md> [accessed 2016-02-17]. Feb. 2016.
- [P*git16] git. *git Website*. <https://www.git-scm.com/> [accessed 2016-02-18]. Feb. 2016.
- [P*GNO14] GNOME Wiki. *Community*. <https://wiki.gnome.org/Community> [accessed 2015-07-17]. Mar. 2014.
- [P*GNO15a] GNOME Wiki. *GNOME Love*. <https://wiki.gnome.org/GnomeLove> [accessed 2015-07-15]. Feb. 2015.
- [P*GNO15b] GNOME Wiki. *The GNOME Tour: finding your way around GNOME*. <https://wiki.gnome.org/GnomeLove/ProjectTour> [accessed 2015-07-17]. June 2015.
- [P*Goo15a] Google Developers. *Google Summer of Code*. <https://developers.google.com/open-source/gsoc/> [accessed 2016-01-05]. 2015.
- [P*Goo15b] Google Groups. *rack-devel mailing list*. <https://groups.google.com/group/rack-devel> [accessed 2015-11-18]. Oct. 2015.
- [P*Goo15c] Google Groups. *Ruby on Rails: Talk*. <http://groups.google.com/group/rubyonrails-talk> [accessed 2015-11-10]. Nov. 2015.
- [P*Goo15d] Google Groups. *spree-user mailing list*. <https://groups.google.com/group/spree-user> [accessed 2015-11-12]. Nov. 2015.
- [P*Goo16] Google Developers. *Gerrit Code Review*. <https://www.gerritcodereview.com/> [accessed 2016-03-14]. Mar. 2016.
- [P*Gre12] Lars Gregori. *VirtualBox Plugin snapshot testing branch*. <https://github.com/jenkinsci/virtualbox-plugin/tree/snap> [accessed 2016-03-14]. Mar. 2012.
- [P*Han04] David Heinemeier Hansson. *Initial commit to Ruby on Rails*. <https://github.com/rails/rails/commit/db045dbbf60b53dbe013ef25554fd013baf88134> [accessed 2015-11-06]. Nov. 2004.
- [P*Hoo13] Hood & Strong LLP. *Mozilla Foundation and Subsidiaries – December 31, 2012 and 2011*. Independent Auditors' Report and Consolidated Financial Statements. https://static.mozilla.com/moco/en-US/pdf/Mozilla_Audited_Financials_2012.pdf [accessed 2015-12-10]. Nov. 2013.
- [P*Ica97] Miguel de Icaza. *The GNOME Desktop project*. email to the GTK mailing list. <https://mail.gnome.org/archives/gtk-list/1997-August/msg00123.html> [accessed 2016-04-26]. Aug. 1997.
- [P*Jen15] Jenkins developers. *Jenkins Website*. <https://jenkins-ci.org/> [accessed 2015-12-16]. 2015.
- [P*Jen16] Jenkins Developers. *Extend Jenkins*. <https://wiki.jenkins-ci.org/display/JENKINS/Extend+Jenkins> [accessed 2016-01-05]. Jan. 2016.

- [P*Joh13] Christian John. *Fork of the VirtualBox Plugin snapshot testing branch for Secure Build Servers*. <https://github.com/paluno/virtualbox-plugin/tree/snap> [accessed 2016-03-14]. Mar. 2013.
- [P*Kod16] Koding. *Koding Website*. <https://koding.com/> [accessed 2016-03-10]. 2016.
- [P*Lee15] Barosl Lee. *Homu*. <http://homu.io/> [accessed 2016-03-14]. 2015.
- [P*Leg15] Legion of the Bouncy Castle Inc. *The Legion of the Bouncy Castle Website*. <https://www.bouncycastle.org/> [accessed 2015-12-16]. 2015.
- [P*Lin15] Linux Kernel Organization, Inc. *The Linux Kernel Archives*. <https://www.kernel.org/> [accessed 2015-12-15]. Dec. 2015.
- [P*Mat10] Josh Matthews. *Getting Involved with Mozilla*. Blog entry. <http://www.joshmatthews.net/blog/2010/03/getting-involve-with-mozilla/> [accessed 2015-12-10]. Mar. 2010.
- [P*MB15] Tom Mornini and Mike Blumtritt. *no rake.gemspec in repository*. <https://github.com/ruby/rake/issues/60> [accessed 2015-11-17]. Sept. 2015.
- [P*Med16] MediaWiki Contributors. *Manual:Developing extensions*. https://www.mediawiki.org/wiki/Manual:Developing_extensions [accessed 2016-01-05]. Jan. 2016.
- [P*Men14] Daniel Mendler. *Olelo wiki*. <http://gitwiki.org/> [accessed 2016-03-09]. Apr. 2014.
- [P*Mer14] Mercurial Wiki authors. *Mercurial Queues Extension*. Wiki article. <https://mercurial.selenic.com/wiki/MqExtension> [accessed 2015-06-17]. Aug. 2014.
- [P*Mer16] Mercurial. *Mercurial Website*. <https://www.mercurial-scm.org/> [accessed 2016-03-14]. 2016.
- [P*MGG15] Evgeny Mandrikov, Lars Gregori, and Henri Gomez. *VirtualBox Plugin*. <https://wiki.jenkins-ci.org/display/JENKINS/VirtualBox+Plugin> [accessed 2016-03-14]. Mar. 2015.
- [P*Mic15a] Microsoft. *Internet Information Services (IIS) 10.0 Express*. <https://www.microsoft.com/en-us/download/details.aspx?id=48264> [accessed 2016-03-03]. Dec. 2015.
- [P*Mic15b] Microsoft TechNet. *Security Update Lifecycle*. <https://technet.microsoft.com/en-us/security/dn436305> [accessed 2015-09-04]. Microsoft, 2015.
- [P*Mic16a] Microsoft Developer Network. *MSBuild Project File Schema Reference*. <http://msdn.microsoft.com/en-us/library/5dy88c2e.aspx> [accessed 2016-03-15]. Microsoft, 2016.
- [P*Mic16b] Microsoft Developer Network. *Office client development*. <https://msdn.microsoft.com/en-us/library/dn833103> [accessed 2016-01-05]. Jan. 2016.

Primary Sources

- [P*Min08] MinGW.org. *MSYS*. <http://www.mingw.org/wiki/MSYS> [accessed 2016-04-26]. Jan. 2008.
- [P*Moi15] MoinMoin contributors. *MoinMoin Website*. <https://moinmo.in/> [accessed 2016-03-10]. Feb. 2015.
- [P*Moz01] Mozilla Foundation. *Bugzilla@Mozilla*. <https://bugzilla.mozilla.org> [accessed 2015-12-10]. Aug. 2001.
- [P*Moz02] Mozilla Foundation. *Phoenix 0.1 (Pescadero) Release Notes and FAQ*. http://website-archive.mozilla.org/www.mozilla.org/firefox_releases/en-US/firefox/releases/0.1.html [accessed 2015-12-10]. Aug. 2002.
- [P*Moz06] Mozilla Corporation. *mozilla.dev.planning*. [news://news.mozilla.org/mozilla.dev.planning](http://news.mozilla.org/mozilla.dev.planning) [accessed 2014-07-23]. Jan. 2006.
- [P*Moz08a] Mozilla Corporation. *mozilla.dev.builds*. [news://news.mozilla.org/mozilla.dev.builds](http://news.mozilla.org/mozilla.dev.builds) [accessed 2015-10-07]. Jan. 2008.
- [P*Moz08b] Mozilla Foundation. *The Mozilla Manifesto*. <https://www.mozilla.org/en-US/about/manifesto/details/> [accessed 2016-03-23]. Nov. 2008.
- [P*Moz09] Mozilla Foundation. *Mozilla Bug 459257 - MozillaBuild fails on WinXP x64 due to errors in guess-msvc/start-* scripts*. https://bugzilla.mozilla.org/show_bug.cgi?id=459257 [accessed 2016-04-26]. July 2009.
- [P*Moz11] Mozilla Foundation. *Release of Firefox 4*. [archive.org, https://web.archive.org/web/20110323003139/http://www.mozilla.com/en-US/firefox/new/](http://web.archive.org/web/20110323003139/http://www.mozilla.com/en-US/firefox/new/) [accessed 2015-01-23]. Mar. 2011.
- [P*Moz12] Mozilla Developer Network. *Windows Build Prerequisites, Revision 320427*. [https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Windows_Prerequisites\\$revision/320427](https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Windows_Prerequisites$revision/320427) [accessed 2015-12-16]. Mozilla Developer Network, Oct. 2012.
- [P*Moz13] Mozilla Wiki Contributors. *Release Management/Release Process*. https://wiki.mozilla.org/Release_Management/Release_Process [accessed 2015-12-10]. May 2013.
- [P*Moz14a] Mozilla Developer Network. *Mercurial Queues*. https://developer.mozilla.org/en-US/docs/Mercurial_Queues [accessed 2015-06-17]. Dec. 2014.
- [P*Moz14b] Mozilla Developer Network. *Mozilla Source Code Directory Structure*. https://developer.mozilla.org/en/docs/Mozilla_Source_Code_Directory_Structure [accessed 2015-12-10]. May 2014.
- [P*Moz14c] Mozilla Foundation. *Bugzilla Database Snapshot*. <https://bugzilla.mozilla.org/page.cgi?id=researchers.html> [accessed 2014-07-23, not available anymore]. May 2014.
- [P*Moz14d] Mozilla Wiki Contributors. *Contribute/Coding/Mentoring*. <https://wiki.mozilla.org/Contribute/Coding/Mentoring> [accessed 2015-09-02]. Sept. 2014.

- [P*Moz14e] Mozilla Wiki Contributors. *Modules/Firefox (Differences between revisions)*. <https://wiki.mozilla.org/index.php?title=Modules%2FFirefox&diff=957835&oldid=688200> [accessed 2015-12-10]. Mar. 2014.
- [P*Moz14f] Mozilla Wiki Contributors. *SummerOfCode*. <https://wiki.mozilla.org/SummerOfCode> [accessed 2015-01-21]. Dec. 2014.
- [P*Moz14g] Mozilla Wiki Contributors. *Tree Rules/Integration*. https://wiki.mozilla.org/Tree_Rules/Integration [accessed 2015-12-10]. Dec. 2014.
- [P*Moz15a] Mozilla Corporation. *Mercurial Repositories list*. <https://hg.mozilla.org/> [accessed 2015-01-15]. 2015.
- [P*Moz15b] Mozilla Corporation. *Mozilla GitHub main page*. <https://github.com/mozilla/> [accessed 2015-01-05]. 2015.
- [P*Moz15c] Mozilla Developer Network. *Add-on SDK*. <https://developer.mozilla.org/en-US/Add-ons/SDK> [accessed 2015-12-10]. Dec. 2015.
- [P*Moz15d] Mozilla Developer Network. *Build Instructions*. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions [accessed 2015-07-16]. June 2015.
- [P*Moz15e] Mozilla Developer Network. *Contributing to the Mozilla codebase*. <https://developer.mozilla.org/en-US/docs/Introduction> [accessed 2015-12-10]. Dec. 2015.
- [P*Moz15f] Mozilla Developer Network. *How to Submit a Patch (Preparation)*. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/How_to_Submit_a_Patch#Preparation [accessed 2015-12-10]. Aug. 2015.
- [P*Moz15g] Mozilla Developer Network. *Linux prerequisites, Section Most Distros - One Line Bootstrap Command*. https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Linux_Prerequisites#Most_Distros_-_One_Line_Bootstrap_Command [accessed 2015-12-10]. Oct. 2015.
- [P*Moz15h] Mozilla Developer Network. *MDN Website*. <https://developer.mozilla.org/> [accessed 2015-10-07]. 2015.
- [P*Moz15i] Mozilla Developer Network. *Mozilla automated testing*. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Automated_testing [accessed 2015-12-10]. Oct. 2015.
- [P*Moz15j] Mozilla Developer Network. *Plugins*. <https://developer.mozilla.org/en-US/Add-ons/Plugins> [accessed 2015-12-10]. Oct. 2015.
- [P*Moz15k] Mozilla Developer Network. *Running automated tests*. https://developer.mozilla.org/en-US/docs/Running_automated_tests [accessed 2015-12-10]. June 2015.

Primary Sources

- [P*Moz15l] Mozilla Developer Network. *Windows build prerequisites, Section MozillaBuild*. https://developer.mozilla.org/en-US/docs/Developer_Guide/Build_Instructions/Windows_Prerequisites#MozillaBuild [accessed 2015-12-10]. Nov. 2015.
- [P*Moz15m] Mozilla Foundation. *Firefox Website*. <https://www.mozilla.org/en-US/firefox/products/> [accessed 2015-10-30]. 2015.
- [P*Moz15n] Mozilla Foundation. *Governance*. <https://www.mozilla.org/en-US/about/governance/> [accessed 2015-12-10]. 2015.
- [P*Moz15o] Mozilla Foundation. *Main hg repository for Firefox*. <https://hg.mozilla.org/mozilla-central/> [accessed 2015-05-05]. May 2015.
- [P*Moz15p] Mozilla Foundation. *Mozilla Foundation License Policy*. <https://www.mozilla.org/MPL/license-policy.html> [accessed 2015-12-10]. 2015.
- [P*Moz15q] Mozilla Foundation. *Mozilla Modules and Module Owners*. <https://www.mozilla.org/hacking/module-ownership.html> [accessed 2015-12-10]. 2015.
- [P*Moz15r] Mozilla Foundation. *Mozilla Persona Website*. <https://login.persona.org/> [accessed 2015-01-15]. Jan. 2015.
- [P*Moz15s] Mozilla Foundation. *Mozilla Public License*. <https://www.mozilla.org/MPL/2.0/> [accessed 2015-12-10]. 2015.
- [P*Moz15t] Mozilla Foundation. *Mozilla Website*. <https://www.mozilla.org/> [accessed 2015-12-10]. 2015.
- [P*Moz15u] Mozilla Foundation. *mozilla/gecko-dev GitHub repository*. <https://github.com/mozilla/gecko-dev> [accessed 2015-10-30]. Oct. 2015.
- [P*Moz15v] Mozilla Foundation. *The Mozilla Foundation*. <https://www.mozilla.org/foundation/> [accessed 2015-12-10]. 2015.
- [P*Moz15w] Mozilla Wiki Contributors. *BMO/Whiteboard*. Wiki Article. <https://wiki.mozilla.org/BMO/Whiteboard> [accessed 2015-10-12]. Jan. 2015.
- [P*Moz15x] Mozilla Wiki Contributors. *Modules*. <https://wiki.mozilla.org/Modules> [accessed 2015-12-10]. May 2015.
- [P*Moz15y] Mozilla Wiki Contributors. *Modules/Firefox*. <https://wiki.mozilla.org/index.php?title=Modules/Firefox&oldid=1076675> [accessed 2015-11-06]. May 2015.
- [P*Moz15z] Mozilla Wiki Contributors. *ReleaseEngineering/TryServer*. <https://wiki.mozilla.org/Try> [accessed 2015-12-10]. Nov. 2015.
- [P*Moz15aa] Mozilla Wiki Contributors. *Tree Rules*. https://wiki.mozilla.org/Tree_Rules [accessed 2015-12-10]. Mar. 2015.
- [P*Nea10] Kyle Neath. *Introducing Organizations*. Github Blog. <https://github.com/blog/674-introducing-organizations> [accessed 2015-10-30]. June 2010.

- [P*Net98] Netscape Communications Corporation. *Netscape Announces mozilla.org, a Dedicated Team and Web Site Supporting Development of Free Client Source Code*. <http://wp.netscape.com/newsref/pr/newsrelease577.html> [original, broken], <https://web.archive.org/web/20021004080737/wp.netscape.com/newsref/pr/newsrelease577.html> [Internet Archive, accessed 2015-11-27]. Feb. 1998.
- [P*Neu07] Christian Neukirchen. *Initial Commit to Rake*. <https://github.com/rack/rack/commit/22f015e82b6c707f8887c6c951a300f597e0d877> [accessed 2015-11-17]. Feb. 2007.
- [P*Neu13] Christian Neukirchen. *Rack: a Ruby Webserver Interface*. <https://rack.github.io/> [accessed 2015-11-17]. Feb. 2013.
- [P*Nig11] Johnathan Nightingale. *Re: Firefox module owner change*. Email. <https://groups.google.com/d/topic/mozilla.dev.apps.firefox/DBh4q25LlXw> [accessed 2015-12-10]. Sept. 2011.
- [P*Nor15a] Xavier Noria. *Rails Contributors - All time*. <http://contributors.rubyonrails.org> [accessed 2015-11-10]. Nov. 2015.
- [P*Nor15b] Xavier Noria. *Rails Contributors - This year*. <http://contributors.rubyonrails.org/contributors/in-time-window/this-year> [accessed 2015-11-11]. Nov. 2015.
- [P*Ols16] Christoph Olszowka. *The popularity rating of Ruby on Rails explained*. <https://www.ruby-toolbox.com/projects/rails/popularity> [accessed 2016-01-13]. Jan. 2016.
- [P*Ope04] Open Source Initiative. *The Open Source Definition*. <http://www.opensource.org/docs/osd> [accessed 2016-04-06]. 2004.
- [P*Ope14] Open Source Initiative. *Licenses that are popular and widely used or with strong communities*. Electronically. <http://opensource.org/licenses/category> [accessed 2014-01-13]. Jan. 2014.
- [P*Ope15a] Open Hub. *Firefox project analysis*. <https://www.openhub.net/projects/9/analyses/26246450.xml> [accessed 2015-11-19, free API key required] <https://www.openhub.net/p/firefox/commits/summary> [accessed 2015-11-19, less accurate data]. Nov. 2015.
- [P*Ope15b] Open Hub. *Open HUB Projects*. <https://www.openhub.net/explore/projects> [accessed 2015-10-27]. Oct. 2015.
- [P*Ope15c] Open Hub. *Ruby on Rails activity facts*. https://www.openhub.net/projects/34/analyses/latest/activity_facts.xml [accessed 2015-11-10, free API key required] <https://www.openhub.net/p/rails/contributors/summary> [accessed 2015-11-10, publicly available]. Nov. 2015.

Primary Sources

- [P*Ope15d] Open Hub. *Spree activity facts*. https://www.openhub.net/projects/13145/analyses/latest/activity_facts.xml [accessed 2015-11-12, free API key required] <https://www.openhub.net/p/spree/contributors/summary> [accessed 2015-11-12, publicly available]. Nov. 2015.
- [P*Ope16a] Open Hub. *Eclipse Platform Project – Estimated Cost*. https://www.openhub.net/p/eclipse/estimated_cost [accessed 2016-03-09]. Ohloh, Mar. 2016.
- [P*Ope16b] Open Hub. *Projects by Activity Level*. https://www.openhub.net/p?sort=activity_level [accessed 2016-03-15]. Mar. 2016.
- [P*Ope16c] OpenStreetMap contributors. *OpenStreetMap Website*. <https://www.openstreetmap.org/> [accessed 2016-02-08]. 2016.
- [P*Ora15a] Oracle. *Hudson Website*. <http://www.hudson-ci.org/> [accessed 2015-12-16]. 2015.
- [P*Ora15b] Oracle. *Javadoc Tool*. <http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html> [accessed 2016-01-05]. 2015.
- [P*Ora16a] Oracle. *VirtualBox Website*. <https://www.virtualbox.org/> [accessed 2016-02-17]. Feb. 2016.
- [P*Ora16b] Oracle Corporation. *Oracle VM VirtualBox User Manual*. Version 5.0.16. <http://www.virtualbox.org/manual/> [accessed 2016-03-14]. Mar. 2016.
- [P*Pal15] Matt Palmer. *Contributing*. <https://github.com/rack/rack-contrib/blob/master/CONTRIBUTING.md> [2015-11-18]. June 2015.
- [P*Peb15] Pebble. *CloudPebble Website*. <https://cloudpebble.net/> [accessed 2016-03-10]. 2015.
- [P*Peb16] Pebble. *Pebble Guides – App Resources*. <https://developer.pebble.com/guides/app-resources/> [accessed 2016-03-10]. 2016.
- [P*Pra10a] Winston Prakash. *Hudson Execution and Scheduling Architecture*. <http://www.hudson-ci.org/docs/HudsonArch-Execution.pdf> [accessed 2016-04-26]. Oracle, Nov. 2010.
- [P*Pra10b] Winston Prakash. *Hudson Security Architecture*. <http://www.hudson-ci.org/docs/HudsonArch-Security.pdf> [accessed 2016-04-26]. Oracle, Dec. 2010.
- [P*RA15] Greg Roelofs and Mark Adler. *zlib Website*. <http://zlib.net/> [accessed 2015-12-10]. Aug. 2015.
- [P*Rac12] Rack community. *Release a new patch version of Rack 1.4 with good release number*. <https://github.com/rack/rack/issues/322> [accessed 2015-11-18]. Jan. 2012.
- [P*Rac15a] Rack community. *rack/rack github repository*. <https://github.com/rack/rack-contrib> [accessed 2015-11-17]. July 2015.

- [P*Rac15b] Rack community. *rack/rack wiki*. <https://github.com/rack/rack/wiki> [accessed 2015-11-17]. 2015.
- [P*Rac15c] Rack developers. *COPYING*. <https://github.com/rack/rack/blob/master/COPYING> [accessed 2015-11-17]. Feb. 2015.
- [P*Rac15d] Rack developers. *Module: Rack*. <http://www.rubydoc.info/github/rack/rack/Rack> [accessed 2015-11-17]. Nov. 2015.
- [P*Rac15e] Rack developers. *Rack applications on GitHub*. <https://github.com/rack/rack/blob/master/SPEC> [accessed 2015-11-17]. Aug. 2015.
- [P*Rac15f] Rack developers. *Rack applications on RubyDoc*. <http://www.rubydoc.info/github/rack/rack/master/file/SPEC> [accessed 2015-11-17]. Aug. 2015.
- [P*Rac15g] Rack developers. *rack/rack github repository*. <https://github.com/rack/rack> [accessed 2015-11-17]. Nov. 2015.
- [P*Rai13] RailsGuides. *Ruby on Rails 4.0 Release Notes*. http://guides.rubyonrails.org/4_0_release_notes.html [accessed 2015-11-09]. June 2013.
- [P*Rai15a] RailsGuides. *Contributing to Ruby on Rails*. http://edgeguides.rubyonrails.org/contributing_to_ruby_on_rails.html [accessed 2015-11-09]. Nov. 2015.
- [P*Rai15b] RailsGuides. *Rails on Rack*. http://guides.rubyonrails.org/rails_on_rack.html [accessed 2015-11-09]. Nov. 2015.
- [P*Rai15c] RailsGuides. *Ruby on Rails Guides*. <http://guides.rubyonrails.org/> [accessed 2015-11-09]. Aug. 2015.
- [P*Rak13] Rake developers. *RAKE – Ruby Make*. <http://rake.rubyforge.org/> [accessed 2015-11-17]. 2013.
- [P*Rak14] Rake developers. *jimweirich/rake github repository*. <https://github.com/jimweirich/rake> [accessed 2015-11-17]. Nov. 2014.
- [P*Rak15a] Rake developers. *RAKE – Ruby Make*. <http://docs.seattlerb.org/rake/> [accessed 2015-11-12]. 2015.
- [P*Rak15b] Rake developers. *ruby/rake github repository*. <https://github.com/ruby/rake> [accessed 2015-11-17]. Nov. 2015.
- [P*Red15] Red Hat, Inc. *Cygwin Website*. <https://cygwin.com/> [accessed 2015-12-16]. 2015.
- [P*Rub12] RubyGems.org. *bacon*. <https://rubygems.org/gems/bacon> [accessed 2015-11-17]. Dec. 2012.
- [P*Rub14a] Ruby on Rails developers. *MIT License*. <https://github.com/rails/rails/blob/master/railties/MIT-LICENSE> [accessed 2015-11-11]. Dec. 2014.
- [P*Rub14b] RubyGems.org. *All versions of rake*. <https://rubygems.org/gems/rake/versions> [accessed 2015-11-17]. Dec. 2014.

Primary Sources

- [P*Rub14c] RubyGems.org. *rake*. <https://rubygems.org/gems/rake> [accessed 2015-11-16]. Dec. 2014.
- [P*Rub15a] Ruby on Rails. *Ruby on Rails Website*. <http://rubyonrails.org/> [accessed 2015-11-09]. Nov. 2015.
- [P*Rub15b] Ruby on Rails. *The core team*. <http://rubyonrails.org/core/> [accessed 2015-11-06]. 2015.
- [P*Rub15c] Ruby on Rails developers. *A Virtual Machine for Ruby on Rails Core Development*. <https://github.com/rails/rails-dev-box> [accessed 2015-11-10]. June 2015.
- [P*Rub15d] Ruby on Rails developers. *Rails License*. <https://github.com/rails/rails#license> [accessed 2015-11-06]. Sept. 2015.
- [P*Rub15e] Ruby on Rails developers. *rails.gemspec*. <https://github.com/rails/rails/blob/master/rails.gemspec> [accessed 2015-11-06]. Sept. 2015.
- [P*Rub15f] Ruby on Rails developers. *rails/rails github repository*. <https://github.com/rails/rails> [accessed 2015-11-09]. Nov. 2015.
- [P*Rub15g] Ruby on Rails developers. *Ruby on Rails API*. <http://api.rubyonrails.org/> [accessed 2015-11-09]. Aug. 2015.
- [P*Rub15h] RubyGems.org. *All versions of rack*. <https://rubygems.org/gems/rack/versions> [accessed 2015-11-18]. June 2015.
- [P*Rub15i] RubyGems.org. *All versions of railties*. <https://rubygems.org/gems/railties/versions> [accessed 2015-11-10]. Nov. 2015.
- [P*Rub15j] RubyGems.org. *All versions of spree*. <https://rubygems.org/gems/spree/versions> [accessed 2015-11-11]. Nov. 2015.
- [P*Rub15k] RubyGems.org. *bundler*. <https://rubygems.org/gems/bundler> [accessed 2015-11-06]. July 2015.
- [P*Rub15l] RubyGems.org. *hoe*. <https://rubygems.org/gems/hoe> [accessed 2015-11-16]. Sept. 2015.
- [P*Rub15m] RubyGems.org. *rack*. <https://rubygems.org/gems/rack> [accessed 2015-11-17]. June 2015.
- [P*Sch07] Sean Schofield. *Initial commits to railscart*. <https://code.google.com/archive/p/railscart/source/default/commits?page=7> [accessed 2015-11-11]. July 2007.
- [P*Sch15] Sean Schofield. *The Future of Spree Open Source Software*. <https://spreecommerce.com/blog/future-of-spree-oss> [accessed 2015-10-28]. Oct. 2015.
- [P*SEL13] SELinux developers. *SELinux Website*. <http://selinuxproject.org/> [accessed 2016-02-22]. May 2013.

- [P*Sha11a] Mike Shaver. *changes in Firefox review policy*. email. <https://groups.google.com/d/topic/mozilla.dev.apps.firefox/nGByxSzFFlw> [accessed 2015-12-10]. Sept. 2011.
- [P*Sha11b] Mike Shaver. *Modules/Firefox (Differences between revisions)*. <https://wiki.mozilla.org/index.php?title=Modules%2FFirefox&diff=353093&oldid=351420> [accessed 2015-12-10]. Sept. 2011.
- [P*Sin15] Sinatra developers. *Rack Middleware*. <https://github.com/sinatra/sinatra#rack-middleware> [accessed 2015-11-17]. Sept. 2015.
- [P*Sla15] Slashdot Media. *SourceForge Website*. <https://sourceforge.net> [accessed 2015-09-04]. Sept. 2015.
- [P*Sof16] Software in the Public Interest, Inc. *Debian Website*. <https://www.debian.org/> [accessed 2016-02-17]. Feb. 2016.
- [P*Spr13] Spree developers. *Core Team*. <https://github.com/spree/spree/wiki/Core-Team> [accessed 2015-11-12]. Apr. 2013.
- [P*Spr15a] Spree Commerce Inc. *Developer Guide: Contributing to Spree*. <https://guides.spreecommerce.com/developer/contributing.html> [accessed 2015-11-11]. Mar. 2015.
- [P*Spr15b] Spree Commerce Inc. *Developer Guide: Extensions*. https://guides.spreecommerce.com/developer/extensions_tutorial.html [accessed 2015-11-11]. Apr. 2015.
- [P*Spr15c] Spree Commerce Inc. *Developer Guide: Getting Help*. https://guides.spreecommerce.com/developer/getting_help.html [accessed 2015-11-12]. Jan. 2015.
- [P*Spr15d] Spree Commerce Inc. *Developer Guide: Navigating the Source*. <https://guides.spreecommerce.com/developer/navigating.html> [accessed 2015-11-11]. Jan. 2015.
- [P*Spr15e] Spree Commerce Inc. *Do I need a Spree storefront to use Wombat?* <https://support.wombat.co/hc/en-us/articles/202421430-Do-I-need-a-Spree-storefront-to-use-Wombat-> [accessed 2015-11-11]. 2015.
- [P*Spr15f] Spree Commerce Inc. *Our Leadership Team*. https://spreecommerce.com/about_us [accessed 2015-11-12]. 2015.
- [P*Spr15g] Spree Commerce Inc. *SpreeCommerce Website*. <https://spreecommerce.com/> [accessed 2015-11-11]. 2015.
- [P*Spr15h] Spree Commerce Inc. *The Spree API: Documentation*. <https://guides.spreecommerce.com/api/> [accessed 2015-11-11]. 2015.
- [P*Spr15i] Spree Commerce Inc. *Wombat Website*. <https://wombat.co/> [accessed 2015-11-11]. 2015.
- [P*Spr15j] Spree developers. *Spree License*. <https://github.com/spree/spree/blob/master/license.md> [accessed 2015-11-11]. Jan. 2015.

Primary Sources

- [P*Spr15k] Spree developers. *spree/spree GitHub repository*. <https://github.com/spree/spree> [accessed 2015-11-11]. Oct. 2015.
- [P*Spr15l] Spree developers. *spree/spree/frontend/spree_frontend.gemspec*. https://github.com/spree/spree/blob/master/frontend/spree_frontend.gemspec [accessed 2015-11-11]. Oct. 2015.
- [P*Sto15] Hank Stoeve. *Issue Stats for spree/spree*. <http://issuestats.com/github/spree/spree> [accessed 2015-11-11]. 2015.
- [P*Tc16] Peter Thoeny and contributing authors. *TWiki Website*. <http://twiki.org/> [accessed 2016-03-09]. Mar. 2016.
- [P*The12] The Apache Software Foundation. *log4j FAQ, Why should I donate my extensions to log4j back to the project?* <https://logging.apache.org/log4j/1.2/faq.html#a4.1> [accessed 2015-05-19]. May 2012.
- [P*The14] The Cincinnati Enquirer. *James Nolan Weirich*. <http://www.legacy.com/obituaries/cincinnati/obituary.aspx?n=james-nolan-weirich&pid=169779270> [accessed 2015-11-17]. Feb. 2014.
- [P*The15a] The Apache Software Foundation. *Apache HTTP Server Project*. <https://httpd.apache.org/> [accessed 2015-04-27]. Apr. 2015.
- [P*The15b] The Apache Software Foundation. *Apache Subversion*. <https://subversion.apache.org/> [accessed 2016-03-14]. Dec. 2015.
- [P*The15c] The Eclipse Foundation. *Desktop IDEs*. <http://eclipse.org/ide/> [accessed 2015-03-02]. Mar. 2015.
- [P*The15d] The GNOME Project. *GNOME Bugzilla*. <https://bugzilla.gnome.org/> [accessed 2015-09-08]. Sept. 2015.
- [P*The15e] The GNOME Project. *GNOME Website*. <https://www.gnome.org/> [accessed 2015-07-16]. July 2015.
- [P*The16] The Eclipse Foundation. *Eclipse Downloads*. <http://www.eclipse.org/downloads/> [accessed 2016-04-26]. Apr. 2016.
- [P*Tig09] TightVNC Group. *TightVNC Java Viewer version 1.3.10*. <http://www.tightvnc.com/doc/java/README.txt> [accessed 2016-03-14]. 2009.
- [P*Tra15] Travis CI. *Travis CI*. <https://travis-ci.org> [accessed 2015-09-04]. Sept. 2015.
- [P*tra15] trac developers. *Trac Component Architecture*. <http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture> [accessed 2016-01-05]. Dec. 2015.
- [P*Tuc14] James Tucker. *[ANN] Rack, Change of Maintainer & Status*. <https://groups.google.com/forum/#!msg/rack-devel/P8oOycVBaH0/1bm4eERJWPQJ> [accessed 2015-11-18]. Aug. 2014.
- [P*Wei03] Jim Weirich. *Initial Commit to Rake*. <https://github.com/ruby/rake/commit/de564de4e2f189e1b133e4adf05ab8fc820a044b#diff-a8514d6a381bae5851617358adb0621d> [accessed 2015-11-16]. Oct. 2003.

- [P*Wik15a] Wikimedia. *MediaWiki Website*. <https://www.mediawiki.org/> [accessed 2016-03-09]. Dec. 2015.
- [P*Wik15b] Wikipedia. *Flagged revisions/Sighted versions*. http://en.wikipedia.org/wiki/Wikipedia:Flagged_revisions/Sighted_versions [accessed 2016-04-26]. Wikipedia, Nov. 2015.
- [P*Wik15c] Wikipedia. *Relevance*. <http://en.wikipedia.org/wiki/Wikipedia:Relevance> [accessed 2016-03-09]. Wikipedia, Dec. 2015.
- [P*Wik16a] Wikipedia. *Be Bold*. https://en.wikipedia.org/wiki/Wikipedia:Be_bold [accessed 2016-03-09]. Wikipedia, Jan. 2016.
- [P*Wik16b] Wikipedia. *Edit warring*. https://en.wikipedia.org/wiki/Wikipedia:Edit_warring [accessed 2016-03-10]. Wikipedia, Mar. 2016.
- [P*Wik16c] Wikipedia. *Vandalism*. <http://en.wikipedia.org/wiki/Wikipedia:Vandalism> [accessed 2016-03-09]. Wikipedia, Feb. 2016.
- [P*Wik16d] Wikipedia. *Wikipedia Website*. <https://www.wikipedia.org/> [accessed 2016-02-08]. 2016.
- [P*XWi16] XWiki community. *XWiki Website*. <http://www.xwiki.org> [accessed 2016-03-09]. Mar. 2016.

Secondary Sources

- [ACB09] Paul J. Adams, Andrea Capiluppi, and Cornelia Boldyreff. “Coordination and productivity issues in free software: The role of brooks’ law”. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. Sept. 2009, pp. 319–328.
- [AD05] Ademar Aguiar and Gabriel David. “WikiWiki weaving heterogeneous software artifacts”. In: *WikiSym ’05: Proceedings of the 2005 international symposium on Wikis*. San Diego, California: ACM, 2005, pp. 67–74. ISBN: 1-59593-111-2.
- [ADG08] Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. “Expertise Identification and Visualization from CVS”. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR ’08. Leipzig, Germany: ACM, 2008, pp. 125–128. ISBN: 978-1-60558-024-1.
- [Ale+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. New York: Oxford University Press, 1977.
- [Ale79] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979. ISBN: 0195024028.
- [AM07] John Anvik and Gail C. Murphy. “Determining Implementation Expertise from Bug Reports”. In: *Mining Software Repositories, 2007. ICSE Workshops MSR ’07. Fourth International Workshop on*. May 2007, pp. 2–2.
- [AM11] John Anvik and Gail C. Murphy. “Reducing the effort of bug report triage: Recommenders for development-oriented decisions”. In: *ACM Trans. Softw. Eng. Methodol.* 20.3 (Aug. 2011), 10:1–10:35. ISSN: 1049-331X.
- [Ant+08] Ionannis Stefan Koch, Ksenia Fraczek, and Anis Hadzisalihovic. *Study of Available Tools*. EU Framework deliverable. FLOSSMetrics Consortium, Oct. 2008.
- [AR08] Craig Anslow and Dirk Riehle. “Towards end-user programming with wikis”. In: *Proceedings of the 4th international workshop on End-user software engineering*. WEUSE ’08. Leipzig, Germany: ACM, 2008, pp. 61–65. ISBN: 978-1-60558-034-0.
- [ASP10] Abu Zafar Abbasi, Zubair A. Shaikh, and Syed Arsalan Pervez. “XYLUS: A Virtualized Programming Environment”. In: *Proceedings of 2nd IEEE International Conference on Information and Emerging Technologies (ICIET-2010)*. June 2010.

- [Aut12] Authors of the Portland Pattern Repository's Wiki. *Wiki IDE*. <http://c2.com/cgi/wiki?WikiIde> [accessed 2016-03-09]. Portland Pattern Repository's Wiki, May 2012.
- [Avi+04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *Dependable and Secure Computing, IEEE Transactions on* 1.1 (2004), pp. 11–33. ISSN: 1545-5971.
- [Bal13] Vipin Balachandran. "Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation". In: *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*. San Francisco, CA, USA: IEEE Press, 2013, pp. 931–940. ISBN: 978-1-4673-3076-3.
- [Bar16] Dominik Bartholdi. *Total Jenkins installations*. <http://stats.jenkins-ci.org/jenkins-stats/svg/total-jenkins.svg> [accessed 2016-03-14]. Mar. 2016.
- [Bay+12] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. "The Secret Life of Patches: A Firefox Case Study". In: *Reverse Engineering (WCRE), 2012 19th Working Conference on*. 2012, pp. 447–455.
- [BB10] Hind Benbya and Nassim Belbaly. "Understanding Developers' Motives in Open Source Projects: A Multi-Theoretical Framework". In: *Communications of the AIS* 27 (Jan. 2010), pp. 589–610.
- [BGD07] Christian Bird, Alex Gourley, and Premkumar Devanbu. "Detecting Patch Submission and Acceptance in OSS Projects". In: *Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 26–. ISBN: 0-7695-2950-X.
- [Bir+07] Christian Bird, Alex Gourley, Premkumar Devanbu, Anand Swaminathan, and Greta Hsu. "Open Borders? Immigration in Open Source Projects". In: *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*. May 2007, p. 6.
- [BK07] Michael Bächle and Paul Kirchberg. "Ruby on Rails". In: *Software, IEEE* 24.6 (Nov. 2007), pp. 105–108. ISSN: 0740-7459.
- [Bla76] Duncan Black. "Partial justification of the Borda count". English. In: *Public Choice* 28.1 (1976), pp. 1–15. ISSN: 0048-5829.
- [Bro10] Frederick P. Brooks, Jr. *The Design of Design*. Addison-Wesley, 2010.
- [Bro87] Frederick P. Brooks, Jr. "No Silver Bullet—Essence and Accidents of Software Engineering". In: *Computer* 20 (4 Apr. 1987), pp. 10–19. ISSN: 0018-9162.
- [Bro95] Frederick P. Brooks, Jr. *The Mythical Man-Month*. 20th Anniversary Edition. Addison Wesley, 1995.

Secondary Sources

- [BS08] Andrew Begel and Beth Simon. “Struggles of New College Graduates in Their First Software Development Job”. In: *SIGCSE Bull.* 40.1 (Mar. 2008), pp. 226–230. ISSN: 0097-8418.
- [Bug15] Yegor Bugayenko. *10 Hosted Continuous Integration Services for a Private Repository*. <http://www.yegor256.com/2014/10/05/ten-hosted-continuous-integration-services.html> [accessed 2016-03-14]. Nov. 2015.
- [Bur+06] Luciana S. Buriol, Carlos Castillo, Debora Donato, Stefano Leonardi, and Stefano Millozzi. “Temporal Analysis of the Wikigraph”. In: *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*. WI ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 45–51. ISBN: 0-7695-2747-7.
- [CA09] Andrea Capiluppi and Paul J. Adams. “Reassessing Brooks’ Law for the Free Software Community”. English. In: *Open Source Ecosystems: Diverse Communities Interacting*. Ed. by Cornelia Boldyreff, Kevin Crowston, Björn Lundell, and Anthony I. Wasserman. Vol. 299. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2009, pp. 274–283. ISBN: 978-3-642-02031-5.
- [CAH03] Kevin Crowston, Hala Annabi, and James Howison. “Defining open source software project success”. In: *in Proceedings of the 24th International Conference on Information Systems (ICIS 2003)*. 2003, pp. 327–340.
- [Can+12] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. “Who is Going to Mentor Newcomers in Open Source Projects?”. In: *Proceedings of SIGSOFT’12/FSE-20*. 2012.
- [CC06] Gerardo Canfora and Luigi Cerulo. “Supporting change request assignment in open source development”. In: *Proceedings of the 2006 ACM symposium on Applied computing*. SAC ’06. Dijon, France: ACM, 2006, pp. 1767–1772. ISBN: 1-59593-108-2.
- [CF09] Jorge Colazo and Yulin Fang. “Impact of license choice on Open Source Software development activity”. In: *Journal of the American Society for Information Science and Technology* 60.5 (2009), pp. 997–1011. ISSN: 1532-2890.
- [CFP11] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. 1.7. <http://svnbook.red-bean.com/> [accessed 2016-04-26]. O’Reilly Media, 2011. ISBN: 0596510330.
- [CH05a] James O. Coplien and Neil B. Harrison. *Organizational patterns of agile software development*. Pearson Prentice Hall, 2005.
- [CH05b] Kevin Crowston and James Howison. “The social structure of free and open source software development”. In: *First Monday* 10.2 (Feb. 2005).
- [CH06a] Kevin Crowston and James Howison. “Assessing the Health of Open Source Communities”. In: *Computer* 39 (5 May 2006), pp. 89–91. ISSN: 0018-9162.

- [CH06b] Kevin Crowston and James Howison. “Hierarchy and centralization in Free and Open Source Software team communications”. English. In: *Knowledge, Technology & Policy* 18 (2006), pp. 65–85. ISSN: 0897-1986.
- [Cha09] Scott Chacon. *Pro Git*. 1st Edition. <https://git-scm.com/book/en/v1> [accessed 2016-04-26]. Apress, 2009. ISBN: 1430218339.
- [Che76] Peter Pin-Shan Chen. “The Entity-relationship Model—Toward a Unified View of Data”. In: *ACM Trans. Database Syst.* 1.1 (Mar. 1976), pp. 9–36. ISSN: 0362-5915.
- [CJ04] Mihai Christodorescu and Somesh Jha. “Testing malware detectors”. In: *SIGSOFT Softw. Eng. Notes* 29.4 (July 2004), pp. 34–44. ISSN: 0163-5948.
- [CLM03] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. “Evidences in the evolution of OS projects through Changelog Analyses”. In: *Taking Stock of the Bazaar: Proceedings of the 3rd Workshop on Open Source Software Engineering*. May 2003, pp. 19–24.
- [CM07] Andrea Capiluppi and Martin Michlmayr. “From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects”. In: *OSS2007: Open Source Development, Adoption and Innovation (IFIP 2.13)*. Vol. 234/2007. Springer, 2007, pp. 31–44.
- [CMP07] Stefano Comino, Fabio M. Manenti, and Maria Laura Parisi. “From planning to mature: On the success of open source projects”. In: *Research Policy* 36 (2007), pp. 1575–1586.
- [Con68] Melvin E. Conway. “How do Committees Invent?” In: *Datamation* 14.5 (Apr. 1968), pp. 28–31.
- [Cos16] CosmoCode. *WikiMatrix Website*. <http://www.wikimatrix.org/> [accessed 2016-03-09]. Mar. 2016.
- [Cox98] Alan Cox. “Cathedrals, Bazaars and the Town Council”. In: *Slashdot*. 1998.
- [Cro+05] Kevin Crowston, Hala Annabi, James Howison, and Chengetai Masango. “Effective Work Practices for FLOSS Development: A Model and Propositions”. In: *HICSS '05: Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2005. ISBN: 978-0-7695-2268-5.
- [CSD10] Indushobha Chengalur-Smith, Anna Sidorova, and Sherae L. Daniel. “Sustainability of Free/Libre Open Source Projects: A Longitudinal Study”. In: *Journal of the Association for Information Systems* 11.11 (Nov. 2010), pp. 657–683.
- [Cun96] Ward Cunningham. “EPISODES: A Pattern Language of Competitive Development”. In: *Pattern Languages of Program Design 2*. Ed. by John M. Vlissides, James O. Coplien, and Norman L. Kerth. AddisonWesley, 1996. Chap. 23, pp. 371–388.

Secondary Sources

- [Dab+12] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. “Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository”. In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW ’12. Seattle, Washington, USA: ACM, 2012, pp. 1277–1286. ISBN: 978-1-4503-1086-4.
- [Dag+10] Barthélemy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. “Moving into a new software project landscape”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 275–284. ISBN: 978-1-60558-719-6.
- [Dav+14a] Jennifer L. Davidson, Umme Ayda Mannan, Rithika Naik, Ishneet Dua, and Carlos Jensen. “Older Adults and Free/Open Source Software: A Diary Study of First-Time Contributors”. In: *Proceedings of The International Symposium on Open Collaboration*. OpenSym ’14. Berlin, Germany: ACM, 2014, 5:1–5:10. ISBN: 978-1-4503-3016-9.
- [Dav+14b] Jennifer L. Davidson, Rithika Naik, Umme Ayda Mannan, Amir Azarbakht, and Carlos Jensen. “On older adults in free/open source software: reflections of contributors and community leaders”. In: *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. July 2014, pp. 93–100.
- [Dil99] Don A. Dillman. *Mail and Internet Surveys: The Tailored Design Method*. Wiley, 1999. ISBN: 0471323543.
- [DMG07] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007. ISBN: 0321336380.
- [DR08a] Paul A. David and Francesco Rullani. “Dynamics of innovation in an “open source” collaboration environment: lurking, laboring, and launching FLOSS projects on SourceForge”. In: *Industrial and Corporate Change* 17.4 (2008), pp. 647–710.
- [DR08b] Amit Deshpande and Dirk Riehle. “Continuous Integration in Open Source Software Development”. In: *Open Source Development, Communities and Quality*. Ed. by Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi. Vol. 275. IFIP International Federation for Information Processing. Springer Boston, 2008, pp. 273–280.
- [DR08c] Amit Deshpande and Dirk Riehle. “The Total Growth of Open Source”. In: *Fourth International Conference on Open Source Software*. Sept. 2008.
- [Duc05] Nicolas Ducheneaut. “Socialization in an Open Source Software Community: A Socio-Technical Analysis”. English. In: *Computer Supported Cooperative Work (CSCW)* 14.4 (2005), pp. 323–368. ISSN: 0925-9724.

- [DWA03] Paul A. David, Andrew H. Waterman, and Seema Arora. *FLOSS-US – The Free/Libre & Open Source Software Survey for 2003*. http://www-siepr.stanford.edu/programs/OpenSoftware_David/FLOSS-US-Report.pdf [accessed 2016-03-23]. Sept. 2003.
- [Ebe+08] Anja Ebersbach, Markus Glaser, Richard Heigl, and Alexander Warta. “The Wiki Concept”. In: *Wiki*. Springer Berlin Heidelberg, 2008, pp. 11–34. ISBN: 978-3-540-68173-1.
- [Edw48] Allen L. Edwards. “Note on the “correction for continuity” in testing the significance of the difference between correlated proportions”. English. In: *Psychometrika* 13.3 (1948), pp. 185–187. ISSN: 0033-3123.
- [Eic+01] Stephen G. Eick, Todd L. Graves, Ian F. Karr, J.S. Marron, and Audris Mockus. “Does code decay? Assessing the evidence from change management data”. In: *Software Engineering, IEEE Transactions on* 27.1 (Jan. 2001), pp. 1–12. ISSN: 0098-5589.
- [End+14] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. “How Do API Documentation and Static Typing Affect API Usability?” In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 632–642. ISBN: 978-1-4503-2756-5.
- [Fes57] Leon Festinger. *A Theory of Cognitive Dissonance*. Stanford University Press, June 1957.
- [Fin03] Martin Fink. *The Business and Economics of Linux and Open Source*. Prentice Hall, 2003. ISBN: 0-13-047677-3.
- [Fow06] Martin Fowler. *Continuous Integration*. <http://martinfowler.com/articles/continuousIntegration.html> [accessed 2016-04-26]. May 2006.
- [Fow96] Martin Fowler. “Accountability and Organizational Structures”. In: *Pattern Languages of Program Design 2*. Ed. by John M. Vlissides, James O. Coplien, and Norman L. Kerth. AddisonWesley, 1996. Chap. 22, pp. 353–370.
- [Fri+10] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. “A degree-of-knowledge model to capture source code familiarity”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE ’10. Cape Town, South Africa: ACM, 2010, pp. 385–394. ISBN: 978-1-60558-719-6.
- [Fri+14] Thomas Fritz, Gail C. Murphy, Emerson Murphy-Hill, Jingwen Ou, and Emily Hill. “Degree-of-knowledge: Modeling a Developer’s Knowledge of Code”. In: *ACM Trans. Softw. Eng. Methodol.* 23.2 (Apr. 2014), 14:1–14:42. ISSN: 1049-331X.
- [GA04] Cristina Gacek and Budi Arief. “The Many Meanings of Open Source”. In: *IEEE Software* 21 (2004), pp. 34–40.
- [Gai80] John Gaito. “Measurement scales and statistics: Resurgence of an old misconception”. In: *Psychological Bulletin* 87.3 (May 1980), pp. 564–567.

Secondary Sources

- [Gar15] Gartner. *Gartner Says Tablet Sales Continue to Be Slow in 2015*. Press Release. <https://www.gartner.com/newsroom/id/2954317> [accessed 2016-02-25]. Jan. 2015.
- [GD05] Marylène Gagné and Edward L. Deci. “Self-determination theory and work motivation”. In: *Journal of Organizational Behavior* 26.4 (2005), pp. 331–362. ISSN: 1099-1379.
- [Gel10] Jaco Geldenhuys. “Finding the Core Developers”. In: *39th Euromicro Conference on Software Engineering and Advanced Applications*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 447–450. ISBN: 978-0-7695-4170-9.
- [Ger03] Daniel M. German. “The GNOME project: a case study of open source, global software development”. In: *Software Process: Improvement and Practice* 8.4 (2003), pp. 201–215.
- [GG05] Alan Grosskurth and Michael W. Godfrey. “A Reference Architecture for Web Browsers”. In: *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*. Sept. 2005, pp. 661–664.
- [GH12a] Volker Gruhn and Christoph Hannebauer. “Components of a Wiki-based software development environment”. In: *E-Learning, E-Management and E-Services (IS3e), 2012 IEEE Symposium on*. 2012.
- [GH12b] Volker Gruhn and Christoph Hannebauer. “Using Wikis as Software Development Environments”. In: *Proceedings of the 11th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT_12)*. Frontiers in Artificial Intelligence and Applications. IOS International Publisher, Oct. 2012.
- [GHJ13] Volker Gruhn, Christoph Hannebauer, and Christian John. “Security of Public Continuous Integration Services”. In: *Proceedings of the 2013 Joint International Symposium on Wikis and Open Collaboration (WikiSym + OpenSym 2013)*. ACM, 2013. ISBN: 978-1-4503-1852-5.
- [Gho05] Rishab Aiyer Ghosh. *Understanding Free Software Developers: Findings from the FLOSS Study*. <http://www.flossproject.org/papers/ghosh-2005.pdf> [accessed 2016-03-23]. 2005.
- [Gîr+05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. “How developers drive software evolution”. In: *Principles of Software Evolution, Eighth International Workshop on*. Sept. 2005, pp. 113–122.
- [GPD14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. “An Exploratory Study of the Pull-based Software Development Model”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 345–355. ISBN: 978-1-4503-2756-5.
- [GR05] Tal Garfinkel and Mendel Rosenblum. “When virtual is harder than real: security challenges in virtual machine based computing environments”. In: *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*. Santa Fe, NM, 2005.

- [GS12] Mário Luís Guimarães and António Rito Silva. “Improving Early Detection of Software Merge Conflicts”. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. 2012.
- [GS67] E. Eugene Grant and Harold Sackman. “An Exploratory Investigation of Programmer Performance Under On-Line and Off-Line Conditions”. In: *Human Factors in Electronics, IEEE Transactions on HFE-8.1* (Mar. 1967), pp. 33–48. ISSN: 0096-249X.
- [Ham+13] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, A. E. Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. “Who Does What During a Code Review? Datasets of OSS Peer Review Repositories”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13*. San Francisco, CA, USA: IEEE Press, 2013, pp. 49–52. ISBN: 978-1-4673-2936-1.
- [Ham95] Sally Hambridge. *Netiquette Guidelines*. RFC 1855. <https://tools.ietf.org/html/rfc1855> [accessed 2016-04-26]. Oct. 1995.
- [Han+16] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. “Automatically Recommending Code Reviewers Based on Their Expertise: An Empirical Comparison”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ASE 2016*. Singapore, Singapore: ACM, 2016, pp. 99–110. ISBN: 978-1-4503-3845-5.
- [Han15a] Christoph Hannebauer. *GitHistoryAnalyzer*. <https://github.com/paluno/GitHistoryAnalyzer> [accessed 2015-11-18]. Nov. 2015.
- [Han15b] Christoph Hannebauer. *PatternStatistics*. <https://github.com/paluno/PatternStatistics> [accessed 2015-11-24]. Nov. 2015.
- [Han16] Christoph Hannebauer. *ReviewerRecommender fork of Expertise Explorer*. <https://github.com/paluno/ExpertiseExplorer/tree/review-recommendation> [accessed 2016-03-03]. Mar. 2016.
- [Har09] Matthew Harward. “CoderChrome: Augmenting source code with software metrics”. Honours Report. University of Canterbury, Christchurch, New Zealand, Nov. 2009.
- [Har15] Michael Hartl. *Ruby on Rails Tutorial: Learn Web Development with Rails*. Ed. by Mark L. Taub. Third edition. Addison-Wesley Professional Ruby. Addison-Wesley Professional, 2015.
- [HBG14] Christoph Hannebauer, Matthias Book, and Volker Gruhn. “An Exploratory Study of Contribution Barriers Experienced by Newcomers to Open Source Software Projects”. In: *Proceedings of the 1st International Workshop on CrowdSourcing in Software Engineering. CSI-SE 2014*. Hyderabad, India: ACM, 2014, pp. 11–14. ISBN: 978-1-4503-2857-9.

- [Her+06] Israel Herraiz, Gregorio Robles, Juan José Amor, Teófilo Romera, and Jesús M. González Barahona. “The Processes of Joining in Global Distributed Software Projects”. In: *Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner*. GSD ’06. Shanghai, China: ACM, 2006, pp. 27–33. ISBN: 1-59593-404-9.
- [HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. ISBN: 0321601912.
- [HG16a] Christoph Hannebauer and Volker Gruhn. “Implementation of a Wiki Development Environment”. In: *Proceedings of the 15th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT_16)*. Frontiers in Artificial Intelligence and Applications. IOS International Publisher, 2016.
- [HG16b] Christoph Hannebauer and Volker Gruhn. “Motivation of Newcomers to FLOSS Projects”. In: *Proceedings of the 12th International Symposium on Open Collaboration (OpenSym 2016)*. ACM, 2016.
- [HLG14] Christoph Hannebauer, Claudius Link, and Volker Gruhn. “Patterns for the Distribution of Power in FLOSS Projects”. In: *Proceedings of the 19th European Conference on Pattern Languages of Programs*. EuroPLoP ’14. Irsee, Germany: ACM, 2014, 35:1–35:7. ISBN: 978-1-4503-3416-7.
- [HM76] James W. Hunt and Malcolm Douglas McIlroy. *An Algorithm for Differential File Comparison*. Computing Science Technical Report 41. Bell Laboratories, July 1976.
- [HNH03] Guido Hertel, Sven Niedner, and Stefanie Herrmann. “Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel”. In: *Research Policy* 32.7 (2003). Open Source Software Development, pp. 1159–1177. ISSN: 0048-7333.
- [HO01] Alexander Hars and Shaosong 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, 9 pp.
- [HPG15] Christoph Hannebauer, Michael Patalas, and Volker Gruhn. *Lab package for the evaluation in the study on reviewer recommendations*. <https://www.uni-due.de/~hw0433/>. 2015.
- [HW09] Michael Hielscher and Christian Wagenknecht. “Programming-Wiki: Online programmieren und kommentieren”. In: *Zukunft braucht Herkunft*. Ed. by Bernhard Koerber. Vol. P-156. Lecture Notes in Informatics (LNI). Gesellschaft für Informatik (GI), Sept. 2009, pp. 281–292.
- [HWG10] Christoph Hannebauer, Vincent Wolff-Marting, and Volker Gruhn. “Towards a Pattern Language for FLOSS Development”. In: *Proceedings of the 2010 Conference on Pattern Languages of Programs*. Hillside Group. Reno/Tahoe Nevada, USA: ACM, 2010.

- [HWG11] Christoph Hannebauer, Vincent Wolff-Marting, and Volker Gruhn. “Contributor-Interaction Patterns in FLOSS Development”. In: *EuroPLoP 2011*. July 2011.
- [Jeo+09] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. *Improving Code Review by Predicting Reviewers and Acceptance of Patches*. ROSAEC MEMO ROSAEC-2009-006. Research On Software Analysis for Error-free Computing Center, Seoul National University, Sept. 2009.
- [JHC15] Jing Jiang, Jia-Huan He, and Xue-Yuan Chen. “CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation”. In: *Journal of Computer Science and Technology* 30.5 (2015), pp. 998–1016. ISSN: 1860-4749.
- [JKK11] Carlos Jensen, Scott King, and Victor Kuechler. “Joining Free/Open Source Software Communities: An Analysis of Newbies’ First Interactions on Project Mailing Lists”. In: *System Sciences (HICSS), 2011 44th Hawaii International Conference on*. Jan. 2011, pp. 1–10.
- [JS07] Chris Jensen and Walt Scacchi. “Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study”. In: *29th International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 364–374.
- [JSW11] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. “The Onion Patch: Migration in Open Source Ecosystems”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE ’11. Szeged, Hungary: ACM, 2011, pp. 70–80. ISBN: 978-1-4503-0443-6.
- [Jun+11] Jae-Yoon Jung, Kwanho Kim, Dongmin Shin, and Jonghun Park. “FlowWiki: A wiki based platform for ad hoc collaborative workflows”. In: *Knowledge-Based Systems* 24.1 (2011), pp. 154–165. ISSN: 0950-7051.
- [Kel06] Allan Kelly. “Patterns for Technology Companies”. In: *Proceedings of the 11th European Conference on Pattern Languages of Programs*. 2006.
- [Kel07] Allan Kelly. “More patterns for Software Companies Product development”. In: *Proceedings of the 12th European Conference on Pattern Languages of Programs*. 2007.
- [KFL94] Calvin Ko, Gero Fink, and Karl Levitt. “Automated detection of vulnerabilities in privileged programs by execution monitoring”. In: *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*. Dec. 1994, pp. 134–144.
- [KH11] Sebastian Kleinschmager and Stefan Hanenberg. “How to Rate Programming Skills in Programming Experiments?: A Preliminary, Exploratory, Study Based on University Marks, Pretests, and Self-estimation”. In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU ’11. Portland, Oregon, USA: ACM, 2011, pp. 15–24. ISBN: 978-1-4503-1024-6.

Secondary Sources

- [Kit+07a] Aniket Kittur, Ed Chi, Bryan A. Pendleton, Bongwon Suh, and Todd Mytkowicz. “Power of the Few vs. Wisdom of the Crowd: Wikipedia and the Rise of the Bourgeoisie”. In: *ALT.CHI at CHI 2007*. 2007.
- [Kit+07b] Aniket Kittur, Bongwon Suh, Bryan A. Pendleton, and Ed H. Chi. “He says, she says: conflict and coordination in Wikipedia”. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. CHI '07. San Jose, California, USA: ACM, 2007, pp. 453–462. ISBN: 978-1-59593-593-9.
- [KK08] Aniket Kittur and Robert E. Kraut. “Harnessing the Wisdom of Crowds in Wikipedia: Quality Through Coordination”. In: *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*. CSCW '08. San Diego, CA, USA: ACM, 2008, pp. 37–46. ISBN: 978-1-60558-007-4.
- [KK09] Mark H. Kryder and Chang Soo Kim. “After Hard Drives—What Comes Next?” In: *Magnetics, IEEE Transactions on* 45.10 (Oct. 2009), pp. 3406–3413. ISSN: 0018-9464.
- [KK10] Aniket Kittur and Robert E. Kraut. “Beyond Wikipedia: coordination and conflict in online production groups”. In: *Proceedings of the 2010 ACM conference on Computer supported cooperative work*. CSCW '10. Savannah, Georgia, USA: ACM, 2010, pp. 215–224. ISBN: 978-1-60558-795-0.
- [KM06] Mik Kersten and Gail C. Murphy. “Using Task Context to Improve Programmer Productivity”. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '06/FSE-14. Portland, Oregon, USA: ACM, 2006, pp. 1–11. ISBN: 1-59593-468-5.
- [Knu84] Donald E. Knuth. “Literate Programming”. In: *The Computer Journal* 27.2 (1984), pp. 97–111.
- [Kop06] Timo Koponen. “Life cycle of Defects in Open Source Software Projects”. In: *OSS2006: Open Source Systems (IFIP 2.13)*. Springer, 2006, pp. 195–200.
- [Kri02] Sandeep Krishnamurthy. “Cave or Community?: An Empirical Examination of 100 Mature Open Source Projects”. In: *Social Science Research Network Working Paper Series* 7.6 (June 2002).
- [Kri06] Sandeep Krishnamurthy. “On the intrinsic and extrinsic motivation of free/libre/open source (FLOSS) developers”. In: *Knowledge, Technology & Policy* 18 (4 2006). 10.1007/s12130-006-1002-x, pp. 17–39. ISSN: 0897-1986.
- [Kro+12] Georg von Krogh, Stefan Haefliger, Sebastian Spaeth, and Martin W. Wallin. “Carrots and Rainbows: Motivation and Social Practice in Open Source Software Development”. In: *MIS Quarterly* 36.2 (2012).
- [KSL03] Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. “Community, Joining, and Specialization in Open Source Software Innovation: A Case Study”. In: *Research Policy* 32.7 (July 2003), 1217–1241(25).

- [Lan+94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. “A taxonomy of computer program security flaws”. In: *ACM Comput. Surv.* 26.3 (Sept. 1994), pp. 211–254. issn: 0360-0300.
- [LaT+14] Thomas D. LaToza, W. Ben Towne, Christian M. Adriano, and André van der Hoek. “Microtask Programming: Building Software with a Crowd”. In: *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*. UIST ’14. Honolulu, Hawaii, USA: ACM, 2014, pp. 43–54. isbn: 978-1-4503-3069-5.
- [LC01] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. isbn: 0-201-71499-X.
- [LH03] Karim Lakhani and Eric von Hippel. “How open source software works: “free” user-to-user assistance”. In: *Research Policy* 32.6 (2003), pp. 923–943. issn: 0048-7333.
- [Lin10] Claudius Link. “Patterns for the commercial use of Open Source: Legal and licensing aspects”. In: *Proceedings of the 15th European Conference on Pattern Languages of Programs*. EuroPLoP ’10. Irsee, Germany: ACM, 2010, 7:1–7:10. isbn: 978-1-4503-0259-3.
- [Lin11] Claudius Link. “Patterns for the commercial use of Open Source: License patterns”. In: *EuroPLoP 2011*. Version 331. 2011.
- [Lin12] Claudius Link. “Patterns for the commercial use of Open Source: Economic aspects and case studies”. In: *EuroPLoP 2012*. Version 398. 2012.
- [Lor53] Frederic M. Lord. “On The Statistical Treatment of Football Numbers”. In: *The American Psychologist* 8.12 (Dec. 1953), pp. 750–751.
- [Lot+09] Roberto A. Lotufo, Rubens C. Machado, André Körbes, and Rafael G. Ramos. “Adessowiki on-line collaborative scientific programming platform”. In: *Proceedings of the 5th International Symposium on Wikis and Open Collaboration*. WikiSym ’09. Orlando, Florida: ACM, 2009, 10:1–10:6. isbn: 978-1-60558-730-1.
- [LS04] Hanno Langweg and Einar Snekkenes. “A classification of malicious software attacks”. In: *Performance, Computing, and Communications, 2004 IEEE International Conference on*. 2004, pp. 827–832.
- [LT02] Josh Lerner and Jean Tirole. “Some Simple Economics of Open Source”. In: *The Journal of Industrial Economics* 50.2 (2002), pp. 197–234. issn: 1467-6451.
- [LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. “Maintaining Mental Models: A Study of Developer Work Habits”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. Shanghai, China: ACM, 2006, pp. 492–501. isbn: 1-59593-375-1.

- [LW03] Karim Lakhani and Robert G Wolf. “Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects”. In: *Social Science Research Network* (2003). Ed. by J Feller, B Fitzgerald, S Hissam, and K REditors Lakhani, pp. 1–27.
- [MA00] David W. McDonald and Mark S. Ackerman. “Expertise Recommender: A Flexible Recommendation System and Architecture”. In: *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work. CSCW '00*. Philadelphia, Pennsylvania, USA: ACM, 2000, pp. 231–240. ISBN: 1-58113-222-0.
- [Mai+15] Patrick Mair, Eva Hofmann, Kathrin Gruber, Reinhold Hatzinger, Achim Zeileis, and Kurt Hornik. “Motivation, values, and work design as drivers of participation in the R open source project for statistical computing”. In: *Proceedings of the National Academy of Sciences* (2015).
- [Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- [Mar99] Klaus Marquardt. “Patterns for Plug-Ins”. In: *Proceedings of the Fourth European Conference on Pattern Languages of Programs*. 1999.
- [Mas+09] H  la Masmoudi, Matthijs den Besten, Claude de Loupy, and Jean-Michel Dalle. ““Peeling the Onion” – The Words and Actions That Distinguish Core from Periphery in Bug Reports and How Core and Periphery Interact Together”. In: *OSS2009: Open Source Ecosystems: Diverse Communities Interacting (IFIP 2.13)*. Vol. 299/2009. Springer, 2009, pp. 284–297.
- [McC76] Thomas J. McCabe. “A Complexity Measure”. In: *Software Engineering, IEEE Transactions on SE-2.4* (Dec. 1976), pp. 308–320. ISSN: 0098-5589.
- [McN47] Quinn McNemar. “Note on the sampling error of the difference between correlated proportions or percentages”. English. In: *Psychometrika* 12.2 (1947), pp. 153–157. ISSN: 0033-3123.
- [MD98] Gerard Meszaros and Jim Doble. “A Pattern Language for Pattern Writing”. In: *Pattern Languages of Program Design 3*. Ed. by Robert Martin, Dirk Riehle, and Frank Buschmann. Software Patterns Series. Addison-Wesley, 1998. Chap. 29, pp. 529–574.
- [MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. “Two case studies of open source software development: Apache and Mozilla”. In: *ACM Trans. Softw. Eng. Methodol.* 11.3 (2002), pp. 309–346. ISSN: 1049-331X.
- [MH02] Audris Mockus and James D. Herbsleb. “Expertise browser: a quantitative approach to identifying expertise”. In: *Proceedings of the 24th International Conference on Software Engineering. ICSE '02*. Orlando, Florida: ACM, 2002, pp. 503–512. ISBN: 1-58113-472-X.
- [Mid+10] Vishal Midha, Prashant Palvia, Rahul Singh, and Nir Kshetri. “Improving Open Source Software Maintenance”. In: *Journal of Computer Information Systems* 50.3 (2010), pp. 81–90.

- [MM07] Shawn Minto and Gail C. Murphy. “Recommending Emergent Teams”. In: *Mining Software Repositories, 2007. ICSE Workshops MSR ’07. Fourth International Workshop on*. 2007, pp. 5–5.
- [Moo75] Gordon E. Moore. “Progress in digital integrated electronics”. In: *Electron Devices Meeting, 1975 International*. Vol. 21. IEEE, 1975, pp. 11–13.
- [MP09] Cerstin Mahlow and Michael Piotrowski. “Opportunities and Limits for Language Awareness in Text Editors”. In: *Proceedings of the Workshop on NLP for Reading and Writing – Resources, Algorithms and Tools (SLTC 2008)*. Ed. by Rickard Domeij, Sofie Johansson Kokkinakis, Ola Knutsson, and Sylvana Sofkova Hashemi. Vol. 3. NEALT Proceedings Series. Northern European Association for Language Technology (NEALT), 2009, pp. 14–18.
- [MPT78] Malcolm Douglas McIlroy, Elliot N. Pinson, and Berk A. Tague. “UNIX Time-Sharing System: Foreword”. In: *The Bell System Technical Journal* 57.6 (1978), pp. 1899–1904.
- [MRB06] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. “Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code”. In: *Management Science* 52.7 (2006), pp. 1015–1030.
- [MSH92] Brad A. Myers, David Canfield Smith, and Bruce Horn. “Languages for Developing User Interfaces”. In: ed. by Brad A. Myers. A K Peters, 1992. Chap. Report of the ‘End-User Programming’ Working Group, pp. 343–366.
- [MWA12] Andréa Magalhães Magdaleno, Cláudia Maria Lima Werner, and Renata Mendes de Araujo. “Reconciling software development models: A quasi-systematic review”. In: *Journal of Systems and Software* 85.2 (2012), pp. 351–369. issn: 0164-1212.
- [Nar10] Elizabeth Naramore. *Why People Don’t Contribute to OS Projects, and What We Can Do About It*. <http://naramore.net/blog/why-people-don-t-contribute-to-os-projects-and-what-we-can-do-about-it>, last checked: 2010-05-19. Mar. 2010.
- [Nor10] Geoff Norman. “Likert scales, levels of measurement and the “laws” of statistics”. English. In: *Advances in Health Sciences Education* 15.5 (2010), pp. 625–632. issn: 1382-4996.
- [Nur+09] Mehrdad Nurolahzade, Seyed Mehdi Nasehi, Shahedul Huq Khandkar, and Shreya Rawal. “The role of patch review in software evolution: an analysis of the mozilla firefox”. In: *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. IWPSE-Evol ’09. Amsterdam, The Netherlands: ACM, 2009, pp. 9–18. isbn: 978-1-60558-678-6.
- [OC90] Paul W. Oman and Curtis R. Cook. “A Taxonomy for Programming Style”. In: *Proceedings of the 1990 ACM Annual Conference on Cooperation*. CSC ’90. Washington, D.C., USA: ACM, 1990, pp. 244–250. isbn: 0-89791-348-5.

Secondary Sources

- [ØJ07] Thomas Østerlie and Letizia Jaccheri. “A Critical Review of Software Engineering Research on Open Source Software Development”. In: *Proceedings of the 2nd AIS SIGSAND European Symposium on Systems Analysis and Design* (2007).
- [OSu09] Bryan O’Sullivan. *Mercurial: The Definitive Guide*. O’Reilly Media, Inc., 2009. ISBN: 978-0-596-80067-3.
- [Par72] David Lorge Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Commun. ACM* 15 (12 Dec. 1972), pp. 1053–1058. ISSN: 0001-0782.
- [Pre99] Lutz Prechelt. *The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really?* Interner Bericht. Fakultät für Informatik, Universität Karlsruhe, 1999.
- [Pri08] Michael Price. “The Paradox of Security in Virtual Environments”. In: *Computer* 41.11 (2008), pp. 22–28. ISSN: 0018-9162.
- [PZB14] Shaun Phillips, Thomas Zimmermann, and Christian Bird. “Understanding and Improving Software Build Teams”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 735–744. ISBN: 978-1-4503-2756-5.
- [QSu16] Q-Success. *Usage statistics and market share of Apache for websites*. <http://w3techs.com/technologies/details/ws-apache/all/all> [accessed 2016-02-25]. Feb. 2016.
- [Rat13] Andreas Ratzka. “User Interface Patterns for Multimodal Interaction”. English. In: *Transactions on Pattern Languages of Programming III*. Ed. by James Noble, Ralph Johnson, Uwe Zdun, and Eugene Wallingford. Vol. 7840. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 111–167. ISBN: 978-3-642-38675-6.
- [Ray00] Eric S. Raymond. “The Cathedral and the Bazaar”. In: (2000). <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/> [accessed 2016-03-23].
- [RC04] Charles Reis and Robert Cartwright. “Taming a professional IDE for the classroom”. In: *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. SIGCSE ’04. Norfolk, Virginia, USA: ACM, 2004, pp. 156–160. ISBN: 1-58113-798-2.
- [Reu07] Jenni Susan Reuben. “A Survey on Virtual Machine Security”. In: *Fall 2007 Seminar of Network Security*. http://www.tml.tkk.fi/Publications/C/25/papers/Reuben_final.pdf [accessed 2016-04-26]. Helsinki University of Technology. 2007.
- [RG05] Mendel Rosenblum and Tal Garfinkel. “Virtual machine monitors: current technology and future trends”. In: *Computer* 38.5 (2005), pp. 39–47. ISSN: 0018-9162.

- [RG06] Gregorio Robles and Jesus Gonzalez-Barahona. “Contributor Turnover in Libre Software Projects”. In: *OSS2006: Open Source Systems (IFIP 2.13)*. Springer, 2006, pp. 273–286.
- [RGH09] Gregorio Robles, Jesus Gonzalez-Barahona, and Israel Herraiz. “Evolution of the core team of developers in libre software projects”. In: *Mining Software Repositories, 2009. MSR ’09. 6th IEEE International Working Conference on*. May 2009, pp. 167–170.
- [RGS08] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. “Open source software peer review practices: a case study of the apache server”. In: *Proceedings of the 30th international conference on Software engineering*. ICSE ’08. Leipzig, Germany: ACM, 2008, pp. 541–550. ISBN: 978-1-60558-079-1.
- [Rie+09] Dirk Riehle, John Ellenberger, Tamir Menahem, Boris Mikhailovski, Yuri Natchetoi, Barak Naveh, and Thomas Odenwald. “Open Collaboration within Corporations Using Software Forges”. In: *IEEE Software* 26 (2009), pp. 52–58.
- [Rie11] Eric Ries. *The Lean Startup*. Crown Business, Sept. 2011.
- [RJ98] Don Roberts and Ralph Johnson. “Patterns for Evolving Frameworks”. In: *Pattern Languages of Program Design 3*. Ed. by Robert Martin, Dirk Riehle, and Frank Buschmann. Software Patterns Series. Addison-Wesley, 1998. Chap. 26, pp. 471–486.
- [RM02] Christian Robotoom Reis and Renata Pontin de Mattos Fortes. “An Overview of the Software Engineering Process and Tools in the Mozilla Project”. In: *Proceedings of the Open Source Software Development Workshop*. 2002, pp. 155–175.
- [RTH13] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails 4*. Ed. by Susannah Davidson Pfalzer. 4th Revised edition. The Facets of Ruby Series. O’Reilly UK Ltd., 2013.
- [Rul06] Francesco Rullani. *Dragging developers towards the core. How the Free/Libre/Open Source Software community enhances developers’ contribution*. LEM Papers Series 2006/22. Laboratory of Economics and Management (LEM), Sant’Anna School of Advanced Studies, Pisa, Italy, Sept. 2006.
- [SAA10] Klaas-Jan Stol, Paris Avgeriou, and Muhammad Ali Babar. “Identifying Architectural Patterns Used in Open Source Software: Approaches and Challenges”. In: *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering*. EASE’10. UK: British Computer Society, 2010, pp. 91–100.
- [SAM06] Katherine J. Stewart, Anthony P. Ammeter, and Likoebe M. Maruping. “Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects”. In: *Information Systems Research* 17.2 (2006), pp. 126–144.

Secondary Sources

- [Sau11] Christoph Sauer. “Wiki as business application platform: the MES showcase”. In: *Proceedings of the 7th International Symposium on Wikis and Open Collaboration*. WikiSym ’11. Mountain View, California: ACM, 2011, pp. 221–222. ISBN: 978-1-4503-0909-7.
- [SC94] Anselm Strauss and Juliet Corbin. “Grounded theory methodology: An overview.” In: *Handbook of qualitative research*. Ed. by N. K. Denzin Y. S. Lincoln. Thousand Oaks, CA, US: Sage Publications, Inc, 1994, pp. 273–285.
- [Sch+08] Charles M. Schweik, Robert C. English, Meelis Kitsing, and Sandra Haire. “Brooks’ versus Linus’ law: an empirical test of open source projects”. In: *Proceedings of the 2008 international conference on Digital government research*. dg.o ’08. Montreal, Canada: Digital Government Society of North America, 2008, pp. 423–424. ISBN: 978-1-60558-099-9.
- [SD06] Gregory Simmons and Tharam Dillon. “Towards an Ontology for Open Source Software Development”. In: *OSS2006: Open Source Systems (IFIP 2.13)*. Springer, 2006, pp. 65–75.
- [Sec12a] SecurityFocus. *Jenkins Multiple Cross Site Scripting and Directory Traversal Vulnerabilities*. <http://www.securityfocus.com/bid/52384/> [accessed 2016-04-26]. Feb. 2012.
- [Sec12b] SecurityFocus. *Jenkins Multiple HTML Injection Vulnerabilities*. <http://www.securityfocus.com/bid/52055/> [accessed 2016-04-26]. Mar. 2012.
- [SEG68] Harold Sackman, Warren J. Erikson, and E. Eugene Grant. “Exploratory Experimental Studies Comparing Online and Offline Programming Performance”. In: *Commun. ACM* 11.1 (Jan. 1968), pp. 3–11. issn: 0001-0782.
- [Seo+14] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. “Programmers’ Build Errors: A Case Study (at Google)”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 724–734. ISBN: 978-1-4503-2756-5.
- [SF09] Peter Seibel and Brad Fitzpatrick. “Brad Fitzpatrick”. In: *Coders at Work*. Ed. by Peter Seibel. Berkely, CA, USA: Apress, 2009. Chap. 2, pp. 49–90. ISBN: 978-1-4302-1948-4.
- [SGR14] Igor Steinmacher, Marco Aurélio Gerosa, and David F. Redmiles. “Attracting, Onboarding, and Retaining Newcomer Developers in Open Source Software Projects”. In: *Workshop: Global Software Development in a CSCW Perspective*. 2014.
- [SH98] Susan Elliott Sim and Richard C. Holt. “The ramp-up problem in software projects: a case study of how software immigrants naturalize”. In: *Proceedings of the 1998 International Conference on Software Engineering*. Apr. 1998, pp. 361–370.
- [Sha06] Sonali K. Shah. “Motivation, Governance, and the Viability of Hybrid Forms in Open Source Software Development”. In: *Management Science* 52.7 (2006), pp. 1000–1014.

- [Shi04] Clay Shirky. *Situated Software*. http://www.shirky.com/writings/situated_software.html [accessed 2016-04-26]. Mar. 2004.
- [Sho+13] Ramin Shokripour, John Anvik, Zarinah M. Kasirun, and Sima Zamani. “Why So Complicated? Simple Term Filtering and Weighting for Location-based Bug Report Assignment Recommendation”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 2–11. ISBN: 978-1-4673-2936-1.
- [SHP15] Sebastian Stünkel, Christoph Hannebauer, and Michael Patalas. *Expertise Explorer on GitHub*. <https://github.com/paluno/ExpertiseExplorer>. 2015.
- [SJW08] Lok Fang Fang Stella, Stan Jarzabek, and Bimlesh Wadhwa. “A comparative study of maintainability of web applications on J2EE, .NET and Ruby on Rails”. In: *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*. Oct. 2008, pp. 93–99.
- [Sma11] John Ferguson Smart. *Jenkins: The Definitive Guide*. O’Reilly Media, 2011. ISBN: 1449305350.
- [SMJ10a] Bhuricha Deen Sethanandha, Bart Massey, and William Jones. “Managing open source contributions for software project sustainability”. In: *Technology Management for Global Economic Growth (PICMET), 2010 Proceedings of PICMET ’10*: 2010, pp. 1–9.
- [SMJ10b] Bhuricha Deen Sethanandha, Bart Massey, and William Jones. “On the Need for OSS Patch Contribution Tools”. In: *Proceedings of the Second International Workshop on Building Sustainable Open Source Communities*. 2010.
- [SS13] Andreas Stefik and Susanna Siebert. “An Empirical Investigation into Programming Language Syntax”. In: *Trans. Comput. Educ.* 13.4 (Nov. 2013), 19:1–19:40. ISSN: 1946-6226.
- [SSG14] Igor Steinmacher, Marco A. Graciotto Silva, and Marco Aurélio Gerosa. “Barriers faced by newcomers to open source projects: a systematic review”. In: *OSS 2014?* 2014.
- [SSN09] Chandrasekar Subramaniam, Ravi Sen, and Matthew L. Nelson. “Determinants of open source software project success: A longitudinal study”. In: *Decision Support Systems* 46.2 (2009), pp. 576–585. ISSN: 0167-9236.
- [Ste+13] Igor Steinmacher, Igor Scaliante Wiese, A.P. Chaves, and Marco Aurélio Gerosa. “Why do newcomers abandon open source software projects?” In: *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on*. May 2013, pp. 25–32.
- [Ste+14a] Igor Steinmacher, Ana Paula Chaves, Tayana Uchoa Conte, and Marco Aurélio Gerosa. “Preliminary Empirical Identification of Barriers Faced by Newcomers to Open Source Software Projects”. In: *2014 Brazilian Symposium on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, 2014, pp. 51–60.

Secondary Sources

- [Ste+14b] Igor Steinmacher, Igor Scaliante Wiese, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. “The Hard Life of Open Source Software Project Newcomers”. In: *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE 2014. Hyderabad, India: ACM, 2014, pp. 72–78. ISBN: 978-1-4503-2860-9.
- [Ste+15a] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. “Social Barriers Faced by Newcomers Placing Their First Contribution in Open Source Software Projects”. In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW ’15. Vancouver, BC, Canada: ACM, 2015, pp. 1379–1392. ISBN: 978-1-4503-2922-4.
- [Ste+15b] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurélio Gerosa, and David F. Redmiles. “A systematic literature review on the barriers faced by newcomers to open source software projects”. In: *Information and Software Technology* 59 (2015), pp. 67–85. ISSN: 0950-5849.
- [SZ08] David Schuler and Thomas Zimmermann. “Mining usage expertise from version archives”. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR ’08. Leipzig, Germany: ACM, 2008, pp. 121–124. ISBN: 978-1-60558-024-1.
- [Tao+12] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun 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. Nov. 2012.
- [TB10] Roger Tourangeau and Norman M. Bradburn. “Handbook of Survey Research”. In: ed. by Peter V. Marsden and James D. Wright. Second. Emerald Group Publishing Limited, 2010. Chap. The Psychology of Survey Response, pp. 315–346.
- [TDH14] Jason Tsay, Laura Dabbish, and James Herbsleb. “Influence of Social and Technical Factors for Evaluating Contribution in GitHub”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 356–366. ISBN: 978-1-4503-2756-5.
- [Ter+16] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, and Chris Parnin. “Gender bias in open source: Pull request acceptance of women versus men”. Feb. 2016.
- [Tho+14] Patanamon Thongtanunam, Raula Gaikovina Kula, Ana Erika Camargo Cruz, Norihiro Yoshida, and Hajimu Iida. “Improving Code Review Effectiveness Through Reviewer Recommendations”. In: *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE 2014. Hyderabad, India: ACM, 2014, pp. 119–122. ISBN: 978-1-4503-2860-9.

- [Tho+15] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. “Who should review my code? – A file location-based code-reviewer recommendation approach for Modern Code Review”. In: *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. Mar. 2015, pp. 141–150.
- [Tho84] Ken Thompson. “Reflections on trusting trust”. In: *Commun. ACM* 27.8 (Aug. 1984), pp. 761–763. ISSN: 0001-0782.
- [TM12] Linus Torvalds and Glyn Moody. *Interview: Linus Torvalds – I don’t read code any more*. [accessed 2015-04-24]. The H Open. Nov. 2012.
- [Tur14] James Turnbull. *Software Archaeology for Beginners – Code, Culture and Community*. FOSDEM 2014 Keynote. https://archive.fosdem.org/2014/schedule/event/software_archaeology_for_beginners/ [accessed 2015-07-31]. Feb. 2014.
- [Und04] Keith Underwood. “FPGAs vs. CPUs: Trends in Peak Floating-point Performance”. In: *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. FPGA ’04. Monterey, California, USA: ACM, 2004, pp. 171–180. ISBN: 1-58113-829-6.
- [Wal42] W. Allen Wallis. “Compounding probabilities from independent significance tests”. In: *Econometrica, Journal of the Econometric Society* 10.3/4 (July 1942), pp. 229–248.
- [Wal80] Abraham Wald. *A Reprint of ‘A Method of Estimating Plane Vulnerability Based on Damage of Survivors’*. Collection of memoranda. CENTER FOR NAVAL ANALYSES ALEXANDRIA VA OPERATIONS EVALUATION GROUP, July 1980.
- [WD10] Joost C. F. de Winter and Dimitria Dodou. “Five-Point Likert Items: t test versus Mann-Whitney-Wilcoxon”. In: *Practical Assessment, Research & Evaluation* 15.11 (Oct. 2010), pp. 1–16. ISSN: 1531-7714.
- [Wei09] Michael Weiss. “Performance of Open Source Projects”. In: *14th Annual European Conference on Pattern Languages of Programming*. 2009.
- [Wei10] Michael Weiss. “Profiting from open source”. In: *Proceedings of the 15th European Conference on Pattern Languages of Programs*. EuroPLoP ’10. Irsee, Germany: ACM, 2010, 5:1–5:8. ISBN: 978-1-4503-0259-3.
- [Wei11] Michael Weiss. “Profiting even more from open source”. In: *Proceedings of the 16th European Conference on Pattern Languages of Programs*. EuroPLoP ’11. Irsee, Germany: ACM, 2011, 1:1–1:7. ISBN: 978-1-4503-1302-5.
- [Wei15] Michael Weiss. “The Business of Open Source: Missing Patterns”. In: *Proceedings of the 20th European Conference on Pattern Languages of Programs*. 2015.

Secondary Sources

- [Wei98] Charles Weir. “Patterns for Designing in Teams”. In: *Pattern Languages of Program Design 3*. Ed. by Robert Martin, Dirk Riehle, and Frank Buschmann. Software Patterns Series. Addison-Wesley, 1998. Chap. 27, pp. 487–501.
- [WGY07] Chong-Guang Wu, James H. Gerlach, and Clifford E. Young. “An empirical analysis of open source software developers’ motivations and continuance intentions”. In: *Information & Management* 44.3 (2007), pp. 253–262. ISSN: 0378-7206.
- [WHG13] Vincent Wolff-Marting, Christoph Hannebauer, and Volker Gruhn. “Patterns for Tearing Down Contribution Barriers to FLOSS Projects”. In: *Intelligent Software Methodologies, Tools and Techniques (SoMeT), 2013 IEEE 12th International Conference on*. Budapest, Sept. 2013, pp. 9–14.
- [Wie02] Karl E. Wiegers. *Peer Reviews in Software – A Practical Guide*. Second printing. Addison-Wesley Information Technology Series. Addison Wesley, 2002. ISBN: 0-201-73485-0.
- [WMZ06] Michael Weiss, Gabriella Moroiu, and Ping Zhao. “Evolution of Open Source Communities”. In: *OSS2006: Open Source Systems (IFIP 2.13)*. Springer, 2006, pp. 21–32.
- [WN13] Michael Weiss and Nadia Noori. “Enabling Contributions in Open Source Projects”. In: *EuroPLoP 2013*. 2013.
- [WND08] Peter Weißgerber, Daniel Neu, and Stephan Diehl. “Small Patches Get in!” In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR ’08. Leipzig, Germany: ACM, 2008, pp. 67–76. ISBN: 978-1-60558-024-1.
- [WSK08] Dirk Wilking, David Schilli, and Stefan Kowalewski. “Measuring the Human Factor with the Rasch Model”. English. In: *Balancing Agility and Formalism in Software Engineering*. Ed. by Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz Walter. Vol. 5082. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 157–168. ISBN: 978-3-540-85278-0.
- [XCY07] WenPeng Xiao, ChangYan Chi, and Min Yang. “On-line collaborative software development via wiki”. In: *WikiSym ’07: Proceedings of the 2007 international symposium on Wikis*. Montreal, Quebec, Canada: ACM, 2007, pp. 177–183. ISBN: 978-1-59593-861-9.
- [XJ10] Bo Xu and Donald R. Jones. “Volunteers’ Participation in Open Source Software Development: A Study from the Social-relational Perspective”. In: *SIGMIS Database* 41.3 (Aug. 2010), pp. 69–84. ISSN: 0095-0033.
- [XJS09] Bo Xu, Donald R. Jones, and Bingjia Shao. “Volunteers’ involvement in online community based software development”. In: *Information & Management* 46.3 (2009), pp. 151–158. ISSN: 0378-7206.
- [Xu+05] Jin Xu, Yongqin Gao, S. Christley, and G. Madey. “A Topological Analysis of the Open Source Software Development Community”. In: *System Sciences, 2005. HICSS ’05. Proceedings of the 38th Annual Hawaii International Conference on*. Jan. 2005, 198a–198a.

- [Yat14] Rebecca Yolande Yates. “Onboarding in Software Engineering”. <https://ulir.ul.ie/handle/10344/4272> [accessed 2016-02-08]. Ph.D. thesis. University of Limerick, Nov. 2014.
- [Yin+04] Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. “Predicting Source Code Changes by Mining Change History”. In: *Software Engineering, IEEE Transactions on* 30.9 (Sept. 2004), pp. 574–586. ISSN: 0098-5589.
- [YK03] Yunwen Ye and Kouichi Kishida. “Toward an understanding of the motivation of open source software developers”. In: *Proceedings of the 25th International Conference on Software Engineering*. May 2003, pp. 419–429.
- [Yu+16] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. “Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?” In: *Information and Software Technology* 74 (2016), pp. 204–218. ISSN: 0950-5849.
- [ZHG16] Firas Zaidan, Christoph Hannebauer, and Volker Gruhn. “Quality Attestation: An Open Source Pattern”. In: *21st European Conference on Pattern Languages of Programs (EuroPLoP 2016)*. ACM, July 2016.
- [ZKB15] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. “Automatically Recommending Peer Reviewers in Modern Code Review”. In: *IEEE Transactions on Software Engineering* PP.99 (2015), pp. 1–1. ISSN: 0098-5589.
- [ZM12] Minghui Zhou and Audris Mockus. “What Make Long Term Contributors: Willingness and Opportunity in OSS Community”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 518–528. ISBN: 978-1-4673-1067-3.