Key Factors for Adopting Inner Source

KLAAS-JAN STOL, Lero, University of Limerick
PARIS AVGERIOU, University of Groningen
MUHAMMAD ALI BABAR, University of Adelaide
YAN LUCAS, Neopost Technologies
BRIAN FITZGERALD, Lero, University of Limerick

A number of organizations have adopted Open Source Software (OSS) development practices to support or augment their software development processes, a phenomenon frequently referred to as *Inner Source*. However the adoption of Inner Source is not a straightforward issue. Many organizations are struggling with the question of whether Inner Source is an appropriate approach to software development for them in the first place. This article presents a framework derived from the literature on Inner Source, which identifies nine important factors that need to be considered when implementing Inner Source. The framework can be used as a probing instrument to assess an organization on these nine factors so as to gain an understanding of whether or not Inner Source is suitable. We applied the framework in three case studies at Philips Healthcare, Neopost Technologies, and Rolls-Royce, which are all large organizations that have either adopted Inner Source or were planning to do so. Based on the results presented in this article, we outline directions for future research.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*Software process models*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development, Software maintenance, Software process*

General Terms: Human Factors, Management, Theory

Additional Key Words and Phrases: Case study, inner source, open-source development practices, framework

ACM Reference Format:
Klaas-Jan Stol, Paris Avgeriou, Muhammad Ali Babar, Yan Lucas, and Brian Fitzgerald. 2014. Key factors for adopting inner source. ACM Trans. Softw. Eng. Methodol. 23, 2, Article 18 (March 2014), 35 pages.
DOI: http://dx.doi.org/10.1145/2533685

1. INTRODUCTION

Numerous software development methods have been proposed to guide organizations in their software development: from the traditional waterfall approach and the V-model to the more recent and highly popular agile methods. While these methods all aim at providing a systematic way to develop software, the emergence of successful open source projects has been remarkable, given the seeming absence of a predefined process—what Erdogmus [2009] provokingly called an *antiprocess*. Open-source communities have produced a number of high-quality and highly successful products, including the

This work is partially funded by Science Foundation Ireland grant 10/CE/I1855 to Lero—the Irish Software Engineering Research Centre (www.lero.ie) and by IT University of Copenhagen. Yan Lucas is currently affiliated with TyCloud, the Netherlands.
Authors' addresses: K.-J. Stol (corresponding author) and B. Fitzgerald, Lero—the Irish Software Engineering Research Centre, University of Limerick; P. Avgeriou, University of Groningen; M. A. Babar, University of Adelaide; Y. Lucas, TyCloud; corresponding author's email: klaas-jan.stol@lero.ie.

DOI: http://dx.doi.org/10.1145/2533685

so-called LAMP stack (consisting of Linux, Apache, MySQL, and PHP/Perl/Python), while defying traditional wisdom in software development [McConnell 1999]. Furthermore, while distributed development has proven extremely problematic and challenging for organizations [Herbsleb and Grinter 1999], open-source software development represents a successful exemplar of distributed development. Several authors have argued for drawing lessons from open-source communities and transferring those best practices to commercial software development [O'Reilly 1999; Asundi 2001; Augustin et al. 2002; Mockus and Herbsleb 2002; Erenkrantz and Taylor 2003; Fitzgerald 2011; Rigby et al. 2012]. The open-source phenomenon has attracted significant attention from the research community seeking to understand how open-source communities can achieve such success. The success of open source has also captured the attention of many organizations who strive to replicate similar successes using the same approach. This phenomenon of organizations leveraging open-source development practices for their in-house software development is what we label *Inner Source*, though different organizations have used various terms to denote this [Stol et al. 2011].[1]

This trend of organizations' interest in reproducing the success of open-source communities for in-house software development provides the background of this study. Inner Source research is still in its nascent phase, and as such, little is known about the circumstances through which it can be successful. Wesselius [2008] highlighted that Inner Source is embedded in a company's existing processes, which are typically hierarchical and focus on top-down control, and posed the question (using Raymond's [2001] *Bazaar* and *Cathedral* metaphors): "How can a bazaar flourish inside a cathedral?" Similarly, but more specifically, Gurbani et al. [2006] concluded their study of Inner Source implementation at Alcatel-Lucent with the following question: "It is not clear, in general, how and when to initiate [an Inner Source] project that can serve as a shared resource." Building on this previous work, our research question is thus: What are the important factors for adopting Inner Source?

We adopted a two-phased research approach to address this question. In the first phase, we drew from the extant literature on Inner Source and open source to derive a framework of factors that support Inner Source adoption. The resulting *probing* framework can be used by organizations to assess the extent to which these factors are in place. In the second phase, we report on the application of the framework in three industry case studies, which provides a rich description of how these elements "come to life."

In recent years, organizations are becoming increasingly familiar with the open-source phenomenon, as it has become more mainstream, a development that has been termed OSS 2.0 [Fitzgerald 2006]. At the same time, we have observed an increasing number of organizations showing an interest in Inner Source, making this study timely. One of the novel features of this study is that, to our knowledge, it provides the first high-level overview of the Inner Source phenomenon, in that it draws together relevant literature from both the open-source and Inner Source fields of study. Furthermore, this study addresses a real-world problem by offering a framework that can be readily used by organizations. The empirical element of this study is a significant contribution to the still limited literature on Inner Source. Previous studies (e.g., Sharma et al. [2002]) did not offer such an empirical component. Practitioner reports (e.g., Barnett [2004]) offer a set of general suggestions, which do not consider an organization's specific context. Furthermore, such reports have not followed a scientific approach nor are they supported by any empirical evidence.

The remainder of this article is structured as follows. Section 2 presents a brief discussion of related work in this area. This is followed by a presentation of our research

[1]One of the earliest records using the term 'Inner Source' is an article by Tim O'Reilly [2000].

approach in Section 3. In Section 4, we present a framework that we derived from the extant literature on Open and Inner Source, which defines a number of factors that support Inner Source adoption. Section 5 presents the results of the application of the framework in three case studies. Section 6 presents a discussion of our findings as well as an evaluation of our framework, and presents an outlook on future work.

2. RELATED WORK AND BACKGROUND

2.1. Defining Inner Source

In earlier work, we defined Inner Source to refer to the adoption of open-source development practices within the confines of an organization [Stol et al. 2011]. Whereas well-defined methods, such as the agile Scrum approach [Schwaber and Beedle 2002] have clearly defined tasks (e.g., Scrum meetings), artifacts (e.g., Sprint backlog), and roles (e.g., Scrum Master), this is not so much the case for Inner Source, although common open-source development practices and roles can be identified. Rather than a well-defined methodology, we consider Inner Source to be more of a development philosophy, oriented around the open collaboration principles of egalitarianism, meritocracy, and self-organization [Riehle et al. 2009]. Within Inner Source, a number of common open-source development practices can be observed:

—universal access to development artifacts (i.e., source code) [Lindman et al. 2008; Riehle et al. 2009];
—transparent *fishbowl* development environment [Melian and Mähring 2008, Lindman et al. 2010];
—peer-review of contributions through organization-wide scrutiny of contributions [Gurbani et al. 2006; Melian and Mähring 2008; Riehle et al. 2009];
—informal communication channels (e.g., mailing lists, Internet Relay Chat (IRC) channels) [Stellman et al. 2009; Lindman et al. 2010];
—self-selection of motivated contributors [Gurbani et al. 2006; Riehle et al. 2009];
—frequent releases and early feedback [Gurbani et al. 2006];
—"around the clock" development [Melian and Mähring 2008].

Which of these practices are adopted as part of an Inner Source initiative varies per organization; each implementation of Inner Source is tailored to the particular context of the adopting organization [Gaughan et al. 2009]. Existing methods that a company has had in place for some time may be augmented with open-source practices, such as those just listed. However, a key tenet of Inner Source is universal access to the development artifacts throughout an organization so that anyone within the organization can potentially participate.

As well as common practices, a number of common roles can be identified. Inner Source projects are often grassroots movements, started by individuals, project teams, or departments [Riemens and van Zon 2006; Gurbani et al. 2006; Melian 2007]. As such, the initiator assumes the role of a *benevolent dictator* [Gurbani et al. 2006], as commonly found in open-source projects [Mockus et al. 2002]. As some contributors become experts in parts of the project, they can be promoted to *trusted lieutenants* [Gurbani et al. 2006], and together with the benevolent dictator they form a *core team*. These governance structures are commonly found in open-source projects. In Inner Source, additional roles may emerge; Gurbani et al. [2010], for instance, identified a number of roles in the core team at Alcatel-Lucent, each of which had a specific function in order to tailor the bazaar to a commercial software development context.

As a project matures, it may attract more attention and support from management. Once an Inner Source project has been recognized to have significant business value that is critical to the organization at large, additional funding may be made available

Table I. Reports on Organizations that have Adopted Inner Source

Organization	Terminology	Model
Alcatel-Lucent	Corporate Open Source [Gurbani et al. 2006, 2010]	Project
DTE Energy	*Not specified* [Alter 2006; Smith and Garber-Brown 2007]	Infrastructure
Hewlett-Packard	Progressive Open Source [Dinkelacker et al. 2002; Melian 2007; Melian and Mähring 2008], Inner Source, *Corporate Source* initiative, Controlled Source, *Collaborative Development Program* initiative	Infrastructure
IBM	Community Source [Betanews 2005; Taft 2005, 2006, 2009; Vitharana et al. 2010], IBM's Internal Open Source Bazaar (IIOSB) [Capek et al. 2005], Internal Open Source [Vitharana et al. 2010]	Infrastructure
Microsoft	*Officelabs* [Asay 2007]; *CodeBox* [Ogasawara 2008]	Infrastructure
Nokia	Inner Source [Pulkkinen et al. 2007], *iSource* initiative [Lindman et al. 2008, 2010; Lindman et al. 2013]	Infrastructure
Philips Healthcare	Inner Source, Inner Source Software [Wesselius 2008; van der Linden 2009; Lindman et al. 2010]	Project
SAP	*SAP Forge* initiative [Riehle et al. 2009]	Infrastructure
US DoD	*Forge.mil* [Federal Computer Week 2009; Martin and Lippold 2011]	Infrastructure

for the core team to provide ongoing support and training to the customers of the project.

There have been a number of reports on the adoption of Inner Source (see Table I). Different organizations and authors have used different terminology to refer to the adoption of OSS development principles. Besides the term "Inner Source" used in this paper, other terms for this phenomenon are "Progressive Open Source" and "Controlled Source" [Dinkelacker et al. 2002], "Corporate Source" [Goldman and Gabriel 2005], "Corporate Open Source" [Gurbani et al. 2006], and "Internal Open Source" [Goldman and Gabriel 2005; Vitharana et al. 2010].

Organizations also adopt Inner Source in different ways, since it needs to be tailored to an organization's context [Melian and Mähring 2008; Gaughan et al. 2009]. Lindman et al. [2013] described this process of tailoring the open-source paradigm to an organizational context as "re-negotiating the term 'OSS.'" Nevertheless, Gurbani et al. [2010] observed two main models of Inner Source adoption: *infrastructure-based* and *project-specific* Inner Source. These two models are described further; a detailed discussion of the differences is presented in Stol et al. [2011].

The *infrastructure-based* model is characterized by the availability of suitable infrastructure that allows individuals and teams within an organization to start an Inner Source project (e.g, a software 'forge' [Riehle et al. 2009]). This model maximizes sharing of software packages within an organization, but reuse is rather ad-hoc and support is heavily dependent on the maintainer of a software package. Most organizations listed in Table I have adopted this model. The infrastructure used for this may vary, but a number of organizations have adopted clones of the SourceForge.net platform (e.g., SAP [Riehle et al. 2009]).

The second Inner Source model is what Gurbani et al. [2010] termed the *project-specific* model. In this model there is typically an organizational unit (a core team) that takes responsibility for a certain Inner Source project (a *shared asset*). This responsibility includes the further development of the shared asset and providing support to users of the software throughout the organization. We are aware of two cases that

have adopted this model: Alcatel-Lucent [Gurbani et al. 2010] and Philips Healthcare [Wesselius 2008].

Thus, the two Inner Source models differ mainly in the level of support and dedication a certain Inner Source project receives, which in practice means the availability of resources. Since organizations must spend their resources carefully, budgets for a dedicated maintenance and support team (a key characteristic of the project-specific model) will only be made available for promising software projects that represent business-critical assets to the organization.

A key difference between Open Source and Inner Source is that the resulting software in Inner Source is proprietary, and as such, has no Open-Source license [Goldman and Gabriel 2005]. Of course, organizations may alter their software marketing strategy over time and make the software that was produced as an Inner Source project available as an open-source project, a phenomenon that has been termed *Opensourcing* [Ågerfalk and Fitzgerald 2008]. In such a scenario, an organization may still contribute to, or even lead the software project, in which case we speak of *sponsored* open source [Capiluppi et al. 2012].

2.2. Adopting Open-Source Development Practices

Little research has been done on how organizations can *create an internal bazaar*, let alone whether Inner Source is a viable option for an organization at all. We are aware of a few studies that are related to ours, which we discuss next.

Sharma et al. [2002] presented an extensive and in-depth comparative analysis of traditional organizations and open-source communities. Their analysis was based on three dimensions: *structure*, *process* and *culture*, each of which was further subdivided in a number of themes. Based on this analysis, Sharma et al. proposed a framework for creating, what they called, "hybrid open-source communities."

One of the dimensions used in the analysis by Sharma et al., *culture*, was also the focus of an analysis by Neus and Scherf [2005]. They emphasized that merely adopting the visible, "formal artifacts," such as organizational roles, processes, and tools, is not sufficient, and that an organization must also pay attention to its *cultural identity*, which is not as clearly visible as its formal artifacts.

Interestingly, Robbins [2005] suggested that the very adoption of open-source development tools (a clearly visible artifact, as previously mentioned) will influence an organization's software development processes. Robbins reviewed a number of common open-source development tools and presented an analysis of how using these tools affect software development processes. One example of an organization that has adopted certain OSS development practices through the adoption of OSS tools is Kitware [Martin and Hoffman 2007]. Developers at Kitware also use agile software development practices, which suggests that existing processes can be *complemented* or *augmented* (rather than be replaced) with OSS development practices. Torkar et al. [2011] identified a number of open-source "best practices" suitable for tailoring to fit industrial software development. Based on a review by practitioners, the top three practices considered most suitable were (1) defining an entry path for newcomers, (2) increasing information availability and visibility (what Robbins [2005] referred to as 'universal access to project artifacts'), and (3) letting developers influence ways of working. Rigby et al. [2012], too, identified a number of lessons from open-source development that they consider to be suitable for adoption in proprietary projects.

To conclude, there have been a few investigations into the adoption of open-source practices and tools in conventional organizations. However, each of these investigations is limited in one or more ways. For instance, while the comparative analysis by Sharma et al. [2002] is extensive and insightful, their framework for creating "hybrid-open

source" communities is not grounded in the literature, nor has their framework been empirically evaluated, thereby missing a link to a real-world implementation of Inner Source. Furthermore, Torkar et al. [2011] provide a number of starting points for organizations to adopt individual open-source development practices, but no consideration is made of an organization's particular context. Key factors that are important for adopting Inner Source have not been previously identified.

3. RESEARCH APPROACH

We conducted a two-phased study to address our research goal. In the first phase, we derived a framework to identify the key factors that support an Inner Source initiative. This is further outlined in Section 3.1. In the second phase, we applied the framework in three industry case studies. We deemed a multiple case study methodology appropriate, since it provides real-world cases that illustrate how the various themes identified in the framework "come to life." Details of the case study design and background of the three case companies are presented in Section 3.2.

3.1. Phase I: Derivation of the Framework

A framework is useful to "synthesize previous research in an actionable way for practitioners" [Schwarz et al. 2007], which is why we deemed derivation of a framework a suitable approach to address our research question. As mentioned, there has been limited attention to this topic in the research literature; however, it is interesting—and reassuring—that the various reports are quite consistent with each other.

We first identified all relevant reports (case studies, experience reports) on the adoption of open-source development practices within an organization. Given the nascent state of the Inner Source area as a field of research, we deemed a systematic literature review unsuitable. This was confirmed by a pilot systematic review, through which we identified a very limited number of papers. Another inhibitor to doing a systematic review is a lack of consistent and agreed terminology in the field, as previously mentioned (see also Table I), which makes searching in digital libraries very challenging. Instead, the literature was identified over an extended period of time, through Web searches and following forward and backward references. The identified articles on Inner Source are listed in Appendix A. Some of these are short papers (e.g., Martin and Lippold [2011]), and not all of them provide insights into factors that support Inner Source.

For the framework derivation, we adopted thematic synthesis, following the steps recommended by Cruzes and Dybå [2011], discussed next.

(1) *Extract data.* We first carefully read all relevant papers so as to get immersed in the various studies. Specific segments of text that were of interest to the objective of our review were stored in a spreadsheet, along with the source of each extracted segment. This established an audit trail that allowed us to trace back the various extracted text segments to their original contexts.
(2) *Code data.* After data extraction, we labeled each extracted text segment with keywords. Where multiple keywords were applicable, we listed all.
(3) *Translate codes into themes.* Once all extracted text segments were coded, the various entries in the spreadsheet were sorted by code so as to group all related text segments. An analysis of similar entries resulted in a set of themes. Through an iterative process, we identified nine themes, or elements, that together constitute our framework.
(4) *Create a model of higher-order themes.* We further explored the relationships among the nine elements identified in Step 3, through which we identified three

higher-order themes. These are Software Product, Practices & Tools, and Organization & Community.

(5) *Assess the trustworthiness of the synthesis.* Finally, in order to increase our confidence that the synthesis resulted in a correct representation of the reviewed literature, we applied peer debriefing as a technique among the authors of this paper. This resulted in intense and fruitful discussion as to whether the elements of the framework correctly capture the essence of the reviewed literature.

The synthesized framework is presented in Section 4.

3.2. Phase II: Application of the Framework

In order to illustrate and demonstrate the framework in action, we followed the first phase with an empirical phase, during which we conducted three industry case studies. In particular, one case study was conducted at Philips Healthcare, which had an established track record in Inner Source adoption [van der Linden 2009]. Findings from this case study illustrate how the factors identified manifest themselves in a real-world setting. The other two case studies were conducted at two organizations, Neopost Technologies and Rolls-Royce, both of which had indicated a strong interest in adopting Inner Source and had already started a number of informal initiatives. Findings from these two cases demonstrate how the framework highlights important issues that must be considered for these organizations. All case studies were conducted on-site at the organizations' premises.

3.2.1. Background of the Case Study Organizations.

—*Philips Healthcare* is a large, globally distributed organization which develops and produces product lines of medical devices, such as X-ray and Magnetic Resonance Imaging (MRI) scanners. At the time of our study, the division had a workforce of almost 38,000. Philips Healthcare is organized in a number of product groups, called "business units," each specialized and responsible for the development of their respective products. For brevity, we shall refer to Philips Healthcare as "Philips" in the remainder of this article.

—*Neopost* is a world leader in the mail sector and is a globally distributed organization with a workforce of 5,500, including 300 R&D engineers. The visited location in the Netherlands is a research and development branch of the global Neopost organization, that designs and manufactures inserter machines.

—*Rolls-Royce* is a large organization, employing 11,000 across a wide range of domains, with a large presence in the aerospace domain within which our study was focused. The development teams we studied all worked on development and design tools, rather than the actual embedded software that controls hardware (e.g., aircraft engines). Such design tools are used by design engineers elsewhere in the Rolls-Royce organization.

3.2.2. Data Collection and Analysis. Informed by the established literature that provides guidance and recommendations for performing case studies [Yin 2003; Verner et al. 2009; Runeson et al. 2012], we developed a case study protocol which outlined the objectives, research questions, data collection and data analysis procedures.

We drew on a number of data sources for the three case studies (see Table II) so as to achieve triangulation across different data sources. The data for the three cases were gathered over a period of two years. We conducted semi-structured interviews with a total of 32 participants. All but two interviews were conducted face-to-face; the remaining two were conducted over the phone. Most interviews lasted approximately one hour, and all were recorded with the participants' consent, and subsequently transcribed resulting in approximately 400 pages.

Table II. Data Sources for the Empirical Evaluation of the Framework

Philips	Neopost	Rolls-Royce
Existing reports on Philips' Inner Source initiative.	Internal wiki and project documentation.	Workshop.
Interviews with: - 2 Software architects - 2 Directors - 4 Managers - 2 Team leads - 1 Software designer	Interviews with: - 10 Software developers - 1 System architect - 1 Dept. manager - 2 Team leads - 1 User interface designer	Interviews with: - 1 Chief architect - 4 Team leads - 1 Software developer
2 informal discussions with Inner Source initiator	Meetings (incl. Scrum, future project testing strategy)	

We analyzed the data using qualitative techniques as described by Seaman [1999]. All interview transcripts were thoroughly read, and phrases of interest were coded with the framework elements as seed categories. Throughout the process, short memos were written and exchanged among the authors which were then further discussed as necessary.

In addition to the interviews, we drew from a number of other data sources at the three companies. For the case at Philips, we studied existing reports on its Inner Source initiative, and also had two informal discussions (one hour each) with the initiator of Philips' Inner Source initiative. These discussions gave additional insight into the history of the company's Inner Source program. At Neopost, we studied information available on the internal wiki as well as project documentation. This helped us to understand the current state of knowledge sharing through these media and also supported us in forming an understanding of terminology and domain knowledge which helped in the interviews. Prior to the case study at Rolls-Royce, we held an online workshop during which we presented and discussed the framework. The workshop also included a Q&A session and was attended by approximately 20 people, a few of whom were based in the U.S.

The results of the three case studies are presented in Section 5.

4. A FRAMEWORK OF FACTORS SUPPORTING INNER SOURCE

The framework we derived (see Figure 1) consists of nine elements organized into three themes: Software Product (Section 4.1)[2], Practices & Tools (Section 4.2), and Organization & Community (Section 4.3). For each factor, we present a brief synopsis followed by a review of the relevant literature pertaining to that factor.

4.1. Software Product

4.1.1. Seed Product to Attract a Community. To 'bootstrap' an Inner Source initiative, there must be a "seed" product—a shared asset—that is of significant value to the organization, or at least of high *perceived* value. The seed product must have an initial and runnable version, which can attract a community by first attracting users, who may subsequently become contributors [Wesselius 2008]. This community of users and contributors is a key factor to an Inner Source project's success.

The need for an initial basic architecture and implementation was clearly noted by Raymond's [2001, p. 47] earlier observation.

[2]Where we write "product," this can be interpreted as module, component, or other unit of software.

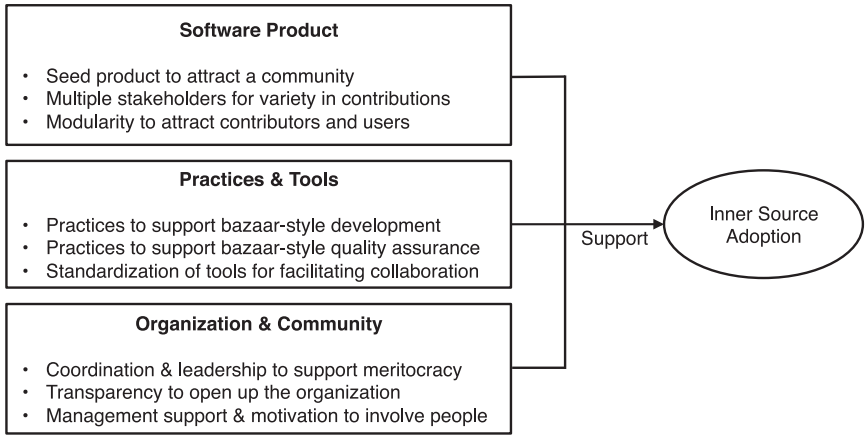

Fig. 1. Factors that support Inner Source adoption.

> "It's fairly clear that one cannot code from the ground up in bazaar-style. One can test, debug and improve in bazaar-style, but it would be very hard to **originate** a project in bazaar mode. Linus didn't try it. I didn't either. Your nascent developer community needs to have something runnable and testable to play with." (Emphasis by Raymond)

Feller and Fitzgerald [2002] summarized this observation as follows: "planning, analysis and design are largely conducted by the initial project founder, and are not part of the general OSS development life cycle." A real-world example of this is Topaz, which was an attempt to re-implement the Perl programming language. Its initiator described this as follows:[3] "when you're starting on something like this, there's really not a lot of room to fit more than one or two people. The core design decisions can't be done in a bazaar fashion."

Senyard and Michlmayr [2004] labeled this initial stage the *Cathedral* phase, after which a project may transition into the *Bazaar* phase. The Cathedral phase is characterized by an initial idea and implementation by an individual or small team; for instance, the Inner Source initiative at Alcatel-Lucent (an implementation of the Session Initiation Protocol (SIP)) started with a single developer [Gurbani et al. 2006]. Once in its Bazaar phase as an Inner Source project, it attracted a significant community of users and developers within the organization which was pivotal for its further development. It has been suggested that the requirements and features of the seed product need not be fully known at the outset so that the project can benefit from organization-wide input and continuously evolve [Gurbani et al. 2006]. If a project is fully specified and implemented (and thus only needs maintenance), there is little need for new contributions from a wider community. As Wesselius [2008] aptly pointed out, "a bazaar with only one supplier isn't really a bazaar. Dinkelacker et al. [2002] found that it is a challenge to find an appropriate initial software domain. Gurbani et al. [2006] hypothesized that lacking an initial seed product, "a research or advanced technology group is a good location to start a shared asset," while Wesselius [2008] argued that the seed product should have a well-defined specification so that its development can be outsourced to a central development group (i.e., a core team). It is important that the shared asset has significant differentiating value to the organization; if it is

[3]`http://www.perl.com/pub/1999/09/topaz.html`.

merely a commodity (e.g., a database system or operating system), there may not be sufficient justification for in-house development [van der Linden et al. 2009].

4.1.2. Multiple Stakeholders for Variety in Contributions. An Inner Source project must be needed by several stakeholders (i.e., individuals, teams, or projects that productize the shared asset) so that members from across an organization can contribute their expertise, code and resources. This in turn will help to establish a sufficiently large pool of users and contributors to establish a vibrant Inner Source project.

Gurbani et al. [2006] argued that different product groups have different needs, and that groups can benefit from other groups' contributions. For instance, the SIP implementation at Alcatel-Lucent benefited greatly from feedback and suggestions of colleagues who were experts in specific fields, such as network optimization and parsing techniques. In other words, input from across an organization can significantly broaden the expertise that is available to a project [Riehle et al. 2009]. Organization-wide feedback can help to solicit a variety of ideas, a sentiment that we would express (inspired by Linus's Law) as: "Given enough mindsets, all ideas are obvious."

The presence of multiple stakeholders suggests that there is considerable interest in the development of a software product; hence, this indicates a good motivation to pool resources and to develop it as an Inner Source project [Gurbani et al. 2006]. Furthermore, having different product groups integrate the shared asset helps to manage its scope and reusability in different contexts [Robbins 2005]. Robbins [2005] outlined the tension with conventional software development, in which teams try to optimize returns on their current project on the one hand, and the cost of providing ongoing support for reusable components on the other hand. From a business perspective, Gurbani et al. [2006] argued the following:

> "It is essential to recognize and accommodate the tension between cultivating a general, common resource on the one hand, and the pressure to get specific releases of specific products out on time."

4.1.3. Modularity to Attract Contributors and Users. An Inner Source project should have a modular architecture. This will facilitate parallel development as different developers (i.e., contributors) can work in parallel on different parts of the project without any merge conflicts. Furthermore, a modular architecture will also help reuse as integration can become easier, thus increasing the number of potential users.

A modular software structure has many advantages and is generally preferred [Parnas 1972] but is particularly important in open-source style development, as it facilitates parallel development, that is, allowing many developers work on different parts of the same product simultaneously [Torvalds 1999; Bonaccorsi and Rossi 2003]. Successful open-source projects tend to exhibit a high level of modularity [O'Reilly 1999; Feller and Fitzgerald 2002]. A modular architecture greatly helps in merging various changes into a single development branch [Gurbani et al. 2006] or even to let developers work on the same repository at the same time (with no merges needed). MacCormack et al. [2006] refer to this as "architecture for participation," the idea that modularity facilitates developers to better understand the code, and which thus enables developers to contribute in a meaningful way. Baldwin and Clark [2006] showed that codebases exhibiting a higher modularity help to increase developers' incentives to join and remain involved in an open-source project, thus helping to keep contributors involved.

Another benefit of modularity is that it supports *reuse* of a component, which means that it helps to attract (and keep) users in the community. Reuse is a key motivation to start an Inner Source project in the first place [Dinkelacker et al. 2002; Vitharana et al. 2010]. An Inner Source project will only attract users if it offers a significant set of functionality (cf. Section 4.1.1). However, when the shared asset becomes too

"heavyweight," it may be perceived as too difficult or inconvenient to use. Dinkelacker et al. [2002] expressed this sentiment as follows: "Using the word 'module' somewhat generically, it's a challenge to strike the balance between module simplicity and utility." This trade-off refers to a component's completeness (utility) thereby broadening a product's appeal [Robbins 2005], and preventing feature creep, making reuse of the component undesirable due to its complexity or size. One of the principles of collaboration in open source is egalitarianism [Riehle et al. 2009], the idea that anyone is welcome to contribute a new feature to "scratch an itch" [Raymond 2001; Robbins 2005]. However, this may also "lead to feature creep and a loss of conceptual integrity" [Robbins 2005]. Losing 'conceptual integrity' may result in a component which is no longer easy to use or whose architecture imposes too many restrictions on the client-application.

4.2. Practices and Tools

4.2.1. Practices to Support Bazaar-Style Development. Software development in an internal bazaar is inherently different from conventional approaches, including contemporary ones such as agile methods. Two aspects related to implementation activities are particularly different in open-source development: requirements elicitation and maintenance. For an Inner Source project to succeed, developers should be comfortable with bazaar-style practices, as conventional approaches may not be appropriate.

In terms of requirements elicitation, the process of identifying a feature and providing an implementation is very different from traditional "requirements engineering" scenarios, and thus this is something that Inner Source developers should be comfortable with. Whereas conventional development approaches may have procedures in place that prescribe how requirements are gathered, stored, and managed, the requirements process in open-source projects may have a much faster turnaround time from initial idea to implementation. Deliberation about features, details, and idealized systems is typically not much appreciated. This is not to say that open-source developers do not discuss requirements at all, but that a sound balance must be found. On the one hand, conversations (e.g., on mailing lists or IRC) may lead to discussions (disputes, even) on the most minute and trivial details, a phenomenon known as "bikeshedding" [Fogel 2005, p.98]. On the other hand, some community members may have the most fantastic plans and idea, but it is actual running code that is valued most. Linus Torvalds, 'benevolent dictator' of the Linux kernel project, expressed this sentiment famously as "Talk is cheap. Show me the code." [Torvalds 2000]. Thus, a common scenario is that developers identify features, provide a "patch" that implements the new functionality after which it is offered to the community for peer review. The fact that the feature was needed in the first place is "asserted after the fact" [Scacchi 2004].

As an open-source project evolves, it is subject to maintenance. Open-source projects slowly evolve, with many minor improvements and mutations, over many releases with short intervals. Or, as Merilinna and Matinlassi [2006] phrased it, "OSS is a moving target—or a dead corpse." A project founder may not have anticipated the project's evolution where the many additions of various features uncover the limitations of the initial design made in the project's Cathedral phase. As a result, contributors feel the need to re-implement a subsystem, or what Scacchi [2004] has termed "reinvention." Raymond cited Brooks [1995, p. 116], who captured this idea well: "Plan to throw one away; you will, anyhow." The Perl programming language is a good example of this. Perl versions one to four were incremental. Perl's creator, Larry Wall, reimplemented Perl 5 from scratch. However, at some point that code base was characterized[4] as "an interconnected mass of livers and pancreas and lungs and little sharp pointy things and the occasional exploding kidney." Around the year 2000, it was decided to re-implement

[4]`http://www.perl.com/pub/1999/09/topaz.html`.

the language once again, resulting in a thorough reinvention.[5] Reinvention in Inner Source may be more restricted than open-source communities due to the pressure of productization and limited resources. In practice, therefore, reinvention is limited only to those parts of a product that are critical bottlenecks. For instance, the parser in the initial SIP implementation at Alcatel-Lucent was optimized for scalability by having an expert on parsing techniques from within the organization 'reinvent' it.

4.2.2. Practices to Support Bazaar-Style Quality Assurance. One of the key tenets of successful open-source projects is a set of mature quality assurance (QA) practices. For an Inner Source project to flourish and achieve a high level of quality, it is important that a set of QA practices are adopted that are suitable for an Inner Source project.

Open-source quality assurance practices can differ significantly from those used in conventional software development. Peer review, which was briefly mentioned, is one of the best known practices in open-source development [McConnell 1999; Rigby et al. 2012]. Feller and Fitzgerald [2002, p. 84] characterized peer review in open-source settings as *truly independent*; that is, peer-review in open-source projects is a self-selected task by other interested developers [Asundi and Jayvant 2007; Rigby et al. 2008]. Open-source developers are more likely to provide genuine feedback given their interest in the success of the project they work on, rather than doing a review because they were told to and possibly having to consider relationships with co-workers when pointing out any issues with their contributions.

Peer review becomes particularly effective when there is a large number of developers in a project, as it can benefit from what is known as Linus's Law [Raymond 2001]: "Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone," more often stated succinctly as "Given enough eyeballs, all bugs are shallow."

Peer review can take place either before or after committing changes to the source code, depending on whether the contributor making the changes has a 'commit bit,' that is, write access. It is an important and effective practice to ensure that any code that is checked in is of good quality, does not contain hacks, and will not lead to an undesirable path of evolution (i.e., prevent future improvements). Direct commit access is usually only given to long-time developers who are trusted to protect the project from bad check-ins; such developers are also known as "trusted lieutenants" [Raymond 2001; Gurbani et al. 2006].

A cardinal sin is to "break the build," that is, checking in code that would prevent the project compiling successfully, as it puts all developers' progress on hold. In order to prevent this from happening, contributors are expected to test their changes first before submitting a patch. Regression test suites and use of specialized testing tools (e.g., Tinderbox) are commonly used in successful open-source projects to provide monitoring of the quality.

Another practice to support quality assurance is to make regular and frequent releases [Robbins 2005; Michlmayr and Fitzgerald 2012]. Releases can be of different types, such as development releases or production releases that are more thoroughly prepared (or 'hardened') for productization. Release management in conventional software development can vary substantially, depending on the development approach that is taken (e.g., waterfall [Royce 1987] versus agile approaches such as Scrum). Likewise, while release management in open-source projects can vary significantly as well [Erenkrantz 2003], successful open-source projects tend to follow Raymond's advice: "Release early. Release often. And listen to your customers" [Raymond 2001, p. 29]. In recent years, a number of successful open-source projects have adopted a

[5]http://www.perl6.org/archive/.

time-based release strategy, as opposed to a *feature-based* release strategy [Michlmayr and Fitzgerald 2012]. Time-based releases offer various benefits, such as less "rushing in" code for new features as releases are frequent, and less fire-fighting as the release process is exercised more often. A regular release cycle also means that there are more opportunities for "bug squashing" sessions (usually held prior to a release), and more regular feedback of users, both of which can contribute to a project's quality. This practice is also very suitable for Inner Source projects; for instance, the SIP stack at Alcatel-Lucent used to be on a two-weekly (development) release schedule.

4.2.3. Standardization of Tools for Facilitating Collaboration. A key success factor for an Inner Source project is that there is a set of common and compliant development tools so as to make contributing easy [Dinkelacker et al. 2002; Gurbani et al. 2006; Riehle et al. 2009]. Differences in the tools that are used across an organization (not uncommon in large organizations) can be an obstacle for developers to contribute, or may necessitate duplication of the code repository, causing additional synchronization efforts. For that reason, a set of common tools must be available throughout the organization.

Tools commonly used by OSS projects are (besides compilers) version control systems (e.g., Subversion), issue tracking software (e.g., Trac), mailing lists, and wikis [Robbins 2005]. While a number of these tools are also commonly used in commercial software development, different business units within large organizations often use a wide range of different tools [Robbins 2005; Gurbani et al. 2006; Riehle et al. 2009]. Riehle et al. [2009] reported that "The biggest hurdle to widespread adoption of SAP Forge is its limited compliance with tools mandatory for SAP's general software development process." Several authors reported that moving code among different version control systems is challenging [Dinkelacker et al. 2002; Gurbani et al. 2006]. For instance, Gurbani et al. [2006] reported their experiences of certain teams replicating the original shared asset's source code into their preferred repository, causing significant merging problems later on.

Therefore, an organization considering adopting Inner Source should pay sufficient attention to addressing this issue. Often, barriers to achieve this are not of a *technical* nature, but rather *organizational* or sometimes *political*. Organizational policies enforced by IT departments of large, global organizations may have significant impact on what can be achieved, even if supported by local management.

4.3. Organization and Community

4.3.1. Coordination and Leadership to Support Meritocracy. An Inner Source project requires a bazaar-style approach to coordination and leadership so as to allow a core team, trusted lieutenants, and other motivated contributors to emerge. Providing flexibility to members across an organization is key to enabling a community to flourish.

Leadership and coordination are two aspects that differentiate open-source projects from traditional projects. Several authors have highlighted the importance of leadership, including a core team that takes responsibility for development and maintenance of a shared asset [Dinkelacker et al. 2002; Gurbani et al. 2006, 2010; Wesselius 2008]. Leadership in traditional organizations is based on seniority and status (e.g., junior vs. senior developer) [Riehle et al. 2009]. This is very different from open-source communities, where projects are typically started by one developer who acts as a "benevolent dictator."[6]

As a project matures, new key developers can emerge based on the concept of meritocracy [Gacek and Arief 2004; Neus and Scherf 2005; Riehle et al. 2009]. Status is earned based on the merit of a developer's contributions, which usually also results in a

[6]E.g., Larry Wall in Perl, Linus Torvalds in the Linux kernel, and Guido van Rossum in Python.

bigger say in important decisions regarding the project. Gurbani et al. [2006] reported the following:

> "Some developers will naturally gravitate towards understanding sizeable portions of the code and contributing in a similar manner, often on their own time. Such developers should be recognized by making them the owner of particular subsystems or complex areas of the code (the "trusted lieutenant" phenomenon)."

Whereas in traditional organizations coordination is based on project plans and release schedules, open-source projects are rather self-organizing [Crowston et al. 2007]. Open-source developers, not concerned with schedules or deadlines, typically select tasks that they are interested in [Feller and Fitzgerald 2002; Robbins 2005; Riehle et al. 2009]. The motivation to do so is often explained as "scratching an itch" [Raymond 2001]. Torkar et al. [2011] identified task selection as an opportunity for commercial software development organizations to adopt a bazaar-style of working; letting go of formal structures and ownership was also suggested by Dinkelacker et al. [2002]. However, Inner Source projects cannot be fully self-organizing, as there are business aspects to consider such as the timely delivery of products that depend on the shared asset. Thus, self-organization is one area where the open-source approach needs to be tailored, or renegotiated to fit a corporate context. In the case of Alcatel-Lucent, for example, Gurbani et al. [2010] identified a number of project management roles within their core team. A key role is that of the *feature manager*, that is, someone who performs a certain level of coordination. For instance, the feature manager identifies features that need to be implemented and will identify potential developers in the wider organization who would be suitable work on a particular feature. This process depends partly on knowledge of who-knows-what and involves finding a balance between developers' availability and engaging in a dialogue with those developers' managers so as to be able to "borrow" them.

The core team and trusted lieutenants previously mentioned must agree on a common vision for future evolution of the project [Gurbani et al. 2006] and guard the common architecture so that additions from one contributing group do not lead to integration problems in another group.

4.3.2. Transparency to Open Up the Organization. Transparency lies at the heart of, and is a prerequisite for, open collaboration projects, even when this 'openness' is limited to an organization's boundaries, as is the case for Inner Source. Transparency is essential to establish visibility and to attract interested developers who may start out as 'lurkers' and transform into active contributors. We discuss three aspects of transparency: organizational culture, artifacts and communication.

Neus and Scherf [2005] suggested learning about the organization's culture and whether there is a strictly obeyed hierarchy. They suggest to do the "Emperor's clothes" test in order to see how open the organization is, describing this as follows:

> "We find out if there are ways in the organization that allow a novice (e.g., an intern) to publicly call attention to the emperor's (i.e., the expert's) lack of clothes (i.e., to raise quality issues), or if all internal communication addressed to a larger audience has to go through some gatekeepers."

An open development culture was also advocated by Riehle et al. [2009], who argued in favor of egalitarianism, and that "projects must exhibit a mind-set that welcomes whoever comes along to help, rather than viewing volunteers as a foreign element." Raymond [2001], too, argued that it is critical that a coordinator is able to recognize good design ideas from others. Dinkelacker et al. [2002] reported that "getting developers

and their management chain comfortable with sharing code and development responsibility across the company" may be challenging.

Open collaboration projects provide universal, immediate access to all project artifacts, allowing anyone to participate [Robbins 2005; Riehle et al. 2009]. While a set of common tools (cf. Section 4.2.3) facilitates the more technical aspects that ensure a common development environment, infrastructure is essential for a transparent process that is based on sharing information and keeping information widely available and up-to-date. The extent to which project artifacts (i.e., source code, issues, documentation, etc.) are accessible to others in commercial software development settings may vary widely from organization to organization. It is not uncommon that such artifacts are private to the developing team.

It is important also that access is straightforward and that there are no barriers to finding information on an Inner Source project, so that potential new community members do not find getting involved to be too cumbersome. At Alcatel-Lucent, for instance, a Centre of Excellence (COE) was established, that provided infrastructure for accessing the source code as well as other information on the Inner Source project [Gurbani et al. 2006]. This way, the COE established a "one-stop shop" for users of the shared asset. It is important to provide sufficient support and maintenance for this and other infrastructure that is needed for Inner Source [Dinkelacker et al. 2002].

Other infrastructure includes means of communication, such as a mailing list for online (and archived) discussions, IRC for synchronous and real-time communication, and Q&A forums. More mature open-source projects can also organize regular meetings in an IRC channel. For instance, the Parrot project has weekly progress meetings to which all developers with a "commit bit" are invited.[7] This is very different from conventional communication mechanisms, such as scheduled meetings, where face-to-face communication is much more common than in open-source projects (where face-to-face communication is minimal if not non-existent). Agile methods in particular value face-to-face meetings over written reports and memos [Fowler and Highsmith 2001]. Though organizations involved in distributed development lack face-to-face communication as well, modern technological infrastructure (e.g., video conferencing) facilitates virtual face-to-face meetings, which are highly uncommon in open-source projects. In any case, meetings in commercial organizations are less "bazaar-like" in that participants are not supposed to join or leave as they please.

4.3.3. Management Support and Motivation to Involve People. A key condition for establishing a successful Inner Source project is to acquire top-level management support on the one hand and to involve interested people (i.e., users and contributors) in an organization on the other hand. Support from top-level management is required so that additional resources can be granted if requested, and also for advertising and advocating an Inner Source project throughout the organization.

Several authors have argued for the importance of management support [Dinkelacker et al. 2002; Gurbani et al. 2006, 2010; Riehle et al. 2009]. Dinkelacker et al. [2002] described how this was a crucial factor for one of HP's Inner Source initiatives, which is named CDP.

> "It has been critical to CDP success to have a group of executive champions. In CDP's case we have two champions, the chief technology officer and the chief information officer. These champions provide the urgency to the organization to start the change process."

[7]One of the authors (kjs) was a contributor to the Parrot project (with a commit bit).

Riehle et al. [2009] wrote that managers of research projects are generally supportive of the volunteer contributions, but that managers of volunteers from regular product groups are typically skeptical in the beginning. However, Riehle et al. also found that management became "neutral" or even supportive once they realized the benefits of early engagement with research projects. An important issue here is that this management support is also expressed in terms of budgets, that is, that budgets and resources are made available to support this. Wesselius [2008] discussed a number of criteria for an internal business model that can support this.

While management support is essential, merely enforcing an Inner Source initiative from the senior management level is not sufficient and in fact goes against the open-source development philosophy that is characterized by *voluntary* participation [Wesselius 2008]. Neus and Scherf [2005] emphasized the importance of passionate people: "to drive change, you need passion," and "people who understand and are excited about the change." In a similar vein, Raymond [2001] wrote the following:

> "In order to build a development community, you need to attract people, interest them in what you're doing, and keep them happy about the amount of work they're doing."

Neus and Scherf [2005] argued that the cultural shift needed for adopting a bazaar-style development approach cannot be forced but merely facilitated. In order to achieve motivation and "buy-in" of staff, it is essential to demonstrate value first, and suggest to do so by solving a concrete problem with a small scope. In order to get started, a few authors have suggested to provide what has been termed a "gentle slope" [Halloran and Scherlis 2002] to provide some handholding [Gurbani et al. 2010] or to define an "entry path" for newcomers [Torkar et al. 2011], which may eventually lead to an active community of contributors.

5. CASE STUDIES

Following the derivation of the framework (see Figure 1) we now present the results of applying the framework in three large organizations. For each factor, we discuss how this has manifested at an existing Inner Source implementation (case I, Philips), after which we present the findings from cases II (Neopost) and III (Rolls-Royce), both of which have initiated informal bazaar-style initiatives. Respondent identifiers are prefixed with the first letter of their organization's name (i.e., P, N and R).

5.1. Software Product

5.1.1. Seed Product to Attract a Community. The seed product at Philips was initially a set of components, or *component suite*, with common functionality related to the DICOM (Digital Imaging and Communications in Medicine) standard. This is a standard for exchanging medical images and workflow information, and as such, is used by most product groups within Philips. For that reason, this set of components was a good starting point, or seed, for Philips' Inner Source initiative. Over time, this component suite evolved into a platform for a software product line, which is still evolving and extended today.

In both Neopost and Rolls-Royce, we identified several potential seed products. At Neopost, a promising component is a Hardware Abstraction Layer (HAL). The HAL is a component that controls the actual hardware of inserter machines. Common operations are abstracted away, which decouples a machine's user-interface application from its hardware. The HAL is written in C++ and has a well-defined interface consisting of a set of *virtual* functions (supporting polymorphism). This allows for easy and modular extensions by others, and it is likely the HAL will need many extensions in the future,

as software support for new hardware components must be added. Given that the HAL will be used in most future machines developed at Neopost, it is of significant value to the organization.

At Rolls-Royce, we identified a number of candidate seed products, some of which are still in a prototyping phase. Participants generally agreed across groups that a prototype would help to gather more requirements, which they felt supported the need for an initial seed product. One such product was a geometry automation framework (GAF). Whereas engineers can manually "*point and click*" designs using a CAD system, the GAF offers frequently needed design constructs. The developers of GAF expected that its users would identify new use-cases and require additional functionality over time. This need for future extensions would make the GAF a particularly interesting candidate for an Inner Source project.

Another promising project we identified was a tool integration framework (TIF). The TIF is a framework that implements a certain workflow, by integrating and invoking a variety of analysis tools that are specific for a certain group (e.g., turbines, compressors). What is common is the workflow and integration capability, but the actual analysis tools that are 'plugged in' (and invoked by the framework) differ per group. The project lead for the TIF described how it already resembled the start of an Inner Source project.

> "I gave the source code to the folks of the turbine group that were supporting TIF, and I said, you can pick this up and use it and start swapping out the compressor codes with turbine code. There's 3 different groups now, different parts of our organization that are all supporting the TIF project." (R6)

Given that the TIF offers potentially significant benefit (and thus, value) to the organization, this project would therefore be a suitable seed product.

5.1.2. Multiple Stakeholders for Variety in Contributions. Philips is a large organization with many product groups that use the Inner Source project as a platform. It is used in more than 300 products delivered by more than ten product groups. Such a large number of stakeholders offers opportunities for large-scale reuse, which is why software offering common functionality is an appropriate choice for a shared asset.

We found that such a large number of different stakeholders also comes with challenges. One challenge in particular is that there are many requirements coming from the various different teams, and the platform team forms a bottleneck. This was one of the reasons that Philips started their Inner Source initiative in the first place, to address this bottleneck. However, the platform team, which plays the role of the core team managing the Inner Source project still has limited capacity. Getting product groups to implement their own requirements has been a challenge.

The candidate components identified at Neopost and Rolls-Royce also have potentially many stakeholders. The HAL just mentioned provides generic functionality needed in virtually all products and will be used in all future machines that Neopost develops. In fact, one participant described how the HAL component would be suitable as an Inner Source project.

> "I can see that work for HAL, to make some kind of OSS development environment for that, so that everybody could contribute." (N7)

One issue we identified in our study is that, while the number of stakeholders in terms of future projects is quite high, the number of developers is rather limited in the visited location of Neopost. This somewhat limits the potential for a wide range of input in terms of requirements, innovative ideas, and so on. A limited number of

developers means that they may have limited time to spare to work on the development of a shared asset.

At Rolls-Royce, the geometry automation library has potentially many users throughout the global organization (approx. 1,600). As one participant indicated, "the potential is huge, really, because geometry is at the heart." One issue that was raised by the interviewees is that the GAF is a new development, and as such, using it will change the users' workflow.

> "we're reshaping their work. So, we're telling them to automate their CAD rather than doing their CAD manually." (R2)

Therefore, it is important that benefits of using the GAF are clearly demonstrated to potential users. One of the interviewees indicated that, for the GAF project to succeed, a 'pull' from the stakeholder needs to be created. But, as the project lead clarified, "Generally people invest the time because they find it useful. They select themselves."

The intent for the GAF project is to have users propose new use cases, which can then be implemented as new features or functionality. Initially, the GAF was targeted towards a single customer within Rolls-Royce, but since other groups have similar requirements, the development group are broadening the scope by looking across various business units within Rolls-Royce.

5.1.3. Modularity to Attract Contributors and Users. Philips' shared asset is a large (several millions of lines of code) and still growing piece of software. The platform initially started out as a 'component suite,' and this model was quite successful. However, the combination of different uses of the components yielded suboptimal results, in that the need for integration testing was not eliminated. For that reason, the core team moved to delivering "component assemblies," which are pre-integrated and tested half-products that product groups can use. While the components may be modular, they are not suitable for an à la carte approach, whereby only those components deemed useful are selected, as one participant described.

> "The components are not so rich that you can choose component A, B, C and D. What you would do is configure the platform and within that configuration you'd choose A, B, C or D. It's more an integrated approach." (P2)

At Neopost, we found that the HAL is a very modular component, whereby the various hardware drivers are stored in separate files, which facilitates parallel development. This was in stark contrast to another component that we identified, whose main purpose is to read and decode instructions from a sheet of paper (as it is processed in a machine). This component was closely coupled with a system application (containing the graphical user interface) that uses it. One developer explained this as follows:

> "And instead of considering [it] as a component, it's just a copy of the [system] application [...] So, saying that you developed a 'component' is arguable, because they really delivered a whole application with a small part that we also use." (N10)

In terms of complexity and modularity of the GAF at Rolls-Royce, the project lead was mainly concerned about presentation. Providing a large amount of functionality was not considered to be an obstacle, but presenting and documenting it in an accessible way was considered to be a potential challenge, as the project lead explained, "I don't care [the user] has access to 10,000 functions and he only uses three, as long he's not bewildered by the other 9,997 that are available."

A project lead working on a different candidate project found that modularity is very important and that the initial modular design is currently broken up into an even finer granularity, describing the rationale as follows:

> "Primarily it's to facilitate parallel development, in that, having it modular means that we can pass a module to [others], with a clear set of requirements." (R4)

5.2. Practices and Tools

5.2.1. Practices to Support Bazaar-Style Development. Philips, offering advanced systems for the medical domain, is operating in a regulated environment and as such its development process is subject to audits by regulators, such as the U.S. FDA. Philips have extensive processes for how requirements move from product groups to the core team. Formal channels for communicating requirements include issue trackers and databases with "all sorts of procedures," but there are also a number of informal channels. These include mailing lists on which developers can provide support to people in other product groups.

Though an open-source way of working suggests a central repository through which developers can improve the code and make changes as they see fit, this is not the case in Philips. While the source code of the common platform is accessible to all business units (facilitating developers in inspecting and understanding how features are implemented), individuals do not typically make changes directly, as one participant explained, "It's not like every developer can check out a piece of the platform, change it, and check it back in." Instead, a key concept in Philips' Inner Source initiative is that of *co-development projects*, whereby one or more members of the core team work closely with a product group to develop (or enrich existing) functionality. Such 'co-development' of features ensures that the new software (a) adheres to the architecture of the shared asset, and (b) implements the new functionality correctly as required by the product group. A core team member that works closely with a product groups is what Gurbani et al. [2010] termed a *feature advocate*.

Developers at Neopost follow the Scrum method to manage software projects, using three-week sprints. It is important to note that while software has an increasingly important role in Neopost's products and is becoming larger and more complex over time, the role of software is to control a machine's hardware. This means that product development is driven by development of the hardware, and the software development process must keep up. We found that Neopost does not have a formal requirements elicitation process in place, but there are a number of sources that the development team relies on. First, the most important source of requirements is the marketing department, which represents the "customer" role in the Scrum method. Other stakeholders throughout the organization also provide requirements, such as service, manufacturing and hardware engineers. All people involved have extensive domain knowledge, and as a result, developers know many of the requirements, given the common features shared by all products. The maintainer of the HAL component just mentioned responded similarly: "It was clear that we needed a hardware abstraction layer, and for me it was clear it had to be independent of the operating system."

We found that most maintenance tasks are of a *corrective* or *adaptive* nature [Swanson 1976]. For instance, the maintainer of the HAL indicated that requested corrections can be implemented in between jobs, but that there is little time for *perfective* maintenance in terms of design overhauls. Improvements such as porting the existing code base from a 16 bit to a 32 bit processor are more time-consuming and therefore may not be accomplished promptly.

Another participant also described how some of the development is already similar to open-source development. In this particular case, the participant described how one project implemented certain features in a component that did not have high priority for its maintainer.

> "They develop those things that they want, but don't have a real priority for us ... and then [N12] reviews the changes. [...] So, it hardly costs us any time, and still [we] have the benefit that the component can do what they need." (N13)

One potential barrier to community-style maintenance is that of testing the changes made to a common component, due to the embedded nature of the software and the variety in hardware. Improvements for one machine may break functionality on another.

We found that the different teams at Rolls-Royce do not employ a strictly enforced or formalized approach to software development in practice; typically this varies per project and team. The approach taken by the GAF team was described as 'agile.' In terms of maintenance as reinvention, we found that all groups are well aware of the need to overhaul the design of certain software projects, which in certain cases does happen.

> "There's a lot of maintenance that is just incremental. A particular bug, in particular code. Every so often you look at it and say, OK, this needs reinvention. What was nice and clean on day 1 is now spaghetti." (R1)

One important issue that must be considered is that reinventing software tools that are used to make CAD designs that are used in production will require validation with respect to the old version of the software. As one participant explained, "You need to accept the big burden of validation when rewriting it." Product design is a time-consuming activity, and remaking older designs using newer (reimplemented) versions of the software tools may not be feasible.

5.2.2. Practices to Support Bazaar-Style Quality Assurance. The operations subdivision of Philips' core team has responsibility for the verification and testing of the platform they deliver. Given the regulated domain, documentation on design and tests have to be delivered. This team runs system (black box) tests. Unit tests (of individual components) are white box tests. The operations team also supports product groups in writing effective tests. In terms of frequent releases, the core team makes a new release approximately twice a year. These are stable and versioned releases, which are fully tested, documented, and supported. Since such releases represent new 'versions' of the platform accompanied with required test and design documentation, the core team must ensure there are no open issues, which makes such releases more costly and thus not very frequent.

Product groups also have the option to use a more recent snapshot that provides more 'cutting-edge' functionality, much like development releases found in open-source development [Michlmayr and Fitzgerald 2012]. These facilitate early feedback to the core team, so that any issues can be resolved early. One product group leader described this as follows:

> "We sit very closely with the platform team, with as many integrations as possible, and keep that feedback loop as short as possible, close to the OSS model, and that works much better. Much less hassle and trouble." (P6)

In terms of quality assurance at Neopost, we identified fragments of a bazaar-style fashion, as one participant described.

> "I thought there was a bug in HAL [...]. I first solved it locally, and then as some kind of review for the maintainer, sent it to him and said, I think this is what's wrong, and this is my solution." (N7)

The maintainer of HAL described this as follows: "If they want to change something in HAL, then they make a 'diff' and send it to me for review. And if I think it's no problem, then I check it in."

The approach to software testing in Neopost varies per project; some projects have automatic unit test frameworks, whereas others have a more manual approach. The nature of testing is difficult due to the fact that much of the software is embedded. At machine level, a test department subjects the machines to extensive tests. After each development sprint, the development team gives a demonstration of the current version of the machine under development to Marketing, who provide immediate feedback. This three-week sprint cycle ensures that the software is always in a runnable state whereas the demonstrations and feedback help in delivering increasingly better versions of the product.

In terms of quality assurance practices at Rolls-Royce, we found that some teams are already using a regression test suite, which was characterized as semi-automated. Some of the teams have a strong focus on tests, using a Test-Driven Development (TDD) approach, and require new members of the core team to be well-versed in TDD. Additionally, teams have realized that increasing the pace of releases is useful, as one participant described.

> "Two reasons really. One is the user impatience for fixes. The other is, it seems to be easier to manage... less risky in each release. We went from nearly never to a [release per] year and then to every 6 months, and now every couple of months." (R5)

5.2.3. Standardization of Tools for Facilitating Collaboration. Philips use a toolset that is provided using a Software-as-a-Service (SaaS) model by an external supplier. The core team has an "Operations" subdivision that provides operational support for the development environment that is rolled out throughout the organization. This ensures that all product groups have the same development environment, which prevents the various problems associated with different toolsets.

At Neopost, too, all developers at the visited location use the same integrated development environment. Some teams use additional tools, for instance, one team uses ReviewBoard[8] which facilitates the coordination of a peer-review process. Most development is done in C++ targeting embedded platforms and controllers. This focus on a common technology reduces the need for a wide variety of development environments and tools (e.g., compilers, debuggers).

In our case study at Rolls-Royce, we found that there is a wide variety of software, written in COBOL, FORTRAN, C, C++, Java and Python. Furthermore, there is a plethora of development environments and platforms, including variants of Unix, Linux, and Microsoft Windows, as one participant explained, "We're developing for Unix, and multiple flavors of PC, which makes things very complicated." Given this large variety in technologies and platforms, the company also needs to maintain the development environments and tools. An important consideration here is that the organization cannot afford the loss of design capability, which is why there must be a tight control of the IT infrastructure. On the other hand, targeting development at multiple platforms helps to increase the software's robustness, as some compilers are more 'pedantic' than others in terms of compiler errors.

[8] `www.reviewboard.org`.

Software development machines at Rolls-Royce are managed by a central IT department that has responsibility for a wide range of machines throughout the organization. The majority of these are not used for software development, and thus there is less flexibility in terms of software tools that are made available.

5.3. Organization and Community

5.3.1. Coordination and Leadership to Support Meritocracy. In Philips, a Steering Committee decides on the new features that will be delivered in the next version, which provides a high-level form of coordination. Sometimes, the core team gives priority to certain business units for a certain release when planning the implementation of new features. Some groups are not eager to contribute but are more passively waiting for the platform team to implement new functionality. By more actively contributing, a product group gets more control over new features and functionality, which helps ensure that the new code does what the product group needs. One participant remarked that the groups involved from the beginning have been most successful in getting their requirements implemented.

Neopost follows the Scrum methodology as previously mentioned, which provides the overall framework for daily coordination, and therefore provides much flexibility. Scrum meetings take place in the morning. At departmental level, all Scrum Masters gather with the department manager on a weekly basis for a Scrum-of-Scrums meeting. Software teams are co-located, which allows for quick, ad-hoc face-to-face meetings. Software engineers tend to specialize in certain parts of the whole system, thus enabling them to continue working in parallel on the same system. This specialization makes them the "contact person" for their particular subsystem. This suggests that such a person has taken on the role of a "trusted lieutenant" (in bazaar-terminology), which differentiates the process from the agile method (Scrum) that is followed. Agile methods emphasize the concept of Collective Ownership, whereby the development team as a whole is responsible for the code, and any developer may make changes [Schwaber and Beedle 2002; Cohn 2009]. Another important factor at Neopost is that machine development is hardware-driven, in that the software supports the hardware. As a result, it is very important that software and hardware engineers coordinate any changes amongst themselves as any changes in the hardware may have impact on the software. Some projects organize weekly face-to-face meetings to achieve this coordination.

Interviewees at Rolls-Royce felt that they had a relatively high level of freedom in how they coordinated their work, thus facilitating a greater level of self-organization. One participant explained it this way.

> "There's nothing really in place that coordinates our work beyond the informal chats about what we need. What's driving task selection for me is talking to customers and seeing, what common geometry do I need to build. There's no *gaffer* [boss] coming along to say '*you got to do this.*'" (R2)

Furthermore, within teams developers work on different parts of the software, thereby developing a certain level of speciality (which, similar to Neopost, facilitates the emergence of "trusted lieutenants"). It is generally known within teams "who's doing what." One participant highlighted the importance of a common vision and direction for a project where different people are "trying to pull it in different directions." This suggests there is room for a benevolent dictator who coordinates and integrates contributions by others.

5.3.2. Transparency to Open Up the Organization. Philips uses tools and infrastructure provided by CollabNet to support an (internal) online community and facilitate a

transparent process. This includes a mailing list on which people can post questions and comments. Developers and architects follow these lists and respond to issues that they are familiar with. This does not mean that all communication is online; architects still have face-to-face meetings to discuss architectural roadmaps. Chief architects also provide training sessions targeting product groups. However, the day-to-day knowledge sharing regarding issues and usage of the shared asset is greatly improved by the online communication as one participant explained.

> "Our Inner Source initiative really helps [for knowledge sharing] because we see that the communication between developers is much more interactive. Since we've changed from a central team that delivers a binary platform to the new Inner Source model in which everybody could see the source code and also contribute, we saw the community growing. People started to inform each other about dos and don'ts about the design. And people found out much quicker whether others were using the platform correctly. That community became much more lively when we adopted the Inner Source model." (P3)

Neopost's internal LAN provides open access to all project artifacts to all employees. Source code is managed by a version control system, and other infrastructure includes an issue tracker and a wiki installation that is used for sharing knowledge and documentation. Thus, most of the infrastructure to facilitate an online community is present. One potential issue is that of online communication. There are a few issues that should be considered. First, the number of engineers in the R&D department is small enough that people often know whom to talk to, so there is no need to 'broadcast' a question. On the other hand, the visited location also collaborates with teams in France, the United Kingdom, and Vietnam. Such collaboration could potentially be fertile contexts for online communities. One factor that must be considered here is that of culture. There are large cultural differences between the teams in Vietnam and the Netherlands, and even between the teams in France and the Netherlands. The culture in the Dutch branch is rather open and liberal, whereas the branches in France and Vietnam in particular are more hierarchical. Such cultural differences may cause a reluctance among developers to post questions.

At Rolls-Royce, universal access and transparency of the development process is a challenge that should be addressed prior to adopting Inner Source. Infrastructure to support transparency and universal access varies widely throughout the organization. Some software development areas, such as embedded software development, use highly standardized infrastructure based on commercial solutions for file-sharing, whereas other areas have little standardization, or use non-version controlled solutions that may inhibit universal access. One of the attractions of Inner Source for Rolls-Royce is the ability to reduce cost by adopting a common solution "stack."

We also found that corporate firewalls are a major obstacle to sharing artifacts across the globe. Some teams use secure collaboration platforms that are based on standard commercial solutions, but these only offer 'static' (or read-only) sharing capabilities. One project lead explained it as follows:

> "I've been pushing for something like open source for a long time. I think, having the ability to communicate and exchange ideas is missing. People are coming to the table with lots of different skills, some kind of Inner Source type of mechanism would really help with that." (R4)

5.3.3. Management Support and Motivation to Involve People. Philips Management strongly supports the Inner Source initiative. In the late nineties, management had started initiatives to improve reuse, which resulted in a software product line organization [van der Linden et al. 2007]. The organization has since acquired many companies,

which have become product groups, all of which were encouraged to use the common platform. The Inner Source initiative helped to address a number of issues related to knowledge sharing, improving productivity. Furthermore, given the one-to-many relationship between the platform team and the product groups, the former could prove to be a bottleneck for the latter, an issue that could be alleviated by introducing Open Source-development principles. However, while management support is imperative, there is no central authority that prescribes to a product group what it should do, as a director of the technology office explained.

> "We don't have the right to tell them, *'tomorrow you'll do this.'* That's not how it works; it's more like building a case, discussing what would be wise." (P2)

Therefore, in order for any Inner Source initiative to succeed, product groups had to buy in. One co-development coordinator explained this as follows:

> "The driver to do this [collaboration] is to ensure that what's implemented works for us. It's not to help others with that new functionality, it's primarily for us. But, once it's finished, then you'd also discuss this beforehand with the core team that the component can go back into the platform." (P8)

Support at Neopost was mostly at the middle management level. Interest in Inner Source at Neopost is mostly grounded in the organization's goal to increase software reuse. A number of years ago, a small development group was set up to (a) develop a common reference architecture, and (b) develop common, reusable components within that architecture to be used by all future projects. In effect, this group resembled a core team that would manage a set of shared assets. However, due to the need for achieving a quick time-to-market, at some point developers were needed in other projects, greatly reducing the team's capacity to continue its goal. Though management did support the concept of a component group, resources were lacking to sustain this initiative.

> "In general we can only develop within projects, so to say. That means that you typically develop project-specific components. [...] That's mostly a matter of budget; there is no money." (N4)

Developers also felt that resources in terms of time were limited to study other people's contributions or to answer questions, as explained by one developer, "it's not like we're twiddling thumbs all the time, so you do need to have the time for it."

We found a similar result at Rolls-Royce, where some departments had limited capacity in terms of staff and had to work on software with only a few developers. As a result, there may be little scope for software development where it is not a core function. In fact, software development was not always recognized and considered to be an important part of engine design. More recently, this situation changed and software is now seen as playing an important role.

We also found that at a departmental level, there is much local support "for everything we do," as one participant phrased it. This is mainly due to a possibly dramatic positive impact on the business. However, in the end, it is a matter of priorities as set by higher-level management, who may not appreciate the "how" (i.e., Inner Source) as much as the "what" (delivering high quality products). One participant explained it as follows:

> "It's a priority thing, not a principle thing. Generally, management will say, yes this is good stuff, do it, but do this other stuff as well (*laughter*)." (R5)

6. CONCLUSION AND FUTURE WORK

There have been several reports of organizations adopting open-source development practices for their internal software development, what we term "Inner Source" in this article. Despite an increasing interest, there has been little attention to the question as to what factors contribute to an Inner Source initiative. Without this, organizations may not have a clear understanding of what Inner Source is and how it works. To address this practical need, in this article, we have presented a framework of factors that support Inner Source. The framework has been rigorously derived from the scientific literature, supported by a number of other sources such as book chapters. Therefore, we argue that the framework provides a rich and sound starting point for supporting Inner Source adoption.

In order to demonstrate and apply the framework in practice, we conducted three industry case studies at organizations at varying points in the adoption of Inner Source: Philips Healthcare, Neopost Technologies, and Rolls-Royce. The application of the framework provides an in-depth account of how the factors, as identified in the literature, "come to life" when applied to real-world contexts. We discuss our findings (presented in Section 5) in Section 6.1. We then present an evaluation of our work as well as a discussion of limitations of our study in Section 6.2. Section 6.3 presents an outlook on future work.

6.1. Discussion

We can make a number of observations based on the findings presented in the previous section. Our discussion is organized using the three categories of the framework: Software Product, Practices and Tools, and Organization and Community.

6.1.1. Software Product. The first category relates to the software that is to be managed as an Inner Source project. In terms of identifying a 'seed' product, this was fairly straightforward in the two case organizations that have not fully adopted Inner Source (Neopost, Rolls-Royce). Interviewees immediately suggested potential products (or components) that could be further developed in an Inner Source project. The need for a shared asset to be reusable by a variety of users also seemed intuitive. In both Neopost and Rolls-Royce, participants indicated how "this could work" for a specific software component.

One issue that emerged from our findings is that there is a tension between attracting a sufficient number of stakeholders to a new Inner Source project on the one hand, and trying to balance a large number of stakeholders on the other hand. Maintaining this balance will have a direct impact on how an Inner Source project is managed. For instance, the shared asset at Philips is used in over 300 products by more than ten product groups. Due to this large scale, there must be a tighter control by the core team to accommodate all stakeholders needs. This is clearly the case at Philips, whose Inner Source model focuses more on the engineering practices rather than the governance practices found in open source.

While the participants in all three cases had clear ideas on the choice of software that can serve as an Inner Source project, the nature and structure of the software can vary widely. For instance, the platform at Philips evolved from a set of components implementing an industry standard to an integrated base product that represents a significant part of all products. While Neopost's HAL serves a similar purpose in that it offers a set of common functionality, its size and structure is much smaller and simpler and does not have far-reaching consequences for the client application's architecture. Also, Philips' platform stands in stark contrast to the GAF and TIF components developed at Rolls-Royce, which are essentially tools targeting design engineers rather than a component that becomes part of a product.

6.1.2. Practices and Tools. In Section 4.2, we argued that there must be room for using 'bazaar-style' development and quality assurance practices for an Inner Source project to thrive. However, from our case studies it becomes clear that this may mean different things to different organizations. For instance, the Philips case illustrates how open-source practices have been tailored to fit the specific corporate context (i.e., co-development projects). As briefly suggested in Section 6.1.1, the nature of the Inner Source product (in terms of scale and business value) may affect what practices can be adopted. At Neopost, software development is very much driven by hardware development and needs to keep up due to the strong interdependency of software and hardware. At Rolls-Royce, there was a clear need for long-term compatibility and availability, thus limiting the potential for 'reinvention' of implementations. In all cases, we conclude that an organization's specific business needs and context affect which open-source practices can be adopted, and how they are adopted.

In terms of using a common set of development tools, different organizations may have varying challenges. Given that most development at Neopost is embedded software, the range of software technologies used is rather homogeneous (i.e., C and C++). At Rolls-Royce the variety was much larger, ranging from Python scripts to large legacy code bases written in COBOL, thus requiring tool support for each of the used technologies. Philips have made a clear commitment to their Inner Source initiative and ensured the required tools are available. Overall, the freedom to introduce new development tools was not a trivial issue in the organizations that have not yet adopted Inner Source.

6.1.3. Organization and Community. Adopting Inner Source is, after all, an organizational change, although the extent and degree of change will vary widely per organization. In all three cases, we found that the nature of Inner Source adoption will depend on existing organizational structures and how amenable those structures are. We observed in both Neopost and Rolls-Royce that certain developers take ownership of a certain software component, which clearly suggests there is room for trusted lieutenants to emerge. At the same time, it is very important that a core team provides overall management of a shared asset: within Philips, the core team plays a pivotal role in planning and participating in co-development projects. We also argue that transparency is important (Section 4.3.2). Our study shows that this was a very important point within Philips as it made their internal community "much more active" (Section 5.3.2). However, introducing this transparency can be challenging for a number of reasons. First, when developers are co-located (as is the case for Neopost), there may be little incentive to "take a conversation online." Second, cultural differences between different divisions of an organization may limit participation. For that reason, education, training, and advertising an Inner Source initiative to all involved developers is very important. Another issue is that of the necessary infrastructure required for sharing development artifacts. Large organizations with various branches tend to use different infrastructure for storing and sharing artifacts, and it can be quite challenging to adopt an organization-wide standard for this.

Finally, while management support is an absolute requirement for Inner Source adoption, there is a strong parallel to open-source projects, in that the developers involved must be motivated and have sufficient incentives to adopt this model of working. For an organization's management, there may be various drivers, such as increasing software reuse and reducing cost. However, while Inner Source can offer a number of benefits, it is not a panacea [Stol et al. 2011]. The parties involved must be supported with budgetary and other resources. The literature suggests that identifying champions and giving them the space to advocate an Inner Source initiative is essential [Dinkelacker et al. 2002; Gurbani et al. 2006]. Our contacts at the case study organizations are such champions; however, it is obviously a challenge to convince top-level

management to commit to an Inner Source initiative. In both Neopost and Rolls-Royce, participants of our study suggested that while Inner Source initiatives are welcome, in practice a lack of budget or time inhibits pursuing them fully. At Philips, the Inner Source initiative has been widely advocated throughout the organization, and top-level management have made a clear commitment to Inner Source.

6.2. Evaluation

In any research effort it is important to address the question of validity. The standard validation criteria include items such as *internal validity*, *external validity*, *reliability*, and *objectivity* [Guba 1981]. However, many authors have pointed out that these criteria are better suited to evaluating quantitative research where constructs are measured explicitly and traits, such as convergent and discriminant validity, are then assessed [Creswell 2007, p. 203]. Such measurements are not typically feasible or practical in qualitative research studies [Leininger 1994]. As a consequence, researchers have argued that *trustworthiness* is an appropriate way to judge the validity of qualitative research [Angen 2000; Creswell 2007; Guba 1981; Lincoln and Guba 1985]. In operationalizing the trustworthiness criterion, Lincoln and Guba [1985] identified four validation criteria for qualitative research as alternatives to the standard criteria previously mentioned that are typically adopted in quantitative research studies. These are *credibility* (paralleling *internal validity*), *transferability* (*external validity*), *dependability* (*reliability*), and *confirmability* (*objectivity*). These are also recommended by Cruzes and Dybå [2011] as appropriate for assessing the trustworthiness of a research study in a software engineering context. Previously, Sfetsos et al. [2006] used these in a qualitative study of Extreme Programming. The following evaluation discusses how each of these aspects of our study's trustworthiness was addressed as well as some limitations of our study. Some tactics (e.g., triangulation) help to address several of the aspects below; they are discussed for each aspect separately.

6.2.1. Credibility. A study's credibility refers to the extent to which we can place confidence in the findings, or that the findings are plausible [Guba 1981]. One important issue regarding our study's credibility is the extent to which we have correctly identified a set of factors that are relevant to Inner Source adoption. In other words, how does our research design establish the credibility of these nine factors? We sought to establish credibility of our findings through a number of approaches, which we discuss next.

Peer debriefing refers to discussing the research with others so as to expose it to feedback by colleagues or practitioners, who can play devil's advocate [Creswell and Miller 2000]. We discussed the work at various occasions with colleagues in this area, during which we received feedback related to the factors of the framework. In one particular instance, we discussed each of the framework's elements in great detail with the initiators of a highly successful Inner Source project. During this process, we found that all factors resonated with the initiators' experiences, but some clarifications as to how to interpret and phrase the factors were highlighted as well. Furthermore, we engaged in a process that has been termed 'venting' [Goetz and LeCompte 1984], during which we discussed the framework through informal conversation with a number of practitioners interested in this topic at several occasions.

Triangulation is a technique that seeks convergence among multiple and different sources of information; these can be different types of data, different investigators, or different research methods. Whereas our framework was based on a synthesis of the extant literature (a single data source), the empirical phase of our study involved three case studies, thereby demonstrating its usability in three different contexts. In all three contexts the framework solicited relevant input for each of the framework's nine factors.

6.2.2. Transferability. Transferability refers to the extent to which a study's findings can be applied in other settings. In other words, this refers to the question: Is our framework applicable in a variety of contexts? Guba [1981] presented some tactics to assess transferability, which we discuss next.

Given that Inner Source is a relatively new phenomenon, we had to identify organizations who would be active in Inner Source implementation. Thus, our sample of organizations has particular characteristics. Purposive sampling refers to a researcher's judgement as a principle of selection of participants or cases [Robson 2002]. That is, we selected three cases that we believe were suitable to apply the framework. Philips was selected as a case as they have an established track record of Inner Source adoption, and therefore is a suitable choice to demonstrate that all factors of our framework are relevant from an Inner Source perspective. The cases of Neopost and Rolls-Royce are both also relevant for two reasons. First, both organizations were actively interested in adopting Inner Source, and thus represent the target audience for the framework. Second, both organizations have already started a number of initiatives that resemble elements of Inner Source. For instance, some Neopost developers already used the principle of 'self-selecting' a defect to work on, for which a patch was sent to the software package's maintainer. Also, other developers use peer-review as a practice, and Neopost's component group initiative resembled a core team.

In this study, we focused on large multi-national organizations in the private sector with very significant software development functions. Although we deem our choice of cases to be relevant, the choice was also affected by the organization's willingness to allow us to conduct the studies. One limitation of our study is that generalizations cannot be drawn from these three cases, and other contexts may exist in which our framework is less suitable. If we had studied different organizations with different profiles, for example, small organizations or public sector companies, the reaction to the framework and issues identified could be quite different.

Furthermore, the framework, while useful as a probing mechanism, was operationalized quite differently across the three case study organizations, even though they were quite homogeneous. Thus, it would be interesting to see the differences in the framework if applied to a heterogeneous sample of organizations.

"Thick" descriptions permit comparison of contexts that are considered relevant. As Creswell and Miller [2000] pointed out, "the purpose of a thick description is that it creates verisimilitude, statements that produce for the readers the feeling that they have experienced, or could experience, the events being described in a study." In order to facilitate an understanding of the three cases, we have attempted to provide detailed descriptions of the contexts at the three organizations. Nevertheless, it is challenging to convey all details of a particular organization's context to its full extent due to space limitations. One potential limitation of our study, therefore, is that this feeling of experience referred to by Creswell and Miller is only achieved to some extent.

6.2.3. Dependability. The dependability of a study refers to the 'stability' of data, or the extent to which findings are reliable and any variance in those findings can be 'tracked' and explained. We used a number of tactics, described next.

In the empirical phase, for each of the three case studies, we drew from several sources (see Table II), another form of triangulation. For instance, the case at Philips was based on a number of existing reports on their Inner Source initiative, complemented with an extensive set of interviews, as well as informal discussions with an informed expert. The other two cases also drew from several sources.

Throughout the research process we established an audit trail, in both the derivation of the framework and the empirical component of the study represented by the three case studies. The derivation of the framework was done by capturing all relevant text

fragments of the literature in a spreadsheet and labeling these fragments with codes. Sources were listed for all fragments so as to enable us to go back to the original context from which a fragment was taken. Steps in the audit trail of the empirical work include recording of all interviews, transcription of the interviews, and writing and exchanging memos amongst the authors regarding both findings and research design. The coding process and memos exchanged was performed over an extensive time period (approx. one year).

6.2.4. Confirmability. Confirmability refers to the neutrality aspect of a study's trustworthiness, and as such is concerned with a researcher's predilections that may bias a study's findings. One such concern in qualitative studies is that of "multiple realities," referring to the subjectivity of the researcher's understanding [Swanson and Chapman 1994, p. 75]. Furthermore, this may also be affected by the attitudes of the participants involved in a study.

Member checking is one tactic we employed in our research design, which refers to checking interpretations with members of groups from which data were solicited [Guba 1981; Creswell and Miller 2000]. We sent a draft of an earlier version of this article to key informants at all three case study organizations for review and feedback.

Triangulation is a tactic previously discussed in relation to credibility and dependability, and can also help to establish confirmability. For instance, in the derivation of the framework, all factors were based on at least two references to the literature. Therefore, none of the nine factors in the framework is based on a single source but always 'confirmed' by a second source.

Notwithstanding the preceding, in our study, interviewees were already interested in Inner Source adoption, and in a sense were already converts. Thus, the extent to which they would view any obstacles as capable of being overcome, and their enthusiasm to participate could skew their responses to our framework, in a manner that would differ from more skeptical participants. In much the same way, the authors are also advocates of Inner Source, and thus may be similarly blind to issues.

6.3. Future Work

This article contributes to the Inner Source literature by deriving a set of important factors in the adoption of Inner Source. The concepts identified in the framework can be used in future studies of Inner Source. We identified a number of directions to guide future research, that are outlined next.

Phased Adoption Model. As mentioned earlier, open-source projects typically have a Cathedral phase after which they transition to a Bazaar phase. In this article, we have argued that this would also be necessary to initiate an Inner Source project. Gurbani et al. [2006] described how the Inner Source project at Alcatel-Lucent evolved through several stages. Wesselius [2008] described the evolution of the business model at Philips Healthcare, from a traditional "taxing" model to a model that was more amenable to an Inner Source approach. This suggests that the adoption of Inner Source follows a number of phases to perform the cultural change within an organization. Identifying these phases would be useful to better understand the steps to take to start an Inner Source initiative, which in turn can provide concrete guidance to organizations.

Tailoring Dimensions. Each case of Inner Source is different and tailored to the context of an organization. Whereas established methods, such as the agile methods Scrum and XP, are well defined and their amenability to tailoring has been explored [Conboy and Fitzgerald 2010], such tailoring is very different for Inner Source, which has no documented reference model. Gurbani et al. [2010] have observed two general models (infrastructure-based and project-specific), a further taxonomy of Inner Source adoption is needed to better understand along which dimensions such initiatives can vary.

The factors identified in this article provide a useful starting point for this. This in turn will also help to identify common Inner Source adoption patterns, which would provide useful and practical guidance to other organizations that wish to adopt Inner Source. One dimension that is of particular interest is governance. The self-organizing nature found in open-source projects may not be suitable for most commercial organizations that have product release schedules. The initial work by Gurbani et al. [2010] in which they identified a number of key roles in the core team provides a useful starting point for this. We expect other governance patterns to emerge over time.

Practices and Patterns in Inner Source. One concept that emerged from the Philips case study is that of co-development. This is one form of translation, or '*re-negotiation*' as Lindman et al. [2013] termed it, of open-source practices to a corporate environment. For instance, where in open-source development it is quite normal to "check out" the code, add a new feature (or correct a defect) and send a patch, the case of Philips showed how this is managed through the concept of co-development. Nevertheless, such practices (relating to the development practices factor of the framework) are derived from the open-source philosophy and different from conventional approaches, and thus still relevant to Inner Source. Further identification of such translations that occur in Inner Source initiatives is an important step so they can be catalogued and made available to other organizations that are interested in adopting Inner Source. Whereas the current study has focused on factors that support Inner Source, we expect that future work will provide detailed insights of suitable practices as well as how to achieve many of the benefits associated with open-source development.

Emerging Practices and Tools. New developments, such as the emergence of new tools have an impact on how developers achieve their tasks and communicate. For instance, in recent years, the use of microblogging through sites such as Twitter and Yammer are becoming increasingly popular, both in closed-source and open-source environments [Wang et al. 2014]. Microblogging can help in building a community as well as in communication. One particularly interesting trend is that each artifact can have a link, through which it is immediately accessible to others. For instance, new contributions (commits), issues recorded in an issue tracker, emails (on public mailing lists), and bulletin boards all can have a link, which greatly helps in contributing to a community's transparency. How Inner Source organizations can benefit from this and how such trends can help to establish Inner Source communities are open questions.

APPENDIX. RELEVANT SOURCES ON INNER SOURCE TO DATE

Table III. Relevant Sources on Inner Source Literature to Date

Year	Authors	Title
2001	Dinkelacker and Garg	Corporate Source: Applying Open Source Concepts to a Corporate Environment
2002	Dinkelacker et al.[a]	Progressive Open Source
2002	Sharma et al.	A framework for creating hybrid-open source software communities
2002	Robbins	Adopting OSS Methods by Adopting OSS Tools
2005	Neus and Scherf	Opening minds: Cultural change with the introduction of open-source collaboration methods
2005	Robbins[b]	Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools
2005	Goldman and Gabriel	What is Open Source? (book chapter in 'Innovation Happens Elsewhere')
2005	Gurbani et al.	A Case Study of Open Source Tools and Practices in a Commercial Setting
2006	Gurbani et al.[c]	A Case Study of a Corporate Open Source Development Model

Table III. Continued

Year	Authors	Title
2007	Martin and Hoffman	An Open Source Approach to Developing Software in a Small Organization
2007	Pulkkinen et al.	Support for Knowledge and Innovations in Software Development - Community within Company: Inner Source Environment
2007	Gaughan et al.	An Examination of the Use of Inner Source in Multinational Corporations
2007	Smith and Garber-Brown	Traveling the Open Road: Using Open Source Practices to Transform Our Organization
2007	Melian	Progressive Open Source: The Construction of a Development Project at Hewlett-Packard
2008	Wesselius	The Bazaar inside the Cathedral: Business Models for Internal Markets
2008	Lindman et al.	Applying Open Source Development Practices Inside a Company
2008	Melian and Mähring	Lost and Gained in Translation: Adoption of Open Source Software Development at Hewlett-Packard
2008	Oor and Krikhaar	Balancing Technology, Organization, and Process in Inner Source
2009	Riehle et al.	Open Collaboration within Corporations Using Software Forges
2009	van der Linden	Applying Open Source Software Principles in Product Lines
2009	Lindman et al.	OSS as a way to sell organizational transformation
2009	Stellman et al.	Inner Source (book chapter in 'Beautiful Teams')
2009	Gaughan et al.	An Examination of the use of Open Source Software Processes as a Global Software Development Solution for Commercial Software Engineering
2010	Lindman et al.	Open Source Technology Changes Intra-Organisational Systems Development–A Tale of Two Companies
2010	Vitharana et al.	Impact of Internal Open Source Development on Reuse: Participatory Reuse in Action
2010	Gurbani et al.	Managing a Corporate Open Source Software Asset
2011	Morgan et al.	Exploring Inner Source as a Form of Intra-Organisational Open Innovation
2011	Torkar et al.	Adopting Free/Libre/Open Source Software Practices, Techniques and Methods for Industrial Use
2011	Martin and Lippold	Forge.mil: A Case Study for Utilizing Open Source Methodologies Inside of government
2011	Stol et al.	A Comparative Study of Challenges in Integrating Open Source Software and Inner Source Software
2011	Stol	Supporting Product Development with Software from the Bazaar
2013	Lindman et al.	Open Source Technology in Intra-Organisational Software Develompent–Private Markets or Local Libraries

[a]Revision of Dinkelacker and Garg [2001].
[b]Revision of Robbins [2002].
[c]Revision of Gurbani et al. [2005].

ACKNOWLEDGMENTS

We are grateful to all participants of our study at the three case study organizations for their time and input. We wish to thank Lorraine Morgan for fruitful discussions on the framework presented in this article. We also thank the anonymous reviewers whose constructive comments have led to significant improvements.

REFERENCES

Pär J. Ågerfalk and Brian Fitzgerald. 2008. Outsourcing to an unknown workforce: Exploring opensourcing as a global sourcing strategy. *MIS Quart.* 32, 2, 385–409.

Allan Alter. 2006. Can IT use open source methods to write internal code? CIO Insight, `http://www.cioinsight.com/c/a/Expert-Voices/Can-IT-Use-Open-Source-Methods-To-Write-Internal-Code/`.

Maureen Jane Angen. 2000. Evaluating interpretive inquiry: Reviewing the validity debate and opening the dialogue. *Qualitat. Health Res.* 10, 3 , 378–395.

Matt Asay. 2007. Microsoft Office experiments with open source (development). O'Reilly ONLamp. http://www.oreillynet.com/onlamp/blog/2007/02/microsoft_office_experiments_w.html.

Jai Asundi. 2001. Software engineering lessons from open source projects. In *Proceedings of the 1st Workshop on Open Source Software Engineering*. Joseph Feller, Brian Fitzgerald, and André van der Hoek (Eds.).

Jai Asundi and Rajiv Jayvant. 2007. Patch review processes in open source software development communities: A comparative case study. In *Proceedings of the 40th Annual Hawaii International Conference on Systems Sciences (HICSS)*.

Larry Augustin, Dan Bressler, and Guy Smith. 2002. Accelerating software development through collaboration. In *Proceedings of the 24th International Conference on Software Engineering*. 559–563.

Carliss Y. Baldwin and Kim B. Clark. 2006. The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Manag. Sci.* 52, 7, 1116–1127.

Liz Barnett. 2004. Applying open source processes in corporate development organizations. Forrester Report.

Betanews. 2005. IBM turns to open source development. http://betanews.com/2005/06/13/ibm-turns-to-open-source-development/%.

Andrea Bonaccorsi and Cristina Rossi. 2003. Why open source software can succeed. *Res. Policy* 32, 7, 1243–1258.

Frederick P. Brooks. 1995. *The Mythical Man-Month*. Addison Wesley Longman, Inc.

Peter G. Capek, Steven P. Frank, Steve Gerdt, and David Shields. 2005. A history of IBM's open-source involvement and strategy. *IBM Syst. J.* 44, 2, 249–257.

Andrea Capiluppi, Klaas-Jan Stol, and Cornelia Boldyreff. 2012. Exploring the role of commercial stakeholders in open source software evolution. In *Open Source Systems: Long-Term Sustainability*, Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi (Eds.), Springer, 178–200.

Mike Cohn. 2009. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley.

Kieran Conboy and Brian Fitzgerald. 2010. Method and developer characteristics for effective agile method tailoring: A study of XP expert opinion. *ACM Trans. Softw. Eng. Methodol.* 20, 1 (2010).

John W. Creswell. 2007. *Qualitative Inquiry & Research Design* 2nd Ed. SAGE Publications.

John W. Creswell and Dana L. Miller. 2000. Determining validity in qualitative inquiry. *Theory Pract.* 39, 3, 124–130.

Kevin Crowston, Qing Li, Kangning Wei, U. Yeliz Eseryel, and James Howison. 2007. Self-organization of teams for free/libre open source software development. *Inf. Softw. Technol.* 49, 6, 564–575.

Daniela S. Cruzes and Tore Dybå. 2011. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement*. 275–284.

Jamie Dinkelacker and Pankaj K. Garg. 2001. Corporate source: Applying open source concepts to a corporate environment. In *Proceedings of the 1st Workshop on Open Source Software Engineering*. Joseph Feller, Brian Fitzgerald, and André van der Hoek (Eds.).

Jamie Dinkelacker, Pankaj K. Garg, Rob Miller, and Dean Nelson. 2002. Progressive open source. In *Proceedings of the 24th International Conference on Software Engineering*. 177–184.

Hakan Erdogmus. 2009. A process that is not. *IEEE Softw.* 26, 6, 4–7.

Justin R. Erenkrantz. 2003. Release management within open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*. Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani (Eds.).

Justin R. Erenkrantz and Richard N. Taylor. 2003. Supporting distributed and decentralized projects: Drawing lessons from the open source community. In *Proceedings of the 1st Workshop on Open Source in an Industrial Context*. Marc Sihling (Ed.).

FCW. 2009. DOD launches site to develop open-source software. Federal Computer Week. http://fcw.com/articles/2009/01/30/dod-launches-site-to-develop-open-source-software.aspx.

Joseph Feller and Brian Fitzgerald. 2002. *Understanding Open Source Software Development*. Pearson Education Ltd.

Brian Fitzgerald. 2006. The transformation of open source software. *MIS Quart.* 30, 3, 587–598.

Brian Fitzgerald. 2011. Open source software: Lessons from and for software engineering. *IEEE Computer* 44, 10, 25–30.

Karl Fogel. 2005. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media.

Martin Fowler and Jim Highsmith. 2001. The agile manifesto. *Softw. Develop.* 9, 28–32.

Cristina Gacek and Budi Arief. 2004. The many meanings of open source. *IEEE Softw.* 21, 1, 34–40.

Gary Gaughan, Brian Fitzgerald, Lorraine Morgan, and Maha Shaikh. 2007. An examination of the use of inner source in multinational corporations. In *Proceedings of the 1st OPAALS Conference*.

Gary Gaughan, Brian Fitzgerald, and Maha Shaikh. 2009. An examination of the use of open source software processes as a global software development solution for commercial software engineering. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 20–27.

Judith Goetz and Margeret D. LeCompte. 1984. *Ethnography and Qualitative Design in Educational Research*. Academic Press.

Ron Goldman and Richard P. Gabriel. 2005. *Innovation Happens Elsewhere*. Morgan Kaufmann, Chapter 3: What is Open Source?

Egon G. Guba. 1981. Criteria for assessing the trustworthiness of naturalistic inquiries. *Edu. Commun. Technol.* 29, 2, 75–91.

Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb. 2005. A case study of open source tools and practices in a commercial setting. In *Proceedings of the 5th Workshop on Open Source Software Engineering*, Joseph Feller, Brian Fitzgerald, Scott A. Hissam, Karim Lakhani, and Walt Scacchi (Eds.).

Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb. 2006. A case study of a corporate open source development model. In *Proceedings of the 28th International Conference on Software Engineering*. 472–481.

Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb. 2010. Managing a corporate open source software asset. *Commun. ACM* 53, 2, 155–159.

T. J. Halloran and William L. Scherlis. 2002. High quality and open source software practices. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*. Joseph Feller, Brian Fitzgerald, Frank Hecker, Scott A. Hissam, Karim Lakhani, and André van der Hoek (Eds.).

James D. Herbsleb and Rebecca E. Grinter. 1999. Architectures, coordination, and distance: Conway's Law and beyond. *IEEE Softw.* 16, 5, 63–70.

Madeleine Leininger. 1994. Evaluation criteria and critique of qualitative research studies. In *Critical Issues in Qualitative Research Methods*, Janice M. Morse (Ed.), SAGE Publications.

Yvonne S. Lincoln and Egon G. Guba. 1985. *Naturalistic Inquiry*. SAGE Publications.

Juho Lindman, Mikko Riepula, Matti Rossi, and Pentti Marttiin. 2013. Open source technology in intra-organisational software development–private markets or local libraries. In *Managing Open Innovation Technologies*, Jenny Ericsson Lundstrom, Mikael Wiberg, Stefan Hrastinski, Mats Edenius, and Pär J. Ågerfalk (Eds.), Springer.

Juho Lindman, Matti Rossi, and Pentti Marttiin. 2008. Applying open source development practices inside a company. In *Open Source Development, Communities and Quality*, Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi (Eds.), Springer, 381–387.

Juho Lindman, Matti Rossi, and Pentti Marttiin. 2009. OSS as a way to sell organizational transformation. In *Proceedings of the Information Systems Research Seminar in Scandinavia*.

Juho Lindman, Matti Rossi, and Pentti Marttiin. 2010. Open source technology changes intra-organizational systems development–A tale of two companies. In *Proceedings of the 18th European Conference on Information Systems*.

Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. 2006. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Manage. Sci.* 52, 7, 1015–1030.

Guy Martin and Aaron Lippold. 2011. Forge.mil: A case study for utilizing open source methodologies inside of government. In *Open Source Systems: Grounding Research*, Scott Hissam, Barbara Russo, Manuel Gomes de Mendonça Neto, and Fabio Kon (Eds.), Springer, 334–337.

Ken Martin and Bill Hoffman. 2007. An open source approach to developing software in a small organization. *IEEE Softw.* 24, 1, 46–53.

Steven C. McConnell. 1999. Open-source methodology: Ready for prime time? *IEEE Softw.* 16, 4, 6–8.

Catharina Melian. 2007. Progressive open source. Ph.D. Dissertation. Stockholm School of Economics.

Catharina Melian and Magnus Mähring. 2008. Lost and gained in translation: Adoption of open source software development at Hewlett-Packard. In *Open Source Development, Communities and Quality*, Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi (Eds.), Springer.

Janne Merilinna and Mari Matinlassi. 2006. State of the art and practice of open source component integration. In *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 170–177.

Martin Michlmayr and Brian Fitzgerald. 2012. Time-based release management in free and open source (FOSS) projects. *Int. J. Open Source Softw. Process.* 4, 1, 1–19.

Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.* 11, 3, 309–346.

Audris Mockus and James D. Herbsleb. 2002. Why not improve coordination in distributed software development by stealing good ideas from open source?. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*. Joseph Feller, Brian Fitzgerald, Frank Hecker, Scott A. Hissam, Karim R. Lakhani, and André van der Hoek (Eds.), 19–25.

Lorraine Morgan, Joseph Feller, and Patrick Finnegan. 2011. Exploring inner source as a form of intraorganisational open innovation. In *Proceedings of the 19th European Conference on Information Systems*.

Andreas Neus and Philipp Scherf. 2005. Opening minds: cultural change with the introduction of open-source collaboration methods. *IBM Syst. J.* 44, 2, 215–225.

Todd Ogasawara. 2008. Microsoft CodeBox: Lessons from the open source community. `http://www.oreillynet.com/onlamp/blog/2008/05/microsoft_codebox_lessons_from.html`.

Patrick Oor and René Krikhaar. 2008. Balancing technology, organization, and process in inner source: Bringing inner source to the TOP. In *Dagstuhl Seminar Proceedings*. `http://drops.dagstuhl.de/opus/volltexte/2008/1548`.

Tim O'Reilly. 1999. Lessons from open source software development. *Commun. ACM* 42, 4, 33–37.

Tim O'Reilly. 2000. O'Reilly network: Ask Tim: Is licensing what makes open source succeed? `http://www.linuxtoday.com/infrastructure/20001216013060PCYSW`.

David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12, 1053–1058.

Mirja Pulkkinen, Oleksiy Mazhelis, Pentti Marttiin, and Jouni Meriluoto. 2007. Support for knowledge and innovations in software development—Community within company: Inner source environment. In *Proceedings of the 3rd International Conference on Web Information Systems and Technologies*. 141–150.

Eric S. Raymond. 2001. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media.

Dirk Riehle, J. Ellenberger, T. Menahem, B. Mikhailovski, Y. Natchetoi, B. Naveh, and T. Odenwald. 2009. Open collaboration within corporations using software forges. *IEEE Softw.* 26, 2, 52–58.

Bram Riemens and Kees van Zon. 2006. PFSPD short history. `http://pfspd.sourceforge.net/history.html`.

Peter C. Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel M. German. 2012. Contemporary peer review in action: Lessons from open source development. *IEEE Softw.* 29, 56–61. Issue 6.

Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. 2008. Open source software peer review practices: A case study of the Apache Server. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, 541–550.

Jason Robbins. 2002. Adopting OSS methods by adopting OSS tools. In *Proceedings of the 2nd Workshop on Open Source Software Engineering*. Joseph Feller, Brian Fitzgerald, Frank Hecker, Scott A. Hissam, Karim R. Lakhani, and André van der Hoek (Eds.).

Jason Robbins. 2005. Adopting open source software engineering (OSSE) practices by adopting OSSE tools. In *Perspectives on Free and Open Source Software*, Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani (Eds.), MIT Press, 245–264.

Colin Robson. 2002. *Real World Research* (2nd Ed.). Blackwell Publishers.

Winston W. Royce. 1987. Managing the development of large software systems. In *Proceedings of the 9th International Conference on Software Engineering*. Originally published in *Proceedings of WESCON'70*. 328–338.

Per Runeson, Martin Höst, A. Rainer, and Björn Regnell. 2012. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, Inc., Hoboken, NJ.

Walt Scacchi. 2004. Free and open source development practices in the game community. *IEEE Softw.* 21, 1, 59–66.

Ken Schwaber and Mike Beedle. 2002. *Agile Software Development with Scrum*. Prentice Hall.

Andrew Schwarz, Manjari Mehta, Norman Johnson, and Wynne W. Chin. 2007. Understanding frameworks and reviews: A commentary to assist us in moving our field forward by analyzing our past. *SIGMIS Data B. Adv. Inf. Syst.* 38, 3, 29–50.

Carolyn B. Seaman. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.* 25, 4, 557–572.

Anthony Senyard and Martin Michlmayr. 2004. How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC)*.

Panagiotis Sfetsos, Lefteris Angelis, and Ioannis Stamelos. 2006. Investigating the extreme programming system–An empirical study. *Empirical Softw. Eng.* 11, 2, 269–301.

Srinarayan Sharma, Vijayan Sugumaran, and Balaji Rajagopalan. 2002. A framework for creating hybrid-open source software communities. *Inf. Syst. J.* 12, 1, 7–25.

Phillip Smith and Chris Garber-Brown. 2007. Traveling the open road: Using open source practices to transform our organization. In *Proceedings of the Agile Conference (AGILE)*. IEEE.

Andrea Stellman, Jennifer Greene, and Auke Jilderda. 2009. Inner source. In *Beautiful Teams: Inspiring and cautionary Tales from Veteran Team Leaders*, Andrew Stellman and Jennifer Greene (Eds.), O'Reilly Media, 103–111.

Klaas-Jan Stol. 2011. Supporting product development with software from the bazaar. Ph.D. Dissertation. University of Limerick.

Klaas-Jan Stol, Muhammad Ali Babar, Paris Avgeriou, and Brian Fitzgerald. 2011. A comparative study of challenges in integrating open source software and inner source software. *Inf. Softw. Technol.* 53, 12, 1319–1336.

E. Burton Swanson. 1976. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*. 492–497.

Janice M. Swanson and Linda Chapman. 1994. Inside the black box: Theoretical and methodological issues in conducting evaluation research using a qualitative approach. In *Critical Issues in Qualitative Research Methods*, Janice M. Morse (Ed.), SAGE Publications.

Darryl K. Taft. 2005. IBM adopts open development internally. `http://www.eweek.com/c/a/Linux-and-Open-Source/IBM-Adopts-Open-Development-Internally/`.

Darryl K. Taft. 2006. IBM to employ open-source development style for tools. `http://www.eweek.com/c/a/Application-Development/IBM-to-Employ-OpenSource-Development-Style-for-Tools/`.

Darryl K. Taft. 2009. Community-source development appeals in tough times. `http://mobile.eweek.com/c/a/Application-Development/CommunitySource-Development-Appeals-in-Tough-Times/`.

Richard Torkar, Pau Minoves, and Janina Garrigós. 2011. Adopting free/libre/open source software practices, techniques and methods for industrial use. *J. Associ. Inf. Syst.* 12, 1, 88–122.

Linus Torvalds. 1999. The Linux edge. In *Open Sources: Voices from the Open Source Revolution*, Chris DiBona, Sam Ockman, and Mark Stone (Eds.), O'Reilly Media.

Linus Torvalds. 2000. Linux Kernel mailing list. `https://lkml.org/lkml/2000/8/25/132`.

Frank van der Linden. 2009. Applying open source software principles in product lines. *UPGRADE* X, 3, 32–40.

Frank van der Linden, Björn Lundell, and Pentti Marttiin. 2009. Commodification of industrial software: A case for open source. *IEEE Softw.* 26, 4, 77–83.

Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.

June M. Verner, J. Sampson, V. Tosic, N.A. Abu Bakar, and Barbara A. Kitchenham. 2009. Guidelines for industrially-based multiple case studies in software engineering. In *Proceedings of the 3rd International Conference on Research Challenges in Information Science*. IEEE, 313–324.

Padmal Vitharana, Julie King, and Helena Shih Chapman. 2010. Impact of internal open source development on reuse: Participatory reuse in action. *J. Manage. Inf. Syst.* 27, 2, 277–304.

Xiaofeng Wang, Ilona Kuzmickaja, Klaas-Jan Stol, Pekka Abrahamsson, and Brian Fitzgerald. 2014. Microblogging in open source software development: The case of Drupal using Twitter. *IEEE Softw.* DOI:10.1109/MS.2013.98.

Jacco Wesselius. 2008. The bazaar inside the cathedral: Business models for internal markets. *IEEE Softw.* 25, 3, 60–66.

Robert K. Yin. 2003. *Case Study Research: Design and Methods* (3rd. Ed.). SAGE Publications, Thousand Oaks, CA.

Received July 2012; revised April, August 2013; accepted September 2013