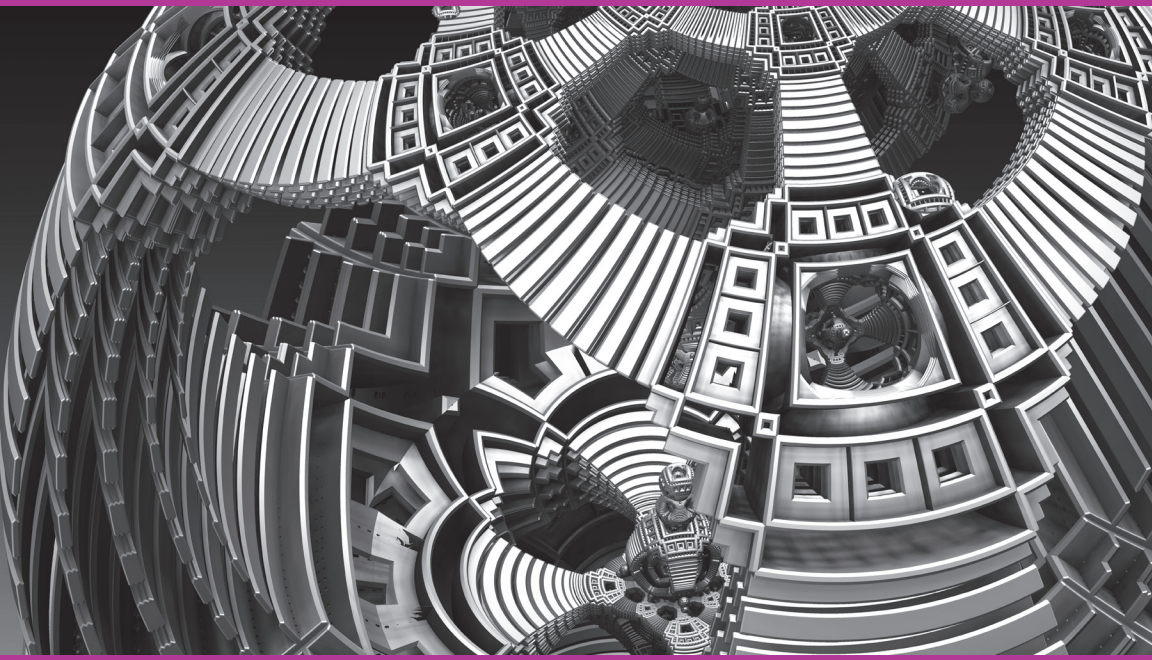
O'REILLY®

Compliments of PayPal

Adopting InnerSource

Principles and Case Studies



Danese Cooper & Klaas-Jan Stol
with foreword by Tim O'Reilly

YOU DON'T HAVE TO GO THERE ALONE

Along the way you will face the forces of time pressures, competitors, budgets, old code, missing docs, obscure architectures, dead languages, intransigence, clueless managers, low morale, entrenched processes, power struggles, acquisitions, mergers, personal fiefdoms, time zones

LEARN	USE	SHARE
Training	Tools	Vision
Studies	History	Culture
Patterns	Templates	Results

BECAUSE NONE OF US HAVE TIME TO MAKE ALL THE MISTAKES

INNERSOURCECOMMONS.ORG

Adopting InnerSource

Principles and Case Studies

Danese Cooper and Klaas-Jan Stol

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Adopting InnerSource
by Danese Cooper and Klaas-Jan Stol

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Justin Billing
Copyeditor: Octal Publishing, Inc. and Charles Roumeliotis
Proofreader: Jasmine Kwityn
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Melanie Yarbrough

July 2018: First Edition

Revision History for the First Edition
2018-06-15: First Release

This work is part of a collaboration between O'Reilly and PayPal. See our *statement of editorial independence*.

978-1-492-04183-2

[LSI]

Table of Contents

Foreword

Tim O'Reilly

I'm told that I coined the term "InnerSource" in 2000, and sure enough there's a blog post to prove it. I don't remember writing it. What I do remember is an earlier conversation, in the summer or fall of 1998, not long after the so-called Open Source Summit in April of that year, to talk to an IBM team that was thinking of embracing this new movement.

They were cautious. How might it affect IBM's business? How would they continue to own, control, and profit from their software? What kind of license might they use to get the benefit of user contribution but still manage and control their creations? The GNU Public License seemed hostile to business; the Berkeley Unix and MIT X Window System licenses were permissive but perhaps gave too much freedom. The just-released Mozilla Public License tried to find a new balance between the needs of a corporate owner and a community of developers. Should they use that or develop their own license?

I was never a big fan of the idea that licenses defined what was most important about free and Open Source software. I'd begun my career in computing in the heady days of the Berkeley Software Distribution of Unix, BSD 4.1, and AT&T's Version 7. I had seen how Unix, based on the original architecture developed by Ken Thompson and Dennis Ritchie at Bell Labs, could attract a wide range of outside contributions from university computer science researchers despite being owned by AT&T. Many of the features that made the system most valuable had been developed at UC Berkeley and other universities. It was the architecture of Unix, not its license, that allowed these contributions to be treated as first-class components of the system. The system defined a protocol, so to speak, for the cooperation between programs, a design by which anyone could bring a new program to the party, and it just worked, as long as it followed that protocol.

I'd then watched as the Internet and its killer application, the World Wide Web, had followed the same model, defining itself not by license but by the rules of interoperability and cooperation. I loved Linux, but it seemed a kind of blindness

in Open Source advocates to focus so much on it. Open Source was the native development methodology of the internet, and the internet was its greatest success story. Network-centric architectures require interoperability and loose coupling. And the Internet allowed distributed development communities to evolve, and shifted power from corporations to individuals.

I'd also been steeped in the culture of the Perl programming language, the idea that "there's more than one way to do it," and the sprawling extensibility of CPAN, the Comprehensive Perl Archive Network, which allowed programmers to share modules they'd built on top of Perl without ever touching the source code of the language interpreter itself.

So I was convinced that much of the magic of Open Source was independent of the license. The things to think about were collaboration, community, and low barriers to entry for those who wanted to share with each other. And so I told Dan Frye, Pierre Fricke, and the other attendees at that IBM meeting, that yes, they could and should release software under Open Source licenses, but if they weren't ready to do so, there was no reason that they couldn't take advantage of these other elements. Given a large enough pool of customers using the same software, I told them, there was no reason they couldn't create a "Gated Open Source Community."

For that matter, given a large enough pool of developers inside a company, there was no reason, I told them, why they couldn't apply many of the principles and practices of Open Source within their own walls. That was what later came to be called InnerSourcing. I defined it at the time as as "helping [companies] to use Open Source development techniques within the corporation, or with a cluster of key customers—even if they aren't ready to take the step all the way to releasing their software as a public Open Source project."

Not long afterwards, I heard the first stories of InnerSourcing in the wild. And as is so often the case, they weren't planned. In 1998 or 1999, two Microsoft developers, Scott Guthrie and Mark Anders, wanted to create a tool that would make it easier to build data-backed websites. They built a project on their own time over the Christmas holidays; other Microsoft developers heard about their work and adopted it, and eventually word reached Bill Gates. When the CEO called the two into his office, they didn't know what to expect. In the end, their project became one of Microsoft's flagship software offerings, ASP.NET. Twenty years later, Scott Guthrie heads all software development at Microsoft, and what was once a renegade InnerSource project is now a major part of Microsoft's software strategy.

Eric Raymond, the author of *The Cathedral* and *The Bazaar*, and one of the first theorists of the Open Source movement, once coined what he called Linus' Law, "Given enough eyeballs, all bugs are shallow." I propose a corollary, which we might call Scott's Law, or The Law of Innersourcing: "Given enough connected

developers, all software development emulates the best practices of Open Source software."

CHAPTER 1

The InnerSource Approach to Innovation and Software Development

With Andy Oram

InnerSource is a software development strategy rapidly spreading throughout large corporations—and it is also more. At its essence, InnerSource enables software developers to contribute to the efforts of other teams, fostering transparency and openness to contributions from others. But beyond that, InnerSource represents a collaborative and empowering way of involving employees in making and implementing decisions throughout the corporation. It embodies a philosophy of human relations, an approach to rewards and motivations, and a loose, adaptable set of tools and practices.

This book presents case studies at a range of companies to show when and why InnerSource may be useful to your organization. The case studies candidly discuss the difficulties of getting InnerSource projects started, along with the progress so far and the benefits or negative fallout. We hope that readers will be inspired to advocate for InnerSource within their software development groups and organizations.

InnerSource is an adaptation of Open Source practices to code that remains proprietary and can be seen only within a single organization, or a small set of collaborating organizations. Thus, InnerSource is a historical development drawing on the Open Source movement along with other trends in software. This chapter introduces InnerSource and locates it in the history of software.

Old Patterns of Development: Closed and Slow

Most people still think of the normal software development model as a team working by itself, communicating with the outside world merely by accepting lists of requirements and feature requests, then shipping prototypes and final ver-

sions of the software for testing. Many newer models of development challenge this norm, specifically the free and open software movement and various Agile and Lean development strategies such as Scrum, Extreme Programming, and Kanban.

The closed model is neither as old nor as rigid as usually thought. Early software developers shared their code. In the first decades of computing, it was hard enough to get the code working in the first place. Developers saw no commercial value in the code, or held an academic attitude toward open publication, or just lacked a business model for licensing it and collecting money. In addition, companies outsourced much development and every team depended on libraries from third parties, leading to much frustration and gnashing of teeth because of mismatches in requirements.

Commercial software—Microsoft being the obvious poster child—became commonplace during the 1970s and 1980s. In response, a conscious free software movement emerged, codified in Richard Stallman's GNU Manifesto of 1985. It formalized code sharing, and the Open Source movement added a recognition that there was tangible business value to sharing. InnerSource taps into this business value while sharing code just within the walls of a single institution, or a consortium.

In a parallel historical development, software got bigger and more complex, eventually crying out for some kind of formal development model. Practitioners attempted to enforce strict adherence to a model of requirements definition, coding, and testing, spawning the movement called "software engineering" whose historical milestones include a 1968 NATO conference and a 1970 paper by Winston W. Royce[1] listing the rigid order of events in what he derisively termed a "waterfall model."

Much derided now, the waterfall model and other accoutrements of software engineering made sense given the coding constraints of that era. Here are some reasons that the waterfall model caught on, along with the changes in the environment that render it inappropriate for most software development today (even where high-risk products such as cars and power plants are concerned):

Coding speed

Early computer languages, although astonishing advances over assembly language, required many lines of code. Modern strategies such as modules and patterns were slow in coming. Any small change in requirements could add weeks to the schedule. And loose language definitions, weakly supported by

1 Winston W. Royce, "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON* (August 1970): 1–9. Reprinted in the Proceedings of the International Conference on Software Engineering, 1987.

compilers, meant that a typo in a variable could lead to subtle errors requiring hours of investigation.

Nowadays, common coding activities have been encapsulated so expertly that one can implement a web server or a machine learning algorithm in a dozen or so lines of code. Tweaking and updating are routine activities and are expected.

Testing speed
: Coding was not originally the most time-consuming part of development—testing took that prize. For much of computer history, developers tested their code by inserting temporary PRINT statements. Submitting code to quality assurance and developing tests included many manual steps.

  Test frameworks, mocking tools, continuous integration, and branch support in version control systems make testing go much faster. Any change can be integrated into the project and run through automated tests quickly. The degree of automation makes test-driven development—where the developer creates tests in tandem with the code being tested, and strives for complete coverage—appealing to many organizations. Static and dynamic code checking also help.

Speed of build and deployment
: Old applications were monolithic and might require overnight builds. They had to be manually installed on physical systems, and any change meant downtime and lost revenue.

  Nowadays, most organizations strive for some form of modularization and packaging—aided by new tools and practices such as containers and virtualization, and infrastructure-as-code—the most recent instantiation of this trend being microservices. One can make a trivial fix and replace all running instances without the end users noticing at all.

As you can see, trends in hardware speed, software development, and networking have made collaboration methods feasible that are suppler than the waterfall method. Less obviously, they facilitate coding communities and collaborative development as well, and thus lead to the power of Open Source. The waterfall method always appealed to managers, sales forces, and customers because it promised products on supposedly reliable deadlines, but all too often those deadlines proved laughable (a constant theme of the popular *Dilbert* cartoon, for instance). Less rigid development methods have proven to be actually more reliable.

Factors Driving Open Source

The story behind Open Source has often been told, in works ranging from the popular book *The Cathedral and the Bazaar*[2] to the more academic *Wealth of Networks*.[3] Some communities definitely aired philosophical views that code should be free like speech, but others sensed early in the movement an even more powerful incentive: that they could achieve more by working together than by having each organization set up barriers around their code and figure out a way to monetize it. Environmental factors that contributed to the tremendous growth of free and Open Source software included:

Ubiquitous networking
: As more of the world joined the internet, and connections got geometrically faster in the 1990s and 2000s, more and more developers could download code and submit changes. Collaboration on mailing lists, chat channels, and other forums also became relatively frictionless.

Faster computers
: Even the cheapest laptop could run a GNU/Linux operating system (in fact, that operating system was more suited to a low-resource environment than proprietary systems) fully loaded with cost-free development tools.

Improvements in collaboration
: Free and Open Source developers took advantage of all common communication tools to create worldwide communities. A simple innovation made communication much more valuable: archives for mailing lists, which preserve the history of every decision made by a group potentially for decades. Centralized version control was supplanted by distributed version control, making it easy to work on a separate branch of common code and create extensive new features in isolation from the larger community.

Cascading effects of coding
: Each advance in languages, libraries, and tools allowed more developers to pile on to projects and enhance their efficiency. Current Open Source developers stand on the shoulders of the giants who came before.

Proprietary Hierarchies

But how can communities compete with hierarchical environments? Critically, a set of best practices developed over time. Lost in history are the thousands of poorly run Open Source projects that did not handle their communities well:

2 Eric S. Raymond, *The Cathedral and the Bazaar* (Sebastopol: O'Reilly Media, 1999).

3 Yochai Benkler, *Wealth of Networks* (New Haven: Yale University Press, 2007).

projects torn apart by nasty mailing list exchanges, projects whose leaders did not respond in a timely manner to contributions, projects that developed an inner sanctum mentality and stopped attracting new developers. Successful Open Source projects today share a number of healthy behaviors, which translate well to InnerSource and will be covered in this book.

InnerSource has to compete with hierarchical and closed development models, and there are certainly traits in those models that appeal to hierarchical organizations:

- Managers like to know who is responsible for a given task or deadline, a mentality associated with the distasteful phrase "one throat to choke." The adversarial competitiveness denoted by that phrase contrasts strongly with the ethos of Open Source and InnerSource. Managers' comfort with that situation is illusory: placing the responsibility on one team leader, or even one team, does not guarantee that goals will be met.
- Resources are easier to plan and distribute when work is assigned to a single team with a known set of developers. But this simplicity is also illusory, because schedules in software development are notoriously unreliable.
- Small, geographically colocated teams can be astonishingly efficient. However, they can also lack diversity and develop groupthink. In the long run, software projects can suffer because end users were not sufficiently represented during design decisions, and because the tacit knowledge within the team gets lost over time, disrupting maintenance.
- If software is developed to be licensed and monetized, it's tempting to define the investment and compare it to the expected income. But because of unanticipated changes of direction, results may be disappointing.

This review of corporate practices leads to a more general look at the research into knowledge workers, economic motivators, and innovation. Although the literature in these areas is huge, we can pick out a few common themes:

- Creative people like to feel in control and empowered to experiment. InnerSource gives them free rein while ensuring their work meets corporate standards before going into the product.
- Beyond a basic salary and set of benefits, intrinsic motivations drive more innovation and participation than extrinsic ones.[4] The chance provided by InnerSource to see one's idea adopted, and the ability to work directly on a

4 Rachel Mendelowitz, "Here's What So Many Leaders Get Wrong about Motivating Employees," *Fortune*, July 3, 2016, *http://fortune.com/2016/07/03/leaders-motivate-employees-business/*.

project that matters, are powerful incentives to positive behavior among contributors.

- Diverse teams produce more useful products that meet a greater range of needs.[5] This diversity includes demographic considerations (race, nationality, gender, etc.) as well as diverse viewpoints and experiences. Open Source and InnerSource maximize diversity, although project managers must consciously aim for it and take steps to nurture diverse participants.

The Open Source Way

What makes Open Source software development different? Initially, of course, the difference came from the sheer topology of the teams. Asynchronous collaboration over the internet ran in direct opposition to the then-recommended optimal way to build software. This sort of peer-based massive collaboration relies on the complete transparency of both the actual code and the project's governance. This transparency is important to several aspects of Open Source: it allows people donating work to see how their donation is governed, it allows self-selection of tasks and self-organization of developers, and it allows work to be distributed globally in an asynchronous fashion, constructed holistically.

Open Source also promotes higher quality code, since transparent code and processes open every line up to wider review than is practical within a single proprietary team. This massive peer review has famously given rise to Linus's law, coined by Eric Raymond, "given enough eyeballs, all bugs are shallow," meaning that defects are fixed more readily in Open Source because more people can read the code and come up with a broader array of possible fixes. In other words, with a sufficiently large workforce, there is bound to be someone who knows the solution for a given defect. This is not a perfect guarantee, of course. The famous Heartbleed Bug, which remained in the Open Source OpenSSL security library for years, showed that subtle issues can be missed by the community.

As work on an Open Source project progresses, a natural meritocracy emerges. The best contributors are given more responsibility for the running of the project. They are rewarded for excellent technical contributions, but also for taking time to mentor new contributors. Open Source communities are increasingly paying attention to the tone of mentor interactions, seeking to increase civility to be as welcoming as possible to the largest pool of contributors. For the same reason, communities are taking documentation more seriously. Traditionally a separate, labor-intensive effort (and therefore relegated to an afterthought), it now

5 David Rock and Heidi Grant, "Why Diverse Teams Are Smarter," *Harvard Business Review*, November 4, 2016, *https://hbr.org/2016/11/why-diverse-teams-are-smarter*.

emerges semi-spontaneously as Open Source projects archive communications between mentor and mentee. In fact, all project-related communication is collected, indexed, and archived, creating de facto documentation that allows rapid onboarding of new contributors.

Another difference between Open Source and traditional methods of software development is "mission attachment," held in common by a majority of the participants. This is partly nurtured by the idealistic philosophy of "free software," which pins itself on ensuring "code freedom" so that contributed code is perpetually available to the whole community to use, modify, distribute, and deploy. These "4 Freedoms" first recognized the value of freedom in code, but other contributions are also recognized. Ideally, nontechnical contributions such as cleaning up documentation count as well in the meritocracy. The 4 Freedoms also allow contributors to "fork" the project, cloning a complete copy so that the person initiating the fork can make modifications as desired. This is a failsafe measure to protect contributors from misuse of their contributions—they can always fork and start a new project if they disagree with the direction of the original project to which they contributed. This is what has happened, for example, to the OpenOffice.org project, which led to the LibreOffice fork of the project and the community.

The concept of *Enlightened Self-Interest* must be introduced to explain the motivations behind Open Source development. This is the idea that all participants are ultimately motivated not only by altruism, but also by personal needs to get something specific done—what Raymond characterized as "scratching an itch." Many well-known Open Source projects started out because of the creators' personal "itches": Larry Wall had to create reports but wasn't quite happy with the tools available to him, which included C, Awk, and the Bash shell, and so he created Perl. Linus Torvalds created the Linux kernel simply because no Unix implementation was available for his 80386 machine (see *Open Sources* by O'Reilly[6] for a more detailed history). These motivations to start projects can be personal needs, but projects may also be driven by other types of motivations, such as curiosity, the directives of an employer, or the desire to increase personal reputation.

Why Does Open Source Development Succeed?

While a detailed explanation of why Open Source development succeeds is beyond the scope of this chapter, Open Source software development seems to have found solutions to some of the long-standing challenges in the software industry: notably developer motivation and timing.

6 Chris DiBona and Sam Ockman, *Open Sources: Voices from the Open Source Revolution* (Sebastopol: O'Reilly Media, 1999).

A key challenge of large software development projects is coordination: how can large Open Source communities deliver complex products such as the Linux operating system without project plans, requirements documents, and a large number of managers who would ensure that the necessary work gets done?

The answer is of course not straightforward, but a number of key factors play a role. For starters, when Linux was first announced, it attracted many other volunteer developers who were also interested in getting a Unix clone on their personal computers. Developers are motivated for several reasons: they have a sincere interest, they may use coding as a form of learning (which seemed to be Torvalds's original motivation), or they simply enjoy contributing. In any case, the motivation to contribute is reinforced by the sense of achievement engendered when contributions are adopted and used by others. This level of involvement is not something that you would typically find on a commercial software development project—hired developers are told what software to develop, and while this doesn't mean they can't enjoy their work, it is unlikely that developers are passionate about all the software they are developing.

A second factor that facilitates the implicit coordination found in large projects is modular design, or what Tim O'Reilly has called the "Architecture of Participation."[7] Modularity refers to the level of independence of different subsystems. For example, the Linux kernel comprises several different subsystems, and furthermore there is a clear distinction (inherited from Unix) between the kernel and external drivers that communicate with hardware. Modularity enables a large number of people to work on different subsystems without getting in each other's way. Having good tool support (e.g., configuration management tools and version control systems) is of course important too.

Traditionally software development projects commonly face the famous three-way tension between delivering high-quality software in a timely fashion and within budget. Open Source projects tend to deal with this tension in a different way.

Open Source projects have a strong emphasis on quality, for several reasons. Open Source projects generally rely on rigorous peer review. And while Open Source projects increasingly see contributions from firm-employed developers, many traditional volunteer developers are intrinsically motivated and take pride in their code—they won't take shortcuts so that they can deliver more quickly. Whereas commercial software development projects usually have a delivery date —either announced to the market, or agreed with a paying customer—Open Source projects often don't have a deadline. Because there is no paying customer, it is far easier to simply delay a release if developers aren't quite satisfied with the

7 Tim O'Reilly, "The Open Source Paradigm Shift," in *Perspectives on Free and Open Source Software*, eds. Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani (Cambridge: MIT Press, 2005).

quality, although such delays may affect the project's credibility if that happens too often. Indeed, Linus Torvalds has delayed releases of the Linux kernel when he wasn't confident the code was sufficiently stable.

Frederick Brooks describes in his famous essay "No Silver Bullet" that adding more staff to a project that is already late will make it later—this has been coined *Brooks's law*.[8] The explanation for this is that adding more people to a project adds significant communication overhead. The induction of new staff will effectively delay the current project team in getting the project finished in time. In Open Source software projects this isn't so much of an issue, because much of the discussion and information about design decisions is captured online. Granted, not all Open Source projects succeed in capturing this well, and when the amount of information exchange becomes too large, it should be formalized in well-structured documentation to summarize the essence. But due to the dispersed nature of the community, developers are more independent and forced to figure things out on their own.

We can now turn to the big question: can we borrow the best practices of Open Source to solve the challenges of software development in industry? This idea was suggested about 20 years ago, and is called InnerSource.

What Is InnerSource?

InnerSource is, simply, the use of Open Source principles and practices inside proprietary organizations. InnerSource empowers individual engineers to contribute code from within their team to another team in order to speed up the process of adding functionality (features) or fixing defects. InnerSource also democratizes development and control over the direction of the project by encouraging pull requests over feature requests. The traditional development practice put all decision making and control in the hands of a single team that maintained the code, who were petitioned by users to add enhancements through feature requests. In contrast, with Open Source and InnerSource, anyone who wants to make a change downloads his or her own version of the code through a pull request (a term popularized by the prevailing code management system GitHub), adds the desired feature, and submits the code for approval.

Mirroring the Open Source approach, submitted contributions are reviewed by someone sufficiently knowledgeable to determine whether the submission is ready to integrate into the master codebase, and to guide the contributor to make any necessary changes. This process of review and providing guidance is preferably done in writing so that the exchange of information can be indexed into a

8 Frederick P. Brooks Jr., "No Silver Bullet – Essence and Accident in Software Engineering," *Proceedings of the IFIP Tenth World Computing Conference* (1986): 1069–1076.

persistent, searchable archive, supporting the desired Open Source attribute of lazy accretion of actionable documentation.

Companies adopt InnerSource in different ways. Vijay Gurbani and his colleagues[9] described a simple taxonomy that distinguished two different modes of adoption:

Project-specific InnerSource
: A dedicated team has responsibility for a particular software asset, typically something that is a key asset to the business (as opposed to simply a development tool, for example). The dedicated team is funded by other business units.

Infrastructure-based InnerSource
: In this model, the organization provides the necessary infrastructure for storing and sharing source code and documentation, and to facilitate communication. Anybody can create a new project, but each project initiator is responsible for maintaining their projects. This means that the level of support that a user can expect will vary greatly.

This taxonomy is useful because it established some basic terminology that we can use when we discuss InnerSource programs.

It's also useful to briefly discuss what InnerSource is not. InnerSource is not simply adopting GitHub or GitLab within your organization and arguing that all source code is now transparent. While tooling is important, it's only an enabler and not a guarantee for success. Also, InnerSource does not mean paying your developers to work on Open Source projects: that's simply sponsoring Open Source development. When you release the source code you're working on to the public with an Open Source license, it's not InnerSource.

InnerSource is also not a development method like Rapid Application Development (RAD) or Scrum, which defines a number of roles (e.g., Scrum Master), ceremonies (e.g., the daily standup), and artifacts (e.g., the sprint backlog). No fixed set of "roles" and "ceremonies" constitute InnerSource.

Instead, InnerSource represents a different philosophy for developing software in-house, which can complement existing methods such as Scrum. Rather than simply adopting new tools and practices per se, InnerSource involves changing the philosophy of working and of leveraging and empowering the developers within your organization. Because every organization is different, there is no single recipe to adopt InnerSource, and that's what makes it so hard!

9 Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb, "Managing a Corporate Open Source Software Asset," *Communications of the ACM* 53, no. 2 (2010): 155–159, *https://doi.org/10.1145/1646353.1646392*.

A History of InnerSource

The term "Inner Source" was originally coined by Tim O'Reilly in 2000.[10] At the time, O'Reilly served on the Board of Directors of CollabNet, a company he cofounded with Apache Software Foundation cofounder Brian Behlendorf and Bill Portelli in 1999. CollabNet's mission "is to bring Open Source–style collaborative development to the software industry as a whole." As such, CollabNet was the first company that helped its customers to engage strategically with Open Source software, and among its first customers were Hewlett-Packard[11] and Philips,[12] both of which implemented InnerSource programs.

It's vital to have an appreciation for how important tooling was back in those days. Although software version control systems were already widely used in industry in the 1990s, there was a confusing array of systems, both Open Source and commercial. Among the commercial offerings were ClearCase and Rational, and Open Source solutions included CVS and Subversion (SVN). (Git's first release was in 2005, and GitHub, the company that made Git available to the masses, was only founded in 2008.) When the first InnerSource programs were started, the diversity of version control systems was far more prevalent than today. This variety in tools was, to use Frederick Brooks's terminology, an "accidental" complexity of software development rather than an "essential" complexity.[13] Difficulties arising from tooling are not *inherent* to software development, but they can represent major challenges to software teams.

Though the term "Inner Source" was coined by a single person, different organizations that started InnerSource programs in the early 2000s did so independently. As such, different organizations used different terms.[14] Hewlett-Packard used an umbrella term "Progressive Open Source" within which "Inner Source" refers to one approach; the program at Bell Labs has been named "Corporate Open Source," and other terms in use included "Internal Open Source" or "Community Source," although those names were also used to describe variants of

10 The term "Inner Source" was coined in a response to Matt Feinstein's question on O'Reilly's attitude on Open Source and OpenGL. The original response is available online. To make the term easier to find (try searching for "inner source" and you'll find references that are not software related), we've removed the space in between and adopted "camel case" capitalization.

11 HP's program is documented in a paper by Jamie Dinkelacker, Pankaj Garg, Rob Miller, and Dean Nelson, "Progressive Open Source," *Proceedings of the 24th International Conference on Software Engineering* (2002): 177–184.

12 Philips's program is documented in an article by Jacco Wesselius, "The Bazaar Inside the Cathedral: Business Models for Internal Markets," *IEEE Software* 25, no. 3 (2009): 60–66.

13 See Frederick Brooks's essay, "No Silver Bullet: Essence and Accidents of Software Engineering."

14 Klaas-Jan Stol, Paris Avgeriou, Muhammad Ali Babar, Yan Lucas, and Brian Fitzgerald, "Key Factors for Adopting Inner Source," *ACM Transactions on Software Engineering and Methodology* 23, no. 2 (2014).

Open Source practiced publicly by proprietary organizations that were mostly unsuccessful.

These early visionary individuals, teams, and companies were the first ones to adopt InnerSource. However, InnerSource did not attract attention from the wider software industry immediately. Younger companies, particularly ones founded in the "Internet Age"—Google and Facebook are the archetypal examples—were of course more agile and were already familiar with "The Open Source Way." Other organizations, however, were not paying attention to this trend, and so the adoption of the "InnerSource paradigm" was quite limited until after the triumph of Open Source in the marketplace.

Why Do Companies Adopt InnerSource?

There are many reasons why companies adopt InnerSource.[15] Here we discuss some of the most important ones.

Breaking Down Silos and Bottlenecks

Perhaps the most important motivation to adopt InnerSource is to break down the silos that inevitably exist in large organizations that have optimized at some point for specialization or ownership culture. Partly this is explained by *Conway's law*, which states that organizations tend to build systems that are structured according to the organization's communication structures.[16] Having different teams take ownership of a specific component with a well-defined interface makes a lot of sense, as long as the interface is respected by both the implementer of the component and its users. After an interface is defined, communication between different teams can be minimized, leading to a high level of specialization within those organizational units. The very existence of hierarchy and different organizational units usually leads to an "us versus them" attitude, for several reasons:

Local optimization
: Mid-level managers who are responsible for a team tend to be concerned mostly with the performance of their team, because ultimately that is considered to reflect their performance as managers. This is a clear example of "local optimization," rather than optimizing for the company as a whole. Consequently, there is no direct motivation for these managers to help other teams.

15 Klaas-Jan Stol and Brian Fitzgerald, "Inner Source—Adopting Open Source Development Practices within Organizations: A Tutorial," *IEEE Software* 32, no. 2 (2015): 55–63.

16 Melvin Conway, "How Do Committees Invent?" *Datamation* 14, no. 4 (April 1968): 28–31.

Not invented here
: The "not invented here" syndrome seems to be a fixture of the software engineering profession, to the point that many professionals are suspicious of the quality and maintainability of any software they didn't personally write.

Developer incentives
: Engineers who have in the past been measured and rewarded solely on their individual coding skill and speed are often reluctant to spend time reviewing others' code.

Developer priesthood
: A division into units and teams inevitably leads to a high degree of specialization. Engineers who work on their specific product or component have a deep understanding of their code, including its function, form, glitches, and shortcomings. They know the rationale for why the software is the way it is. Outsiders tend to lack that, which may lead to misunderstandings and disagreements.

Perceived job security
: Finally, the software engineering industry has long been plagued by practitioners who believe that hoarding expertise is the only job security.

All of these impediments must be mitigated for InnerSource to effectively reduce bottlenecks, and to improve collaboration across an organization.

Reuse

An important reason to adopt InnerSource is to increase reuse, which software engineering researchers and professionals have long considered the Holy Grail of software engineering. After all, constructing software is a lot faster when you can reuse already-extant high-quality components developed elsewhere in your organization. A key barrier to reuse, of course, is the lack of transparency that simply leaves developers unaware of other potentially similar efforts within your organization. The increased level of transparency that InnerSource provides takes away at least this barrier. Another issue is that available components tend to have slightly different use cases, or different features from the ones required in other teams. Again, the transparency offered by InnerSource can help here, by enabling and facilitating closer collaboration between a component's "owner" and its users. For example, teams might suggest—and help implement—additional features, or rearchitect an existing component in collaboration with its owner. Sure, this is easier said than done, and other things will have to be put into place, in particular organizational commitment at the highest level manifested as time and budget.

Knowledge Sharing and Full Stack Teams

Specialization and ownership culture both drive the creation of silos of knowledge. Over time, cross-silo knowledge can be lost within an organization, which can be damaging if you need to quickly mobilize resources to another area of the stack (onboarding can be difficult). Best practices, such as establishing an effective DevOps practice, assume that at least some full-stack or cross-platform experts exist to take a holistic view when considering sweeping changes. Once full-stack knowledge has been lost to a specialization and ownership culture, it can be difficult to stimulate regrowth. InnerSource mentoring and extrinsic rewards can begin to break down silos and reward cross-stack collaboration.

Innovation

At least one very large organization originally implemented InnerSource to help break down barriers to innovation. In very regimented organizations, thinking innovatively may have been intentionally suppressed until it is effectively extinguished. Giving engineers who were previously closely managed a measure of autonomy, emotional safety, and the sanctions to pursue unorthodox hunches via InnerSource methods can unleash their innovative creativity in productive ways while protecting production outcomes through applied mentorship and review. Sun Microsystems' Cofounder and Chief Scientist Bill Joy famously had the insight that "no matter who you are, most of the smartest people work for someone else."[17] This is true *within* a company, as well, and allowing input from outside your silo can really open your eyes to new possibilities.

Improving Quality

As previously mentioned, the effect of Open Source on the quality of the resulting software has been summarized as Linus's law, which talks about increasing the resources brought to bear for code review. But the mere awareness that code will potentially be reviewed by a large number of strangers has the effect of causing us to strive to put our best foot forward, because nobody wants public humiliation. It has been well documented that Open Source developers tend to be more careful when developing code they will post for the world to see. Another Open Source maxim, "release [code] early, release often," means important defects that could negatively affect an entire project, such as security holes, are generally discovered and patched much more rapidly than in proprietary settings.

17 See the Wikipedia entry on Joy's law.

Staff Mobility and Process Standardization

Most InnerSource implementations assume transparent code hosting using a distributed development platform such as GitHub Enterprise, Bitbucket, Mercurial, or the like. That standardization of tooling reaps an inherent benefit of making it quite a bit easier to onboard new hires or transferred employees. Recruitment and onboarding are both easier because the development environment and processes, such as the review cycle and merge and deploy protocols, are now much more standardized within the organization.

InnerSource Today

The role of software has become much more important for many organizations that have not traditionally developed software. For example, software is transforming sectors where software did not loom large originally, such as the automotive sector: the amount of software in cars has risen exponentially in recent decades, and this trend is likely to continue. So, organizations outside the software industry now also find themselves in a position where they have to deliver complex and innovative software of high quality and within budget. These organizations that are new to software are now eagerly looking for new approaches that can help them overcome the bottlenecks found in traditional development approaches.

The limited attention for InnerSource changed in recent years. In 2014, one of us (Danese) was hired by PayPal to head the company's Open Source programs. She quickly came to the conclusion that PayPal would benefit greatly from an InnerSource practice. In 2015 she gave an influential keynote at the annual OSCON conference stating her conclusion that InnerSource would be an important practice going forward as more organizations undertake work to modernize their engineering practices in the direction of Open Source. In that talk, she also announced the creation of the InnerSource Commons, an industry group that seeks to bring together like-minded professionals who are interested in implementing InnerSource.

Since then, we have seen a significant increase in interest and momentum. To establish the community, we started a Slack channel that now (mid-2018) counts over 270 individuals representing over 70 different companies—and these numbers are growing fast. Twice a year, the InnerSource Commons community organizes a Summit during which attendants share their knowledge and experiences. The community follows the Chatham House Rule, which states:

> When a meeting, or part thereof, is held under the Chatham House Rule, participants are free to use the information received, but neither the identity nor the affiliation of the speaker(s), nor that of any other participant, may be revealed.

PayPal also sponsored the publication of two booklets with O'Reilly. The first one is a brief introduction, titled *Getting Started with InnerSource*, authored by Andy Oram. The second one, by Silona Bonewald, also currently employed by PayPal, is titled *Understanding the InnerSource Checklist*, in which Bonewald presents a checklist of preconditions that must be in place before organizations can adopt InnerSource successfully.

A dedicated team within the InnerSource Commons community has also started to document the various lessons learned and has adopted the concept of "patterns" to encode this knowledge. Ultimately, our goal is to develop an InnerSource pattern language. The work on distilling patterns is an ongoing activity, and this "patterns subcommittee" is actively disseminating its work in various ways, including webinars, videos, and papers.[18]

Why We Wrote This Book

Our work with the InnerSource Commons community has taught us that there is broad interest in a collection of case studies that both illuminate the motivations of organizations on their InnerSource journey and justify experimentation with InnerSource as a practice. Many individuals have told us they are seeking permission to start InnerSource experiments at their places of work, but they need persuasive examples. Others have already begun experimenting but aren't sure how to plan for scaling out the practice across the organization or how to measure success and justify broader adoption. Still others are unsure how to get started.

In order to get InnerSource to flourish inside an organization, you must first understand what aspects of the existing culture stand in the way of transparent collaboration and acceptance of contributions from outside a team. Some common cultural impediments to InnerSource include (but are not limited to):

- A general fear of change
- The "not invented here" syndrome
- A belief that developers external to a team are less skilled or will submit defect-ridden code
- A lack of sufficient time or resources to get existing work done
- An unwillingness to engage in mentorship, or lack of knowledge on how to be a mentor
- Mid-managerial conflict over the team's charter

18 Erin Bank et al., "InnerSource Patterns for Collaboration," *Proceedings of the 24th Conference on Pattern Languages of Programs*, 2017.

These existing cultural forces will vary per organization, and even per business unit or team. Identifying and providing extrinsic motivators to change away from deeply ingrained beliefs and behaviors can be tricky (but necessary).

With so many people and companies interested in this topic, we feel the time is ripe to present a set of interesting case studies of InnerSource in practice. Because each case of InnerSource differs from the next, together these cases represent many different experiences in different contexts. This can be very useful to other individuals and companies who want to learn what other companies have done. Furthermore, while each chapter presents a rich description of one specific case study, we also think that you, the reader, may want to get some support in starting off in your own organization. So we've included a chapter with practical tips for crafting your first experiment.

Who Should Read This Book

This book targets professionals at all levels. For executive managers, this book presents convincing evidence (we believe!) that InnerSource is the way to develop software in the future. Yes, InnerSource adoption will cost resources: you need to make available some budget to roll out InnerSource and provide support to the people on the ground. But, we ask you to see this as a long-term investment. No community has ever ramped up within a short time. Things like learning to collaborate and building trust take time. Additionally, money alone isn't enough: InnerSource is not a product or service that you simply purchase; hiring a consultant alone is not enough. You need to identify and support the "change agents" that make things happen—these are the champions you'll need to evangelize and talk to the naysayers. Remember, ultimately InnerSource is about empowering people, and Good Things will happen.

For mid-level managers, we hope this book provides inspiring stories that encourage you to revisit your responsibilities as a team manager. We're not saying to take on more responsibilities, but instead to redefine them. Rather than optimizing for the team you're responsible for, we'll try to convince you that supporting your developers to participate in an InnerSource initiative is ultimately a Good Thing. Obviously, we also recognize that you as a manager will need to get the means and resources, and therefore we also wrote this book for your bosses.

For developers, we think this book is interesting because it tells the stories of so many other developers, who also don't have decision-making powers, and who increased their productivity, their happiness, and the ability to use their creativity. As we've already pointed out, InnerSource is about empowering people, and ultimately this means developers and users of software. The various case studies in this book illustrate how individual developers were empowered, how they increased their job satisfaction, and how they overcame the various challenges that resulted of lacking any decision-making power. For developers, initiating an

InnerSource program that gets full management support is not trivial, but we hope the stories in this book provide some inspiration. Furthermore, InnerSource gives companies a taste of the benefits of Open Source, which has become an essential part of the software engineering ecosystem that can't be ignored. InnerSource offers an internal training ground for companies on the way to a full Open Source investment, either by joining existing Open Source foundations or by open-sourcing their own assets.

Finally, as the first dedicated book on InnerSource, we believe it will be of interest to software engineering students and academics who study software development methods and tools. As we mentioned before, we consider InnerSource to be the future of software development, and as such we think students should learn about it, just like they learn about traditional and agile methods. For researchers, we believe this book is useful because it compiles a series of detailed case studies of InnerSource.

How This Book Is Organized

This book tells the stories of several companies that have started the journey to adopt InnerSource. As we mentioned, InnerSource is not a defined development method or framework, such as Scrum. Instead, it's a development paradigm, and each instance is unique and tailored to the specific context of the organization.

InnerSource often refers to the "Open Source development paradigm"—and in particular we refer to "The Apache Way." In Chapter 2, Jim Jagielski, cofounder and director of the Apache Software Foundation (ASF), discusses InnerSource and introduces "The Apache Way."

Chapter 3 presents one of the early cases of InnerSource. It describes the Session Initiation Protocol (SIP) stack project at Bell Laboratories, which was part of Lucent Technologies at the time the project originated 20 years ago. This chapter, contributed by the project's "Benevolent Dictator for Life" (BDFL) Vijay K. Gurbani, and his co-authors James D. Herbsleb and Anita Garvert, describes the origins, motivations, and evolution of the project.

In Chapter 4, Georg Grütter, Diogo Fregonese, and Jason Zink present the InnerSource initiative at Robert Bosch GmbH, or "Bosch" for short. Bosch is a large German company that operates in many different domains, including the automotive sector and consumer electronics. Bosch started the Bosch Internal Open Source (BIOS) program around 2009 within a specific R&D setting.

Chapter 5 presents a case at PayPal, which started adopting InnerSource back in 2014 when the company hired Danese Cooper as Director of Open Source. PayPal has run a number of InnerSource experiments to evaluate process improvements. In 2016 PayPal's InnerSource Team ran a large experiment to determine whether an InnerSource mandate would work on a core component where the

teams had previously been reluctant to do InnerSource. In that same year, a small grassroots InnerSource experiment quietly launched and ran itself out of PayPal's Chennai office. The two cases confirmed each other's findings.

In Chapter 6, Isabel Drost-Fromm presents the journey of Europace toward InnerSource. Europace is a medium-sized company in the financial sector that was searching for new ways to become more self-organizing.

Chapter 7 presents the case of Ericsson, a global leader in the telecommunications domain. John Landy discusses his experiences and lessons learned with setting up the Community Developed Software (CDS) program. The case presents a greenfield development project, in which Ericsson adopted a platform-based architecture, but without the corresponding platform organization. The key reason for doing so is that platform teams tend to become bottlenecks, as feature teams make a large number of feature requests.

Finally, after reading these exciting cases, we hope you feel sufficiently inspired to try InnerSource in your own company. For that reason, we lay out a set of guidelines for adopting InnerSource in Chapter 8. Based on the recurring patterns that we observe in the case studies, we offer advice about how to choose and structure your first InnerSource experiment.

Visit Us Online

On this book's website you'll find much more information. We'd also very much like to hear from you—any suggestions, feedback, or comments are welcome. We hope you enjoy reading this book as much as we enjoyed writing it!

Acknowledgments

This book isn't just the result of a couple of months of writing. Klaas has conducted research in this domain for about 10 years, during which his research has been kindly supported by the Irish Research Council (under IRCSET grant RS/2008/134 and New Foundations grants in 2013 and 2014) and Science Foundation Ireland (grant 15/SIRG/3293 and 13/RC/2094 to Lero—the Irish Software Research Centre), allowing him to travel across the globe to visit many companies for his field research. The insights gained from these field trips have been foundational for this book.

CHAPTER 2

The Apache Way and InnerSource

With Jim Jagielski

At its core, InnerSource applies the "lessons learned" from successful, healthy Open Source projects to guide and direct enterprise IT development. Another way to look at InnerSource is applying the principles and tenets of Open Source development to internal processes and principles. With this in mind, it's critical for those adopting InnerSource to understand the *what* and *how*, but even more importantly the *why* of those tenets, as well as which particular ones to emulate. We have found that the best model by far are tenets used by the Apache Software Foundation (ASF), collectively termed "The Apache Way."

In a nutshell, The Apache Way can be condensed into what is the unofficial motto of the ASF: Community Before Code. This does not mean that the code (or the software project) is unimportant, but rather that secure, innovative, enterprise-quality, and healthy code depends on the health and vitality of the community around it. This realization emerged at the origin of the Apache Web Server project and the Apache Group.

Origins of The Apache Way

Back in 1995, the most popular web server was the NCSA Webserver (HTTPd), which was written and maintained pretty much exclusively by Rob McCool. There was a large and growing user community, but no real developer community at all. When Rob left to join Netscape, this left the development of HTTPd stagnant. Here was a large, incredibly important software project that countless people and businesses depended on, but that was now no longer actively developed or maintained. Out of necessity, as well as self-interested altruism, a small group of individuals, the Apache Group, started to exchange fixes and improvements (called "patches") and started collaborating on a mailing list. Like a phoenix, the project itself slowly was reborn. Although this rebirth was incredibly

successful, in fact, more successful than we ever anticipated, we realized how lucky and fortunate we were that we just happened to have the right group of people, at the right time, to be able to achieve this. We also realized how unlikely it would be to catch lightning in a bottle a second time. What was clear was that we wanted to create a development environment that could weather the coming and going of developers, corporate interests, and other external factors, to ensure that no one else dependent on that project, individual or company, would ever be left in a lurch like we were.

The way we accomplished that was to focus on the community around that software project, to make facile collaboration a priority, to encourage a flat hierarchy, to value consensus, to make unaligned volunteers first-class citizens, and to maintain an incredibly tight "feedback loop" between users and developers (after all, those of us who formed the Apache Group were users in the first place, and became developers by necessity). Basically, our path was a recognition that volunteers were the life-blood of successful Open Source projects and the source of its energy, health, and innovation.

This understanding, and the focus on volunteers and contributors as a basis for measuring and instilling community health, serves as the basis for the main core values of The Apache Way.

Fundamentals of The Apache Way

Three fundamentals lie at the core of The Apache Way:

- Meritocracy
- Transparency
- Community

Let's look at each of these in turn.

Meritocracy

In some circles, the phrase "meritocracy" is avoided, due to negative connotations caused by experiences where it is abused and perverted, restricting diversity and inclusion instead of expanding them. Because this concept is so important to The Apache Way, it is crucial that it be clearly understood.

Within Apache, meritocracy means that the more you do, the more merit you gain, and the more merit you gain, the more you are able to do. You are recognized and rewarded for your contributions, independent and separate from other factors. In other words, merit is based on the actions and contributions of the person, and not who or "what" the person is.

We want members of our community to actively participate, not just passively observe. By contributing in some way—code, documentation, fundraising, publicity—they gain an understanding of the project's needs and challenges, and become more committed to it as individuals or organizations. If our project was a restaurant we wouldn't just let people review its recipes; we'd invite them into the kitchen. This is central not only to the goal of meritocracy, but to the others as well.

This definition of merit, and meritocracy, allows for an extremely flat, level, and nonhierarchical playing field for all contributors. Within the ASF, and within all Apache projects, peers have "proven" themselves and thus are worthy of, and given, respect and trust. Also important in this concept is that once earned and awarded, merit does not "expire." When looking back at the origins of The Apache Way, when all contributions were provided by volunteers doing work "on the side," one can see how vital this was. After all, life happens and it was not unheard of that a contributor would step away for months at a time, only to return as their time freed up. If they were required to "regain merit," it would discourage them from returning. The principle is part of the larger goal of making the bar as low as possible for involvement within a project.

Another aspect of meritocracy is that, when done correctly as within Apache, it avoids a self-sustaining oligarchy, where those with power and merit maintain and consolidate it. By creating an environment and culture where new contributions and new contributors are encouraged and welcomed, and where a governance structure makes it clear what benefits and rewards they gain via their contributions, the community itself will thrive and grow. Key in this is that no one's "vote" or opinion is more valued, or weighted more heavily, than someone else's. Whether one has been a member of the project for six months or six years, their vote and their voice carries the same weight.

Treating newcomers and marginal participants equal to established members is a principle that might not seem productive, but it has proven its importance again and again. Its basis is the recognition that contributors will come and go. If, when joining a project, someone knows that they cannot "compete" with someone who has been there longer, they really have no incentive to join. People want a say in the projects they join; they want to exercise some influence on the direction of the project. If potential influence is squashed simply due to being "new" in the project, why bother in the first place? This risk of repelling contributors is elegantly avoided by Apache's very flat peer system.

The final value of meritocracy is that it provides the encouragement and incentive to get involved. It sends a clear signal to the external community and ecosystem that contributions are not only welcomed and wanted, but actually rewarded. It is the core path in that incredibly important feedback loop between users and developers/contributors.

Transparency

To describe how the Apache Group approaches this fundamental, let's first consider software development in a typical enterprise environment. A roadmap is created by marketing and management, which defines what features and capabilities the software must have. Next that roadmap is presented to the software development team, which consists of individuals who are assigned work and tasks toward the development effort. Discussion is limited to that specific team, usually via face-to-face meetings or the normal "hallway/water-cooler" conversations, with little, if any, of it documented. Very little collaboration is done within the team, and certainly not external to the team. In effect, we have the standard siloed, self-contained, and insular software development scheme so prevalent even today.

Now compare this with how software development is done via community-based Open Source projects. The roadmaps are not as detailed or stringent, and the creation and adjustment of the roadmap are done by the developers and users of the project, usually via mailing lists or wikis. The "team" is geographically and culturally diverse, spread out over time zones and locations that make face-to-face meetings impossible. The people on the "team," rather than being assigned, are there by choice, a self-selected group of individuals who are also self-organized. Areas of responsibility are fluid, with contributors working on areas that interest them. Instead of working on the project because it is their "job," they ideally work on it because they are personally invested at an emotional level with the project (and its community). Even in situations where the person is paid for their work, there is a fundamental difference between someone who contributes because they are paid to do so (a "hired gun," so to speak, or, alternatively, a "line cook"), and one who is lucky enough to be paid for something they would do on their own free time, and frequently still do ("a chef"). We have noticed that people working on Open Source projects on company time often are volunteers or at least empowered to choose the project to which they contribute, not people assigned by their managers. This shows a high level of identification with the project.

Comparing the top-down and Open Source scenarios just shown, it looks as though the "Open Source Way" would never succeed, and yet today it is clear that it is the optimal way to do software development. But this is possible only when transparency within and beyond the project is valued. In fact, it is vital.

Transparency, of course, is basic to Open Source because the source code must be available. In other words, the code itself is transparent, in that it is visible to all. But when doing software development, that is only one small facet of true transparency. Not only the code but the decisions must be visible: the decision-making process itself, the discussions and conversations—all of this must be transparent.

Transparency is important because it prevents a culture and environment that disenfranchises those not currently in the group; it enables those "outside" to

have a clear and accurate insider's view, which fosters a development community that people feel empowered to join and contribute to. Transparency is required for collaboration not only within the team, but with other teams as well, which drives innovation and reuse. Transparency ensures that all discussions and points of view are represented, with everyone having the ability to understand and question them. Transparency, almost more than anything, drives the required culture inherent within InnerSource, one which breaks down the wall between "us" and "them."

Within Apache, this focus on transparency exhibits itself in several ways. First is the reliance on mailing lists as the primary method of communication. Mailing lists are preferred first and foremost because they are asynchronous, being unbiased in regard to location and time zone. Mailing lists are also self-documenting (all Apache mailing lists are archived), which ensures that the "tribal knowledge" held within the current group is shared and known universally. It provides a view into development history and the reasoning behind decisions. Mailing lists are also useful for their ability to thread conversations and topics, making it easier to follow items of interest and avoid others.

Another way transparency is encouraged is via the public nature of the code itself. When contributors know that their contributions will be seen "by the whole world," the tendency is to put one's best effort into it, which results in better code. This also has two other benefits. The first is letting contributors learn from others. The ability to mentor and be mentored is feasible only if one's work product is visible to a wider group.

The second benefit is maintaining the crucial personal attachment between a contributor and their contribution. The importance of keeping a history of who contributed what is unfortunately seldom recognized. Open Source recognizes and acknowledges that developers are really artists, whose medium of choice is code. As with all artists, they want to hone their skills as well as share their craft, and the Open Source movement provides the environment to do so. In most typical software shops, the developers write their code and then "throw it over the wall" to QA or production; that breaks the connection between the artist and the art, to the detriment of all. Many practical disadvantages can be observed: developers lose control and insight into their code and how it is actually used in production; QA and testing are done by those unfamiliar with the code and its requirements; and developers lack "responsibility" for the code they produce—in many cases they don't get the 3 a.m. beeper call when their code breaks in production, for example. Transparency—the ability for the contributor to point to their contribution, open and visible to all—maintains that vital connection.

Finally, transparency eases the concerns over the provenance of code inherent in all software development for licensing reasons.

Community

The third leg of The Apache Way is community. Although the other two could also be bundled as subitems under community, this aspect is somewhat deeper. In The Apache Way, community could almost be defined as the entire ecosystem around a software project. It is a recognition that not only is the community larger than the developer community, or even the developer and user community, but encompasses the wider world. Software today impacts everybody, even people (and other creatures) who don't directly use that software.

In essence, our fundamental of community concerns a shared culture that all members of that community understand and promote. The genesis of that culture, and the model of that community, must spring from the contributors.

Within Apache projects, we do several things to reinforce that. I mentioned earlier the concept of collaboration, but it is easy to minimize its meaning, or see it as simply "working together." Collaboration, as it relates to community, is about seeing the individual merit of each person and seeing each person as a vital part of the project itself. It means ensuring an environment, both in culture and in infrastructure, where each person can influence the project and can drive certain tasks or efforts, while still engaging the other community members as well.

One rule of thumb we use toward this goal is to discourage large, substantial "code dumps" into Apache projects. A code dump, for example, could take place if I were to refactor the code or create a new feature completely independently, and then, once complete, commit that to the codebase. In that scenario, I'm not really working with the rest of the contributors, I'm simply doing my own thing and adding stuff as the mood strikes. At Apache, what I would do is create a thread on the development mailing list describing what I was considering, maybe commit a preliminary work-in-progress (or, alternatively, create a public branch of that work-in-progress), and encourage others to work on it *with me*. That is true collaboration. That is a true community.

Another major part of community is driving consensus. In an environment such as Apache, no single person decides what feature to add, or what patches and contributions to accept. Instead, it is a community decision, which implies that there needs to be consensus on that decision. Within Apache we use voting to gauge this consensus; usually someone will propose something and ask for a vote, at which point people will post email messages saying +1 ("sounds good to me; I support it"), +0 ("no opinion"), −0 ("I don't like it, but don't want to stop it from happening"), or −1 ("I am against this and don't think we should do this"). Now you may expect that what we do is tally up the votes and let the majority vote "win." But you would be wrong. In general, if someone, even one person, votes a −1, we step back and continue to discuss the issue. We work on resolving disagreements and ensuring consensus within the group. Again, this is how a community should be run.

The Apache Way serves as the base model for InnerSource. Understanding The Apache Way, and using it to guide one's InnerSource strategy, has been the basis for successful InnerSource journeys for countless companies. The next several chapters present several of these journeys.

CHAPTER 3

From Corporate Open Source to InnerSource: A Serendipitous Journey at Bell Laboratories

Vijay K. Gurbani, James D. Herbsleb, and Anita Garvert

Year, place founded: 1996, Murray Hill, New Jersey (USA)

Area(s) of business: Telecommunications equipment manufacturer (software and hardware)

Revenues: US$9.44 billion (2005)

Offices: Global, headquartered in New Jersey (USA)

Number of employees worldwide: 30,500 (2006)

Number of R&D engineers: 10,000+

Year of InnerSource adoption: 1998

Although the radical disruption posed in the 1990s by internet voice calls may now be forgotten (everyone is now used to crossing continents through Skype or similar services), the innovation known as Voice over IP (VoIP) was a major extension of the internet's capabilities. On an economic level, VoIP challenged the incumbent telecom companies and their business model of high settlement costs to carry voice traffic across local and international boundaries. These settlement costs were, in turn, passed to the people making long-distance and overseas calls. In this environment arose an internet multimedia signaling protocol that enabled everyday internet users to bypass the telecom service providers for an essentially cost-free means to communicate with anyone in the world. This chapter traces the development methodology that, in anticipation of InnerSource,

enabled VoIP at Lucent Technologies, Inc., and its research arm, Bell Laboratories.

Background on Internet Protocols for Voice Communication

In March 1999, the Internet Engineering Task Force (IETF) released a multimedia signaling protocol called Session Initiation Protocol (SIP).[1] The standardization of SIP in 1999 coincided with the burgeoning use of the internet. By 1995, the last NSFNET backbone service was dismantled, and the internet as we know it today—composed of multiple backbones operated by different service providers and linked at inter-exchange points—started to take shape.[2] One of the most promising technologies of the time was Voice over IP (VoIP). The dominant protocol for VoIP in the mid-1990s was ITU-T's H.323. SIP appeared as a lightweight alternative to H.323 and soon gained mindshare and marketshare when the 3rd Generation Partnership Project (3GPP)[3] chose it as the signaling protocol for the IP Multimedia Subsystem (IMS) architecture in November 2000.

The choice of SIP as the signaling protocol for 3GPP/IMS certainly gave the protocol a boost in visibility, but the choice also implied that SIP, originally designed as a lightweight, peer-to-peer rendezvous protocol, would over time morph into a more carrier-grade protocol that supported centralized control of telecommunication services. Over the upcoming decade, telecom companies—equipment vendors such as Lucent Technologies (and its descendant, Alcatel-Lucent), Ericsson, Avaya, as well as service providers such as AT&T, Verizon, Sprint, Vodafone, France Telecom—gradually adopted SIP, jettisoning previous protocols like H.323. While incumbent protocols like H.323 were more complex than SIP was at the time, there was another reason for gravitating toward SIP: it was a traditional internet-style protocol with text (UTF-8) encoding, and therefore its use fit in well with the pantheon of other internet-style protocols like the World Wide Web and email.

In this chapter, we discuss our work on a SIP server written by Vijay Gurbani, one of the authors of this chapter, and the effort that propelled the SIP server

1 Henning Schulzrinne, Eve Schooler, Jonathan Rosenberg, and Mark J. Handley, "SIP: Session Initiation Protocol," IETF RFC 2543 (1999), *http://bit.ly/2JD1v76*.

2 Susan R. Harris and Elise Gerich, "Retiring the NSFNET Backbone Service," *ConneXions* 10, no. 4 (April 1996), *http://bit.ly/2JES7zG*.

3 3GPP is a collaboration between telecommunications service providers and vendors to develop a standardized, globally acceptable specification for the third-generation (3G) mobile phone network. The 3G mobile phone network was envisioned to use internet as the transport; voice would be packetized and delivered as data packets end to end. We are currently in the 4G era, with 5G looming ahead as a standard by 2020.

from a standalone asset used by one division in a large company (Lucent Technologies, Inc.) to a shared resource used by the entire corporation. We trace the evolution of the project that lasted eight years (1998–2006), and spanned the merger of Lucent Technologies, Inc., and Alcatel to form a new corporation, Alcatel-Lucent.

Vijay started the work on the SIP server as part of a research mandate on how the relationship between the two prevalent networks at that time—the incumbent circuit-switched network that provided the ubiquitous dial tone in people's homes and the new internet that was fast becoming equally ubiquitous—could lead to novel services. We describe the work as it proceeded in research divisions of the corporation, including Bell Laboratories, the R&D arm of Lucent Technologies, Inc., and later, Alcatel-Lucent, and highlight its contributions to what is now known as InnerSource. The asset that we created was intrinsic to the core business of the company, which was to manufacture telecommunication devices (gateways, firewall devices, base stations, application servers) and foster innovative services over the newly emerging VoIP network. The telecommunication evolution toward IP was inexorably tied to SIP, as it was the canonical protocol spoken between the network devices and software applications. In our work, we characterized this phenomenon as Corporate Open Source and established a taxonomy of projects under it.[4,5] Our taxonomy (briefly explained in Chapter 1) classified the corporate use of Open Source in two classifications: a lightweight *infrastructure-based* InnerSource and a more rigorous *project-specific* InnerSource. In this chapter, we further describe the latter to examine lessons learned and insights from our work.

As documented by the chapters in this book, InnerSource is used widely within corporations. However, we believe that our work, as described in this chapter and related literature,[4,5,6] represents an early experiment in using large-scale InnerSourcing to effectively create a software platform that was endemic to the core business of the company (telecommunications signaling) and intimately tied to the product strategy of the company. Many of the products that the company sold, and continues to sell, use the assets created as part of this InnerSource experiment.

4 Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb, "Managing a Corporate Open Source Software Asset," *Communications of the ACM* 53, no. 2 (2010): 155–159.

5 Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb, "A Case Study of a Corporate Open Source Development Model," *ACM International Conference on Software Engineering* (2006): 472–481.

6 Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb, "A Case Study of Open Source Tools and Practices in a Commercial Setting," *5th ACM workshop on Open Source Software Engineering* (2005): 1–6.

SIP: A Brief Background

Here we provide a very brief background on SIP to help readers appreciate some of the intricacies and complexities of the protocol that can clarify decisions made at a later stage of our effort.

SIP is a multimedia signaling protocol. Its job is to find the intended recipient with whom a multimedia session is to be established by routing a SIP request over a SIP network. More generally, one can think of SIP as the underlying protocol when a mobile phone subscriber dials a number and reaches his or her intended recipient; the protocol interprets the digits that constitute the dialed number and routes the session request to the recipient. The session established is not limited to a voice session, and neither are phone numbers the only way to indicate a session recipient. SIP is versatile enough to set up video, gaming, and instant messaging sessions. Besides phone numbers, it can route requests to email-like identifiers such as "*sip:vkg@acm.org*," called SIP URIs (Uniform Resource Identifiers). In fact, most people use SIP already even if they are not explicitly aware of this: when they dial mobile phones, or phones in an enterprise setting, or VoIP home phones, the network uses SIP to route multimedia session requests. SIP is also used in Apple's Facetime application and serves as a signaling protocol for the Jitsi application from Atlassian enterprise productivity software.

The SIP network is composed of the following key roles, each requiring a complete implementation of a SIP stack:

SIP user agents (UAs)
: A user agent is a SIP endpoint, i.e., it is the artifact that a human user interacts with to set up a session, receive notification of an incoming session, or tear down an existing session. Examples of SIP user agents are a mobile phone and a VoIP application.

SIP registrar
: A SIP registrar is a network server that stores a mapping from a SIP URI to an IP address where the recipient UA can be found.

SIP proxy
: A SIP proxy is a network server that aids in the routing of SIP messages. Under normal circumstances, a SIP UA indicates interest in setting up a session to its nearest proxy server. The proxy server then routes the request downstream to the next proxy server responsible for the recipient's address of record. A SIP proxy can be logically viewed as a conjoined UA server and a client. As a server it receives requests from an upstream client, and as a client it sends requests to a downstream server.

SIP redirect server
: A SIP redirect server is a SIP UA that redirects incoming session requests to an alternate set of URIs.

SIP back-to-back user agent (B2BUA)
: A SIP B2BUA is a conjoined UA client (can initiate session requests) and UA server (can accept, or reject, or redirect session requests). B2BUAs are used in service provider networks to implement certain centralized services. The difference between SIP proxies and B2BUAs is that the latter maintain detailed session state that the former does not; a B2BUA may also be privy to the media stream that a proxy is normally not concerned with.

A SIP session is established through a three-way acknowledgment similar to that used by TCP. It starts when a UA client sends a request called an INVITE. The INVITE request is routed from the UA client to its nearest proxy, and from there the request is routed over the SIP network to the proxy responsible for the recipient's domain. Each intermediary SIP server that handles the request will subsequently be in the path of the response. When the request reaches the recipient's domain, the proxy consults the SIP registrar responsible for the domain and forwards the request to the recipient UA server. The server optionally issues a series of 100-class responses that are known as *provisional responses* because they attempt to set up a session. The provisional response is followed by exactly one final response, which (similar to other internet protocols such as HTTP and SMTP) can be chosen from the classes 200 through 600. The final response is routed over the same set of SIP intermediaries that the request traversed to get to the UA server. Upon receiving the final response, the UA client sends an ACK request, thereby finishing the three-way SIP session setup handshake. Other SIP requests (like BYE and REGISTER) do not require an ACK. This grouping—sending an INVITE request, receiving one or more provisional responses followed by a final response, and then sending an ACK—constitutes a SIP transaction. (A non-INVITE request will not have an ACK as part of its transaction.) Every SIP entity described requires a SIP transaction layer to handle the protocol machinery that sends out requests and responses, performs retransmissions, and so on.

The transaction layer is, essentially, the heart of SIP processing. Every SIP entity has one, and every SIP entity's transaction layer behaves differently depending on its role in the SIP ecosystem. A SIP UA client's transaction layer is responsible for routing requests to the UA server and matching incoming responses to outstanding requests. A SIP UA server's transaction layer must be prepared to handle requests that arrive over one of the multiple transports supported by SIP: UDP, TCP, TLS, DTLS, and SCTP. It must also send provisional and final responses to the outstanding requests. A proxy's transaction layer is more complex because it accepts requests as a UA server and routes those requests toward the intended UA server while acting as a UA client. In SIP, a proxy may fork an incoming

request to multiple downstream branches, so a proxy's transaction layer must be prepared to match incoming responses and ACK requests to the right branch. SIP transaction handling can get arbitrarily complex given the rules of SIP processing and support for multiple transports.

The Project: The Common SIP Stack (CSS)

Bell Lab's SIP implementation project lasted for a period of eight years, from 1998–2006, and proceeded in three phases, as described in the following subsections.

Phase 1 (1998–2000): The Foundational Years

In 1998, SIP was not yet a standard. It was proceeding through the process to become a standard in the Internet Engineering Task Force (IETF), the main standards body for protocols that run on the internet. The IETF process defines a series of intermediate protocol documents called Internet-Drafts (I-Ds), which are successively refined until a critical mass is reached for the I-D to become a standard document called Request for Comment (RFC).

Vijay's home R&D department was, at the time, exploring the interplay between this new protocol and the rest of the traditional telephone infrastructure. Thus, he started to write the code that would eventually lead to a standards-compliant SIP proxy server and a SIP registrar. Vijay attended SIP working group IETF meetings, where the protocol was successively refined. He participated in the 4th, 5th, and 6th SIP bakeoffs (see the sidebar "SIP Bakeoffs" on page 35) held in April 2000, August 2000, and December 2000, respectively. By 1999, SIP had become an RFC,[7] although work on the iterative IETF model of refining the protocol continued until a second SIP RFC,[8] which was released in 2002 and is still in force. Because of Vijay's sustained participation in the IETF and the iterative nature of protocol development in the IETF, the SIP code benefited greatly from the insights and experience gained by developing protocols in the consensus-based approach epitomized by the IETF.[9] By the end of 2000, Vijay had produced an RFC-compliant SIP proxy server that interoperated with other vendor

7 Schulzrinne et al., "SIP: Session Initiation Protocol."

8 Eve Schooler, Jonathan Rosenberg, Henning Schulzrinne, Alan Johnston, Gonzalo Camarillo, Jon Peterson, Robert Sparks, and Mark J. Handley, "SIP: Session Initiation Protocol," IETF RFC 3261 (2002), *http://bit.ly/2yaZNnU*.

9 IETF's guiding credo is "rough consensus and running code;" practically speaking, this means that anyone writing an IETF protocol specification has all the rights and privileges to seek consensus on changing the protocol behavior if he or she can technically demonstrate that a portion of the protocol does not work as expected, or the portion can be improved and he or she can quantify that improvement in a technically unambiguous manner.

implementations, and in fact, was used to find bugs in many vendor implementations.[10]

SIP Bakeoffs

IETF organized a series of bakeoffs, or interoperability events, where implementers brought their SIP assets and tested them against each other. Bugs in implementations were fixed in real time. The interoperability test suite consisted of both simple (three-way SIP handshake) and complex (forking, response aggregation) SIP operations. The bakeoff allowed the IETF to focus on problematic areas with the protocol that could be better specified, or even completely replaced by an alternative, when the next revision of the I-D came out. The IETF maintains a list of issues found at each SIP bakeoff.

At this time, the reach of the SIP assets developed by Vijay was mostly limited to his home business division; ideas were crafted around the SIP proxy server and subsequently productized, but the source code was mostly available only within Vijay's department. He was the primary developer working on the SIP codebase during Phase 1, with the home department providing the partial time of 1–2 additional developers to work with him toward the end of Phase 1.

With the burgeoning acceptance of SIP by 2000, Vijay started to contemplate sharing the SIP code widely within the company. Two conditions in the company and the larger environment prompted his decision: first, his management was supportive of Open Source in general, including the need to share the codebase. Second, as SIP became the mandatory signaling protocol in 3GPP, it became the lifeblood of the various products being sold by the company (at that time, Lucent Technologies). Given that the telecommunications ecosystem was steadily moving toward using SIP as the core signaling protocol, it was evident that most telecommunication equipment would use it in some form: line cards would use SIP, telephony gateways would use SIP, as would phones and even software VoIP applications.

The stage was thus set for Phase 2.

Phase 2 (2001–2003): Opportunistic Partnering

During this phase, the SIP code started to be shared widely and found its way into the critical path of projects such that a more organized approach would be

10 The need for other vendors to interoperate their assets with a standards-compliant SIP server was so acute that Vijay ran the SIP proxy server on the DMZ of the company's network so external vendors could use his server to test their implementation's interoperability with the specification.

needed (Phase 3). But before we go into Phase 3, several consequential steps occurred in Phase 2, which we describe here.

First, the code started to be shared widely by Vijay with the rest of the company. A website was established from where the technical community could download pre-compiled binaries of the SIP server and registrar, or they could download the source code. The source code was written to be portable across the Linux operating system and Solaris. The only requirement that Vijay imposed on the product groups and individual researchers for using the code was that they feed back the bug fixes to him as well as any new features developed using the source code as the base, a provision similar to the well-known GNU GPL. The code started to benefit in three very important ways:

Linus's law
: "Given enough eyeballs, all bugs are shallow." This ensured that the code was peer reviewed and tested by other researchers and developers.

Contribution of experts
: The code was already architected to execute on multiple processors to meet the needs of concurrency (using the thread pool software design pattern) and scalability (using the dispatcher software design pattern); ideas on making these better started trickling in. The code started to benefit from the collective wisdom of the deep R&D culture prevalent in the company, especially the tips, techniques, and strategies that permeated the company with respect to telecommunications signaling. For example, developers across the company who were experts in performance would offer heuristics and tricks on using atomic instructions for certain operations to help with concurrency. In one instance, a researcher offered insights into high-performance parsing strategies as SIP is computationally expensive to parse. Developers and researchers versed in security would offer insights on better techniques for the cryptographic hashing algorithms used in SIP digest authentication. Yet other developers contributed an asynchronous approach to Domain Name Service (DNS) queries, so that a thread did not block for a DNS response to arrive.

Enhanced code portability to heterogeneous platforms
: The code was ported to other platforms beyond Linux and Solaris; for instance, developers ported the code to the Windows operating system and contributed the changes to the mainline effort.

Second, and perhaps just as important as the first step, was that the ratification of SIP by 3GPP as the mandatory signaling protocol drove the business divisions to seek a SIP stack upon which to build their products. Consequently, Vijay and his management started to reach out to the business divisions to acclimate them to the SIP asset, and to share the experience gained by contributing to the core protocol development in the IETF and attending the bakeoffs. With Vijay becoming

the de facto "SIP expert" in the company, different business divisions started to use the source code developed by him.

Requests started to trickle in for establishing a *framework* approach to the source code. As a SIP proxy, the transaction layer was embedded in the behavior of the proxy. Consequently, the code was refactored[11] to separate the logical behavior of a SIP entity (proxy, registrar, etc.) from the transactional handling required by the entity. Because all SIP entities operate on the notion of a transaction, the code was refactored to extract a transaction library called *siptrans* that handled all the transaction-related complexities, including forking and heterogeneous transport support. *siptrans* retained the concurrency, scalability, and security features that were put into the codebase earlier, but added an event-driven model where the framework would raise an event when a certain message arrived at the transaction layer. The application writer would register a C function callback, which would be invoked by *siptrans* to service the event. With *siptrans*, all a new project or user that wanted to use SIP had to do was to implement the logical behavior of the SIP entity using the transaction library.

Third, official change requests from product groups started to arrive; while Vijay provided support for interested users and product groups as they came in, the assets had organically grown and dispersed within the company to an extent that *siptrans* became a critical path for many businesses. These product groups sought organized support as they integrated the code into their development environment and sold products based on it.

Phase 3 (2004–2006): Corporate Open Source

By this time, the SIP assets were feature-rich and mature. The SIP stack contained many features and additions being standardized by IETF. The code had, by now, been taken to three additional SIP bakeoffs: 7th (March 2001), 9th (December 2001), and 11th (October 2002). By the 11th bakeoff, Vijay felt that the utility of participating in them decreased based on the metric of number of changes required in the base SIP stack to be compliant; there simply weren't many changes required anymore because the core of *siptrans* was stable in terms of protocol conformance.

The Common SIP Stack group

Meanwhile, the various projects that had used *siptrans* continued to contribute the changes back to the master source. Because the project had matured, the

11 Refactoring of the code was done before, albeit in a limited manner. Because all SIP entities need to parse SIP messages, an early change request was to create a standalone SIP parser library. This library was subsequently used by many projects and assets in Lucent Technologies, including the 5ESS Digital Switch and the Alcatel-Lucent OmniSwitch 9900 Wireless Network Guardian.

changes being contributed now were more than software patches; instead, entire features were being added to the SIP stack. In addition, other product groups wanted to use the SIP assets but were hesitant to invest in the integration of the source code in their development environment without considerable support from the base SIP team, which consisted of Vijay and 1 to 2 developers helping out on an ad hoc basis. The need for organized support led to the creation of a group that would coordinate the code drops to the different organizations and projects and provide support for these groups to start using the shared asset. This group was called the Common SIP Stack (CSS) group and was headed by Anita (another author of this chapter), working in conjunction with Vijay. At the same time, Vijay and Anita started an academic collaboration with James Herbsleb (the second author of this chapter) to document how the asset was being used across the corporation, and more importantly, to understand how we applied traditional Open Source development techniques and tools to a corporate environment.[12][13][14]

The CSS had two goals:

- Maintain an independent and common source code repository such that all projects take their deliverable from CSS.
- Evangelize the technology and the implementation by creating awareness of the resource within the company.

To do these things, the division coordinating SIP assembled a core team and created a SIP Center of Excellence (COE), which was envisioned as a one-stop shop for SIP assets in the company. The SIP COE was a company-central website from which other product groups and projects within the company could obtain information on the shared asset and instructions on how to download, compile, and execute the source code. Furthermore, it had a release schedule for the upcoming version of the SIP stack, as well as links to bug reports and developers assigned to each bug.

The core team

The core team consisted of three individuals in anchor roles and a dynamic population of developers, a quality assurance team, and trusted lieutenants as described next. The three anchor roles were composed of the product manager (and liaison, a role played by Anita), a chief architect and benevolent dictator (a role played by Vijay), and a project manager who was responsible for release schedules and feature arbitrage.

12 Gurbani et al., "Managing a Corporate Open Source Software Asset."

13 Gurbani et al., "A Case Study of a Corporate Open Source Development Model."

14 Gurbani et al., "A Case Study of Open Source Tools and Practices."

The primary responsibility of evangelizing the product fell on the product manager and the chief architect; evangelization of the technology was a multipronged affair. The product manager coordinated with the product groups that wanted to adopt the code in order to convince them of the utility of using an in-home, best-of-breed SIP asset owned by the company. A primary issue here was whether a product group would help defray the costs of developing some new feature that was specifically needed by that group. The defrayal could be through in-kind contribution (by lending a developer to the CSS to code, integrate, and test the feature) or through some monetary contribution. All such interactions were handled by the product manager.

The chief architect and benevolent dictator helped evangelize the asset by focusing on the technical contributions of the asset compared to third-party SIP stacks, many of which the chief architect had encountered as part of the SIP bakeoffs. The chief architect also made a presentation to the senior leadership team of the company to convince them of the advantages of having a homegrown SIP asset, namely:

- A feature-rich, standards-compliant server developed by in-house experts knowledgeable in signaling, security, scalability, and reliability
- No licensing encumbrances
- No dependency on a third-party vendor's release schedule, and thereby more control on planning and releasing products
- Strategic advantage in being at the forefront of developing and deploying the protocol

The presentation was quite successful, because in February 2006, an email message went out from the then president of Bell Laboratories, Alcatel-Lucent to the R&D community of the company asking that each product group first evaluate the internal SIP asset before requisitioning a third-party SIP stack. The chief architect also established communication channels with the company's supply-chain management (SCM) that allowed SCM to redirect third-party SIP stack requisition requests to the SIP COE.

Trusted lieutenants

Besides evangelization, the chief architect was involved in two other important tasks: working with trusted lieutenants to add new features in the code, and architecting the SIP code such that it served multiple constituencies.

The chief architect worked closely with a set of trusted lieutenants, individual developers who spearheaded nontrivial features in the codebase. For example, one new feature that was added in the code had to do with signaling compression (sigcomp). The process of sigcomp compressed the signaling messages at the

sender side and decompressed them on the receiver side. The product group that wanted this feature contributed a developer to the CSS who worked closely with the chief architect to seamlessly engineer the compression and decompression in the appropriate threads that received and transmitted the messages.

Another example involved a trusted lieutenant for high-performance SIP parsing. This phenomenon of key developers that rise through a meritocracy to become responsible for substantial portions of the software has been observed in both Open Source and InnerSource projects.[15] In our project, it exhibited itself on multiple occasions with an interesting outcome: some of the developers who were volunteered by their home product group to the CSS to finish a feature rose to become trusted lieutenants within the project and expressed interests in staying on in the CSS project even after the feature was completed.

By this point in its trajectory, SIP as a protocol was evolving rapidly as new extensions were being added into the protocol by IETF and 3GPP; between the issuance of RFC 3261 in 2002 and the end of Phase 3, more than 80 extensions were proposed to the protocol.[16] The products built on the SIP stack were targeted mostly to the national (and international) service providers,[17] who were interested in the 3GPP extensions being added to the protocol. With the acceptance of SIP for use by 3GPP, the protocol morphed from an inherent peer-to-peer rendezvous protocol to a carrier-grade telecommunication signaling protocol that supported centralized control of telecommunication services. Individual researchers and a few product groups were, instead, keen on vanilla RFC3261 SIP assets (i.e., not encumbered by 3GPP extensions).

An immediate consequence of this was that the codebase had to be modular and interfaces had to be well defined, such that 3GPP extensions could be added in without perturbing the mainline RFC3261 behavior of the protocol. Furthermore, these extensions, once added, were not used uniformly by all product groups. Depending on the specific SIP role (user agent, proxy, B2BUA) that a product group would be developing into products, certain extensions would need to be pulled in while others left out. Thus, architecting the system to make it modular, agile, and applicable to both constituencies was an early requirement.

15 Klaas-Jan Stol, Paris Avgeriou, Muhammad Ali Babar, Yan Lucas, and Brian Fitzgerald, "Key Factors for Adopting Inner Source," *ACM Transactions on Software Engineering and Methodology* 23, no. 2 (2014).

16 Jonathan D. Rosenberg, "A Hitchhiker's Guide to the Session Initiation Protocol (SIP)," RFC 5411 (February 2009).

17 Service providers are the companies that (e.g., AT&T, Sprint, Vodafone, etc.) provide wireless network service to the users.

A growing community

In this phase, the size of the development community around the SIP asset increased dramatically, with 30 developers and testers working on the asset. As the asset became more widely used, it became apparent that certain roles were needed to ensure a smooth delivery of the codebase from the CSS to an individual product group:

- Construction, verification, and load bring-up engineers who worked directly with the product group to provide support for release management tasks.
- A release advocate to ensure that the code changes for all features were submitted on time, and to keep track of all business division–specific impacts for the particular release.
- Delivery advocates, assigned to each business division that intended on using the common asset but was new to the concept of InnerSource. They helped in the surprisingly difficult task of build integration of the SIP assets into the infrastructure used by the business division.
- A feature advocate to see a particular feature to completion.

The delivery advocate played a crucial role: he or she served as a bridge between the CSS and the specific product group. The delivery advocate was assigned to the product group to ensure that the SIP asset could be integrated into the specific tools, processes, collected lore, and compilation dependencies of the product group. Furthermore, the delivery advocate worked with the particular product group to ensure that its contributions to the common asset were assimilated in a manner conducive to the architecture of the common asset. The delivery advocate was the subject matter expert in load building of the common asset and had to understand the requirements and business infrastructure of the product group to successfully integrate the common asset into the processes of the product group.

A member of the core team was also assigned as feature advocate to shepherd a particular feature to completion. In this role, the feature advocate approves design documents, performs code inspections, and ensures that the change aligns with the overall software architecture. While the delivery advocate handled the load building and integration issues, the feature advocate worked closely with the product group to specify the particular feature of interest to the product group. Often, the feature advocate would work closely with the developers of the product group to specify and implement the new feature in a manner consistent with the modular architecture of the common asset. We believe that this pattern of assigning specific roles for an InnerSource project that is intimately tied to a product strategy of the company will endure as more corporations use InnerSource. All but nontrivial InnerSource projects will require the core team to act as shepherds to foster the adoption and integration of the common asset within a

product group, leaving the product group freer to focus on its specific business needs.

The roles of release, delivery, and feature advocates rotated among the developers working in CSS to allow them a breadth of experience managing a software release life cycle. On a cursory look, it appears that these roles are associated more with traditional software development than the agile, lightweight, and process-free methodology for developing Open Source software. However, as we argue in just a moment, these roles were essential for our InnerSource project to succeed.

Reflections, Insights, and Discussion

Our work was an early experiment of a large-scale InnerSource project that exhibits one crucial difference from other InnerSource projects. At the time we were conducting our experiment, early concepts of InnerSourcing were being used by other corporations but in a limited fashion: their visibility and effect were limited to a project or a department. The software assets we created, by contrast, were intimately tied to the product strategy of the company.

This reality led to an important insight from our work: for an InnerSource software asset that is tied to the product strategy of a company, it does not suffice to simply hand a product group an archive file with instructions to compile and install the source code. The processes for work assignment, feature prioritization, and product deliverable planning are optimized within that product group to match its business structure. Furthermore, different product groups have specific affinities for build- and bug-reporting tools and source code control systems. All of this makes it infeasible to simply "toss the source code over the wall" and expect that a product group will integrate it into their environment without any support. The roles of delivery and feature advocates evolved primarily to deal with the complexity of differing expectations and varied tool sets used by product groups.

Advertising and Encouragement

The chief architect and the product manager should always be on the forefront of ensuring that the common assets are known widely within the company. This can be accomplished by establishing alliances with SCM and educating the sales and marketing teams on the benefit of having a best-of-breed in-house software asset. Beyond that, another insight from our work was the need for a top-down approach to help foster the use of the common asset. During Phase 1 and 2, Vijay's department leadership team encouraged the sharing of the asset within—and even outside—the department. During Phase 3, such encouragement arrived from the senior leadership team of the company in the form of an email message by the then president of Bell Laboratories. Without such momentum and encour-

agement, common assets will have limited impacts. The technical staff of a corporation can create common assets and ensure that they are the best-of-breed (the bottom-up approach), but at some point, an inflection will be required from the top to elevate these projects within the company.

Establishing alliances with SCM and garnering top-down encouragement is admittedly easier to do when the common asset is tied to the product strategy of the company, but even if that turns out not to be the case, department-wide or product-group wide sponsorship and championing is essential for the asset to succeed.

The SIP Asset: By the Numbers

We conclude this section with a brief objective look at the numbers behind the project. Figure 3-1 shows the software release frequency by the year, starting from 2000–2001, when the project entered Phase 2. The number of releases accelerated during Phase 3 as there were many developers doing active and parallel development to the software while delivering it to multiple business divisions.

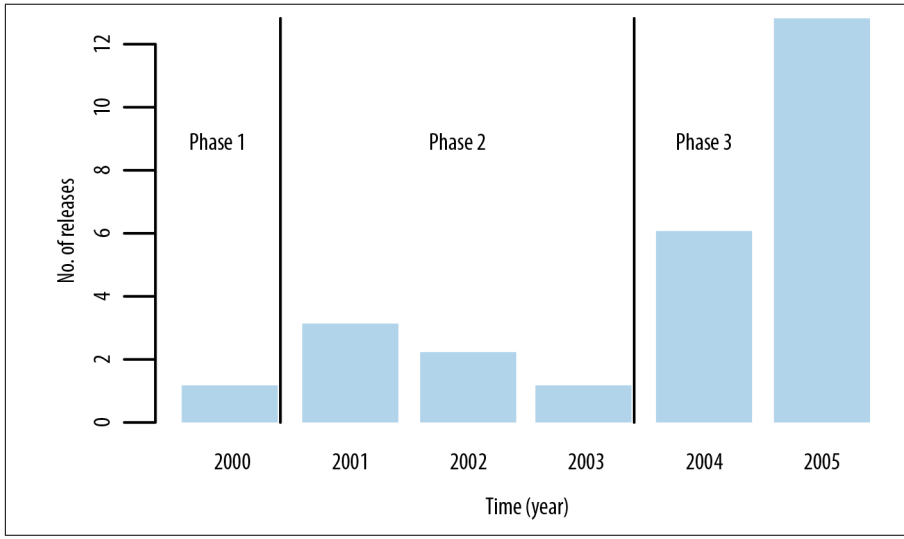


Figure 3-1. Software release frequency

Figure 3-2 shows the number of unique downloads during each phase—we consider only the first time a new project or business division downloaded the software, with no counting of subsequent downloads by the same project or business unit. Here, Phase 2 witnessed the most downloads: 87. Phase 3 witnessed about half the number of downloads (40). This could be attributed to the elevation of the software from being downloaded by individuals to being used primarily on a project-wide basis by each business division.

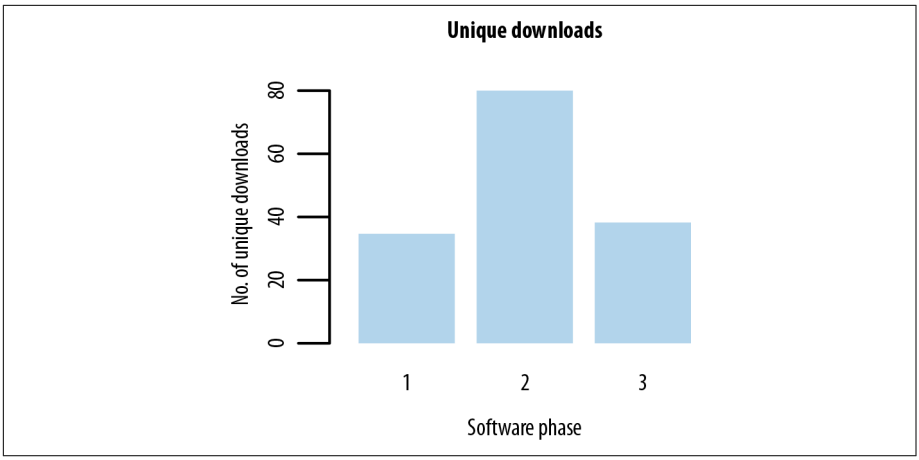


Figure 3-2. Unique software downloads

Figure 3-3 shows the evolution of the codebase across the phases in terms of normalized lines of code. We define normalized lines of code as the subset of the source code tree that is required to compile the software base completely. Specifically, this count does not include comments in the code, nor does it include all the support software that was built in parallel to test the functionality of the server.[18]

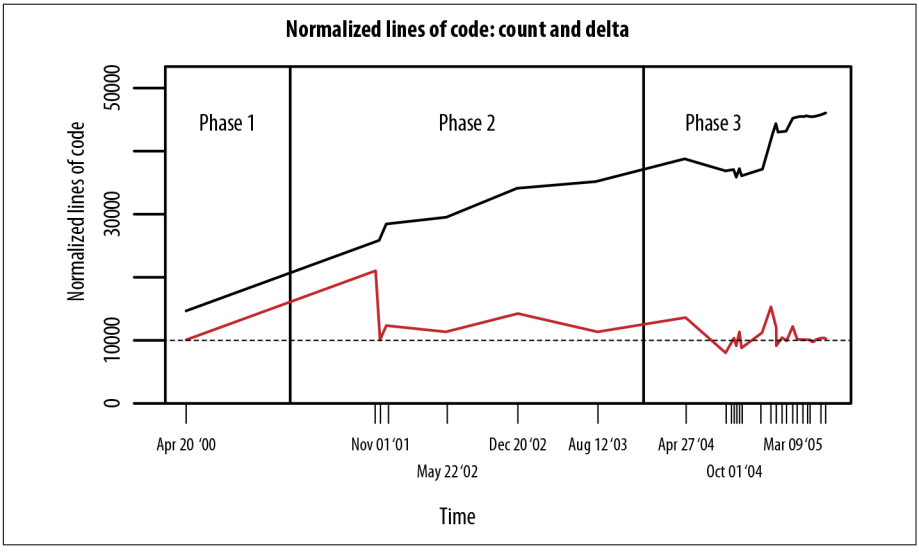


Figure 3-3. Normalized lines of code

18 Gurbani et al., "A Case Study of a Corporate Open Source Development Model."

The top curve in Figure 3-3 corresponds to the increase in the size of the code across the phases; by and large, it has a positive slope reflecting the constant addition of new functionality. The curve at the bottom of the figure tracks the changes in the lines of code (the delta) as the software progressed. The largest positive slope of the delta is between April 20, 2000, and November 1, 2001, which constitutes the formative stage of the software during which the asset was evolving to serve a larger set of audiences, including various business divisions.

Looking Back

With the passage of time, it is gratifying to reflect that our work has proved to be foundational for the larger InnerSource community[19] in three ways: first, it served as one of the initial detailed and successful case studies of InnerSource,[20] second, it presented the first known taxonomy of classifying InnerSource projects as project-based or infrastructure-based (see Chapter 1), and finally, it identified a set of roles to formalize an InnerSource core team.[21]

Our success is due to many factors, some that we controlled and others that acted as controlling influences on us. Certainly, we were lucky to be at the cusp of a technology (internet telephony and VoIP) that was poised to extend its reach as networks became faster and far more pervasive. However, luck alone does not account for the success we had with our work in creating a shared software asset and the associated artifacts like the SIP COE, whose main objective was to propagate and evangelize the software in various business divisions of the company. Business divisions in a large corporation are almost autonomous entities with their own budget, vision, headcount, and development methodologies. The asset we created was deemed fundamental to the operation of many business divisions within a company whose primary output was to manufacture communication products that used a canonical, standards-compliant communication protocol. To effectively do so, there was simply an advantage to nurturing core expertise across the company that allowed it to create and leverage a best-of-breed, internally developed SIP stack.

Acknowledgments

Vijay would like to thank Warren Montgomery and Jack Kozik for providing a department where sharing of code and serendipitous experimentation was the norm, the absence of which would have limited the success of the project. The

19 Maximilian Capraro and Dirk Riehle, "Inner Source Definition, Benefits, and Challenges," *ACM Comput. Surv.* 49, no. 4 (2016): 1–36.

20 Gurbani et al., "A Case Study of Open Source Tools and Practices."

21 Gurbani et al., "Managing a Corporate Open Source Software Asset."

authors also thank the many contributors to the project, including developers and testers and all others who collectively ensured the success of the project. The authors gratefully acknowledge support from NSF grants 1546393, 1633083, the Alfred P. Sloan Foundation, and the Google Open Source Program Office.

CHAPTER 4

Living in a BIOSphere at Robert Bosch

With Georg Grütter, Diogo Fregonese, and Jason Zink

Year, place founded: 1886, Stuttgart, Germany

Area(s) of business: Automotive, consumer goods, industry technology, energy and buildings

Revenues: ca. €78.1 billion (2017)

Offices: 440 subsidiaries in 60 countries, 125 R&D locations

Number of employees worldwide: 402,000

Number of software engineers: 20,000

Year of InnerSource adoption: 2009

Robert Bosch GmbH is a large corporation active in areas ranging from mobility and industrial solutions to energy and building technology and consumer goods. Like any true startup, Bosch was founded in a garage; it all began in Stuttgart, Germany, in 1886. Since then, Bosch has grown to over 400,000 employees, called "associates," who are distributed across 120 research and development offices spanning five continents. Due to the extremely distributed nature of the company, Bosch, like any other distributed organization, faces challenges in achieving collaboration across business units (often called "silos"), across different locations, and across different time zones. For such a large company to remain successful, it is essential to facilitate and improve collaboration. In pursuit of that goal, two associates in corporate research who were interested in free and Open Source software started playing with the idea that the Open Source development model could address many of the issues associated with distributed development. And so the seed for the Bosch Internal Open Source (BIOS) initiative was planted.

Why InnerSource

Being a large organization, the company owns many assets such as buildings and machines, but what is far more important is the knowledge base that the company has built up over the years: sharing this knowledge among its associates is key for the company to innovate and continue to prosper. Like any company, Bosch continuously aims to improve its processes to achieve better efficiency. The BIOS initiative fit well with that goal. The company saw a number of benefits of the Open Source development model, and hoped to achieve these through the BIOS initiative:

- InnerSource helps to increase the efficiency of distributed software development, by fostering collaboration across different business units, and to increase knowledge sharing.
- InnerSource offers an internal training ground for staff to learn how to participate in Open Source communities, which is becoming important for any software organization.
- InnerSource as a modern approach to software development helps to attract talented software developers. Bosch is moving to become an Internet of Things company, and as such it depends on being able to hire new developers.

Further benefits were discovered as the BIOS project succeeded. In particular, we found that developers were more innovative because they could develop software for other teams without requiring approval and resources from those teams' managers.

Starting the BIOS Journey

Like many other large companies that are distributed across the globe, Bosch has experienced the common challenges caused by geographic and time distances, which can make software development processes quite inefficient. The stories of several hugely successful Open Source projects—the Linux kernel being a prime example—which mastered the challenge of distribution did not go unnoticed. These successes inspired a few associates to study how Bosch could benefit from the lessons learned in Open Source software development, not just internally but with the ultimate goal of making Bosch a successful player in the Open Source arena itself. At the time, Bosch didn't have any expertise or experience with engaging in existing Open Source projects and communities, or starting new ones. Instead, they started a number of other Open Source–inspired initiatives, including a wiki to share knowledge, an issue tracker, and an internal research project which, among other things, explored how software development within Bosch could benefit from adopting Open Source development practices: the

Bosch Internal Open Source (BIOS) initiative. The prospect of increasing the efficiency of distributed software development was what convinced management to support this initiative.

The BIOS initiative was started in 2009 and has evolved since then. We can identify roughly two main phases.

Establishing and Growing the BIOSphere

The first phase started with the official sponsorship of the BIOS initiative by the company's executive management in 2009. In this phase, the BIOS initiative was positioned as an "experiment," and was put under the stewardship of Corporate Research, who were in charge of the overall research project. As such, they managed the budget and bore final responsibility. The initial experiment started very small, with the specific goal to evaluate whether the application of the Open Source development paradigm could help overcome the challenges of distributed development, and in particular, help to overcome any bottlenecks to efficiency. For the first years, up to the end of 2015, "entry" of new communities was guarded by a formal review committee. The number of communities it could approve was limited by the committee's budget for funding community leaders and contributors.

This first experiment booked a number of successes (which we will describe), eventually leading to an extension in 2012 that provided another three years of funding for the BIOS initiative under the same conditions. The original question as to whether Open Source development practices could be leveraged within Bosch was answered positively, so the program left the research stage. The initiative's stewardship changed from Corporate Research to the Corporate Engineering Department as a result.

BIOS Values

The BIOS initiative was originally set up as a bubble within the company, a "safe space" in which to apply and experiment with the Open Source working model. We called this safe space the "BIOSphere." Rather than focusing on Open Source tools and technologies, the main focus was on re-creating the culture that underpins Open Source development. Within BIOS, the culture is based on five values:

Openness
: We make it easy for any associate to join and contribute to a BIOS community. We lower the barriers for entry into BIOS communities as much as we can.

Transparency
: We aim to be radically transparent and share our work products, our communication, and our decision making with all associates in the company. So,

while openness is about getting people in, transparency is about being sure that all product and process artifacts are accessible.

Voluntariness
: The decision to join and contribute to a BIOS community is left to each associate. Associates should work within BIOS because they are intrinsically motivated, not because their manager told them so.

Self-determination
: Associates are free to choose what to work on, which tools and processes to use, and when they work on the projects.

Meritocracy
: Power is vested in BIOS project members based solely on their merits, that is, based on the quality and quantity of contributions made.

These values stood in stark contrast with how Bosch ran other projects at the time. It was quite obvious to us that this would engender quite a bit of friction and risk undermining the "safe space." In order to protect the bubble, we created a protective layer around it by introducing two mechanisms that were instrumental for the success that BIOS eventually enjoyed: the BIOS Review Committee and the BIOS Governance Office (BGO).

BIOS Review Committee

The first mechanism was what we called the *Review Committee*, which comprised vice presidents of the engineering business units. This was the interface between the pilot BIOS communities and management. It served as a gatekeeper to the BIOSphere in terms of which communities would be a part of it, thereby giving managers a constructive way to influence what would happen within the bubble while avoiding micromanagement. The Review Committee also made management support for the BIOS initiative official and thereby provided the necessary executive air cover.

Any Bosch associate was able to propose a new community for the BIOSphere. The Review Committee would meet twice per year. One of these meetings was used to review and approve (or decline) new community proposals; the other meeting would be used solely to review active communities. Proposed communities are evaluated based on a set of clearly defined criteria—the "BIOSness criteria"—so as to ensure that accepted communities contribute to the aim of the BIOS initiative:

- A proposed community must have a clearly defined vision and mission in line with the overall mission of the BIOS initiative. This served two main purposes: first, it ensured that the community goals were aligned with Bosch's overall goals and strategy, and second, it ensured that the BIOSphere

would be used only for communities that helped the BIOS initiative in its initial mission. (The initial mission was to evaluate the suitability and effectiveness of Open Source practices in the confines of the corporation; the mission evolved as the benefits of InnerSource became apparent.) We especially examined whether the community's topic was attractive for a large enough group of associates to increase the chances of actually receiving voluntary contributions.

- The community must have a full-time community leader so as to ensure that there was someone who would try to build a community and inspire others in the company to join. Having a clearly identified community leader would also help ensure the community would be viable and prevent hosting "zombie" communities that would simply take up resources but not contribute to the goals of BIOS.
- The proposed community must adhere to the five BIOS values described earlier.
- The community must be geographically distributed. Because the remit of the BIOS initiative was to evaluate how the Open Source development paradigm could help improve collaboration across locations, it was important to ensure that the space created for this purpose would be used with that aim in mind, rather than providing a playground for just any project.
- The community must be cross-departmental. Similar to the previous criterion, the aim of BIOS was to stimulate cross-departmental collaboration, and thus BIOSphere resources should be reserved for those communities that aim to achieve that.

The Review Committee also had the option to retire (or "sunset") a community, but that happened only once during the six-year period that the Review Committee was active, and in that particular case, the community itself chose to retire due to a lack of participation.

BIOS Governance Office

The second mechanism to keep BIOS on track was the BIOS Governance Office (BGO), which provided a legal framework within the organization and developed the ground rules for the BIOSphere, including the entry criteria discussed earlier. The BGO was responsible for the BIOS license, procurement of hardware and software, and contractings between communities and contributors wherever they were needed. The BGO also tackled organizational tasks, such as organization of Review Committee meetings, coordinating the overall communication between the various stakeholders (such as Bosch management and the community's leaders and developers), managing budgets, and documenting and maintaining a body of knowledge about BIOS. Finally, the BGO supported developers who were applying for a new community by helping them develop a community vision that

was aligned with the BIOSness criteria and, once approved, supporting them with building and evolving their respective communities.

The initiative had an annual budget of about 1.5 million euro, which was used to fund community leaders and contracted contributors. A small portion of it was used to buy hardware and software, specifically when it was not in the standard product catalog at the time. Such technology included development boards such as Raspberry Pis or BeagleBones, smartphones, smartwatches, and other gadgets. Because this was completely within control of the BIOSphere and purchases wouldn't have to go through standard processes, such purchases could be done much more quickly. All BGO tasks were handled by only a single person. Thanks to the BGO, the administrative overhead for developers was minimal, which allowed them to focus their limited time on development. There is a parallel here to Open Source projects; while many community members focus on contributing code, it is not uncommon for some members to support the community by looking after administrative work.

Attracting Contributors

The vast majority of developers working within business units are allocated to one or more projects, and as such their time is completely planned, with virtually no "slack time." In other words, normal developers tend not to have any time to do anything else beyond the work assigned by their managers within their teams. This also means that a proposed community, which would necessarily include a community leader (as one of the acceptance criteria) and perhaps also some developers from one or more business units, could find it difficult to get developers committed. The BIOS Governance Office could use its funding to reimburse a business unit for the engineer's time, effectively buying out the developer for 10, 20, and in some cases even 50 percent of their time. BIOS community leaders were funded 100 percent by the BGO. Because the BIOS budget was corporate money rather than money from a specific business unit, those bought-out developers would enjoy a great deal of independence—in other words, those developers would not be expected to serve their business unit, and instead would have complete freedom to self-direct.

Any developer time buyout and reimbursement arrangement with a business unit was supported by a formal contract. Formal contracts are a traditional mechanism for organizations that managers recognize, so this facilitated a smooth process for managers to sign off on those. While reimbursement of developer time can help convince a manager to let an engineer spend time in the BIOSphere, a more compelling reason is if the business unit stands to benefit from the community work directly—for example, by being able to improve their internal processes or productize it.

All communities in the BIOSphere publish their work products under a specialized BIOS license, which protects all BIOSphere developers from any liability claims. The BIOS License (BIOSL) was created by the BGO based on existing Open Source licenses, and later improved by the central legal department. The license clarifies what developers and business units *can* do with the software, what they *cannot* do, and what they *must* do. For example, it prescribes that all assets created by any BIOS community must be available to all business units within our company. This means that a business unit that did not contribute in any way to a BIOS community would still be able to "free-ride" and incorporate its assets into their products. Ultimately, a business unit is responsible for ensuring compliance, such as ensuring that intellectual property is protected or that OSS license obligations are met, thereby relieving the communities of this responsibility. This means, for example, that business units cannot publish any source code from BIOS communities outside of Bosch, and business units remain liable for their products (which includes any warranty claims), even if the product uses any BIOS assets. In effect, business units must follow the same procedures with BIOS software as they would for Open Source software components in such areas as licensing and accounting.

From BIOS to Social Coding

After the first six years, the BIOS initiative saw a number of changes, which we characterize as Phase 2. Before 2016, the BIOSphere was guarded by the Review Committee mentioned earlier and facilitated and funded through the BGO. At the end of 2015, a consensus formed in top-level management that BIOS was mature enough to continue without both, so they decided to dissolve both the Review Committee and the BGO. Effectively, this removed the "protective layer" around the BIOSphere. With the closing of the BGO, we also lost funding for the community leaders, who subsequently were not able to continue dedicating themselves completely to their respective communities anymore. On the bright side, without the Review Committee, everybody could now start a BIOS community without having to wait for and go through a formal vetting process.

Having experienced the value that BIOS brought to the organization firsthand and fearing that the lack of organizational support for BIOS would endanger what we had built, a group of former BIOS community leaders and the BIOS governance officer started a new bottom-up initiative to further drive the adoption of BIOS. After much lobbying with top-level management we eventually established the Social Coding initiative. The goals of this initiative are:

Changing the norm
: Drive the cultural change toward a more open organization in which the BIOS way of working will become the norm, rather than the exception.

Establishing infrastructure
: Provide a platform for our coders to collaborate across divisional boundaries.

Offering organizational support
: Help the organization use that platform and implement the new culture.

BIOS did not prescribe the use of any specific tools, following the principle of self-determination, but after a couple of years a consensus emerged that using a single collaboration platform would be beneficial. We started to use Stash and eventually migrated to Bitbucket, when the Social Coding initiative was established. Our main reason for using Stash and later Bitbucket was that Bosch had already heavily invested in the Atlassian tools suite.

Access to the platform was provided completely free of charge to any individual and department, unlike the standard infrastructure for which departments were charged monthly fees. By subsidizing the platform, we made sure that people who wanted to start a new community could do so at no cost. This meant that developers didn't have to ask their department managers for permission to get involved.

To convince business units that are hesitant to "open up," and to build a critical mass of developers, the platform now also allows "traditional" projects that were not open to the rest of the company—that is, non-InnerSource projects. Ultimately, we hope that those projects will convert to a true BIOS community by way of exposure to the new culture and by promoting existing BIOS communities as the "Gold Standard" and a role model for collaboration within Bosch.

Finally, and unlike many other companies, we didn't require projects or communities to use our platform. Instead, we opted to make it an offer to everybody to use at their discretion, because we think this is more in line with our values, especially self-determination.

Sustaining Social Coding

The year 2018 saw another change in the funding of the initiative. Both the BIOS and Social Coding initiatives were funded with corporate money, which kept the accounting overhead to a minimum. Beginning in 2018, the Social Coding initiative has to finance itself with business unit money. While we were successful in collecting enough funding for the Social Coding team by way of an internal crowdfunding campaign, the effort for running that campaign, and subsequently for the necessary internal accounting, turned out to be prohibitively high. This is why we decided to take the idea of community one step further, and push for replacing the central Social Coding team with a decentralized community of Social Coding advocates beginning in 2019. Every business unit that would like to hold a stake in (and influence the evolution of) Social Coding and the accom-

panying infrastructure will be encouraged to provide capacity to at least one Social Coding advocate to represent the business unit in the community. It remains to be seen whether that approach to sustaining Social Coding will be successful.

Success Stories

BIOS and Social Coding led to a number of success stories. Here we describe some of these.

Widespread Adoption

Within the time span of the first experiment and its extension (a total of six years), 11 communities were accepted within the BIOSphere, involving a total of 300 developers from 15 business units in 11 countries across three continents. As the BIOS initiative evolved into the Social Coding initiative, the number of users on our platform has increased from 300 to almost 8,000 in the past two years and is continuing to grow constantly. We now facilitate collaboration of developers in more than 150 business units in 28 countries. The total number of projects grew to over 600, almost one third of which are organized as BIOS communities—that is, they are accessible by all Bosch associates. The remaining two thirds of projects on the platform, however, are closed projects, accessible only by a specific business unit or project. Despite this, many business units that are running closed projects have adopted elements of the Open Source style working models that Social Coding promotes for their product development as well. Interestingly, the ratio of open versus closed projects on our platform approaches that of GitHub.

Diverse Ecosystem of Communities

The topics of communities varies greatly, ranging from production and engineering tools, to technology and product demonstrators, libraries and APIs, and new products. We develop throwaway prototypes and demonstrators very quickly, but also mission-critical software running in our production plants. Some examples include:

Teleheater
: An app for remotely controlling our HVAC systems, including the required infrastructure and embedded hardware. The system was presented as the flagship innovation at a major trade fair in 2011 and won an international design award for its interaction concept as well as an internal innovation award.

COM4T IP-Stack
: A networking stack for resource-constrained devices, allowing them to take part in the Internet of Things without requiring much of the devices' very precious and limited RAM. It is now used in a variety of IoT-related products —for example, the XDK Cross Domain Development Kit (xdk.io).

Tram Collision Avoidance
: A visualization for a collision avoidance system for trams building on Bosch radar and optical sensors used in the automotive domain and an API for accessing the CAN bus developed in the BIOSphere. The BIOS team rapidly developed the visualization in cooperation with the potential customer and was instrumental in eventually acquiring new business. The system is now being field-tested in many European cities by public transport authorities.

This type of innovation happened both in existing business units and in completely new areas that emerged spontaneously.

Improved Collaboration

At a technical level, the collaborations across different business units became smooth and frictionless. Previously existing technical barriers for this kind of collaboration were effectively removed.

BIOS communities were given complete autonomy with respect to how they performed their work. Initially, there was a general expectation that this would lead to a lack of formal planning and quality control, neglect of corporate processes, and a plethora of non-standardized toolsets in the BIOSphere which would ultimately lead to chaos. However, in fact the opposite happened: a strong culture of craftsmanship and apprenticeship emerged. BIOS now has a well-deserved reputation of producing high-quality software and many business units have modeled aspects of their development processes after processes common in BIOS projects.

Personal Growth

The BIOS initiative became a great success, not only for the involved business units, but also for the associates. Many associates reported they felt their personal growth accelerated dramatically through their BIOS activities. These associates consistently reported that the BIOS work in turn led to increased happiness and work satisfaction. The BIOSphere and its communities very clearly radiated a "coolness" factor, which attracted many excellent developers. Engineers were able to find a new purpose and renew the enthusiasm that they might have lost over the years.

Increased Productivity

The level of engagement and motivation of developers in BIOS communities were extraordinarily high. It is therefore not surprising that the productivity that BIOS communities achieved was far above average. The reduced administration work offered to developers by the BGO, and the permission to let developers adopt only processes that they felt were necessary and helpful, also contributed to that increase in productivity. As a result, a number of small teams and communities had a very disproportionate impact on the organization.

Alignment with Business

A key issue in adopting InnerSource is the potential disconnect between developer self-determination on the one hand and aligning work with the company's business objectives on the other. Interestingly, we found that this alignment happened quite naturally. Many BIOS projects have made direct or indirect contributions to reaching Bosch business objectives and this was recognized by our management. We also discovered that the vast majority of BIOS developers consider it to be the ultimate reward for their engagement if their software is used in a product or brings value to fellow associates by making their daily work simpler, more enjoyable, or more efficient.

Success Factors

In hindsight, we can identify a number of factors that led to the success of the BIOS initiative.

The timing of the BIOS initiative was good, because our management wanted to develop a strategy for Open Source around the time the BIOS initiative started. This was a major success factor for rallying support for the BIOS initiative and getting it off the ground. Although InnerSource is not the same as Open Source, InnerSource can help to create a culture that values transparency and meritocracy and thus help the organization to work with communities of developers. Furthermore, the BIOS initiative addressed an urgent need of the organization: to improve the efficiency of distributed software development in general.

BIOS started as an experiment with low expectations—and perhaps most importantly, it was *declared* as an experiment. Managers were more likely to sign off on it because it was time-limited and because declaring it as an experiment clearly communicated that failure was an acceptable option. For executive managers, such an experiment with a budget for a fixed time period is simply an investment with a risk. Without such limitations, managers are likely more wary that such an experiment might spiral out of control.

The experiment was completely funded by corporate management rather than a specific business unit. This helped to give the initiative the autonomy needed to

ensure that it could pursue a direction that wouldn't necessarily be dominated by any one business unit. By minimizing the influence of individual business units, the corporate-level initiative avoided political or budget-related conflicts that might hamper any cross-business unit collaborations. This is one of the key things that BIOS aimed to improve.

A second success factor was the focus on attracting self-selected and motivated "volunteers." This completely aligns with one of the values we defined previously: voluntariness. Perhaps most importantly (and most impressive), many contributors would contribute to these projects in addition to their normal, daily workload. This self-selection of highly motivated people naturally meant that their motivation was intrinsic and that they were highly enthusiastic. This in turn acted as a natural filter and allowed the recruitment of the most driven associates who were passionate about what they did.

The support rendered by the BGO was an important success factor as well. It unburdened the developers in the BIOSphere of the many administrative processes common in large organizations and thus allowed them to focus entirely on delivering their best work and make the most of the sometimes limited amount of time they had. More importantly, the BGO successfully managed to maintain the protective layer around the BIOSphere as a whole. As a result, the developers in the BIOSphere were able to implement the five values to the fullest extent, which contributed greatly to the extraordinarily high level of motivation and thus productivity we observed.

Another major success factor was the choice of leaders: the BIOS communities were led by enthusiastic and competent community leaders that were able to dedicate 100 percent of their time to establish, promote, and grow their respective communities. They were instrumental in attracting and retaining contributors, which is probably more difficult than in Open Source communities where the number of potential contributors is much higher to begin with. The importance of the 100 percent community leader became even more apparent when we lost the capability to fund them in 2016 and the number of contributions and overall productivity of their communities decreased significantly as a result.

Finally, giving the communities in the BIOSphere the autonomy to decide themselves what to work on and which tools and processes to use contributed to the success of the BIOS initiative. By using a dogfooding approach (i.e., the developers who created processes also used them), only those processes that really helped the community were implemented and continuously refined; everything else went out of the window. Communities also had and made use of the freedom to quickly act when a project opportunity presented itself. Chance favors the prepared.

Challenges

The BIOS initiative has been very successful, overall, but there were certainly also some challenges, some of which we have not yet been able to meet.

One of the primary challenges is to grow communities. It can be really hard to attract contributors from throughout the company when people are focused on their specific business units, and when their time is completely allocated to specific projects owned by that business unit.

We also found that getting buy-in and commitment from management, especially of middle management, was quite challenging. This is because many of the benefits of InnerSource—productivity as a result of developer happiness, employee retention, and personal growth, to name a few—are hard to quantify. Another, more systemic reason for the lack of buy-in is that mid-level managers simply have different goals and motivations, as they are responsible for only a slice of the company. Even if they buy into the goal of the whole organization, supporting cross-team collaboration and allowing their subordinates to contribute to business outside of their area of responsibility may not be in their interest, because offering developer time to outside projects can incur a cost to their own business unit's performance.

In terms of developer career path, one shortcoming of the BIOS initiative was that it was not coordinated with Human Resources. As a consequence, work performed within this initiative was often not rewarded in terms of advancing developers' careers. Time spent voluntarily on BIOS work was sometimes time not spent on reaching the goals set by their organization. Although several individuals have been able to leverage the increased visibility they enjoyed, there was no formal promotion track that was facilitated by HR policy.

We found it very challenging initially to get internal publicity for BIOS. Even after the first six years, very few associates overall were familiar with the BIOS initiative. This challenge later changed to a more interesting issue of brand dilution. The success stories that we shared within the company inadvertently led other teams to use the brand BIOS to suggest they were part of the success, without them actually practicing the values that BIOS advocates. This is not an uncommon phenomenon, and in a conflicted way, an indicator of success: clearly, our InnerSource initiative is perceived as successful, and others want to share in this success. However, if teams simply identify themselves as part of the success without actually practicing it, the BIOS brand is diluted. Worse, business units may simply claim they "do InnerSource" but not make the effort that is required.

Although cross-company collaboration improved, we still faced a cultural divide between BIOS and business units that could cause some friction. In particular, one of the BIOS values is meritocracy, and for that to work well, it is very impor-

tant to acknowledge high-quality contributions, because such credit is the "currency" for BIOS developers. At the outset, most business units were not familiar with this culture. And so when business units simply use BIOS assets to develop their products (and they are free to do so, as mentioned earlier), and don't give credit where credit is due, this leads to feelings of betrayal among BIOS developers because their work is taken without any recognition. Another facet of this challenge is that it was at times hard to sell work on APIs and libraries, work that many software developers gravitate to naturally. The reason for this is that API developers are usually so far removed from where money is actually made in the company that the value can be hard to quantify, and therefore, hard to justify. This is exacerbated by the fact that few managers have experience in software development that would allow them to appreciate the importance of high-quality APIs.

In addition to the challenges just described, a number of other issues emerged related to laws and regulations. First, we had to engage with the legal department and develop a license for all assets that would be created within the BIOSphere. This wasn't straightforward because of the sheer number of legal entities and countries involved. For the same reason, we also had to consider things such as export control: if code is being shared across legal entities in different countries, you must observe export control regulations, such as US laws controlling the export of strong encryption algorithms. Furthermore, we had to consider the federal tax authorities and address the problem of transfer pricing.

A concern that should not be taken lightly is the potential misuse of an increased level of transparency regarding individual developer's activities and performance that InnerSource offers. In Germany, for example, the Workers' Council serves as an "ombudsman" to ensure that such transparency is not abused for inappropriate performance evaluations.

Lessons Learned

Based on our experiences with setting up BIOS and the BIOSphere, we've learned a number of lessons that could be useful for other organizations that aim to set up an InnerSource initiative:

Ensure good stewardship
: Our initial steward, Corporate Research, was a perfect fit for what we were trying to do. They had a full understanding of the culture change that we were trying to make, and they shared our enthusiasm for software and Open Source. Furthermore, we had sufficient "executive air cover" through the Review Committee. Such support from a high level within the organization was very important for getting things done.

Create a compelling business case

One problem for any centrally funded change initiative is sustainability. While the BIOS initiative initially received corporate funding, over time we were expected to become self-sustaining, which meant that we would somehow have to get funding support from the various business units. However, creating a compelling business case to those business units that would convince them to give us a portion of their budget is really challenging.

Start measuring, but with care

While any change initiative will get a bit of slack, after a while, managers will expect some evidence that whatever improvements are promised are also actually achieved. Measuring participation and progress is therefore an important thing to do. We didn't do metrics from the outset for a number of reasons: the risk of metrics being gamed, the lack of control about interpretation of measurements and also the difficulty to objectively quantify the less tangible, and in our opinion most important, value contributions we made. However, we eventually arrived at the conclusion that one should try and quantify the less tangible value contributions, such as employee retention or increased learning from the beginning, possibly based on some broad assumptions, rather than giving up on quantifying those value contributions at all.

Think about marketing

We found that getting a clear message out to all business units in a large organization such as Bosch can be very challenging. Even after several years of advocacy, some associates still aren't familiar with BIOS. At the same time, one mistake we made was that we muddied the water ourselves by having *too much* branding. While BIOS and the BIOSphere were very useful (and playful) terms that had real meaning within Bosch, we caused confusion by introducing the term Social Coding, which referred to the second phase of the BIOS initiative.

Plan for early decentralization

We have learned the hard way that it is unlikely that a central team driving change initiatives such as BIOS and Social Coding will be funded for extended periods of time, regardless of whether they have fully realized their goals. Changing the culture of an organization, especially a large one like ours, will almost certainly take much longer than management is willing to fund the change initiative. Therefore, rather than relying on a continuously funded central team, we think it makes sense to spend significant effort early on to build up and empower a community in the organization that will continue to drive the initiatives even if the central team eventually ceases to exist. The bottom-up nature of both the BIOS and the Social Coding initiatives is now proving advantageous with respect to the community building effort. We are convinced that setting up the BIOSphere as a "safe space" in the organization

was instrumental in getting the initiative off the ground. However, it is easy to be lulled into a false sense of security regarding the longevity of this bubble. We feel it is important to communicate early on that a "safe space" like our initial BIOSphere can be only a temporary solution that eventually needs to be replaced with something else to help sustain InnerSource in the long run.

Seek alternative funding models
: While we have been successful in establishing stable and long-term funding for our collaboration infrastructure, we have been less successful in doing so for BIOS communities. Remember that we used BGO funding for contracting community leaders and contributors and that it was a key factor in growing and sustaining our InnerSource communities. Getting business units to provide funding on the other hand, especially for permissionless innovation projects where the benefit to the business unit is not necessarily obvious, turned out to be hard, even if the overall benefit for Bosch was apparent. At the time, we decided to make our collaboration platform available for free to both open and closed projects, mostly for the sake of organizational simplicity and speeding up the decision process. In retrospect, it would probably have been better if we asked the owners of closed projects to pay a small fee for using the platform, which we could have then used to fund the BIOS communities.

Conclusion

At the time of writing, we are nine years into our InnerSource journey. We have seen tremendous successes and have also encountered major challenges and frustrations. But most of all, we experienced firsthand how InnerSource can truly change the way we work for the better. Today, we can observe promising signs that our culture is indeed changing in the direction we envisioned at the beginning. We are encountering more and more projects at Bosch that "get" the BIOS idea and develop in the open from the very beginning.

For many of us, our time working in the BIOSphere and leading the first BIOS communities was the most productive, the most fun, and quite simply the best time in our careers up to now. At this point, we can't imagine working in any other way and are convinced that InnerSource, similar to Open Source, will continue to make a difference and is ultimately here to stay.

Acknowledgments

We'd like to express our gratitude to Stefan Ferber and Hans Malte Kern for lobbying for, and introducing, InnerSource at Bosch. Thanks to Gerd Zanker and Robert Hansel for their pivotal roles in BIOS and Social Coding.

CHAPTER 5

Checking Out InnerSource at PayPal

Year, place founded: 1998, Palo Alto, California (USA)
Area(s) of business: Financial services
Revenues: US$13.09 billion (2017)
Reach of business: 200 countries worldwide
Number of employees worldwide: 18,100
Number of engineers: 8,000
Year of InnerSource adoption: 2015

Much has already been published about PayPal's journey with InnerSource, the idea that using Open Source collaboration methods inside proprietary engineering organizations could go a long way towards improving quality and reducing resource bottlenecks while building competency in collaborative development. This, in turn, should increase the chances of successful Open Source participation down the road; the first booklet PayPal wrote is the most-downloaded non-code asset on public GitHub at the moment.

After a little background, this chapter will describe the year-long experiment with the Symphony module, PayPal's longest and most intentional InnerSource experiment to date. Symphony helped to convince senior executives within the company that the InnerSource method is an effective way to increase collaboration in a heavily siloed engineering organization. We'll also describe a parallel experiment by the India Domestic team, which independently decided to employ InnerSource to gain agency for a focused regional effort.

A Little Background

In 2014, longtime Open Source activist Danese Cooper was hired by PayPal to set up an Open Source programs office and to develop an Open Source strategy both as a way of giving back to the movement that had helped PayPal launch itself and also to attract more "Alpha Geek" engineering employees. A scan of the company's potential Open Source assets revealed that the biggest impediment to transformational Open Source participation was the *internal* engineering culture, which had been optimized over a number of years for deep specialization and very aggressively schedules for task completion. Engineers who were already under intense time pressures were reluctant to serve as mentors or evaluate outside code contributions.

In 2015 Danese and her team started evangelizing inside PayPal about InnerSource. PayPal had already decided to retool with GitHub Enterprise to increase code transparency, and there were already a few PayPal teams wanting to work more collaboratively, particularly teams working with Open Source technologies such as Node.js. Most of these Open Source–aware teams worked on the customer-facing portions of the PayPal product stack (e.g., new customer onboarding), but they often faced significant pushback when they tried to collaborate with more traditional core teams.

PayPal's past engineering management had chosen to optimize for ownership, believing that each team's pride in owning its code would help drive quality. One universal side effect of ownership culture is the mistaken belief that only your team is qualified to work on your code. Gifts of code from contributors outside a silo elicited confusion at best, and at worst derision, particularly if the contributor normally worked on the frontend but was trying to contribute to the backend code. It is a common bias to unconsciously rank the expertise of one's immediate team above any unbidden contributor's work.

Attributes of InnerSource

It is worth clarifying what is meant by InnerSource—after all, each organization adopting InnerSource has its own specific way of doing so. For PayPal, InnerSource meant the following:

- Encouraging the use of pull requests instead of feature requests.
- Trusted Committers who review submitted code and return advice if changes are needed.
- Written communication that is archived and searchable, in place of verbal communication that is lost to future potential contributors.

- Suitable extrinsic rewards to overcome reticence and kickstart collaborative work.

Because people learn in a variety of ways, Danese developed a diagram to explain InnerSource, centered on the legacy issues experienced by many large companies with software practices of a certain age. PayPal had recently undertaken a well-publicized Agile transformation, but Agile didn't address the practice of allowing executive escalation to interrupt planned work. A key feature of the PayPal Agile practice is the use of Scrum across all engineering teams, so work is organized into two-week sprints with a demo at the end of each sprint. Frustration about being evaluated on planned work completion, when in any given sprint those plans could be blown up by an escalation, was a serious pain point for engineering middle management. The diagram therefore starts with a depiction of two wedges of cheese (i.e., executives) escalating a feature request to preempt planned work. Recognition of that problem and its high cost was universal among viewers.

PayPal's initial experiment was driven by this problem alone, and by using InnerSource, the team was able to reduce the time they were spending during each sprint on escalation-driven interrupts from 65% before InnerSource, to under 5%, allowing them to focus on completing planned work. For this reason, this diagram is most often called the "Cheese Story" (Figure 5-1).

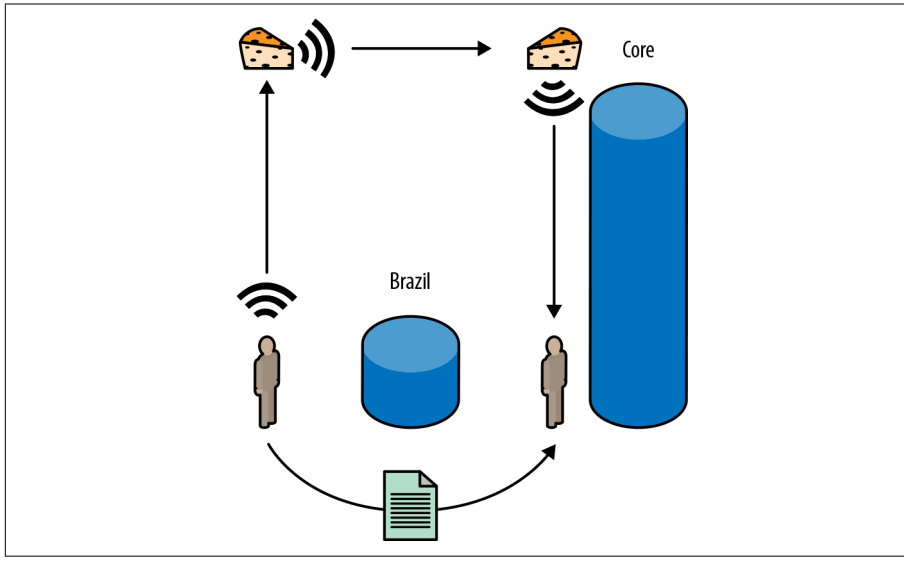


Figure 5-1. Escalation to "cheeses" to get work integrated

The diagram builds to depict the feature request morphing into a pull request (e.g., the engineer who needs the feature attempts to write it him or herself, and contributes that code to the target silo). Borrowing from The Apache Way (see Chapter 2), the engineer on the receiving side becomes a "Trusted Committer" (TC),[1] meaning he is responsible for reviewing contributed code and determining whether it can be merged. If it isn't yet acceptable to merge, instead of rewriting that contributed code, the TC returns written feedback about what the contributor needs to do to make it acceptable. The pair iterate until the contribution is good enough to merge (Figure 5-2).

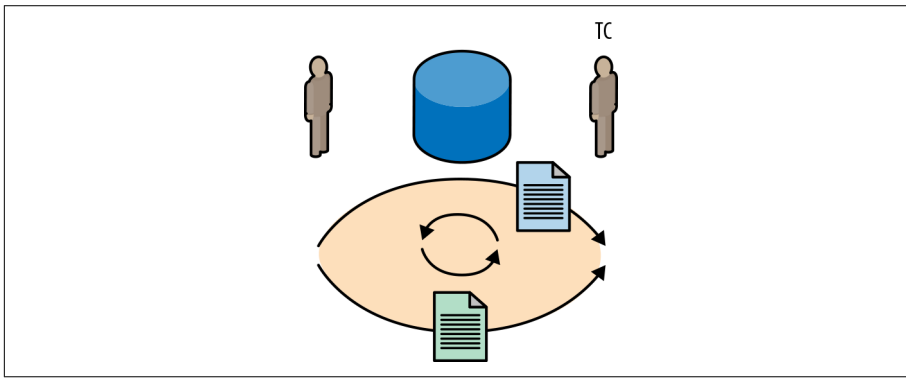


Figure 5-2. Mentorship—a Trusted Committer sends written advice to a contributor about how to improve his contribution so it can be accepted into the larger silo

What happens next in the diagram reveals one of the hidden benefits of The Apache Way. The written conversation between the contributor and TC is collected into a persistent, searchable archive (Figure 5-3).

1 João Miranda, "InnerSource: Internal Open Source at PayPal," *InfoQ*, October 30, 2015, *http://bit.ly/2JK0RAt*.

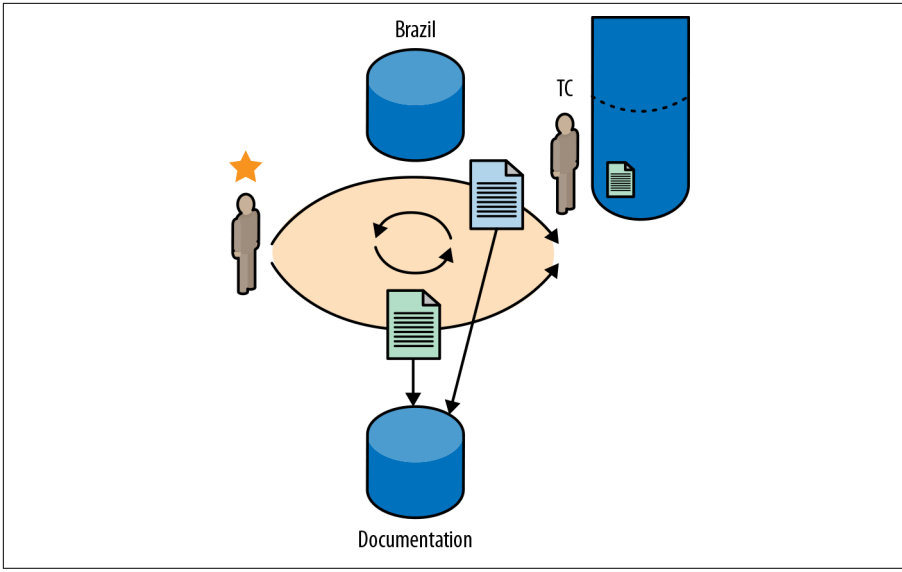


Figure 5-3. Documentation—the entire mentorship conversation is archived in a persistent, discoverable way and becomes de facto documentation, mirroring the function of Mailman archives at the Apache Software Foundation (ASF)

Although the time it takes to have the initial conversation can seem expensive, this actionable documentation (actionable because it focuses on real information about how to write code that can be merged into that silo) can then be referenced by the TC the next time a potential contributor tries to submit code, which greatly reduces the time cost of mentoring subsequent contributors. Moreover, tracking which conversations are searched most often can help determine where development of more formal documentation could be useful.

The diagram continues building to highlight other benefits of InnerSource that were discovered during the first InnerSource experiment at PayPal. A first benefit is that the practice of InnerSource granted the different silos insight into the question of how to best modularize their codebase, a task that had formerly seemed difficult to undertake. Second is noticing that as contributors become adept at writing code for the silo, they can start to act as TCs for aspiring new contributors, thus further freeing up the original TC. Thus, increasing full-stack knowledge in the company is seen as another benefit of InnerSource at PayPal. A third benefit is seen by teams that decide to divert all their own work through the TCs, as more careful code inspections result when you make the TC responsible for the merge of their team members' code and there is a resulting bump in overall quality (Figure 5-4).

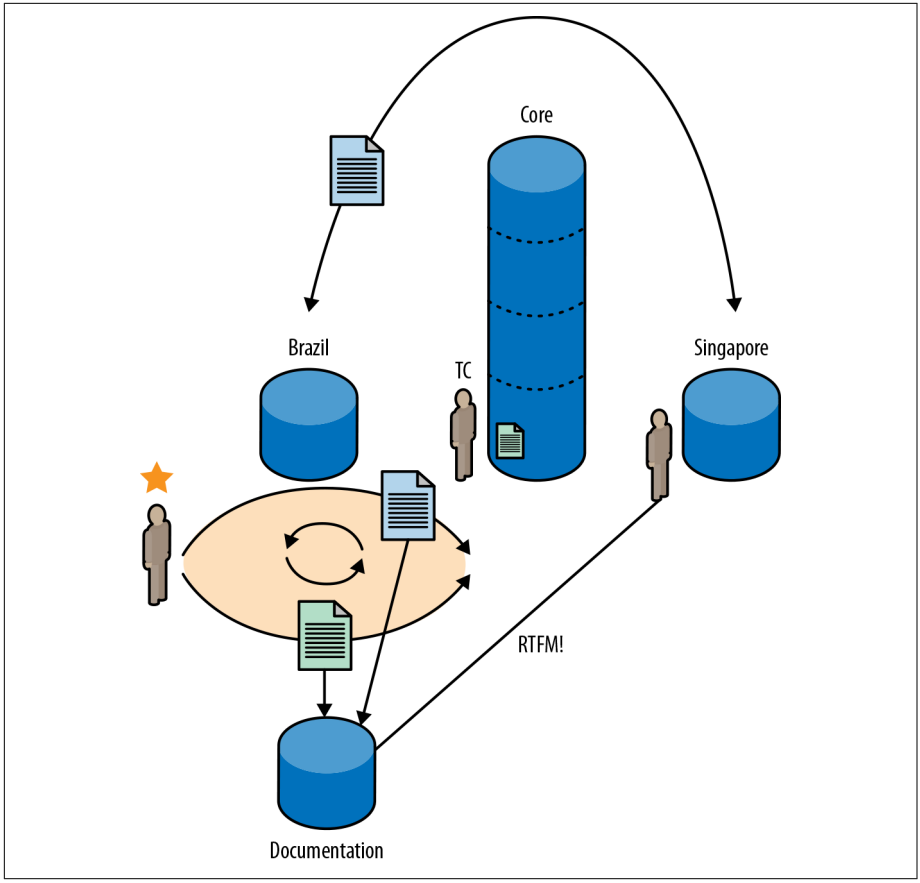


Figure 5-4. Other benefits of InnerSource include increased velocity due to accretion of actionable documentation, deeper understanding of where/how to modularize, and building of full-stack knowledge as serial contributors become guest Trusted Committers

The last part of the diagram build addresses motivations of the players involved and how to introduce extrinsic motivators[2] to deal with asymmetry of motivation. Of course, the contributor has an intrinsic motivation (he needs the feature to complete his assigned work, as indicated by the gold star he gets when he completes his work), but the TC's motivations aren't so clear. Initially most engineers don't want to be confined to code review and mentoring for even just one two-week sprint, because in the past they've been rewarded mostly for the code they

2 Richard M. Ryan and Edward L. Deci, "Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions," *Contemporary Educational Psychology* 25, no. 1 (2000): 54–67.

produce. An extrinsic motivator such as explicit recognition for excellent mentoring can really help to overcome TC reticence (Figure 5-5).

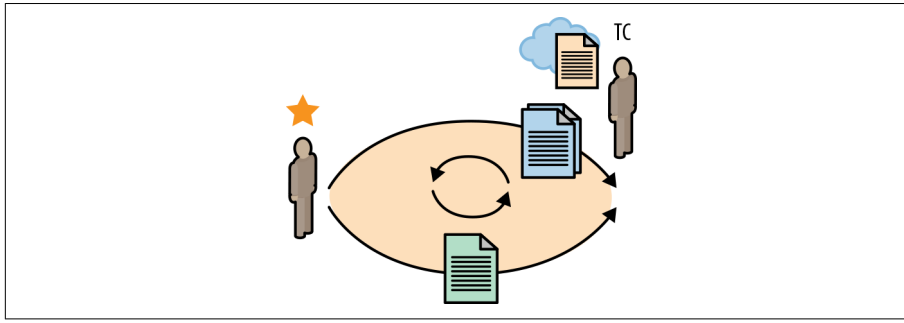


Figure 5-5. Possible extrinsic rewards

The CheckOut Experiment

The first InnerSource experiment within PayPal was started by the CheckOut team (some of these experiences were documented in the first booklet we cowrote with O'Reilly editor Andy Oram).[3] CheckOut was one of the busiest core teams. A majority of PayPal's customer flows routed through CheckOut (because most customers use PayPal to buy and sell things), and with an elite team of only 25 engineers it was difficult for the CheckOut team to consider budgeting time to review and merge other teams' submitted code, preferring to write any code aimed at fulfilling requests for changes themselves.

But those types of requests were increasing. In an effort to enrich diversity of experience, PayPal engineering started engaging in "Accu-hiring": acquiring smaller companies at least in part as a way of bringing in whole teams of new engineering employees at once. These new employees were often very comfortable with more modern development methodologies, and some of them began submitting pull requests rather than feature requests when they needed changes in legacy code. Pressures on the team intensified as PayPal ramped up new feature development and delivery, and often the urgency of completing such requests was increased by executive escalation. Such escalation is a common antipattern where planned work goes uncompleted because all resources are diverted to address a somewhat artificial emergency created by executives using their aurthority to circumvent planning procedures and demand preference for their team's request. This practice is common in companies wherever serious resource bottlenecks hamper the timely resolution of feature requests, or wherever strict hierarchies create opportunities for such power plays, because people throughout

3 Andy Oram, *Getting Started with InnerSource* (Sebastopol: O'Reilly Media, 2015), *http://www.oreilly.com/programming/free/getting-started-with-innersource.csp*.

the company judge effectiveness as a function of the individual executives' ability to "cut through red tape." Realizing that they wouldn't be able to keep up with this increasing demand while still completing improvements they planned themselves, the CheckOut team began to experiment with InnerSource.

Luckily, most of PayPal's codebases had recently been made visible to all engineering employees through a company-wide GitHub Enterprise implementation spearheaded by the CTO. Removing barriers to engineers freely accessing all company code is a necessary first step to InnerSource, regardless of which code management tool is used. But just making it possible to see all the code doesn't automatically increase collaboration by itself, because old patterns are hard to change.

The first change CheckOut made was identifying 10% of the engineers on the team to take on the role of Trusted Committer. These staff members were tasked with reviewing not only guest pull requests submitted in InnerSource fashion, but *all* pull requests, even those written by their own team. So that none of the Trusted Committers would object to being pulled away from actually coding, they assigned the Trusted Committer role on a revolving basis, each member serving as a TC for one or sometimes two two-week sprints at a time. They advertised which of the team members were currently serving as Trusted Committer in an *About.md* document in their GitHub repository. This way, anybody outside the team taking an interest in the project would know to whom InnerSource pull requests should be submitted. The team also took the bold step of announcing they would no longer respond to executive escalations unless there was a showstopping bug involved. To make this last piece work, they requested support from the CTO's office for the duration of the experiment.

Their results were amazing! As mentioned before, they managed to reduce the amount of escalation-driven work they did per sprint down to less than 5% (from a pre-InnerSource high of 65%). They also significantly increased overall quality by instituting real code reviews across the board and were able to prove that reducing escalations was key to addressing perceived bottlenecks. Moreover, as guests started querying the structure of this monolithic system, the original developers found themselves explaining design decisions. These conversations helped them gain missing insight into how to restructure those systems to make them more modular, where initially they had been struggling to decide about how and where to modularize. Lastly, they were able to identify guest contributors whose diligence and excellent contributions entitled them to assume guest Trusted Committer status, further distributing the workload of mentoring new guest contributors. Almost everyone was very pleased with this new way of working.

Unfortunately, PayPal's first InnerSource experiment ended in an anti-pattern lesson. The experiment had been designed to only address "Engineer to Engineer"

workflows and left out Product Management Owner (PMO) roles. The PMOs were officially tasked with assigning and tracking work (stories in Agile parlance) to complete new features, and they still believed escalation was the most effective method for them to get their work done. The ensuing struggle ended with several of the members of the CheckOut team opting to leave PayPal rather than return to the old way of working.

The Onboarding Experiment

Around the time the CheckOut team started their experiment, another PayPal group was trying a different method to bootstrap collaboration more quickly. *Onboarding* is the process of signing up a new customer or merchant with PayPal. Since Onboarding is practically the first interaction people have with PayPal's services, it strives to be easy to understand and frictionless in every market. Regional differences mean the Onboarding experience needs to be tailored to reduce friction. Originally that tailoring was handled by feature request, but it was inefficient to describe a desired change and then wait for the core Onboarding team to write it. So Regional Sales Engineers (RSEs) started wanting to get directly involved in tweaking the Onboarding experience for their region by contributing code to make the feature happen instead of waiting for someone else to write the code.

But instead of setting up InnerSource between the teams as it was done for CheckOut, the decision was made to run a one-time bootcamp for all potential guest contributors (such as RSEs) to more quickly bring them up to speed all at once. Seventy-five engineers converged at PayPal's headquarters in San Jose, California, from all over the world to spend two weeks building relationships and learning about Onboarding's architecture and codebase. This mass training period greatly increased mutual understanding, but it brought up another antipattern. Because the content was delivered mostly verbally, a persistently archived and searchable knowledge base was not created as a byproduct (as it would have been if the training had been delivered in writing through mentoring sessions as code was submitted and reviewed, e.g., in InnerSource style). Although the engineers who went through the program did gain knowledge and relationships, there was a significant productivity cost on both host and guest teams. Moreover, because the conversations were not captured in writing, the loss of productivity during training would necessarily be repeated each time a new group of engineers needed the training. This experience reaffirmed the desirability of written mentoring, which can be archived and made discoverable so the experts need only take the productivity hit one time.

Executive Air Cover

These limited successes meant that pressure to increase effective collaboration continued to build within the company, but most core teams were still pushing back due to competing pressure to complete work in progress, and in some cases a belief that InnerSource wouldn't work or was more trouble than it was worth. Proponents realized we needed some executive "air cover" to set up a wider and hopefully more definitive experiment. PayPal's CTO at the time worked with his lieutenants to identify core codebases for a higher profile experiment. His involvement went a long way to guarantee cooperation and allow a longer-term study of the effects of InnerSource on PayPal Engineering processes and practices.

Symphony, the codebase that was ultimately chosen for the next InnerSource experiment, was a key component in the PayPal software stack that was both monolithic and commonly involved in serving customers. Symphony was already scheduled to be divided into frontend and backend components, and there would be tendrils of code that needed rewriting to accommodate that split, even as there would also be guest contributions coming in to allow new features to be written. In December 2015 meetings started between the CTO, relevant engineering executives, senior managers and Product Owners, and the InnerSource team about how to accomplish the transformation of Symphony.

The first work was exposing everyone's misgivings about InnerSource collaboration. One of the senior executives had previously worked at Google and felt strongly that we were talking about using the much storied "20% time" concept to, in essence, extract more work out of the same employees. There were also fears about the capability of different types of engineers (for instance, the backend engineers felt the frontend engineers were less capable of grasping backend complexities). Everybody wanted some type of Service-Level Agreement (SLA) from the others, be it for expected wait time before getting an answer to a query, or for a contributor's willingness to fix any issues that might arise from incorporation of contributed code. There was much interest in what we would be measuring to assess success of the experiment, and a universal desire to be shielded from executive escalations for the duration of the experiment. Danese helped the meeting attendees negotiate a set of working agreements in a *Contributing.md* file, delivered some training to newly minted Trusted Committers and guest contributors, and set up a regular weekly cadence of Scrum meetings to keep everybody on track. In January 2016 we started tracking the Symphony InnerSource experiment.

Meanwhile, in India

One of the interesting team dynamics within PayPal Engineering is created by having a large engineering organization in Chennai, India. This arrangement can actually be an aid to InnerSource, as time-zone and physical distances between team members encourages written rather than verbal communication, but it still requires training about how to work differently. About two-thirds of the Symphony team was Chennai-based, so as part of setting up the Symphony experiment Danese spent significant time in India delivering training to Trusted Committers and guest contributors. Some of the people attending the training were not involved in Symphony but were looking to understand InnerSource and why we were using it. One of these who became a real champion of InnerSource at PayPal was a Senior Director charged with leading a team to build new products for the domestic India market.

With a population of over 1.3 billion people and a rising middle class, India had long been identified as a potential growth region for PayPal. However, efforts to address this potential market were slow to coalesce. The original plan for 2016 was to spin up a large coordinated effort in Chennai, but during annual plan review other work became more urgent and, in the end, the India Domestic project was only partially funded for 2016. Since the team was now underresourced, they knew they would have to work differently to still achieve their goal of being restored to full funding in the following planning year. Toward this end, they voluntarily took advantage of InnerSource training in early 2016 and undertook their own InnerSource experiment.

Learning Goals

We knew from the CheckOut and Onboarding experiments that InnerSource could be an effective strategy to reduce unplanned work interruptions while building full-stack knowledge (i.e., broader knowledge of all the parts of PayPal's software stack, not just isolated knowledge of a specific assigned area), and we had a theory that capturing written mentorship advice in a persistent and searchable archive would build the actionable documentation we needed to increase velocity in the future. But, we still had major questions that needed answers before we could commit to rolling out InnerSource at a company-wide scale. These included:

- Could InnerSource work in situations where there was a lot of initial pushback?
- Could InnerSource work along with other methodology initiatives such as Agile and Continuous Integration (CI)?
- What kinds of training or coaching would be most effective?

- What extrinsic rewards would best overcome reticence?
- What metrics would be most useful?
- What cultural impediments to successful InnerSource would be encountered and how would we mitigate them?

At PayPal we use the metaphor of hospitality to help InnerSource participants think about how to behave. Hosts welcome guests, but only if they obey the house rules. Guests expect hospitality in return for respect for the host. Symphony was chosen for our official experiment, in part, because many of the developers on both guest and host sides were skeptical about the value of the exercise. It was thought that if we could win the Symphony team over, we could proceed to scale out across the company with some confidence. Even the VP in charge of Symphony was skeptical. He had worked at companies where cross-company pull requests are relatively common, and he said that in his experience they got merged only when executives from the outside teams exerted direct pressure, so he wasn't a believer that we could live without executive escalation.

The India Domestic team, meanwhile, was just trying to prove that their project could work. They needed to make progress with minimal resources, and much of the work they needed to do was within codebases they neither owned nor could directly influence through escalation. They saw practicing InnerSource as the only way to gain enough agency as guest contributors to achieve their goals.

Beginning Symphony and InnerSource Brand Dilution

The kickoff meeting for Symphony happened in late December 2015 and included all the executives who would be supplying air cover, including the CTO, all the VPs whose employees would be touching Symphony code, and the original ownership organization including directors, senior managers, and managers whose employees had written Symphony. InnerSource contributions were measured in sprints, and the first work started early in 2016.

The Symphony InnerSource pilot lasted a whole year. Throughout 2016 there were competing pressures on the InnerSource team with regard to focus. Many of our air cover executives wanted us to focus solely on Symphony and the lessons it was teaching us about PayPal's engineering culture for the year, while others wanted us to continue to take on, train, and support additional teams in addition to working with Symphony. The InnerSource team tried to keep both camps happy by lending only limited support to non-Symphony teams. Pressures were eased with this minimum progress and advocacy for continued focus on Symphony from executive sponsors, but there was also significant brand dilution for the term "InnerSource."

Teams not yet trained in InnerSource concepts started declaring they were "doing InnerSource" without a real understanding of the methods developing around Symphony, and through lack of understanding they fell into the trap of reinforcing rather than mitigating people's fears. People were having bad experiences and blaming our InnerSource program. In particular, rumors grew among rank-and-file engineers that InnerSource was a form of "forced conscription" where one team could force another to do their work for them. Interestingly, when we studied this claim, we found the actual problem was straight out of *Lost in Translation*. An aspiring guest contributor would write an Agile story for a proposed InnerSource contribution (signaling their intention to write the code and contribute it to a Host). When considering the request, the Host team would rewrite the story so it made engineering sense from *their* point of view. Once approved for InnerSource, the original submitters would sometimes not recognize the rewritten story as one they had initiated and would end up feeling exploited when they were asked to code up the story. "This InnerSource sucks!" was a common response to the misunderstanding. The InnerSource brand was suffering.

To stem this brand dilution, we wrote the "InnerSource Checklist"[4] and spun up a special 15-minute briefing presentation which was delivered top down starting at the CTO's staff meeting (to catch engineering executives not involved in Symphony) and filtering down to middle managers in a 12-week-long push just to establish a baseline understanding of what the term "InnerSource" actually means at PayPal, and hopefully to stop people from claiming they were "doing InnerSource" when they actually were not.

Initial Symphony Training

But before Symphony's InnerSource experiment could get started, the first job was getting everyone involved aligned, which meant training all participants in their InnerSource roles early in the first quarter of 2016. Although we now have developed many training assets, at that point in time we were starting from scratch. We identified four trainings that everyone would eventually need:

Introduction to InnerSource
: This includes some history, a discussion of the "Cheese Drawing" and the issues it touches on about working around bottlenecks and excessive escalations while improving quality, and some projected goals for the methodology. This also formed the basis for later training developed for Senior Management to combat brand dilution.

4 Silona Bonewald, *Understanding the InnerSource Checklist* (Sebastopol: O'Reilly Media, 2017), *http://bit.ly/understand-IS-checklist*.

The Role of the Contributor
: A primer on how to be a good guest, including seeking advice early and often, accepting a written critique and working through it, and staying available to support your work for 30 days after deployment of a contributed pull request.

Trusted Committership
: A primer on how to be a good host, including why the role of Trusted Committer is important, why and how to mentor a contributor, and why written advice is important. It also touches on how achieving Guest Trusted Committership could help your PayPal career.

InnerSource for Product Managers
: A discussion of how the role of Product Manager changes under InnerSource, and how to negotiate with other PMOs to get InnerSource work scheduled.

The InnerSource team developed a set of slides and a Q&A for each of the four trainings, then delivered each of them in two-hour sessions (eight hours total per person!) in both San Jose and Chennai. We advertised these internal trainings widely within each location and allowed as many employees per session as the rooms could hold. We kept track of the actual Symphony participants' attendance in order to satisfy the requirement that all directly involved parties receive training, but most sessions included about 20% self-selected interested others. Currently the InnerSource Commons community and PayPal are working on online versions of these four trainings, to support scaling up InnerSource across the company, with the first one completed by this writing.

In general, the trainings are about uncovering and addressing fears evoked by a new methodology. Change is hard for most people, and InnerSource is no exception. Trainees had a number of understandable fears, and the trainers (members of the InnerSource team) found the most effective way to mitigate those fears was to get trainees to articulate and then take the time to really discuss and work through the fears. People need to feel heard. Trainers took a nondefensive stance and tried to value all the feelings people were sharing. Over the span of several weeks when the trainings were originally delivered, several common fears surfaced, including:

- Skepticism that escalations would actually cease
- Fear of increased workload, of being conscripted, of having code publicly scrutinized, or of being shamed for lack of code quality
- Fear of losing agency associated with their position
- Fear of insufficient ability with written English, since written comments would persist

- Fear of wasting time on unsuitable code, that gift code wouldn't have acceptable quality
- Fear of accepting gift code and then having the burden to maintain it

The training presentations were designed to test comprehension and stimulate conversation. They featured quizzes describing possible scenarios (taken from real life) and offering multiple-choice nonobvious answers. This method proved very fruitful for getting people to admit their fears. For example, this question emerged at the "InnerSource for Product Managers" training.

InnerSource as practiced at PayPal is:

- A way to conscript extra resources into your project
- A magical way of creating programmers to help you make your deadlines
- Outsourcing internally
- Open Source at PayPal
- A way to increase collaboration, cross stack knowledge, ease bottlenecks, and assist teams in getting needed changes implemented in stacks they don't own

These possible answers usually caused a deep discussion of why conscription doesn't work, how InnerSource is different from Open Source, and whether the last answer (which happens to be the correct one) is magical thinking.

The Contributing.md File

The need for mutually agreed-on rules of engagement to help address common fears and skepticism about InnerSource was fulfilled by the *contributing.md* file (the "md" extension signals it's written in Markdown) as the main place to memorialize those agreements. We held a meeting with key executives, managers, and developers to negotiate and document the rules of engagement (see Figure 5-6).

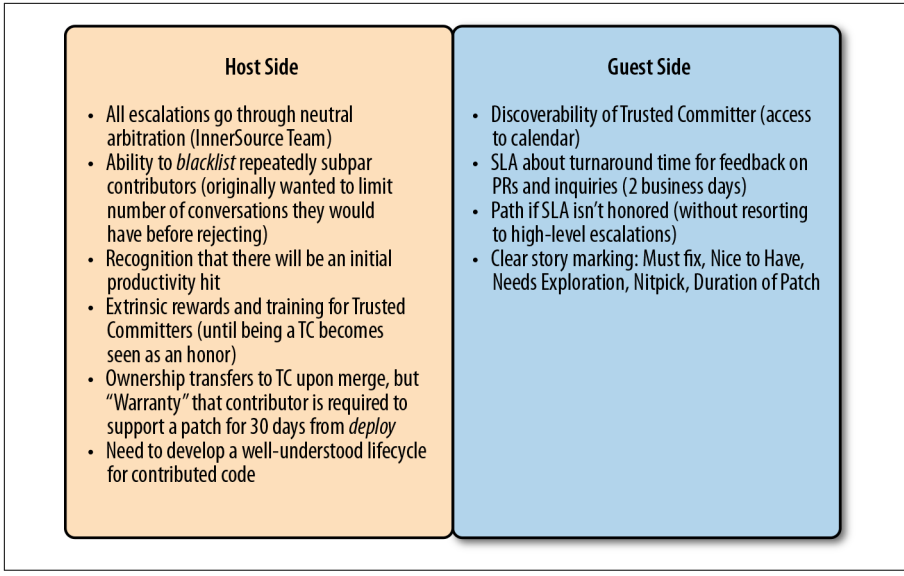


Figure 5-6. List of agreements: Symphony contributing.md v1.0

This list tells you a lot about the fears that were most prevalent and the mitigations we identified together, including:

- The ability to refuse escalations
- The "30 days after deploy" warranty that the guest contributor must honor
- Threat of a blacklist that any contributor can land on if she is unwilling or unable to follow mentorship advice (note this has never actually been employed)
- The "2-day Service-Level Agreement (SLA)" within which submitted pull requests must receive feedback from a Trusted Committer
- The escalation path if that SLA fails

The InnerSource team added a weekly 15-minute check-in call (as a specialized Scrum) with the actual team leads to listen to any issues they were encountering, and monthly 30-minute check-ins with the sponsoring executives to keep them in the loop and ask their advice or support on specific issues.

Cadence of Check-Ins

The 15-minute check-ins were most useful because they were a chance for engineers enduring pain points to alert us about them. It was during the check-ins that we first figured out the brand dilution described earlier within the planning

process, which led guest contributors to conclude that InnerSource was forced conscription.

Although we were officially trying to forestall development of a comprehensive metrics package (because we weren't sure we knew enough to establish meaningful measures that wouldn't drive anti-patterns), we identified a light set of metrics to collect per sprint (chiefly lines of code successfully merged per sprint and InnerSource stories completed per sprint), so as to have something to track in the monthly executive meetings. We chose lines of code per sprint as our primary metric because it was easy to collect manually (by counting all lines associated with stories tagged as InnerSource) and did give some indication of relative progress within the pilot. It is important to be careful with metrics, because you influence behavior by what you measure. The tendency to optimize for any metric is strong, especially among engineers. Measuring lines of code can influence engineers to be more verbose in their writing, just as measuring number of check-ins can influence engineers to check in smaller and more frequent patches. Nevertheless, we decided ease of collection outweighed these risks.

Even the best training must be reinforced. Deeply ingrained cultural norms have a way of showing up in practices long after teams should be following new patterns. Weekly check-ins helped us keep Symphony on track. Some examples of issues that came up include:

Cheating on InnerSource processes in an attempt to optimize
: In an ongoing effort to streamline and waste no effort, we heard several times that teams wanted to limit InnerSource to stories with an estimated 2.5 or more "story points." They figured they could fulfill small requests the old-fashioned way, by writing those themselves (since they could code them up faster than explaining how to someone else). We had to remind them that teaching a man to fish was more efficient than fishing for him in the long run.

Staff turnover, resulting in a need to retrain
: During the year we were working closely with Symphony, the senior staff changed three times. Frequent job shifts were another cultural norm at PayPal, which management is currently working to curtail. The net effect of these staffing changes was that we had to completely retrain three times.

Workload of reporting metrics
: Even though we were only asking for lines of code committed per sprint, the Symphony team noticed that they could automate collection of those metrics with a lightweight tool. "Mr. Gherkins" is a set of scripts that tag repos for InnerSource (to make them easily discoverable), collect and report metrics, and let the contributor know when code has been deployed and the 30-day warranty countdown has begun. We open-sourced Mr. Gherkins through InnerSource Commons in 2016.

Difficulty scheduling stories
: This example was already partially covered, but it was a serious problem so I'm including it here for completeness. The Symphony team was experiencing pushback from guest teams asked to submit code for requested features (e.g., to handle them through InnerSource). By joining scheduling calls, the InnerSource team were able to trace the problem to the practice by the Symphony team and PMOs of restating a given story from their point of view. The originators of the story could no longer recognize their story when it was pushed back to them for InnerSourcing, and that shock led to circulation of the rumor mentioned earlier that "InnerSource equals forced conscription." The InnerSource team was able to make a case for stories going through less transformation to mitigate this issue.

Difficulty scheduling meetings
: This issue was reported by the India Domestic team and was a consequence of geographically separated teams. India Domestic found out early on that getting San Jose to honor meeting commitments took extra work. They developed a habit of sending reminders and tight agendas to make sure they made the meetings as effective as possible for all parties. They figured out through trial and error which days of the week were most advantageous.

Prevetting planned contributions
: This is another best practice out of India Domestic that the Symphony team also found useful. Contributor teams were encouraged to notify the receiving Trusted Committers by filing an issue as soon as an InnerSource contribution is contemplated. This allowed both sides to maximize resource planning. It also allowed the receiving Trusted Committers to fend off any misguided planned contributions. Once we started doing this with all InnerSource projects, we saw a higher throughput of contributed and merged lines of code.

Outcomes

Figures 5-7 and 5-8 present two graphs derived from actual InnerSource code merged as a function of time for both the Symphony and India Domestic projects. These were prepared for executive review and clearly show that significant amounts of code can be produced by guests for a host that participates in an InnerSource program. This is code that the host would have been asked to write, so it represents a real offloading of work for the host teams. But it also represents less unproductive time spent waiting on behalf of the guests and also significant learning on both sides.

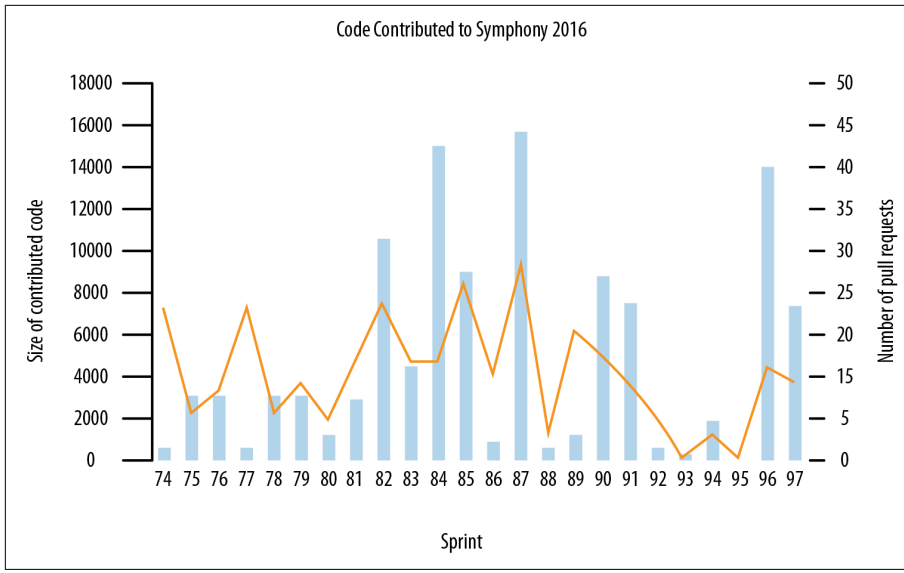


Figure 5-7. InnerSource code merged by Trusted Committers into Symphony in 2016 as a function of time (organized by two-week sprint)

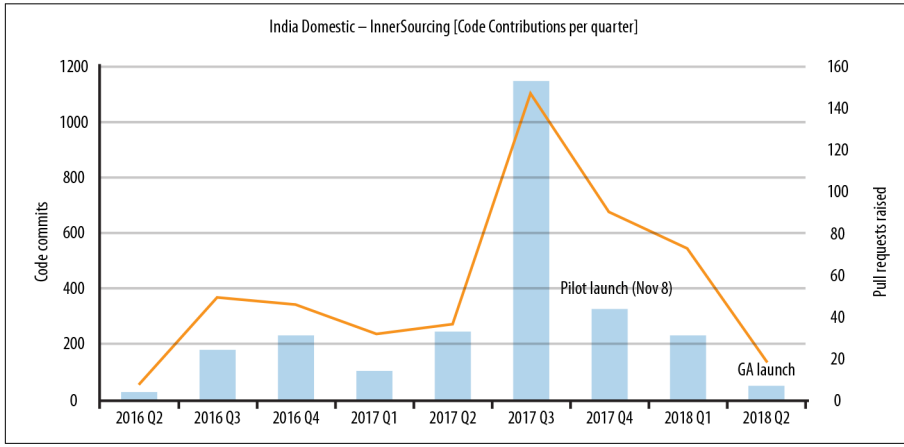


Figure 5-8. InnerSource code merged between Q2 2016–Q2 2018 by the India Domestic project team (organized by quarterly sprints)

In both cases, significant contributions of code were successfully merged from InnerSource pull requests, although it took both teams a while to get into the swing of it. In the Symphony graph (Figure 5-7), the yellow line represents number of InnerSource pull requests per sprint. Tracking this number, we learned that as the Symphony team gained comfort with the InnerSource process, they were able to effectively review and mentor in larger and more complex (and hence fewer) pull requests per sprint. So, the process was gaining efficiency as

trust in the process grew. Over the year, Symphony averaged about 4,728 lines of merged InnerSource code per sprint for a total of 113,471 total lines of code merged. The average number of InnerSource pull requests merged per sprint was about 11.

India Domestic's graph (Figure 5-8) also shows significant progress but with a different pattern with a single noticeable spike. The spike occurred because there was an accretion of contributions ready to merge during a feature freeze. PayPal has long used a seasonal moratorium on merging new code to control risk of production destabilization during the most critical periods of customer activity, such as the holiday shopping season. Since the engineers continue to code during a feature freeze, this necessary practice of merge moratorium engenders a surge in new code merges whenever a moratorium is lifted. As with Symphony, it also took some time for the team to really hit their stride in terms of alignment and working cadence to produce and gain acceptance for their InnerSource contributions, and that stride was optimal in the quarter before product launch after they had learned the lessons of submitting smaller changes and working within the host's guidelines for the best outcomes.

The Rhythm of InnerSource Work

Along with the discipline of working in two-week sprints, overlaying InnerSource work into the equation seems to have created a specific cadence. Trusted Committer was a rolling responsibility on Symphony and India Domestic teams, with most engineers taking on TC duties in shifts in this project lasting one or at most two consecutive sprints. The "sawtooth" graph shape (where a single very successful sprint is followed by a much lower number of lines of code merged in the next sprint) is common across both projects. This indicates a break in the mentorship pipeline, as large merges actually take multiple sprints to review before they can be merged, and often teams change Trusted Committers just after a big merge. The new Trusted Committer begins the next large piece of code review, but very little is merged in the interim.

Along with the passive accretion of actionable documentation as a consequence of written mentorship from Host Trusted Committer to guest contributor, there are a couple of additional benefits of InnerSource. One is gaining new insight on the part of Hosts into the complexities of their software silo. The CTO's office had long been requesting that all silos modularize their code, but many struggled with refactoring and modularization before InnerSource and it was slow going to achieve adequate modularization. As mentioned before, answering guest contributor questions proved invaluable in providing insight to where modularization would have the biggest impact to make a given piece of functionality easier to understand and contribute to.

Another benefit is identification and indoctrination of remote talent. In both experiments, Hosts learned of one or more Guests who achieved a high level of reliability in pull requests they submitted after an initial round of TC mentorship. One engineer in Singapore so distinguished himself that he became a Guest Trusted Committer on the Symphony team!

Thus, we determined that InnerSource brought several benefits, some of which have already been mentioned:

- Useful documentation generated through the mentoring process without extra effort
- New insight by Hosts into their software's complexity and where best to modularize it
- Identification and training of remote talent
- Most of all, developer satisfaction

Of course, the biggest win is when engineers express a preference to continue with InnerSource. The Symphony team ended the year by announcing a rearchitecting plan that would be run using only InnerSource methods from the start. Once team members see the potential of InnerSource realized in their daily work, they are generally hungry to continue to work this way.

The Future of InnerSource at PayPal

The InnerSource experiments that PayPal ran in 2016 proved that InnerSource could work even at the very center of PayPal's stack, although of course with significant mentorship to keep the team moving in the right direction. It was recognized by senior management that InnerSource would involve both cultural and procedural changes. Based on these results, the company decided in 2017 to devote resources to scaling InnerSource across all of PayPal.

The India Domestic team was also able to make enough progress exclusively using InnerSource that their 2017 plan was fully funded.

Through our experiments at PayPal, we gained many insights and experiences. We conclude this chapter with the most important takeaway lessons that may be useful to other companies who are considering adopting InnerSource:

Peer support and training are crucial
: It's easy to dismiss a new approach as "too time consuming" or "too much work" until you've sat in a room with enthusiastic participants. Time and again during training we saw skeptics swayed to at least give InnerSource a try after they heard from peer colleagues who had previously worked on the CheckOut team (or on Symphony) that our claims for this new paradigm were true in their experience. Since the training for InnerSource is really an

exercise in confessing and debunking fears, it's a very important conversation to set up the exercise for maximum chance of success. Without such training sessions to inform people about InnerSource, they may form their opinions based on hearsay or misinterpretations from others.

Regular check-ins help you keep the InnerSource train on the tracks
: It's unrealistic to expect that team members will remember everything discussed during training. They will fall back on the optimizations with which they are already comfortable. Checking in and listening carefully to what teams report in their regular check-ins, along with timely reminders of nuances in the new approach, can save time and frustration.

Metrics matter
: In the beginning you can use simple metrics to sanity check that InnerSource will work for you. One outcome of our Symphony year was a plan for a more comprehensive tool, "SeazMe," which will not only be the aggregation point and persistent archive for our passively accreted documentation, but will also be where we watch at scale for the emergence of anomalies and anti-patterns.

Scaling out InnerSource takes planning
: Management commonly underestimates the amount of work involved in scaling out InnerSource as a practice across the whole company. After Symphony, senior management wanted the InnerSource Team to turn on a magic spigot that would spread InnerSource throughout the rest of the company, but without increasing the size of the InnerSource team and without completing any of the necessary tools.

The India Domestic team further proved that with sufficient guest motivation, InnerSource can work at PayPal even without executive mandate. In the end, that team submitted InnerSource code to no fewer than 23 silos within the company, each of which had varying levels of experience and bias when it came to InnerSource. They developed a number of tactics, including:

Adopt a win-win mindset when approaching InnerSource
: Their focus was not only on getting work done, but also on exploring and articulating how InnerSource contributions would strengthen the host codebase and product. A case in point was enablement of two-factor-authentication experiences during card transactions. Once enabled as part of the India Domestic project, the feature was rolled out in other geographical regions and is adding tremendous value to various markets, such as Europe, with zero code change.

Be patient when scheduling with a Host team
: The India Domestic team proved that being a respectful but persistent guest was a viable tactic within PayPal to eventually gain agency with a given Host. Early initial contact, regular follow-ups with host teams, and scheduling

actual meetings to close out pull requests on a pre-consented basis can keep the InnerSource project moving forward, if slowly.

Plan for rebasing after a pull-request merge
: Early on, the India Domestic team had to circle back and rework previously merged guest contributions when the host undertook a major codebase reorganization. They learned it was best to get in front of likely rebasing activities. They worked with the host team to gain access to automated test regression suites (which were still not included as visible code in GitHub). If no test suites were shared with the guest team, it had to plan for increased capacity regression. Scheduling a "Plan & Review" of identified regression test cases with domain Product Owner and product management teams can give early warning of any issues in an InnerSource plan. If this involves undue effort, then working on ways to further automate the regression suites might be preferable and could benefit both the teams.

Know release cycles and plan to fit in
: Merge moratoriums caused some headaches for the India Domestic team, resulting in that big spike on their graph and apprehension that could have been avoided. Working early with the host team to involve all stakeholders, including the product owner and the architecture review teams, in the planning process would have facilitated better coordination. Another best practice would be host teams publishing release cycles for each quarter.

In the end, PayPal's InnerSource experiments in 2016 proved that InnerSource could work at PayPal for core components under restructuring and could be very helpful in determining how and where to modularize. The Symphony project also exposed anti-patterns such as too frequent job changes (which necessitates constant retraining to create continuity of effort) and how brand dilution can occur when a new practice is co-opted by hopeful imitators adopting buzzwords without understanding, which can create bad experiences and kill adoption. Looking back at the questions we sought to answer, we developed some answers:

Could InnerSource work in situations where there was a lot of initial pushback?
: Yes, but it requires careful training, executive support, and timely efforts to combat brand dilution in overcoming pushback.

Could InnerSource work along with other initiatives such as Agile and CI?
: Yes! The only issue in harmonizing with Agile was that the standard Agile advice to physically co-locate teams tends to cause too much verbal transmission of knowledge. Geographically diverse teams tend to build up better documentation as a result of asynchronous written communication patterns.

What kinds of training or coaching would be most effective?
: In-person training is still the most effective, but it doesn't scale. Experiments in scalable online training were under way as this chapter was being written.

What extrinsic rewards would best overcome reticence?

For PayPal, enlightened self-interest is the best motivator. Coming up with ways to link increased collaboration with career gains will be key to continued uptake of InnerSource as a method.

What metrics would be most useful?

Since you are what you measure, you want to make sure you're tracking metrics that reinforce key behaviors you want to encourage. Now that the ways InnerSource can work inside of PayPal are better understood, a metrics suite is under development. It focuses on counting effective collaboration, mentorship, improved quality, and increased velocity due to the accretion of actionable documentation in the form of mentorship/code review advice.

What cultural impediments to successful InnerSource would be encountered and how would we mitigate them?

As mitigations have been sought for the various challenges encountered and described in this chapter, it has become clear that fitting InnerSource into a complex organization with some history and a clearly defined culture takes diligence, patience, and skill. But those very qualities also make the work rewarding.

Acknowledgments

The InnerSource Program at PayPal has enjoyed many champions. First, I'd like to thank the members of the original InnerSource Team: Duane O'Brien, Cedric Williams, and Silona Bonewald. Kanchana Welagedara was an early team member, and more recently Cia Gilbert has joined as well. Bill Scott and Arnold Goldberg, along with Edwin Aoki, have provided executive-level air cover, funding, and support. Lastly the India Domestic team who did such a great job of building their grassroots InnerSource experiment was led by Satish Vaidyanatha, and included Shankari Sadhasivam, Harish Annam, and Siddick Ebramsha.

CHAPTER 6

Borrowing Open Source Practices at Europace

With Isabel Drost-Fromm

Year, place founded: 2011, Berlin, Germany (subsidiary of Hypoport AG, founded in 1999)

Area(s) of business: Financial services

Revenues: ca. €78 million (2017)

Offices: One location (with several remote colleagues)

Number of employees worldwide: 150

Number of software engineers: 70

Year of InnerSource adoption: 2017

Europace AG is a medium-sized FinTech company founded in 1999 in Berlin, Germany, employing about 150 people. It operates Germany's largest financial marketplace for mortgages and consumer loans in Germany, as well as related insurances. With its transaction platform,[1] Europace creates and develops markets for its customers. Its fully integrated system is a digital marketplace that connects financial advisors with almost 500 partners, including banks, building societies, and insurance companies. Financial advisors use the EUROPACE frontend to choose from the best financing offers for their clients, and Europace APIs enable partners to build their own applications based on the company's system. Several thousand users are involved in more than 35,000 transactions, representing a total value of approximately four billion Euro each month. Europace's

1 A service that comes with APIs for integration in external applications as well as two web fronts that can directly be used by customer employees.

platform is compliant with German and European financial (e.g., WIKR/BAFin) and General Data Protection Regulation (GDPR) laws and standards. The company is a fully owned subsidiary of Hypoport AG, a corporation listed at the German stock market prime standard. This standard has been part of the SDAX since 2015, a German stock market index of 50 small and medium-sized companies.

Looking for New Ways of Organizing

The company has a long track record in using Agile methods such as XP and Scrum at the software development level. The next step was bringing the same agility to the entire organization. In 2015, the company started its journey toward decentralized self-organization. We started looking at different governance forms such as Holacracy and Sociocracy, which seek alternative forms of organizational governance based on consent for decision making. Our goal was to build a lively self-organizing outfit supported by frameworks that allowed us to shift responsibility to personnel with the most expertise. We found the ideals of Holacracy and Sociocracy appealing, but could not find comprehensive guidelines for applying these principles to software development. Ultimately, management's research into Holacracy and Sociocracy helped it to accept the InnerSource mission and worldview more easily than most companies would. At the same time, the emerging practices of the InnerSource community enabled us to fulfill our broader mission on empowering and giving responsibility to our frontline developers.

Our fundamental belief is that Europace employees are experts in their field, and very capable of making the right decisions. After all, they are close to the business and have profound knowledge of their customers. What used to be one software development team was split into four autonomous units with roughly 30 employees each. Every unit has their own set of customers and priorities. After the split, each unit became fully responsible for their products and end-to-end process, and each adopted a DevOps culture to secure the collaboration needed between customer-facing team members and operations staff.

At Europace, we dedicate much attention to growth, organizational development, and improvement. The four units are getting support from several central teams, including one called People and Organization (PnO). PnO supports teams in the transformation from a hierarchical, centralized decision-making organization to a higher degree of self-organization. In addition, PnO coaches support our employees with personal coaching and leadership development. Many other companies offer such a level of support and coaching only to senior leadership and management.

Being a medium-sized company, most of our colleagues are located in one building in Berlin. A few colleagues are located in a city nearby (Lübeck), while another few are telecommuting. Because most colleagues that collaborate on a

day-to-day basis are co-located, much information sharing and decision making traditionally happened face-to-face. As the company grows, we are looking to become more facilitating to remote workers, so that we can hire from a larger pool of potential employees who may not want to move to Berlin. We also think that supporting people to work from anywhere makes us a more attractive employer. Ultimately, we believe that happy employees who can find a good balance between work and family perform better.

The decision to support remote workers, however, meant that we had to identify new patterns for collaboration and decision making. While an increased level of team autonomy meant that a lot of decisions could be taken locally, a downside was that development units were getting increasingly isolated and "siloed." While best practices such as code reviews, and in some cases pair programming were already being adopted, thus facilitating knowledge sharing, we still relied heavily on face-to-face communication. Failing to record decisions and their rationales meant that they were hard to track down over time.

Furthermore, there still was a high technical interdependence of teams. Because of our growth, what once was one team developing one software product was split into four units, each with separate customers. As a result, the product teams are still technologically interdependent, and some components have several dependent teams. Despite our efforts to decouple this into several subsystems, development prioritization today cannot happen entirely autonomously.

Starting the Journey Toward InnerSource

Europace had been using some InnerSource practices, but had not formulated a formal InnerSource program. Starting early in 2017, we established our efforts in a more systematic way.

Europace had been using best practices, such as continuous integration and continuous deployment, and modern tools such as Git for version control, issue trackers (some teams had adopted JIRA, others had moved to Trello), and Slack for communication. Some of the teams had been using the hosted version of GitHub, while others still hosted their code on an internal Git server. All developers had read access to almost every code repository or could get such access fairly easily. Technical foundations to work with pull requests were in place but were being used inconsistently.

Adoption of services and tools was mostly driven by a need for automation. Git interacts with automated tools more easily than Subversion, the source control system we used previously. Even though Europace operates in the traditionally conservative financial services sector, the company has a progressive culture of keeping an eye open for new technologies, and we adopt them when it makes our internal work faster and easier. One example of that attitude is our adoption of

Docker, which we adopted during a time when it still suffered from instability on nonmainstream platforms. Initially, we used Docker only for a greenfield project on the fringe to limit risk for our core product. As the upstream project matured and we got more experienced, we rolled Docker out on a wider scale. We consider ourselves lucky to have some engineers in our teams who understand the value of being involved upstream—watching the projects that are vital to our platform, and occasionally even getting active in these projects. Being active in upstream projects made our engineers train and appreciate Open Source workflows.

Why InnerSource?

As mentioned, Europace already had a lot of the important tooling in place, but the way it was being used varied a lot. For us, adopting InnerSource as a branded concept that we could market within the company helped us to consolidate these efforts into one bigger initiative using this label. It also helped to draw in everyone who previously might not have worked together on these separate efforts and make collaboration something that we could design actively. There were a number of challenges that we hoped to be able to address by adopting InnerSource, which we discuss next.

Capturing communication and decisions

While we had been using Slack for communication, a lot of communication was still happening face-to-face because we were all based in the same building. While this is in the spirit of good Agile practices (one of the 12 principles behind the Agile Manifesto states: "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation"), the extent to which we captured the outcome of this communication in some written form varied a lot. Furthermore, the degree to which people could search and find this archive of our communication and decisions varied even more. As a result, it could be very difficult to trace certain development and product decisions back to their origin, unless you were part of the meeting where those decisions were taken—and you could still remember it.

A clear focus on writing decisions down over verbally communicating them helps to alleviate this problem. If convenient, employees were taught to use Slack, GitHub discussions, or other archived media for discussion; if they found it more convenient to engage face-to-face, on the phone, or in a video call, they were urged to summarize the key points afterward in an archive. In order to achieve that focus, we started with a core group of InnerSource enthusiasts who were leading by example. Going forward, those people were helping their peers to adopt this communication style. The insights learned by those leading employees were gathered and finally published on our company tech blog—with nonsensi-

tive content posted publicly,[2] and access to sensitive content restricted to employees only.

Improving task transparency

Most teams used either Trello boards or JIRA for planning. Although these tools can be configured to make decisions visible to outsiders, teams often deliberately limited both read and write access to these resources to their team only. Standard workflows for these tools are designed around one cohesive team instead of inviting collaborators from the outside to participate. That made cross-team collaboration unnecessarily hard and we've found this to be a common source of misunderstandings. GitHub does a much better job of exposing information by default: it encourages you to keep source code and issue management in one project, makes it easy to link relevant discussions (commits, issues, and pull requests) together from any location, and makes it easy to search across issues, communications on pull requests, and code. Our goal with InnerSource was to make these discussions as transparent as possible. This goal was supported by a general trend toward an increased level of transparency in our organization at all levels of the company. This included support from our senior leadership, who wanted to move away from traditional hierarchical decision-making processes in favor of self-organizing teams.

Finding a better balance of autonomy and collaboration

While we wanted teams to become as autonomous as possible, our pursuit of this goal led to the point where they became isolated silos. This in turn hampered cross-team collaboration because teams had different priorities. Starting a conversation about InnerSource was one of the forces that brought our focus back to collaboration—even across unit boundaries—while at the same time it provided the tools, patterns, and processes for units to remain as autonomous as possible.

Evolving teams and accountability

Teams grow and evolve as members may join and leave a team once features are finished or projects come to an end. When people leave, they take their knowledge and expertise with them, and this can become a real problem if there is nobody left on a team who can be held accountable for certain parts of the code. Introducing the concept of Trusted Committership helped make accountability transparent and self-selected. For one thing, introducing the concept of a Trusted Committer removed the tension between wanting to collaborate, without handing out write access to the codebase from the beginning. That way, mentoring new developers was built into the process, independent of whether they were new

2 See *http://bit.ly/2HPofLf* for the basics and *http://bit.ly/2JLKbZA* for the first cross-team insights.

hires or seasoned developers who were working in a different team. This in turn meant that mentoring itself gained in value as a contribution. Frequent contributors to a project were invited to serve as Trusted Committers as well. We found that people took it upon themselves to take responsibility for what was now their project. This self-selected responsibility is much preferred over inheriting responsibility by virtue of simply being hired into a certain team, because people who do so are signaling that they want to develop as leaders within the community. As someone evolves and rises to become a Trusted Committer, they will become part of the movement, which helps to increase their motivation.

InnerSource Experiments

The first thing that changed a couple weeks after the introduction of the InnerSource program was the more widespread adoption of pull requests as a model for collaboration: having a name for collaboration that comprised that workflow made it easier to talk about it and convince others to join.

Europace units each work to fulfill the needs for one type of platform user (offering mortgage products through our platform; selling mortgage products; offering and selling loan products; and exclusively supporting our biggest customer). Typically one unit has at least ten team members. As units grew, teams of two to eight people were formed. A team would take responsibility for adding new features to the platform, or would improve and refactor existing features. Typically such a task takes from a couple weeks up to a few months to complete. Once a team's task is finished and has reached a stable state in production, team members are (at least as of today) free to form new teams or join other teams.

In one of the units, developers were already striving for a four-eye principle in code development. Although that level of oversight was not strictly enforced, people were strongly encouraged to work in pairs for all development work. Each team would also strive to have at least one more seasoned developer as well as someone who could serve as a product owner for that functionality. As this unit grew toward 30 members, it became unfeasible to get support and guidance from experienced members through pairing alone. Finding a common time slot where people could work on the same piece of code became increasingly a challenge. Instead of pairing up, people could resort to an asynchronous workflow. Using pull requests meant that people could still benefit from a second pair of eyes and receive mentoring and feedback before merging.

The second change happened a couple months later. This time InnerSource was a means to improve cross-unit collaboration. One team within unit A had developed a microservice that a team within unit B wanted to reuse. Deployment of the microservices remained separate, but coding happened in the same repository. Resorting to InnerSource meant that people could stay at their desks and within their teams, but still be able to collaborate. The developers working on

that project were coming from a world where write access to repositories was granted as soon as developers were assigned to a project. In this project, we made a few changes that people had to get used to. Write access was handed out in a meritocratic way, rather than by default. Only after contributors had earned the trust of the maintainers of that service did they get write access to the repository. This essentially converted the new contributors to Trusted Committers; as such they were publicly recognized for their contributions, but they also became responsible and accountable for the further development of that service.

As we were increasingly working from different locations, much of the discussions around requirements, architectural design, and deployment already happened in written form—though in this iteration separate from the code, stored either in a Trello board that wasn't linked to code development or in an ad hoc Slack channel that was created just for this collaboration project. Working across units required us to improve our documentation to onboard new developers as well as to document best practices for making contributions.

The third change was brought about by the second cross-unit collaboration. This time, we decided to get coding discussions closer to the repository, linking issues and the surrounding discussion to the commits made.

Steps Toward InnerSource

At first, the definition of "InnerSource" wasn't much more specific than "apply Open Source collaboration principles to projects inside a company." The practices that we established at Europace aligned closely to best practices within the Apache Software Foundation (ASF) (see Chapter 2) as well as in Open Source projects in general. But to explain practices internally to team members who didn't have much knowledge of the ASF, what helped communicate the bigger picture and identify a viable language were observations and best practices shared by the InnerSource Commons community.

The path toward a more formal InnerSource initiative at Europace started with establishing a specific role responsible for coordinating these efforts. With over a decade close to and within the ASF, I was brought onto the team to give members a deep dive into how globally distributed organizations work. Hiring someone with several years of experience in running Open Source projects, mentoring contributors, and mentoring projects that joined the ASF helped us understand the reasoning behind the principles and patterns InnerSource suggests.

However, it also meant that from the start it was clear that collaboration patterns would have to be evaluated for their utility within our context, namely, a medium-sized company, operating in Germany, located almost exclusively in one office building (with a handful of remote colleagues), subject to German employment and privacy laws. As one obvious example, that means not leaking any pri-

vate data from customers or employees to third-party services like Slack or GitHub (unless they have agreed to this data being processed on the respective service).

We started our approach by focusing on two goals. First, we needed to get people on board with the idea. Second, we had to provide an environment that was separate from the production environment so that we could try out new processes and roles.

Getting People on Board

To get people on board, we first identified different groups of people that would be affected by our InnerSource program.[3] We identified four different groups of stakeholders:

- People who had already been independently leading some of these efforts, prior to our InnerSource program
- People who were likely to block these efforts
- People who likely would greet these efforts with enthusiasm and joy
- People who knew a lot about the company history and context for the initiative, its potential roadblocks, and its benefits

The first step was to get one-on-one conversations going to understand where people stood, what benefits InnerSource could bring to them, and what concerns they might have. To make these conversations happen in a casual manner, we usually had them as lunch conversations.

After it was clear that almost everyone was facing similar challenges in collaboration—in particular when crossing unit boundaries—we brought interested developers, team leaders, and project managers together, again in an informal out-of-office, pay for your own lunch setting. This meeting turned into a monthly recurring roundtable where InnerSource (and Open Source) enthusiasts would exchange their ideas, share experiences, and provide general support to each other. What we managed to achieve was to retain at least one motivated participant from each unit over the course of the entire first year.

To make transparent who was helping drive the InnerSource initiative, we decided to use the concept of Trusted Committership also at the meta level: those who regularly served as InnerSource mentors for their teams, answered questions on InnerSource questions, and encouraged others to adopt InnerSource best practices were asked to become Trusted Committers. Giving them this role in a

3 A special thanks here to Henri Yandell for mentoring through my first days and weeks in the role as Open Source Strategist.

formal way highlighted what they had been doing all along. In many cases it led to increased motivation, because people now felt part of a movement. As a result, we anticipated that they would spend more time and energy driving InnerSource adoption with their peers.

Leading by Example

When starting the InnerSource program formally, the first question that we had to address was: how do we organize the InnerSource program itself? In particular, we had to address questions such as:

- How should InnerSource tasks be made transparent?
- How do we get people to ask questions to the community out in the open?
- How should we track documentation that we collect over time?

We made a conscious decision to design communication processes independently of existing internal communication practices, but reuse tools that were already available in the organization. The goal was to establish a project environment that was as close to your typical (Apache) Open Source project as possible, so others could experience the benefits (but also the challenges) firsthand, instead of just hearing about them. We made that environment as open as possible to invite everyone to participate and shape the InnerSource initiative. It also meant that people who were unfamiliar with the technology and process could try them out in a safe environment that didn't affect any production systems—just in case they would break anything. As a result, we decided to first create what Vijay Gurbani and his colleagues have described as an "infrastructure-based" InnerSource program (see Chapter 1).[4]

We set up a GitHub repository (called "ep-innersource") to track everything related to our InnerSource work. We also created a dedicated Slack channel for discussions about the InnerSource program. Initially, we also mirrored all activity coming from the InnerSource GitHub repository over to that channel. But because this soon became overwhelming, activity from the InnerSource GitHub repository was forwarded instead to a second separate Slack channel. This was inspired by how Apache projects work with mailing lists: any change to the repository, issue tracker, or on the wiki is mirrored to a dedicated commits mailing list. That way, there is a single source to trace back decisions and modifications to project artifacts. In other words, if it didn't happen on the mailing list, it didn't happen.

4 Vijay K. Gurbani, Anita Garvert, and James D. Herbsleb, "Managing a Corporate Open Source Software Asset," *Communications of the ACM* 53, no. 2 (2010) 155–159.

The content and function of the GitHub repository focused on the InnerSource initiative itself. The repository's purpose was twofold. First, it served as a place to list current items that we worked on, in order to be fully transparent. The issue tracker in GitHub proved to be quite useful for this purpose. Second, the GitHub repository served as a place to collect persistent documentation of previous discussions and decisions. Storing these in Markdown format in the repository itself worked well for that purpose.

After running the first two experimental InnerSource projects, we noticed some recurring issues. One issue was deciding who would become responsible for monitoring the service that was being developed as an InnerSource project. At some point I documented what we learned in the pattern format she had first seen on the InnerSource Commons website, and posted the resulting text as a pull request to be included in our internal InnerSource documentation. Through regular GitHub review comments, others who had participated in those experimental InnerSource projects could make suggestions on how to change the text to better reflect reality. After the document was finalized, I submitted it publicly as an InnerSource pattern. The ability for anyone to participate in the process of content creation, seeing the result published externally, and giving credit to contributors, increased the motivation, engagement, and participation of those involved.

InnerSource Principles

Developing a common internal understanding of what InnerSource meant for Europace required some more effort. Together with one of our UX experts, we developed a concise set of six principles of InnerSource at our organization. Again, the resulting draft was shared as widely as possible within Europace. We announced it on the internal InnerSource Slack channel. At the same time, I created a GitHub issue in the ep-innersource project tracking progress on development, and put a link including a brief explanation out on the "general/smalltalk" Slack channel. Finally, I reached out personally to people with the specific request to get the document reviewed. Again, everyone was invited to give feedback, ask questions, and make changes. As the document stabilized, a final message was sent out in the spirit of Lazy Consensus. Everyone was invited to go through one more cycle of review until a week from that message. A lack of response from any particular person implied you agreed to the current state.

The processes described here show how we learned to make decisions out in the open, for everyone to see and participate in, much as described in Jim Whitehurst's book *The Open Organization Workbook*. This meant that people were much less surprised when a certain decision was finally announced, and that those interested in the topic could participate freely and openly. The result was a shared and earnestly accepted set of principles and rules:

Make everything open, transparent, and findable
: Project artifacts (source code, documentation, etc.) should be accessible to anyone within the company and easy to find. Barriers to participate in a project should be as low as possible.

Encourage contributions over feature requests
: All stakeholders of a product see themselves as potential contributors—and are being treated as such by the project. In the spirit of a meritocracy, we assume that good contributions can come from anywhere. Contributions remain suggestions—communication and coordination remain important before investing large amounts of time in an implementation that might be rejected later for reasons that in retrospect look trivial. Contribution rules for participating in a project are openly documented and binding on those participating.

Favor written over verbal communication
: It should be possible to participate asynchronously in a project. In order to enable asynchronous decision making at the project level, communication needs to happen in written form. Project-relevant discussions that don't happen via the project's selected main communication channel should be summarized and archived. That way, all relevant communication can be read and followed by all colleagues, even long after the discussion happened, ensuring that a larger number of people can participate in the project. As a side product, a basic level of documentation of the project history will start to accrete. There are exceptions to the rule, in particular for discussions related to people issues or security issues, so communication needs to happen as publicly as possible, but as private as necessary.

Embrace mistakes
: When communicating in mainly written form, mistakes can no longer be erased but potentially remain readable company-wide. This requires a culture where mistakes are seen as learning opportunities and chances for improvement.

Welcome all contributions
: All contributions (source code, documentation, bug reports, constructive discussion, marketing, user support, UX design, operations) are valued. Contributions to projects are rewarded: those who add value to the project are invited to become Trusted Committers. All Trusted Committers are visible throughout the company.

Maintain a project memory
: Written advice is allowed to accrete in a persistent, searchable archive. All project-relevant decisions, design documents, and other artifacts are kept in

that archive. All communication can be referenced by stable URLs, which ideally can be searched and accessed by all employees.

InnerSource Results

Over time, we saw some major benefits in how our InnerSource projects were collaborating. Adopting InnerSource principles meant that the process of coding as well as the decisions that led to code changes became more transparent. Previously, our culture focused too much on the "minimal documentation" side of the Agile principle of "Working software over comprehensive documentation," and it was often unclear why a certain piece of code had been written the way it was. Moving decision making, architecture design discussions, documentation, and code closer together meant that changes were easier to track without the need to introduce many formal requirements for documentation up front. This in turn raised the interest in InnerSource among people who were not developing source code on a daily basis—notably project management and UX designers.

Using best practices for asynchronous decision making helped to reduce our dependencies on face-to-face communication. It became easier to pull in expertise from people initially not involved in the development of a feature without having to repeat verbally what had been discussed about that feature earlier. Those practices also improved transparency: they helped everyone see which projects were currently under development, what state they were currently in, and what challenges they faced. Making challenges visible helped resolve them more quickly. Also, that way, it became less likely that teams would go off and develop different solutions for the same technological challenges.

Valuing pull requests as a means to collaborate meant that senior project members were able to provide input and support even when not assigned to work on the same feature as the developer who had questions. It also meant that newcomers to the team could get up to speed faster: the ease of getting code reviews through pull requests meant that there was a safety net that encouraged new team members to make valuable contributions quickly after starting on the team. The asynchronicity meant that developers could collaborate across unit boundaries, across different locations, and across different time zones. We no longer had to first colocate people, synchronize their meeting schedules, and release plans.

Establishing Trusted Committership as a reward led to developers feeling more accountable for the projects they had contributed to. They became more engaged with the projects they had gained trusted committership for. Handing it out as something that had to be accepted by the nominee helped with making that accountability something that is self-selected and a conscious decision.

At the code level, we observed an increase in cross-unit collaboration and outside contributions to projects. Both the quality of reviews and the quality of the code

itself improved substantially. We observed a speedup in development, because the work shown on a pull request was a clearer basis for discussion than informal requirements communicated across unit boundaries. Iterating on actual code, even if in a draft state, helped speed up mutual understanding, decision making, and thus implementation.

InnerSource: One Year Later

At the time of writing, about a year after we started our formal InnerSource program, we are seeing that cross-team collaboration is happening. People are using pull requests and have started to see the value of having documentation produced as a side effect of simply communicating online, using either Slack channels, GitHub issues, or pull requests for communication.

Currently our InnerSource projects are relying heavily on GitHub for collaboration. As a result of the GitHub workflow, source code, issues, and code review comments are easy to link together. Particularly where some of the participants are working remotely, in other units, or on different time schedules, much of the regular project communication is happening in written asynchronous form within GitHub issues and pull request code reviews. As a result, there's a lot more communication that automatically accretes over time.

Some projects have started using the concept of Trusted Committers. The concept serves two purposes: as a way to communicate that Trusted Committers are the ones accountable for the project, but also as a way of rewarding valuable contributions to InnerSource projects.

InnerSource Challenges

While InnerSource has brought several benefits, we've also identified some challenges during our transformation.

Building Trust in Written Communication

It can be challenging to trust the judgments of people you work with if you don't meet on a day-to-day basis. It can also be intimidating to expose yourself to critique from others if you don't know them very well, which, again, is an effect of not meeting them every day. In one early experiment involving two units, we had five engineers collaborating on a microservice that was needed by both units. Initially, some of the developers were using GitHub, but still committing directly to master without going through a pull request model that would have allowed others to review and comment on the code. After considerable encouragement, everyone finally agreed to work with pull requests.

I had the chance though to observe people's behavior both online and offline. I noticed a difference between how reviews were conducted online in GitHub

repositories, and what people on the InnerSource project were chatting about offline at their desks: some of the technical issues that people were complaining about in person never seemed to make it into the digital archive. In addition, people were discussing about how certain reactions they had gotten in their code review should be understood.

I concluded that there was still an invisible barrier between participants in different units. As a quick fix, we organized a shared, informal lunch session to get everyone together to chat and socialize as a form of team-building. The goal was for people to get to know each other, to get participants in the InnerSource project more familiar with how we were communicating, and hopefully to make people realize just how much common ground they shared.

In the days and weeks that followed, we saw that communication steadily improved. Spotting friction in a distributed, online setting tends to be much harder than in a co-located setting. In our experience, finding ways to remove that friction in a distributed setting is also more important than in a colocated setting, because textual communication has far less bandwidth than personal communication. As a result, misunderstandings can arise more easily. Making sure that people have met each other in person makes communication through written channels easier.

Scaling InnerSource Beyond First Experiments

One challenge we face is to achieve a more widespread adoption of these best practices. Some people have started to communicate the benefits of those changes in the way they work, inspiring others to pick up this mode as well. Others have started to give presentations on their experiences and write blog posts to spread the word further. However, we need to do more work to evangelize our InnerSource program. We also need to spend more time teaching our colleagues the values and principles that InnerSource brings as well as the tooling and workflow that our initial experiments showed to be effective.

Another challenge that we face in scaling up our InnerSource program is related to the nature of our industry. While for many projects there are no legal restrictions before moving to private repositories on GitHub, there are other codebases for which the situation is not so clear. Currently discussions are ongoing around questions such as:

- Which parts of our codebase must be hosted on-premises?
- Can we host our source code on an external platform at all?
- What kind of information has ended up in our codebase over time: is it only code, or do our repositories contain any access keys, or worse, any kind of personal data that made it into test data?

Getting Other, Nondeveloper Colleagues on Board

Another challenge lies in getting product managers on board as well. Managers often don't perceive themselves as potential contributors, because they don't write any code. However, it makes perfect sense for someone with a deep understanding of project priorities and customer needs to become a Trusted Committer for other types of contributions, including requirements and documentation.

Crossing the Boundary Toward Open APIs and Open Source

At the time of writing, people are starting to see the potential of InnerSource when combined with Open Source activity. Colleagues already active in upstream Open Source projects have an easier time switching between their work at Europace and in those Open Source communities, because they use similar development and communication processes and tooling. This makes it very easy to switch back and forth.

We also saw how our InnerSource program became a "training ground" for participating in Open Source projects. The opportunity to engage in Open Source practices within our organization helped lower the barriers to start contributing to Open Source projects. Some of our colleagues who hadn't been active in Open Source projects became motivated to participate upstream. Other colleagues who worked on external APIs of our platform are using what is being developed through our InnerSource initiative as a basis to streamline communication and collaboration with external partners, as well.

Conclusion and Future Outlook

Although rolling out InnerSource in an organic as opposed to a top-down fashion meant that things were moving fairly slowly in the beginning, we found the change it brought to be sustainable. Moving forward, it helps tremendously to have people on the team that have Open Source experience, because many of the best practices can be carried out without too much tailoring or adapting. It also helps a great deal to turn people into ambassadors to spread the word.

Although it's quite easy to explain the processes and tooling of Open Source and transfer that to an internal development environment, the real challenge lies in instilling an attitude and mindset in downstream users so that they feel empowered as active contributors instead of passive consumers. Despite the challenge of making such a cultural change within the company, this is also where the value of InnerSource lies, because as a company we're interested in improving collaboration and leveraging the motivation, expertise, and creativity of our staff. Furthermore, InnerSource also helps to get more people active in contributing to upstream Open Source projects, which benefits the wider Open Source ecosys-

tem. Ultimately, this leads to better collaborations within communities, with partners, and with customers.

Acknowledgments

A big thank you to my colleagues at Europace who helped with this chapter: Tobias Gesellchen, Leif Hanack, Anja Jesiersky, David Rehman, Sandra Sauske, Julia Schenk, Heike Schmidt, and many more who supported the InnerSource initiative.

CHAPTER 7

Connecting Teams with InnerSource at Ericsson

With John Landy

Year, place founded: 1876, Sweden
Area(s) of Business: Telecommunications
Revenues: ca. €18.9 billion (2017)
Offices: Serves customers in 180 countries
Number of employees worldwide: ca. 97,500
Number of R&D engineers: 23,600
Year of InnerSource adoption: 2013

Ericsson is a Swedish multinational company operating in the telecommunications industry. Founded in 1876, the company now has operations in about 180 countries. Ericsson is a global player in communication network solutions, and its networks carry about 40% of the world's mobile traffic. Ericsson R&D employs about 23,600 people worldwide. Ericsson's R&D operations in Ireland are based in Athlone, which employs over 800 software engineers working on Ericsson OSS (Operation Support Systems).

The Changing Telecommunications Landscape

The telecommunications landscape has changed dramatically in the past 30 years or so. A key driver in this evolution is the emergence of new standards and technologies for mobile networks, which have seen very dramatic growth since the early nineties. Telecommunications traditionally focused on fixed networks that were managed through large exchange stations, but mobile networks play an

increasingly important role, with new standards emerging continuously (GSM, 3G, 4G, and 5G). The nature of this infrastructure is significantly different as well: rather than a relatively small number of fixed network exchanges, mobile networks consist of a very large number of base stations.

As mobile networks continued to grow and evolve in the 1990s and into the 21st century, shaping the world of communication, many other new technologies started to emerge and influence the growth of telecom. IT technologies, the cloud, virtualization, microservices, containers, AI/ML, adaptive policy, and the recently emerging serverless computing are fundamentally changing the telecom networks. The pace of change is enormous and future telecom networks will deliver services across industries.

These technologies are changing the entire telecom network architecture, bringing innovation, agility, and automation to telecom networks. The networks now consist of a cloud-based infrastructure, accelerated data plane, virtual switches, controllers, and orchestrators, providing enormous flexibility.

Network equipment vendors need to absorb and induct all these new technologies into their products and solutions, R&D, and various processes. If a common architecture and reuse of platforms across various solutions and products is not leveraged, the cost of creating and delivering these products will be unsustainable. This is where InnerSource plays a major role within an organization. InnerSource has become vital more than ever in this time of highly accelerated growth and rapidly changing technologies. In this chapter, we investigate various areas where Ericsson teams leveraged InnerSource.

As new standards and technologies emerge, the management systems for these networks (which are called "operations support systems") evolve as they are extended and merged over time. While considerable work has been done to ensure the architectural integrity of these systems as well as to scale up their capacity and performance, it became clear to us at Ericsson that a new operations support system would be needed to support future technologies and trends such as the Internet of Things, which differ dramatically from the older, fixed-network technologies in terms of capacity and performance.

Around 2010, a decision was made to develop a new operations support system called Ericsson Network Management (ENM) to replace the legacy platform. Development of ENM started in 2012 as a greenfield project to replace the legacy systems. Around the same time, Ericsson had started a large-scale Agile transformation using the Scrum development approach. Before this, Ericsson followed a rigorous plan-driven waterfall approach, which included clearly defined phases such as pre-study, feasibility study, execution, and verification. As many other companies following the waterfall approach have experienced, this often led to projects going over time and budget, which is why we started our Agile transformation.

Why InnerSource?

ENM has a layered architecture, each layer consisting of one or more components that offer services to the next layer. In essence, each layer provides platform functionality to enable feature development. As a Product Development Unit (PDU), we decided to create this set of architecture layers without creating corresponding, dedicated platform teams. We didn't want to create platform teams that would shift development efforts from being *product-focused to platform-focused*. From experience in the past, we've seen that feature teams tend to have highly demanding expectations about what features a platform should have, which can lead to large numbers of requests for features that would flood the platform team's backlog, causing it to become a major bottleneck. This is a common problem in platform organizations.[1] Our customers are interested in *features*, not necessarily in the platforms. Our organizational choice was also informed by our Agile and Lean transformation journey. It made us more aware of the principles of Agile and Lean development, which emphasize simplicity and prevention of wasteful effort. Therefore, our normal development teams were organized as *feature teams* who are responsible for delivering features across the layered architecture.

When a team decided that it needed support within an underlying platform for something new—or decided that some code they were developing could benefit other teams by being included in the platform—they would insert the necessary code into the platform. This was the architectural decision that drove our InnerSource program. We recognized that having multiple teams work on shared code required new forms of coordination and decision making. We also realized that Open Source projects offered a good deal of guidance for how to make such collaboration effective.

To this end, the PDU created a small unit called "Community Developed Software" (CDS). The role of CDS was not to provide a traditional platform team, but to espouse and facilitate collaborative development using Open Source development practices.

Scaling Up Development Capacity

Because each horizontal component was potentially a bottleneck, within CDS we were very keen on helping feature teams implement all the functionality that they needed. This way, we would be able to scale up the development capacity, both internally and externally. In CDS, we established core teams (which we discuss later) responsible for providing components with well-defined interfaces that fea-

1 Dirk Riehle, Maximilian Capraro, Detlef Kips, and Lars Horn, "Inner Source in Platform-Based Product Engineering," *IEEE Transactions on Software Engineering* 42, no. 12 (2016): 1162–1177.

ture teams could use and extend to implement their features. Changes to these components would no longer be sitting on a dedicated platform team's backlog, but on the feature team's own backlog.

We hoped that this would lead to a scaling of development capacity, because any team within Ericsson (not just ENM) could use any of the assets and might contribute to its development or improve it. Furthermore, we considered the idea that individual people who were interested in contributing to Open Source might also be interested in contributing improvements to our InnerSource components. The latter didn't really materialize on a large scale.

Reducing Waste

Initially, we were focused on reducing the wasted time and energy negotiating between platform and feature teams, but we also had an unexpected result. If a Product Owner's team worked on a contribution for the platform, it would be on *their* backlog, rather than on the platform team's backlog. As a result, Product Owners became much more pragmatic and critical when deciding whether they needed certain features. Features were built only if they were really necessary to bring the product to market—therefore, we built far fewer features that weren't really needed. In terms of "lean thinking,"[2] this helped us to reduce waste.

Improving Quality

Once we settled on this idea of applying Open Source techniques within our company, we had to address skepticism at different levels, including management and projects. As we started to investigate the best practices of Open Source development, we identified other motivations that could help us to get management buy-in. One reason was to improve quality—or what Eric Raymond has famously called *Linus's law*: "Many eyeballs make all bugs shallow."[3] Our argument was that openness wasn't to be feared but to be embraced, because it could lead to better quality simply by having more engineers look at the source code. Having a higher level of reuse of our components would also help to achieve this because those components would be tested in more contexts.

Starting the Community Developed Software Program

When we started out in early 2013, we weren't aware that other companies, such as Bosch (Chapter 4) and PayPal (Chapter 5), had similar initiatives. We weren't aware that this idea of adopting Open Source development practices to improve internal development was called InnerSource. At this time, the InnerSource

2 James P. Womack and Daniel T. Jones, *Lean Thinking* (New York: Simon and Schuster, 2003).

3 Eric S. Raymond, *The Cathedral and the Bazaar* (Sebastopol: O'Reilly Media, 1999).

Commons community hadn't been founded yet, and so we were finding our own way.

We started out by investigating the principles of the Open Source development paradigm. We realized that we could easily adopt Open Source tooling as well as Open Source development practices and techniques to streamline our processes. However, a more important aspect was that of the development culture. We looked at the Apache Software Foundation (ASF) (see Chapter 2), and attended the 2013 FOSDEM conference (a leading annual convocation of European free and Open Source developers in Brussels) to meet people working in Open Source communities in order to get a feel for the culture of Open Source. We also looked at some successful Open Source communities such as Jenkins.

Creating Core Teams

Our new initiative created core teams responsible for platform components. However, unlike traditional platform teams, these core teams aimed to build a community to develop their components. This was done by adapting our Open Source learnings to work within our organization. We decided to keep core teams small. This was to avoid the expectation they would implement all requirements for feature teams. Instead, they would create the environment for feature teams to extend components as needed.

The core team for a component was responsible for creating a "minimal extensible component" and the environment to support feature teams in successfully extending that component. The core team was empowered to determine how to create this environment, but common tasks included:

- Identifying and staffing key roles
- Establishing the rules of the game, including design rules, test requirements, and review and approval for a contribution
- Onboarding of new members of the community
- Assisting contributors
- Codevelopment of new functionality
- Comaintenance
- Managing technical debt
- Modernization as new technologies emerged

As the CDS unit and process evolved, we identified some common roles:

Product Owner
: A core team's Product Owner looks after the component's backlog, just like a Product Owner in feature teams does. The Product Owner has a "feature

mindset," and engages with the feature teams to help identify and implement their changes on the component.

Architect
: The Architect ensures that the component has an appropriate architecture and maintains its interfaces, so that its desired quality attributes (or "ilities"), such as performance, scalability, and extensibility, can be achieved. For CDS components, extensibility was a particularly important quality.

Code Guardian
: The Code Guardian is responsible for making sure that all code contributed to the component is of sufficient quality and complies with coding standards and rules.

Some core teams may have a few core developers, known in Open Source as "trusted lieutenants" and by some InnerSource programs as "Trusted Committers" (see, for example, Chapter 5). But even their code contributions are rigorously reviewed before committing to the repository to ensure high quality.

The three roles that make up the core team would each be given to people with the extensive expertise that would be needed for that role, and together these roles complement each other. When the goals of these three roles occasionally come into conflict, they find ways to resolve the conflict. While Architects are important to establish a vision for a component and are experts in their specific domain, any tendency to "overengineer" a component would be opposed by the Product Owner, who makes sure that components offer the interfaces that the feature teams really need. Code Guardians play an important role, as well, and typically the people serving as Code Guardian would know the component well.

Setting Up Contribution Rules

Within CDS, we anticipated anxiety among the core teams that contributions wouldn't be of sufficient quality. To prevent this, we knew that it was important to establish clear coding standards and rules to reassure the core teams, who ultimately bear responsibility for the components. This way, contributors would have clear guidance while preparing their contributions, while the core teams would be more confident that contributions were of good quality. This would mean that they wouldn't have to waste any time reviewing any contributions of poor quality. After drafting these coding standards and rules, we had them reviewed by the Governance Council (discussed next) before we published them to our internal community. We also looked at Open Source governance practices such as Apache for inspiration.

Setting Up a Flexible CDS Infrastructure

Once the coding standards were in place, we launched a beta version of our portal and infrastructure to host code repositories, and to facilitate workflows for making contributions. Based on what we observed in Open Source projects, we felt that it was important to have a "one-stop shop" portal that developers could visit to find out about our InnerSource components.

Rather than enforcing a specific infrastructure (such as a specific code repository and issue tracker) on the community, we designed a frontend that encapsulated and hid the specific tools that were used by different component teams. We didn't want to reinvent existing tools, processes, and build systems, and we wanted teams to be able to keep using whatever tools they were using already. Our portal facilitated this, as long as the tool chosen by a team had a programmable interface, such as REST,[4] that permitted programmatic access to the tool. Furthermore, the build infrastructure of many of the components was already in place, and this was often quite a complex setup, due to the complexity of the product. Requiring teams to reinvent their processes and build systems would have hampered participation in our CDS program.

Creating the Governance Council

When we proposed our initiative to the CTO Office, one recommendation we got was to create a Governance Council to oversee our CDS program. The CTO Office felt that a strong technical drive and technical ownership was very important, so the council consisted mainly of technical experts. This would prevent too much involvement from nontechnical managers, who wouldn't focus on the potential benefits but might instead be distracted by focusing on their specific team's resources. We identified people in key technical roles and who we thought were key technical influencers, asking for their input and inviting them to take place on the council. We purposely didn't simply invite *all* people within a given role. Instead, we handpicked those people who stood out for their technical expertise and reputation within the company.

Selecting Components and Development Models

Once we had the portal in place, we had to populate it with components. When we started out, we handpicked a number of common components that we identified in the overall architecture. As we selected components to be incorporated into the CDS program, we were looking for the low-hanging fruit: those that seemed to be most suitable and could lead to quick wins to build up some

4 Roy T. Fielding and Richard N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology* 2, no. 2 (2002): 115–150.

momentum. These tended to be components that were common to different applications and could be easily extended.

One exercise we used to identify changes to components that could be developed as InnerSource components was what we called the "Architectural Runway." This exercise involved Product Owners and Architects from a number of teams who would identify candidate changes needed for their requirements. By physically overlaying the different roadmaps, we could identify common changes. This exercise also helped to establish a timeline based on when teams would need each change, and who would implement the change.

Most engineers would be well aware of the shortcomings of having dedicated platform teams. The traditional scenario to get a feature into a platform would be to create a requirement and have it approved, after which it would sit in the platform's backlog. However, it wouldn't necessarily be delivered if the development team didn't rank it as a high enough priority. This experience helped us to convince people to adopt our new approach of having application teams implement functionality themselves, rather than waiting for a platform team to do it.

Development Zones

Because we recognized and acknowledged that not all components would be suitable for InnerSource development, we defined a number of development models. We refer to our CDS environment as the "green zone": anyone within Ericsson is free to download these components and make changes as they see fit. We pursue the CDS model when we believe a viable community can form around the component, when we think that following this model would avoid a bottleneck, and if the component's architecture is amenable for cross-unit collaboration. Obviously, whether these expectations will hold is not always clear in advance. We can't tell whether a community will form. If it doesn't, we change the component's development model to the traditional way of working.

When a community does form, however, this doesn't mean it will last. It may enjoy a successful series of development bursts, after which the community falls into a natural decline once the component no longer needs further work. In such cases, we simply revert to the standard, non-CDS development approach.

There's a few situations in which the InnerSource approach isn't used. We don't have the authority to force the CDS approach on any one component—as we mentioned, the best we can do is to sit down with the Product Owner and convince them of our case. But for some components we understand that InnerSource simply isn't the best approach, and for those components a dedicated team provides the full implementation. For example, some components play a critical role in achieving certain quality attributes such as performance, and realizing those benefits requires a high degree of specialized knowledge. If a feature

team requires additional functionality or changes, the feature team and component team would discuss what changes are necessary.

We also don't pursue the CDS approach for those components that we want to control more tightly in terms of their evolution and versioning. Typically, these aren't individual components, but "component ensembles,"[5] which were preconfigured and preassembled components that represented a part of a vertical feature. We have moved those component ensembles out of the CDS environment and created a separate space for them, which we call the "blue zone." Any component in the blue zone can only be used "as is"—treating them as more traditional off-the-shelf components—and as such these aren't pure InnerSource components. While these blue zone components can be extended in limited ways, any changes to the core of a blue zone component have to be done in close collaboration with the component's owner.

Collaboration Workflow

Once we identified components and a Code Guardian for each one, we ensured there was a clear branching and merging strategy. While the "one-stop shop" portal provided the necessary infrastructure to access the source code, this wasn't enough to get teams to make contributions. We realized early on that we needed to take away any barriers that could hamper collaborations. We knew that the "user experience" for potential contributors had to be very smooth. They had to easily understand the process for submitting contributions and how Code Guardians would review them.

Luckily, Continuous Integration (CI) was already used by most components, and we leveraged that toolset. A CI infrastructure that would help contributors understand the impact of their contribution (for example, whether it breaks the build) was essential to establishing a community. We documented the collaboration workflow for the CDS to make it easy for developers to navigate the source code and understand the contribution process. Our workflow defined the following activities:

Discover

> We make sure that each InnerSource project is well documented, so that interested engineers and teams can discover projects that may be of interest to them. Getting familiar with the projects on the CDS platform is the first step to participation.

5 Scott A. Hissam, Robert C. Seacord, and Kurt C. Wallnau, *Building Systems from Commercial Components* (Boston: Addison-Wesley Professional, 2001).

Discuss
: We provide discussion forums so that engineers can propose and discuss issues or contributions. A contribution typically starts with a discussion between a contributing team and a core team's Product Owner and Architect.

Issues
: As an Agile shop, we use the widely adopted JIRA issue tracker from Atlassian, which developers can use to create and communicate issues. Once the discussions on the message board lead to agreement on the what and how of a contribution, the contributing team can take control of the story.

Clone and implement
: Once the discussion agrees on the story and a contributor takes on the assignment, development follows a fairly standard process of cloning the source code and making the necessary changes.

Submit
: Once implemented, the change is submitted for peer review through the Gerrit central server. A Code Guardian reviews the code to ensure that the contribution is of sufficient quality and follows our agreed coding standards and rules.

Community-developed software CI
: Once the code passes the CI build system successfully, the contribution is committed, and will become a part of the next release.

Making Collaborations Happen

Our CDS program focused heavily on scaling up our development capacity and preventing bottlenecks. While we were hoping that individual engineers would contribute as well, similar to how individuals contribute to Open Source projects, this didn't really manifest. Instead, we saw that contributions followed a more systematic process of "pre-negotiation," similar to how contributions are discussed in The Apache Way (see Chapter 2). Contributions were usually in the form of user stories or feature requests that were planned together with a component's Product Owner, rather than ad hoc changes to the code. Feature teams would discuss ahead with a core team what contribution they'd make (the "discuss" phase in the contribution workflow outlined previously).

Fostering cross-unit collaborations isn't easy, for a number of reasons. One major issue that often arises in large organizations (and our company is no exception) concerns component ownership. In traditional organizations, business units have clearly defined responsibilities to deliver certain products and services. In order to achieve this, such product-driven units will pursue full control over their software stack. This product-driven nature of the organization leads to business

units that follow their own lead. This, however, presents a major barrier for cross-unit collaborations, because collaborating with other units implies that teams become dependent on others: they will perceive a "loss of control." In reality, InnerSource can empower a team to reuse common software, and make changes themselves as needed.

Second, although organizations may have an overall enterprise architecture, depending on the organizational culture, business units may not perceive such a centralized architecture as "binding." So the relative independence of business units can lead to independent and fully owned software stacks. InnerSource can help overcome this by making the common architecture available as an InnerSource code repository, rather than just a specification to comply with.

When we proposed our InnerSource way of working, it didn't come without some resistance. Engineers who had worked in the domain for a long time would be skeptical, because they interpreted our proposed way of working as requiring them to do the platform team's work in addition to their normal job. Product Owners were also very skeptical of CDS, because they thought that developers would randomly pull stories from everywhere, rather than their own product backlog. These reactions are natural and understandable and have also been reported by other companies that have adopted InnerSource.[6]

The best way we found for a team to make a contribution was to persuade the Product Owner that doing so would be to their own benefit. The key is to sit down with them and explain that putting requested features on their own backlog would be a faster way of getting work done. This change in mindset is a big cultural change, and trying to convince teams to work in this new way didn't always work. There were teams who believed they were working on key critical parts of the software stack and were afraid the CDS program would not meet their product's quality or deadline requirements.

Pillars of Community-Developed Software

Around 2015, in an attempt to reflect on what we achieved with our CDS efforts and to further champion them throughout the organization, we tried to capture the principles of what we were doing. We defined a set of five "pillars" that we believe are key to our CDS program:

Community portal
: The first pillar is the community portal, a one-stop shop with completely open access for all Ericsson engineers worldwide to software, documenta-

6 Klaas-Jan Stol, Muhammad Ali Babar, Paris Avgeriou, and Brian Fitzgerald, "A Comparative Study of Challenges in Integrating Open Source Software and Inner Source Software," *Information and Software Technology* 53, no. 12 (2011): 1319–1336.

tion, and infrastructure. The portal provides a step-by-step guide that helps developers get started. As the portal evolved and matured, it provided information dashboards such as contribution statistics by region and the portion of contributions that came from core teams versus external teams.

Skilled people
: Having the right people in place is key to making InnerSource a success. We're an Agile shop using Scrum, so we already had Product Owners in place for our feature teams. InnerSource provides a way to scale up the Agile way of working to the enterprise level. In our InnerSource implementation, we designed a flexible framework that provided a workflow for feature teams to get their work done while interacting with multiple component core teams. We made sure that each of the key roles in our core teams was performed by highly skilled and committed people, because this helped to establish the credibility of the program and demonstrated our level of commitment to InnerSource.

Best-in-class SDKs
: Our InnerSource implementation isn't only about the openness and transparency of our components within the organization. Simply sharing the source code isn't enough, because developers could still run into problems when they want to extend a feature but don't have a full understanding of the codebase to implement an extension. We observed that for a particular component it would be regularly extended in the same way. In these cases, we created software development kits (SDKs) that made it easier to extend the component without knowledge of the overall codebase.

Responsive Agile
: We are committed to constantly improve our processes so that we can deliver useful functionality to our customers. One key aspect of an Agile approach is to make the project's progress clearly visible. For that, we're using the standard Agile practices and techniques such as burndown charts and sprint demos. Our InnerSource program scales up our agility across application teams. Application teams that want to use specific components can do so—our CDS platform provides complete access to a range of reusable components. The CDS portal also provides quality dashboards, and our development environments depend heavily on Continuous Integration. A community backlog provides full visibility in the plans for the InnerSource components, and we're using bug trackers as a standard way to report any issues with our InnerSource components.

Active community
: The final pillar represents our efforts to build a vibrant developer community, which is perhaps the hardest aspect of adopting InnerSource. While establishing a portal and support channels to help people getting started

takes some time and effort, the hardest part is to convince people and teams to *trust* other teams, and to let go of the "us versus them" and "not invented here" mindsets. There's no checklist to follow in achieving this. Instead, it's important to nurture the community, to listen to feedback, and to respond promptly to concerns. At Ericsson, we use communication mechanisms commonly found in Open Source projects, such as forums and mailing lists. Other practices we adopted from the Open Source world were to organize hackathons and run face-to-face onboarding workshops. These practices helped our teams actively engage with the community.

Success: The User Interface SDK Framework

One of our most successful InnerSource components is our User Interface SDK (UISDK). Every company needs a consistent look and feel in all customer-facing products to ensure brand alignment. Furthermore, most applications need some UI, and creating a UI can require considerable development time, so it makes sense to develop this through a community effort. By making the UISDK an InnerSource project, we were able to better support UX experts, at lower cost.

UISDK is one of the most active CDS projects, used by over 60 teams. In the past year alone, the project received over 360 contributions. We've run over 20 onboarding workshops and 10 hackathons to get teams started with UISDK. The community is very active, with about one million page views since the project started in 2013, and close to 20,000 forum posts.

Lessons Learned

Our CDS program has seen a number of successes that keep inspiring us—as well as our management—to continue our journey. While our journey hasn't been without challenges, we've learned some valuable lessons.

A key lesson that we learned was that our approach to developing platforms without dedicated platform teams can work. Using appropriate architectural patterns, we had well-defined interfaces and SDKs that provided hooks for application teams to develop the features they needed in the platform. So our goal of preventing bottlenecks was achieved. The InnerSource process wasn't suitable for all components, and for those that it didn't fit, we followed more traditional approaches.

We found that when a core team "jelled," we could create a community around the CDS component. The three roles in the core teams were well balanced: the Product Owner helped to prioritize those features that offered maximum business value, the Architect helped to maintain architectural consistency, and the Code Guardian maintained a high level of code quality. But if the personalities

and opinions of these three key roles diverged too much, the resulting conflict reduced the effectiveness of CDS development.

While we achieved cross-organization collaborations, the hope that individuals (rather than teams) would browse the code and polish it, identify defects, and offer improvements to the documentation didn't materialize in our program. Ultimately, all contributions on our InnerSource projects remained in the domain of "work," and this idea that engineers would nurture a "pet project" didn't happen at scale.

In some components, contributions were rejected or didn't pass the code review. When this happens, it's hugely important to have transparency and a clear understanding why. Otherwise, there may be a perception that it is difficult to contribute. The core team must be clear when there is an issue. To aid this, the best contributions are small and frequent. Discipline is key, including always following CI and avoiding shortcuts (reviewing one's own code, etc.).

We found that InnerSource works well for the right deliverable. In particular, we found it very suitable for components that are common to multiple teams and easy to extend. A focus on extensibility in the architecture of the component is critical. If the correct interfaces for collaboration are hard to identify, the component may need to be refactored.

We also found involving the application teams in developing the platform spawned a much clearer focus on building the things that teams actually needed, rather than having "nice-to-have" features. This was a great way to reduce waste in our organization.

InnerSource may not be the answer for all software. For those components that require more control, we simply defined the separate "blue zone" development and contribution model.

We booked a number of successes with cross-team collaborations that we otherwise wouldn't have seen. Although sometimes we've had to convince teams to try to work together, once they did and the result was successful, we made sure to celebrate those successes. The idea here was to let teams and developers experience the collaboration, hoping they wanted to repeat such successes.

Our Agile and Lean transformation has helped us improve development teams' ability to organize their work and deliver software in a timely manner. Our Continuous Integration machinery plays a critical role in this. Agile teams are backlog driven. InnerSource provides another level of collaboration between units by sharing backlogs. This leads to an enterprise-wide agility.

As the CDS portal was gaining wider recognition throughout the company, teams in other countries also started to download and use the CDS components. However, initially those teams were "passive" users rather than active contributors. We

made an effort to "onboard" these teams so that they, too, could start contributing. In order to better track who was using the CDS components, we started recording their usage, simply by requiring users to log in into the portal.

The Future of InnerSource at Ericsson

InnerSource has attracted a considerable amount of momentum within our organization. While our journey hasn't been without any bumps in the road, the idea of learning from Open Source communities has started to fuel a change in our organizational culture. It takes time, patience, and perseverance to introduce new ways of working in any organization—and we are no exception to this rule. Despite some natural resistance that comes along with any type of change, our InnerSource program has sufficient momentum to sustain it. The CDS program has become a mature software ecosystem within our organization.

We are now taking the lessons that we learned and applying them to a new Application Development Platform (ADP), which like the ENM is a greenfield project. As we're doing this, we're "reinventing" what InnerSource means for our company. We envisage this new project becoming a similarly successful ecosystem, but we are now increasingly focused on improving the delivery pipeline. As we mentioned, simply sharing the source code is not enough. You also need a delivery mechanism so that users of a component can simply take an existing component and plug it into their system. We're using contemporary technologies such as Docker containers to simplify the delivery of our components and applications. With the seed for changes in our organizational culture planted, we keep working at building communities within Ericsson.

Acknowledgments

A big thank you to all my colleagues at Ericsson. In particular, thanks to our first Community Developed Software team of Matt Hamilton, Dermot Hughes, Tom Curran, and Pat Mulchrone, who started this journey and contributed to the findings in this chapter.

CHAPTER 8
Adopting InnerSource

We hope that the case studies in this book have given you some inspiration to start an InnerSource experiment at your organization. As should be clear by now, each InnerSource program is unique, and what might work for some companies may not work for yours. Some of the case studies will seem a better fit with your organization than others. But while we recognize there is great variety in how InnerSource is adopted, we can also see great similarities.

Indeed, as we mentioned in Chapter 1, the InnerSource Commons (ISC) has a dedicated working group striving to extract the similarities among approaches and capture them as patterns, much like the software engineering community has done for years with design patterns. The ISC's patterns community has regular meetings to discuss new patterns, which they document in public.

Although we don't present any patterns in this final chapter, we do offer you some guidance based on the lessons learned in the case studies. We'll present a comparative analysis of the five case studies in order to extract some commonalities and differences.

This chapter will also attempt to give you some practical advice about how to bootstrap your first InnerSource experiment. There are many resources available to support you at *http://InnerSourceCommons.org* as well, and using the pointers given in this chapter you should be able to get started.

As we mentioned before, InnerSource isn't a defined method like, for example, Scrum, which has a number of standard roles (Product Owner, Scrum Master), events or "ceremonies" (like the daily stand-up), and artifacts (such as a sprint backlog). While Agile consultants will admit that any company adopting Scrum must tailor it to their context, these roles and events are foundational to the method—without a Scrum Master and a sprint backlog, you can't really do Scrum.

InnerSource is different. InnerSource is a strategy, or for the more philosophically inclined (which we are), a *paradigm*. It's a different way of thinking about how to do large-scale software development. This, however, means that there's no one-size-fits-all approach to adopt InnerSource; as the nuanced case studies in this book clearly demonstrate, InnerSource is as varied as corporate culture. That's why we can't give you a fixed recipe that gives you a successful InnerSource program. There is no set of recommendations to follow, nor is adopting InnerSource a problem that can be solved by simply throwing a lot of funding its way (although the latter helps if it's targeted at the right things). So, where does that leave us?

What we can do is offer a set of guidelines based on long-range analyses of multiple projects. We can back up our recommendations with comparisons and justifications. This doesn't mean that our set of guidelines is complete—nor does it mean they will always work. Instead, we emphasize that adopting InnerSource represents creating an internal community, which consists of different types of actors, both at the individual and aggregate level (e.g., teams and departments). Each of these has motivations, resources, and constraints. Balancing these wisely will lead to success.

Comparison of the Case Studies

We're organizing our analysis using a framework that one of us developed before as part of his research on InnerSource.[1] The framework defines nine "key factors" for adopting InnerSource, which are organized under three themes: product, processes and tools, and community and management. We're not suggesting you limit your thinking to these nine points, but rather use them as "intellectual bins" to arrange your thoughts, develop tactics, and focus your efforts.

Product

The first theme is the product, by which we mean the actual software to be InnerSourced. Three factors are important to consider here: seed product, stakeholders, and modularity.

Seed product

In order to start building a community, you need to have an initial product or early version of that product—a seed product. Without it, contributors have nothing to play with, nothing to run, and nothing to contribute to. You can't design software with a community from scratch; there must be one person, or

1 Klaas-Jan Stol, Paris Avgeriou, Muhammad Ali Babar, Yan Lucas, and Brian Fitzgerald, "Key Factors for Adopting Inner Source," *ACM Transactions on Software Engineering and Methodology* 23, no. 2 (2014).

perhaps a few people, who have a vision and who can create an early implementation or prototype of that vision. As Eric Raymond[2] pointed out: "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."

Despite the variety of communities that we've seen in the case studies, most of the communities were built around some existing software. At Bell Labs, the initial version of the SIP server was written by a single person before it was InnerSourced. At Ericsson, creating the Community Developed Software portal wasn't enough—it was populated with a number of components that had been selected because they had a high potential to benefit from contributions by feature teams. The setup at Bosch (see Chapter 4) is an exception here, because initially, communities had to be proposed without an actual product. However, even in those cases, before any contributors could be attracted, the community leader and perhaps a few core developers would have to create the initial prototype before it could attract any contributors.

Most of the cases in this book started out as experiments. PayPal ran a series of experiments, each of which led to new lessons learned and increased confidence in the value of InnerSource. At Bosch, the whole BIOS initiative (represented by a group of associates) started as a three-year, time-limited experiment with an explicit goal of evaluating the suitability of Open Source development practices. Europace also started with running some experimental InnerSource projects. The teams learned some valuable lessons that were subsequently documented and shared on an internal website, and which was later publicly shared as an pattern to the InnerSource Commons patterns community.

Stakeholders

InnerSource aims to facilitate and improve internal collaborations within the organization. However, collaboration on software assets makes sense only in the presence of multiple stakeholders. If only one team needs a certain, perhaps highly specialized component, there's no point in pursuing InnerSource, because contributions by other teams simply won't be made.

A good example of this requirement was the shared need for a common SIP server at Bell Labs. This case study demonstrates that a new, emerging standard that is needed by many different business units offers good opportunities to pool resources. The project benefited greatly from different types of expertise (cryptography, parsing technology) in the company. Within Europace, the different teams serve different types of customers, but they all do so through a common

2 Eric S. Raymond, *The Cathedral and the Bazaar* (Sebastopol: O'Reilly Media, 1999).

platform. We also saw this at Ericsson: the new architecture provides a set of common platforms, each of which could be used by a range of feature teams. As different feature teams required different features, the common components grew as needed, while the core teams maintained architectural integrity.

Modularity

Modularity refers to the degree to which code is organized into independent modules. It is a desirable system attribute that has been studied for decades.[3] The higher the degree of coupling between modules, the more modules a developer may have to touch when making changes to the system. A high degree of modularity—or loose coupling—is generally preferred because it facilitates parallel work by teams on different modules in the same code base, without getting in each other's way. It also promotes software reuse and simplifies the integration of systems. As more teams get involved in collaboration on a codebase, modularity is becoming an increasingly important attribute of systems—and this is especially true for InnerSource. Tim O'Reilly has referred to this idea as the "Architecture of Participation."[4] A common implementation strategy is to replace or transform legacy monolithic systems to a microservices architecture.

We've seen modularity as a recurring theme in the case studies, though it loomed larger in some cases than in others. At Bell Labs, the original implementation was refactored to make the SIP assets more reusable. At Europace, the single, large team was split up into four smaller teams and the company made efforts to make the software more modular, although this wasn't without its challenges. At PayPal, management had previously mandated that all code be made more modular; few teams complied until InnerSource. The involvement of guest contributors actually helped in this effort, as their pre-contribution questions about how the systems were structured forced hosts to reflect and reconstruct some of the design decisions, which in turn helped them to rearchitect the systems. The teams at Ericsson, having the luxury of starting afresh in their greenfield project (which isn't that common in the software industry) made a strong effort to establish well-defined interfaces and maintain a high level of modularity in the architecture.

Table 8-1 summarizes how the companies in this book reflected key requirements described in this section for the product.

---

3 David L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM* 15, no. 12, (1972) 1053–1058.

4 Tim O'Reilly, "The Open Source Paradigm Shift," in *Perspectives on Free and Open Source Software*, eds. Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani (Cambridge: MIT Press, 2005).

Table 8-1. Comparison of InnerSource projects across the case studies

Case	Seed product	Stakeholders	Modularity
Bell Labs	SIP server, an emerging technology that became a key asset to many parts of the company.	Variety of business units that were in need of a SIP stack ("the lifeblood of various products"). Experts from across the company contributed variety of improvements, e.g., in the parser and cryptography.	Over time the SIP server was refactored to improve reusability. The SIP parser was spun out as a separate module. The standalone SIP server was refactored into a library (framework).
Robert Bosch	Range of innovative prototypes, some of which were productized. Set up as an experimental initiative in which anyone could propose a project.	Any business unit was free to use any BIOS asset as long as its use complied with the BIOS license.	Architectures of different projects varied. Most projects were innovation prototypes, with less attention to optimizing the product's modularity given the relatively small size of communities. Many projects created embedded software for specialized hardware.
PayPal	Series of experiments by different teams, including CheckOut, Onboarding, and Symphony.	There was considerable demand for increased collaboration and transparency, coming from recent hires with more experience using Open Source methods. At the same time, C-level management wanted to overhaul duplicative practices.	The CTO office had requested that all code be refactored to increase modularity. As guest contributors started asking questions about the structure of monolith systems, key developers' explanations of the design decisions helped them to reflect on possible modules.
Europace	EUROPACE platform, which connects the company's partners.	Units serve different types of customers, all through the same platform.	The original development team was split up into four smaller units with a high level of autonomy. Efforts are underway to divide these further into smaller subsystems.
Ericsson	Central portal populated with selected components within a larger architecture; the components make up the "horizontal" layers, which are used by application teams that develop application "verticals."	Any application team that requires services from the components positioned across the different layers. Applications are "verticals" that cut across the horizontal layers of the system.	The core components were designed to have well-defined interfaces that were essential for the application teams to implement their functionality.

Processes and Tools

The second theme refers to the processes and tools. Three factors are important: defining a development process that fits with organizational standards and facilitates InnerSource collaborations, quality assurance, and tools that help teams adhere to those processes and maintain quality.

Development process

Many organizations have established ways of developing software. Whether they use a classic waterfall or more modern Agile methods such as Scrum, InnerSource requires that these processes be augmented to facilitate cross-unit collaborations. What's important here is a clear and simple collaboration process, because a complex process will act as a barrier for people to make contributions —or even to use an InnerSource asset.

The case studies demonstrate a variety of ways to adopt new practices and processes. At Bell Labs, an increasing stream of defect reports and patches led the chief architect to expand the core team with trusted lieutenants, very similar to what happens in large Open Source projects. PayPal and Europace also reported the use of Trusted Committers. Bosch's BIOS program emphasized self-determinism, so that different communities could define their own processes.

Quality assurance (QA)

Closely related to the development process, QA practices are extremely important, and Open Source communities have contributed a number of process innovations to improve QA. Two of these innovations are peer review and time-based releases. Traditional approaches to QA are formal code reviews and walkthroughs, which are quite heavyweight approaches. Peer review, on the other hand, is shown to be a very effective and lightweight method to identify defects because it doesn't require any planning, nor does it require attendance of a team in the same room. Time-based releases improve on the traditional feature-based release approach, because a time-based release schedule offers a regular heartbeat to projects.[5] Software releases can be very big events, especially if they don't happen often. The principle underpinning time-based releases is to do them regularly so they become less of a big deal.

To support QA, PayPal introduced an InnerSource process innovation called the 30-day warranty. Under this process, a guest contributor is responsible for fixing any important bugs reported during the 30 days after the contribution is first deployed to customers. This should reassure the host team that they're not accepting just any code, but code that the guest contributor is confident enough to support. It also addresses a common fear among host teams that they will be held responsible for defects in other people's code, which can hamper their willingness to accept contributions in the first place.

5 Martin Michlmayr, Brian Fitzgerald, and Klaas-Jan Stol, "Why and How Should Open Source Projects Adopt Time-Based Releases?" *IEEE Software* 32, no. 2 (2015): 55–63.

Tools

Having the right tools in place makes or breaks any collaboration effort. While this may seem like a simple technical issue, it can be remarkably complex to get new tools installed at a company. Ensuring that you have compatible tools so that teams from different business units can collaborate is essential, but not sufficient. Certainly, good tool support can greatly help to improve teams' workflow. Many companies today are adopting GitHub or GitLab, though we've also seen companies who use Microsoft Team Foundations or BitBucket. On the other hand, simply installing one of these popular code collaboration platforms isn't enough.

At Bell Labs, the SIP assets were downloaded by different teams, and sometimes teams stored a copy in their own version control system. Any changes made had to be ported back to the master repository, which could prove difficult simply because the version control systems weren't compatible. Bosch's BIOS project decided to let teams use whatever tools they wanted. This was also the case at Ericsson, where a central portal was created behind which teams could use whatever tools they had already—as long as those tools had a programmable interface (such as REST). Several other case studies reported the adoption of GitHub Enterprise (Europace, PayPal). Table 8-2 summarizes how the companies in this book reflected key requirements described in this section for process and tools.

Table 8-2. Comparison of practices and tools across the case studies

Case	Development process	Quality assurance	Tools
Bell Labs	Initial implementation was done in "Cathedral mode," taking feature requests from teams and contributions from experts. The architect worked with trusted lieutenants on nontrivial features. Teams that needed features would "lend" a developer to the CSS group. Over time the size of the community grew to around 30.	The only precondition for sharing code was to share back any bug fixes. Code had been taken to "bakeoffs" with other companies. Trusted lieutenants helped the benevolent dictator review contributions. Specialized roles emerged to maintain quality and help others to integrate the software assets. Feature advocates performed code inspections and reviewed design documentation.	SIP assets were implemented in C and were portable across two platforms initially, with further support for other platforms contributed by others. One of the CSS group's responsibilities was to maintain the canonical source repository. Some teams stored their changes to their own copy of the source code in a different version control system, which complicated the "buy-back" process that was supposed to integrate those contributions back into the main branch.
Robert Bosch	Communities emphasized self-determinism, free to use any methods, practices, and tools.	Focus of efforts was on developing innovative prototypes, some of which led to products in the market. For prototypes, traditional QA processes did not loom large.	BIOS initially allowed full freedom of tools, but with Social Coding all collaborators in the company moved to a common collaboration platform that is free of charge.

Case	Development process	Quality assurance	Tools
PayPal	Mentoring happens through written communication, which leads to passive accretion of documentation. Contributors are encouraged to notify of planned contributions, allowing TCs to "pre-vet" them as well as plan time for reviewing those.	Trusted Committers mentor contributors to write code that can be more easily accepted. About 10% of engineers are Trusted Committers, whose responsibility is to review code contributions from other teams. All contributions come with a 30-day warranty provided by the contributing team. If any issues are identified during this period, the contributing team is expected to provide fixes.	A large-scale retooling effort adopted GitHub Enterprise prior to PayPal's InnerSource experiments.
Europace	Units develop features for different types of customers. Teams develop specific features, after which members can join other teams. Practices include pair programming.	Trusted Committership as a reward to developers made them take that role seriously. More cross-unit collaboration led to better quality code reviews and better quality code.	The company previously used SVN, but moved to GitHub, which provided better integration with automation tools.
Ericsson	Feature teams develop functionality that fits within the architecture of components that core teams have set out.	Review process enforced by a Code Guardian; strong dependence on continuous integration infrastructure.	CDS portal as a frontend that enabled any unit to use their own tools as long as they have a programmable interface (e.g., REST).

Community and Management

Ultimately, it's people who develop software and manage the process. This theme addresses aspects related to the human factors: coordination and leadership, transparency, and management support and motivation.

Coordination and leadership

Although each case study emerged from a unique context, all demonstrate the utility of dedicated roles and teams to help organize development efforts. InnerSource doesn't mean copying an "onion" governance structure that is often associated with Open Source projects (that is, concentric circles of contribution that grow in responsibility as one moves inward), but companies can find hybrid ways to balance a company's business interests and the flexibility that a meritocracy offers. Several of the cases have core teams that are "in charge" of InnerSource projects, and which consist of specialized roles. Both R&D settings (e.g., Bell Labs) and "normal" commercial development (e.g., PayPal) made successful use of a facilitating core team. In PayPal, leadership remains with the traditional corporate roles, but this is augmented with the Trusted Committer role, who serves as a mentor to contributors outside the team.

Transparency

Perhaps one of the most common patterns we can detect in the case studies is their implementation of transparency. All case studies recognized the importance of opening up the environment and making it as easy as possible for anyone to find resources and make contributions. Many of the case studies established a central portal that provides access to InnerSource development artifacts and communication channels. The interfaces differ. Modern platforms such as GitHub and GitLab already provide user-friendly interfaces. But at Ericsson, the portal was simply a frontend for existing tools that were developed in-house. Bell Laboratories' SIP project had a dedicated "Center of Excellence," an intranet website that provided a one-stop shop for anyone interested.

Opening up the artifacts of a project (source code, documentation) is one thing, but transparency also means opening up communication and the decision-making processes. Some companies, such as Europace, have adopted Slack to facilitate such asynchronous and archived communication so that others can consult what was discussed at a later point in time.

Management support and motivation

All case studies reported support from management, though the timing, level, and extent of commitment varied. Some companies (PayPal, Europace) have hired Open Source advocates as change agents. In other cases, management recognized the importance of investing in Open Source technology (Bosch), and others were simply open-minded and supportive of Open Source (Bell Labs), or recognized the benefits of the Open Source paradigm (PayPal, Ericsson). Some of the cases report true grassroots initiatives; for example, Bell Labs' SIP stack was a clear case of "bottom up" InnerSource, which later on became more formalized. Others were initiated from a corporate management level; for example, management at Europace had been investigating alternative ways to make teams more independent.

The case studies report different levels of support, which evolved over time for some. For example, the BIOS program at Bosch received central corporate funding (as opposed to sponsorship from the business units) to create the BIOSphere environment and fund full-time community leaders.

Management support is essential to give credibility to any software process improvement initiative, including InnerSource. Because most large corporations are divided into a set of business units or other types of subdivisions with their own profit and loss sheets, these divisions tend to be managed as silos—the rationale being that, if each of the individual divisions operates optimally, then the company as a whole does so too. However, there is an essential difference between executive management (who see the big picture) and mid-level managers (who are evaluated on how well their division performs).

Although management support is essential, it's not enough to simply declare that the company now adopts an InnerSource strategy. You need motivated people to step up, and they need to be provided the means to do so. One of the BIOS principles at Bosch emphasized the voluntary nature of involvement. As a result, the BIOS program attracted highly motivated associates.

Table 8-3 summarizes how the companies in this book reflected key requirements described in this section for community and management.

Table 8-3. Comparison of community and management factors across the case studies

Case	Coordination and leadership	Transparency	Management support and developer motivation
Bell Labs	Core team of three: architect (and initiator) who served as Benevolent Dictator; product managers; and project manager supported by trusted lieutenants. Additional roles emerged: project liaison and delivery, feature, and release advocates. As more contributions came in, a Common SIP Stack (CSS) group was formed. Product manager/liaison worked with business units	Center of Excellence (COE) as one-stop shop that provided source code, documentation, instructions, and a release schedule.	Management was generally supportive of Open Source and sharing source code with other business units. Chief architect convinced executive management of the many benefits of a homegrown implementation of the SIP stack. The project borrowed experts from across the company, which is an in-kind contribution of business units. Sometimes BUs made monetary contributions.
Robert Bosch	In BIOS any associate could propose a community, which would be evaluated by the Review Committee. Each community had a full-time community leader. Strong emphasis on meritocracy. BIOS Governance Office performed administrative support tasks that shielded developers from this task.	BIOS as a "radically transparent" community, sharing all work products and communication to anyone within the company. Creation of new communities was initially guarded by a Review Committee (in BIOS), but with Social Coding this became a completely open platform.	Stewardship by Corporate Research. Corporate funding to support program, primarily for funding community leaders so they were not dependent of business units, but also to buy out contributors for 10% to 50% of their normal job. Highly motivated and self-selected developers. BIOS's Review Committee comprised of Vice Presidents of the engineering business units; this provided necessary air cover.
PayPal	10% of developers as Trusted Committers, which is a role assigned to developers on a revolving basis. Guest contributors can earn Trusted Committership.	Company-wide access through GitHub Enterprise for all source code; much communication is written down, archived, and searchable.	CTO and several other executives hired Open Source advocate Danese Cooper to help the company engage with Open Source and InnerSource. This action formed automatic air cover for the project. Intrinsic motivators, such as scratching your own itch, were augmented by extrinsic motivators designed to further appeal to enlightened self-interest, such as taking demonstrated InnerSource mastery into account when advancing careers.

Case	Coordination and leadership	Transparency	Management support and developer motivation
Europace	Leaders sought new organizational forms to support self-organizing teams with a high degree of autonomy. Flexible teams form and disband as features are needed and completed, respectively.	All repositories are open. Use of Slack for communication. Explicit emphasis on written (over verbal) communication. Task transparency was a key goal to facilitate remote workers.	Strong support from company leadership, and units supported by a central People & Organization team that strongly emphasizes growth and development of staff. InnerSource fit well within this goal.
Ericsson	Governance Council to oversee the Community Developed Software initiative. Core teams consisted of a Product Owner, Architect, and Code Guardian, emphasizing facilitation of feature implementation by application teams. Development activity was pre-negotiated between feature and core teams.	Community Developed Software as a one-stop shop portal, accessible to anyone within the company.	Business units have high degree of autonomy. CDS is supported by the product line that developed the new platform. Budget was redistributed to give feature teams the means to develop the functionality they need in the components.

Guidelines for Adopting InnerSource

Many companies have decided to start adopting InnerSource—and commonly they start out with a pilot project, or an experiment. We think that introducing InnerSource with one or more experiments with some limited scope is a good idea because it allows you to learn lessons and adapt accordingly. Once some initial success is achieved and InnerSource is gaining some momentum in terms of enthusiasm of people involved, we often see that others within organizations show an eagerness to also get involved.

So, in this section we offer a set of guidelines to get started with an experiment in your organization. We've based these guidelines on our extensive experience in studying InnerSource as a phenomenon, our collaborations with companies that have started InnerSource initiatives (some more successful than others), and the five case studies that we presented in this book.

Product

The first thing we suggest you do to start an experiment is to select a *seed project*. It's important to start an experiment with the right seed project that has some potential for cross-team collaboration, because a project that does not have such potential may have trouble gaining any momentum in building a community. A good seed project is one that does the following:

Removes a bottleneck
: A suitable seed project is one that can solve a known bottleneck or other potentially costly resource-related problem. Bottlenecks tend to form when one team is dependent upon another; rather than waiting for the bottleneck team to do all the work, the InnerSource paradigm encourages dependent teams to "do it themselves."

Has development potential
: A suitable seed project is one that has development opportunities, ranging from low-hanging fruit to adding significant features. What's very important is that a project isn't "finished" or feature-complete or worse, abandoned, because other people may not see any opportunities to contribute. What won't work is picking a software asset that is in maintenance mode or orphaned, and hoping that an army of developers will start to improve the software. In order to attract developers, there needs to be potential for developers to make their mark, something by which they can identify with the project—rather than simply cleaning up someone else's old crufty code.

Has different stakeholders
: It is ideal to select a first experiment with a diverse set of stakeholders in order to maximize your learning opportunities. This point may not be obvious to newbies, but your eventual InnerSource strategy needs to take into account *all* potential stakeholders, lest the ones you omit end up undermining your program. InnerSource works best when all stakeholders are aligned, which means they have to see tangible value in supporting the initiative. Typical stakeholders include, but are not limited to, host engineers, Guest contributors from other teams, Trusted Committers, Product Owners or managers, quality assurance engineers, DevOps or deployment engineers, software architects, middle and senior managers, human resource managers, and technical writers.

Has sufficiently modular structure, or seeks to become more modular
: Peer-based collaboration models such as Open Source have proven that smaller, more loosely coupled modules that are designed to work together make the overall function and performance of a collection of code more coherent and maintainable throughout the necessary changes that future features might require. InnerSource collaboration has proven very helpful in guiding host engineers to see how a monolithic codebase could be modularized to best advantage. So, when designing an experiment, previous lack of success in introducing modularity can indicate where you would expect InnerSource to create tangible value.

You may want to consider a range of experiments, or to set up an "infrastructure-centric" InnerSource program (see Chapter 1), where anyone within your company can upload their software assets. In order to prevent this from becoming a

graveyard of dead projects, we suggest curating these assets based on the attributes just discussed. Alternatively, if you wish to allow the unrestricted creation of projects, you could also divide the repository into two parts: an actively curated part that fosters active communities, and a completely open section that allows anyone to start a community. Hybrid models are possible, of course; for example, you could set a condition (as was done in Bosch's BIOS program) that all participating communities appoint a community leader and a Trusted Committer (who could be the same person), and that they actively aim to grow a community and make sure they maintain a "major branch" that always works.

Process and Tools

Once you've designed one or more experiments that have a strong potential to grow as an InnerSource project, it's important to document how people can contribute, to provide tooling to support collaboration, and to ensure the delivery of consistently high-quality software. It's also important to realize that there is no one-size-fits-all development process. Instead, whatever process you define needs to fit the context of your organization. We've stated several times that InnerSource borrows heavily from The Apache Way (see Chapter 2), but this doesn't mean that every organization needs to adopt those exact collaboration methods dogmatically. Instead, we seek inspiration in The Apache Way. As you design a workflow to make InnerSource work for you, we suggest a process that features the following attributes:

Simplicity
: Simplicity is better because you can explain it clearly and quickly. Simple doesn't mean trivial or weak. Simplicity is important because jumping through a lot of hoops represents a barrier to entry.

Facilitate synchronized schedules
: Make sure your process considers all stakeholders' roles and timing requirements. Pre-vetting planned work and pre-negotiating expected outcomes will allow teams to fit InnerSource activities into existing scheduling. Remember that you're still in a business setting, so an open-ended "it's done when it's done" ethos, although typical in Open Source, will likely meet with pushback inside a company.

Put quality assurance in place
: Introduce peer code review (at a minimum through Trusted Committers, or as a formal practice between pairs of team members) because one truism that Open Source clearly proved is that "many eyes make all bugs shallow" (though, as we pointed out in Chapter 1, it's not a guarantee that your software doesn't contain any critical bugs). Institute regular time-based releases such as sprints to organize work into discrete short timeframes, then evaluate quality before each release. Make sure there is a code branch that always

works, to counteract backlash. Something like PayPal's 30-day warranty might be useful to provide a minimum quality threshold to ensure that teams are serious before launching InnerSource projects.

Deploy compatible code management tools
: It is very important to manage code with tools that allow transparent access to all parties, because incompatibility (and workarounds such as cutting and pasting across company tools) inevitably hampers collaboration. Adopting new tools in a company today can be a major issue, but many InnerSource projects depend on the previous implementation of a transparently distributed code management tool like GitHub, GitLab, or Bitbucket. Instituting new tools can also help drive other code improvement efforts such as Continuous Integration (CI) and Test-Driven Development (TDD) and can give teams confidence in the quality of resulting code.

Community and Management

So, now that you've done all the setup, how do you get this InnerSource machine working? This third theme, community and management, suggests that you start by focusing on the people.

This is also where most of the challenges lay in our case studies. Many companies that we've worked with over the past years have engineering cultures featuring clearly delineated silos. Silos must compete for resources, and typically that competition breeds a general lack of trust, an "us versus them" mentality, and even a sort of xenophobia toward anyone not within the silo. When we explain the concept of InnerSource, teams and business units tend to agree that it's the right way forward—except for their particular team or business unit, because they self-identify as "different." The range of excuses is endless, but common ones include:

- "Our team uses a different language, and we're afraid it's too difficult for other developers to pick it up."
- "The code that our team produces is far better than that of other teams."
- "Our software is more business-critical, and so we couldn't possibly share."
- "Our implementation of common functionality is far superior to that of other teams against whom we must compete."

Here are some factors to consider when addressing community and management issues in your InnerSource experiment.

Coordination and leadership

The first factor refers to getting people on board, and establishing how projects are coordinated and led. Open Source communities often use the term *meritocracy*, and to some extent this can be adopted within InnerSource, as well.

Identify champions

You will likely be more successful if you build an internal community within your company, a collection of enthusiasts who are willing to pursue the effort. A best practice for identifying "champions" is to look for people who have shown an interest in Open Source or are effective influencers with a good combination of technical and social skills—both of which are important in order to be able to build bridges across different teams within the organization.

Establish new roles

It pays to get in front of charter conflict (where team members frustrate each other by duplicating effort in a misguided attempt to establish ownership) by taking the time to clearly define and apportion out leadership roles. Consider the following guidelines for Trusted Committers:

- Make them discoverable, by listing their contact information.
- Make them responsive, by establishing agreements about turnaround time for their work.
- Sequester their time to allow them to focus solely on code review and mentorship during one or more sprints.
- Set out guidelines or even a Code of Conduct for mentorship to make sure contributors feel supported, not shamed, by the review process.

Consider seeking support for a moratorium on most executive escalations during your InnerSource experiment. Executive escalation (as described in the Cheese Story) is generally a signal that the normal workflow is failing to keep up with the pace of the business. Since InnerSource is a reengineering of a process, it needs to be sheltered from the bad habit of executive escalation, at least until the new process has become cemented in practice. Ideally, InnerSource should erase the need for executive escalation by making self-directed engineering more normal, more efficient, and better aligned with planning.

Lastly, the role of Trusted Committer is particularly important to get right, but you may want to define the roles of champion, contributor, and product manager or owner within the InnerSource context, as well.

Prevent brand dilution

Organizational change is generally challenging, and InnerSource is not immune. We've seen on several occasions that once an InnerSource program starts to catch on, or once the term InnerSource becomes a buzzword within a company, suddenly *everybody* seems to be "doing InnerSource"… except that they aren't actually doing it. For example, they implement the principle of transparency (discussed momentarily), but don't accept any contributions from outside the team. Instead, these people tend to implement new ideas

based on their own assumptions or impressions without seeking understanding, and then seek to deflect blame for bad experiences by blaming the buzzword. It can be important to counter this tendency before the company "antibodies" manage to expel InnerSource without ever actually practicing it.

Transparency

The second factor is transparency, which is critical to leveling the internal playing field so that every engineer can learn about and contribute to a host's code or documentation. Open Source has taught us that transparency feeds the community as well, because people invest more of themselves if they have unfettered access. It is true that some developers initially prefer not to see their code potentially subjected to wide scrutiny, but these few are often won over when real code review catches human errors before code merge, saving both money and time downstream. Likewise, some companies still harbor concerns about bad actors, so we suggest some guidelines for increasing safety along with transparency:

Open up everything
: Transparency is often interpreted as "opening up the source code," but actually, for InnerSource to flourish optimally, everything needs to be open by default. This means that you need communication channels such as "chat rooms" (IRC is the classic and freely available solution, but commercial offerings such as Slack are popular today), archived emailing lists (in Open Source the default tool is ezmlm), and possibly also wikis. Communication channels must be archived, so that discussions can be discovered later on by new contributors seeking to solve a similar problem. In addition, it's also important to document how decisions are made: rather than making decisions "behind closed doors" (or at the water cooler), the rule is "take it to the mailing list," so that the community at large can engage in decision-making processes. Finally, almost all cases of InnerSource that we've seen provide a central portal that provides information and pointers to the various resources (including code, documentation, and communication channels). Portals are essential for making the InnerSource program accessible throughout a company.

  Note that many companies persist in the belief that there is danger in allowing all employees to view all code, whether because of trade secrets expressed in the code, or other security concerns such as a fear that employees will fail to realize potential harm from sharing production code snippets as "real-world examples" on public blogs or even illegally contributing them to Open Source projects. In reality these types of problems are very rare. More common is injection of malicious code and other sabotage by disgruntled employees, but the increased likelihood that such sabotage will be detected earlier in code review or by another employee studying the altered code more than compensates for the perceived risk.

Establish house rules

Simply designing an InnerSource project will not cause a vibrant community to spring to life automatically. Some people will misunderstand or abuse your initiative, interpreting InnerSource as an opportunity to assign tasks to other teams. Or they may "dump" their code contributions into your framework without paying much attention to providing test cases, documentation, and coding standards. So you need to establish some house rules that capture the "etiquette" that will reduce friction from participation in InnerSource. A common practice for articulating house rules is via a *contributing.md* file, negotiated between hosts and guests and maintained in the common repository. Furthermore, while any team may be able to download a copy of the source, just like Open Source, it may be important to clarify the conditions under which they can use that source code by providing an InnerSource license.

Market your program

We have learned that marketing matters to the ultimate success of InnerSource within a company, because the way a program is presented affects how people perceive it. For example, the names Bosch Internal Open Source (BIOS), and subsequently the term BIOSphere to refer to an internally protected bubble, were well-chosen. They established the initiative as a brand, with a logo, and an overall philosophy that gave participants a pleasing sense of identity. Creating some "swag" like T-shirts and stickers may also help people to self-identify with and spread the message. So picking an appropriate name that fits with the company, something that clearly delineates and defines what it is you're trying to do, may help people to understand your efforts. One common recipe for failure is not planning for scaling the *marketing* of your InnerSource program once your initial experiments are judged successful. It is a best practice to recruit team members with marketing experience to help design and implement message amplification.

Management support and motivation

The last factor is management support and motivation. As we will posit here, simply establishing real management support can be challenging, but it's not sufficient if there is no interest on the ground from engineers who might want to participate. You want to make sure you find both "air cover" and ground support if your end goal is company-wide InnerSource. Here are some tactics you can adopt to mitigate risk until you can assemble enough support to go wide:

Executive management commitment

It's important to eventually get commitment from at least a couple of executives, because they can provide "air cover." In hierarchical organizations, messaging and visible participation from a senior leader can drive awareness and adoption, and their direct advocacy can preserve funding through suc-

cessive budget cycles. However, gaining attention from executive managers means you will need to speak their language. Reporting progress in terms of value created can be effective, since showing accumulated progress will help them justify their commitment over the long haul. Pitching and securing funding for discrete parts of a large roadmap on a "fixed-time, fixed-funding" basis may help them compartmentalize risk. While several of the case studies in this book present very successful grassroots programs with only a minimal level of executive sponsorship, in our experience those programs can't scale past a certain point without eventually securing executive sponsorship.

Staying under the radar
: Occasionally your well-designed InnerSource experiment may stall because key executives or middle managers just don't get it. Or perhaps your company still harbors unjustified misinterpretations of what Open Source is (metaphorical comparisons such as "Cathedral versus Bazaar" don't help). Or, we've also seen scenarios where your senior executive champion changes role, and can no longer provide you with air cover. This doesn't mean you have to give up. We suggest you keep going, but learn to "stay under the radar" for a while until you have more promising results. Sometimes it's a matter of using different language and terminology. We've seen successful tactics involving renaming a project "technology transfer"; or leveraging more acceptable terms that exploit known strategies, such as "enterprise agility." It may be possible to start your InnerSource efforts with one module of a generally usable component such as deployment infrastructure or a library that would eventually be useful to every business unit, and then gradually widen the scope within that component to grow your program. Although these tactics are certainly not foolproof or a guarantee for success, many successful InnerSource programs started out as grassroots initiatives and grew after proving value creation.

Strategic humility
: It's always a good idea to underpromise and overdeliver. Asking for a "small budget" while promising major benefits sets up a potentially fatal backlash. Some InnerSource programs have succeeded in evading attack from corporate culture antibodies by tactically underplaying the value and goals of the program until localized proof of value is established. Let happy participants evangelize through word of mouth at the beginning of your experiment. Eventually, you should implement a more extensive internal marketing effort but wait to unleash it until you've collected some local success.

Address Fear, Uncertainty, and Doubt
: Fear, Uncertainty, and Doubt (FUD) is a classic marketing tactic meant to discredit something by presenting it in an undeservedly bad light. Spreading FUD can hinder the adoption of any product. When Open Source wasn't yet

generally accepted by the mainstream software industry, it suffered a lot from organized FUD. We see a similar thing happening with InnerSource, where skeptics at all levels of organizations sow discontent without direct experience, out of a general aversion to change. For example, it's not uncommon for business units to refuse to contribute to shared repositories because they fear that other business units will "steal" their code and effectively their business. This highlights an extreme distrust between teams within the same corporation, which in turn warns of deeper dysfunction: an inability to mobilize employees to work together across the company to achieve any sort of common goal or major change, including responding appropriately to external market forces.

Feed participant motivations
: At the end of the day, InnerSource is about shifting power away from managerial command and control toward individual autonomy to give resources a method and permission to work around existing bottlenecks. Relearning how to work is such a fundamental shift that participants will probably experience some discomfort during the transition. Typical expressions of that discomfort manifest as fears about host team sovereignty, the respect awarded to guest contributors by the host team responsible for evaluating their work, impact on people's schedules, service-level promises and time commitments, change and rotation of duties, rejection of unsuitable contributions, and maintenance expectations for merged contributions. It's important to listen to, acknowledge, and then try to mitigate participant discomforts and fears. Qualitative data (success stories) collected from initial experiments can be really helpful.

The InnerSource Commons

We believe the guidelines and tips we set out in this chapter are useful to start up an InnerSource experiment. But we'd like to add a disclaimer that we're not making any promises! As we said before, this isn't just a simple recipe that will lead to success. As the various case studies have reported in very frank ways, there are considerable challenges in making the cultural changes that are needed to improve collaboration within any organization. We believe the benefits more than outweigh the challenges presented by InnerSource, which is why we're advocating for it.

The very best practice we can recommend is joining the InnerSource Commons at *www.innersourcecommons.org* to connect with the network of your InnerSource peers across the tech industry. Together, the Members of the ISC are learning about and documenting patterns and practices associated with InnerSource. All the stories we've told in this book and more have been discussed

within the ISC. It really is the best way to prepare yourself for success with InnerSource, and we hope to see you there.

CHAPTER 9
Epilogue

Brian Behlendorf

InnerSource is an idea that has been long in the main, and it is really gratifying to see it finally gaining widespread momentum.

In 1998, the same year the term "Open Source" was coined, I had just sold my interest in the first company I co-founded, Organic Online. It was the same year that I and the group of volunteer developers known informally as the Apache Group, who built the Apache HTTP Server, began the work of forming a formal nonprofit corporation around our efforts, which led to founding the Apache Software Foundation (ASF) in 1999. But I also needed a new full-time gig, and I believed strongly that the way ASF engineers were using the internet to work together on what became the world's most popular web server—this "Apache Way" of working (see Chapter 2)—was more efficient, agile, and powerful than any other software engineering process. I felt it was possible to make lightning not just strike twice, but over and over again.

So, I joined up with Tim O'Reilly (of O'Reilly and Associates), a venture capital firm named Benchmark, and a number of other collaborators and friends and started a company named CollabNet, to bring Open Source practices and principles to the rest of the software world. While I was perhaps more motivated to help companies publicly release their code, it became clear that enabling private software collaboration within corporate walls was just as valuable, if not more so. Tim coined the term "InnerSource" for this. InnerSource differs from Open Source *only* in that it has no public code sharing component. It happens privately within the confines of a company's firewall, or between companies in a shared but exclusive space.

There is a significant benefit to fostering cross-team collaboration; to fostering reuse and continuous improvements; to allowing teams to see "below the API" when calling into another team's interfaces or code; to allowing one team to

"fork" the code of another and then ask their modifications to be pulled back upstream after some work is done (but not require it); to recognize when starting work on a component that there likely are other teams with a need for that same component, triggering an effort to find them and work together; and to many other patterns that were common in Open Source development even in the 1990s, but unheard of in the enterprise. This was a revelation to teams traditionally driven by a "waterfall" software development mentality, intense political pressures, and tool decisions with unintended consequences such as silos and separation, even within leading technology firms for whom software collaboration should have been second nature.

But collaboration is hard. It requires stepping outside of one's immediate pressures of deadlines and milestones and looking around at others in the organization for opportunities—opportunities to reuse what someone else has done (even if it needs more work), as well as opportunities to promote what you've built to the rest of the org (so they hear about it and come to use it, thereby improving your code along the way). Even when you do have multiple parts of an organization eager to work together, everything from tool choices to time zones to preferred code formatting styles can stymie collaboration.

Solving this does not require charity or selflessness, but it does require a form of enlightened self-interest—a faith that a bit more investment in the short term pays off in greater reuse, fewer architectural errors, and a more agile dev team than you'd have otherwise. It also requires humility, which is always in short supply! And until this book was written, it required an innate, informally shared understanding of how to make it all work.

Today most of the innovation happening in tech wouldn't exist without Open Source, and the methods we discovered back in the 1990s are more relevant than ever. Older companies need to keep up with the pace of change to stay relevant, and that's why I'm so pleased to see this book of stories about how InnerSource can still transform engineering practices and solve problems across a wide range of long-established companies. I'm expecting to see the InnerSource movement continue to grow. I hope that one day every software engineer is taught these techniques in order to realize the true value of peer-based collaboration at scale.

Glossary

Air cover
: Refers to executive management support for a potentially vulnerable change initiative, such as an InnerSource program. Having support at the highest level helps to get things done and to ensure that mid-level managers cannot block an initiative.

Apache Software Foundation (ASF)
: An American nonprofit organization that supports Apache Open Source software projects, including the Apache HTTPd web server. Founded in 1999, the ASF is home to the world's largest single-license Open Source software repository with over 350 projects and more than 5,000 contributors. The Apache Way is a documented method for transparent peer-based collaborative software development upon which InnerSource is based (see Chapter 2).

Benevolent Dictator for Life (BDFL)
: The term BDFL is commonly used in Open Source projects with a single leader, who is often (but not necessarily) its original creator. The term first appeared in 1995 referring to Python's creator Guido van Rossum. A BDFL oversees a project and has the final say in key decisions related to a project's future development direction. Not all Open Source projects have BDFLs.

BIOS
: BIOS stands for Bosch Internal Open Source, which is the name for Bosch's first InnerSource program. Starting in 2009, the BIOS program was a three-year experiment (later extended by an additional three years) to evaluate the utility of Open Source development practices for corporate software development. BIOS evolved into Bosch's Social Coding program.

Bosch Internal Open Source (BIOS)
: *See* BIOS.

Bottleneck
: A metaphor that denotes when the capacity of a system or organization is limited by one component or team—the bottleneck.

Brooks's Law
: An observation by Frederick Brooks, who was project manager for IBM's OS/360 project, that adding more people to a project that is already late, will make it later. The explanation for this is that new people need to be onboarded onto the project, the combinatorial explosion of the number of communication links in larger groups of people, and the limited extent to which tasks can be divided over multiple people.

Champion
: A person who helps advocate an InnerSource program, by serving as a contact

point, a domain expert, or a leader to help other teams onboard and even arbitrate in an InnerSource program.

Charter conflict
: Charter conflict is a common management problem where two or more individuals or teams each believe a given area of effort is their sole responsibility, which leads to both working in the same problem space, often with duplicated effort, and eventual hurt feelings. Clear guidance from leadership can avoid or resolve charter conflict.

Cheese, wedge of cheese
: A metaphor for an executive manager (see Cheese Story).

Cheese Story
: The Cheese Story describes how disagreements and conflicts get escalated up the management chain to executives, which are drawn as a wedges of cheese. When escalation occurs, one division's "cheese" discusses the team's wishes with the other division's cheese, who will then discuss the issue with his or her engineers. This out-of-band intervention runs counter to InnerSource communication, collaboration, and decision-making processes, all of which are generally more direct.

Chief Architect
: A key role in the core team in Bell Laboratories' SIP project. The Chief Architect typically has intimate knowledge of the software asset.

Code Guardian
: A role in Ericsson's CDS initiative. The Code Guardian is one of three roles (besides Product Owner and Architect) in Ericsson's core teams. The Code Guardian is to "guard the quality of the code" by reviewing contributions from individuals and teams.

Community Developed Software (CDS)
: CDS is an InnerSource program within Ericsson that started with the company's attempt to build a platform without a platform organization. Within the CDS, core teams are responsible for designing and delivering components with well-defined interfaces that feature teams can use to implement the functionality they need.

Continuous Integration (CI)
: First coined as a term by Grady Booch, IBM Fellow and co-inventor of the Unified Modeling Language (UML), and also one of the original 12 practices of Extreme Programming (XP), CI is the practice of integrating code changes into the master code repository to ensure that no defects are introduced that break the system as a whole.

Contributing.md file
: A file offered in every project setup that memorializes working agreements between guest contributors and host code owners, as well as the process for making, reviewing, merging, and supporting contributions (the *.md* extension suggests it's written in Markdown, which is supported in many modern version control systems).

Core Team
: Many successful Open Source projects have a core team that consists of a relatively small number of key contributors. Some projects define a core team to consist of a BDFL and Trusted Lieutenants. Others define a core team by the architecture of the project, for instance, the core team may work on the core engine while everyone else works on modules that the core engine acts on. Companies adopting InnerSource sometimes adopt the concept of a core team, but they may redefine what that means for their context.

Corporate Open Source (COS)
: The term originally used at Bell Laboratories for InnerSource.

Delivery advocate
: A role in Bell Laboratories' SIP project's core team that assists business units in

the task of integrating the shared assets into the business units' software. We've also seen the concept of a delivery advocate in other companies, though they didn't use the term.

Executive air cover
: *See* Air cover.

Feature advocate
: A role defined in Bell Laboratories' SIP project's core team. The feature advocate is responsible for seeing a certain feature to completion.

Guest contributor
: A term used in PayPal's InnerSource program, to refer to a contributor external to the owning team (Host team).

Host team
: A term used in PayPal's InnerSource program, to refer to the team that mentors, accepts, and reviews contributions from guest contributors. *See also* Guest Contributor.

Infrastructure-based InnerSource
: A variant of InnerSource whereby an organization provides the necessary infrastructure for anyone to start a new community. In this model, the organization doesn't provide any resources to support a dedicated core team to develop or maintain a specific project. *See also* Project-specific InnerSource.

Inner Source
: The term coined by Tim O'Reilly in 2000 to refer to the idea of leveraging Open Source development practices for corporate software development. The InnerSource Commons has adopted "camel case" spelling: InnerSource.

InnerSource
: Alternative spelling for "Inner Source" (the original spelling coined by Tim O'Reilly). The one-word spelling was chosen by the InnerSource Commons community to make the term more searchable on the internet.

InnerSource Commons
: Founded in 2015, the InnerSource Commons is an organization of individuals working on sharing experiences, developing public educational materials, and supporting each other as they work on InnerSource implementations either for their employers or as consultants. This group meets online at *http://innersourcecommons.org*.

IRC
: Internet Relay Chat. Chat server software that was invented in 1988 and is still commonly used by Free and Open Source projects. Many alternative communication and collaboration platforms exist today with similar features. One of the better known commercial platforms is Slack.

Liaison
: A formalized role in Bell Laboratories' SIP InnerSource project. The Liaison plays a key role in managing the core team's activities, and is the interface to business units who wish to discuss new work requests. Within the SIP project, the liaison works closely with the Chief Architect.

Linus's law
: Originally coined by Eric Raymond in his essay The Cathedral and the Bazaar: "Given enough eyeballs, all bugs are shallow." In other words, bugs that may seem inexplicable to some might be obvious to others, as long as a sufficient number of people that look at the code. The limitations of Linus's law were demonstrated when the "Heartbleed" bug was introduced into OpenSSL in December 2011, released in March 2012, and neither noticed nor fixed until April 2014.

Markdown
: A lightweight markup language to structure text that can be easily converted to other formats, such as HTML. The file extension is *.md*. Many modern tools have native support for it; GitHub

and GitLab both support Markdown with specific extensions.

Modularity
: Modularity refers to the extent to which a software program's parts are coupled. A high degree of modularity means that modules are very loosely coupled, which in turn means that changes in one module result in a minimal number of changes in other modules. A low degree of modularity means that modules are tightly coupled, which in turn means that a change in one module may require significant changes in all coupled modules.

Product Owner (PO or PMO)
: A key role in the Scrum development framework, the most popular Agile development method. The PO represents the voice of the customer, and has as a key responsibility the maintenance of the product backlog of planned features expressed as "stories"—the role also typically includes setting development priorities for the product.

Project-specific InnerSource
: A variant of InnerSource that focuses specifically on one or a few projects that have dedicated teams, typically supported by corporate funding or by business units. The dedicated "core team" is responsible for maintaining a roadmap and the asset's architectural consistency. *See also* Infrastructure-based InnerSource.

Pull request (PR)
: A set of proposed changes to source code that includes metadata such as review comments. A PR is the mechanism used in distributed (or decentralized) source code management systems such as Git and Bitbucket. In centralized code management systems (such as Subversion), changes to source code would typically be captured in a patch that does not include the metadata that a PR has.

Release advocate
: A formalized role in the core team of Bell Laboratories' SIP InnerSource project. The release advocate is responsible for ensuring that all features that were planned for a release are submitted on time, and keeping track of division-specific impacts of a specific release.

Scrum
: Possibly the most popular Agile method for software development. Scrum defines a number of roles (see Scrum Master, Product Owner), artifacts (e.g., product backlog), and ceremonies, such as the Daily Standup (or Daily Scrum). Being an Agile method that recommends a maximum team size of about 7 to 9 people and that emphasizes face-to-face communication, many companies that adopted Scrum have difficulty scaling up their software development efforts. InnerSource can help in this regard.

Scrum Master (SM)
: A key role in the Scrum development framework, the Scrum Master works to ensure a given team is following all the steps of Scrum, including regular short meetings to discuss daily work, pulling specific assignments from a backlog of feature descriptions called stories, and completing a show and tell of progress called a "demo" at the end of every planned work interval (a sprint), typically two to four weeks in duration.

SeazMe
: An Open Source tool developed and released by PayPal that collects information (and ad hoc documentation) from a variety of sources into a persistent and indexed archive, in order to make information discoverable and to enable collecting metrics on InnerSource collaboration between teams.

Silo
: Metaphor to describe the typical arrangement in large corporations,

where each division (or large collection of code) is a separate, independent entity with very little collaboration and communication between them. Silos often compete for budget and resources, and this competition further undermines collaboration and communication.

Slack
: A commercial collaboration platform that shares many of the features of IRC, but also allows the sharing of many document and image types in addition to plain text. Slack is currently widely used by companies as a collaboration and communication tool. Slack stands for "Searchable Log of All Conversation and Knowledge."

Social Coding
: Bosch's current InnerSource program, which descended from BIOS.

Symphony
: The subject of one of the key InnerSource experiments at PayPal. Symphony was a year-long project to rearchitect a key service used by most PayPal features.

Test-Driven Development (TDD)
: Test-Driven Development is one of the original 12 practices of Extreme Programming (XP), one of the most popular Agile development approaches. In TDD, you write the tests for a feature first, and then the feature itself, after which the test should pass, which suggests the feature is complete.

Trusted Committer
: *See* Trusted Lieutenant.

Trusted Contributor
: *See* Trusted Lieutenant.

Trusted Lieutenant
: In an Open Source context, the term Trusted Lieutenant can refer to a member of an Open Source project's core team who has merge access to the major branch of the source code management system. Trusted Lieutenants may assist a project leader (or BDFL) or may be themselves solely responsible for reviewing, critiquing, accepting and merging contributions to a project. Trusted Committer is a synonym. A Trusted Contributor has achieved a level of trust in their contributions, which is a step on the path to Trusted Lieutenant.

About the Authors

Despite her B.A. in French literature, **Danese Cooper** has a 30-year career in tech, including senior engineering management and open source strategy positions at Apple, Microsoft, Symantec, Sun, Intel, and most recently, PayPal. She is a recognized leader in the Open Source movement, for her work at Sun and as CTO of Wikimedia Foundation (home of Wikipedia), and for her many years of service on Boards of Directors or as an advisor to well-known Open Source projects including the Open Source Initiative, Open Hardware Association, Mozilla, Drupal, and Apache. Currently she serves as the Chairperson of the Node.js Foundation. Danese founded *InnerSourceCommons.org* in 2015 as part of a bootstrapping effort for PayPal. She developed and delivers training for organizations and individuals working with InnerSource through DaneseWorks, her consultancy. She speaks internationally on Open Source and InnerSource trends and evangelizes the Open Source ethos far and wide.

As an academic, **Klaas-Jan Stol** has conducted research on Open Source and InnerSource for the last 10 years. He is a lecturer with the School of Computer Science and Information Technology at University College Cork, a Funded Investigator with Lero—the Irish Software Research Centre, and a Science Foundation Ireland Principal Investigator. The goal of his research is to better understand these novel modes of development work, and what their implications are for the software industry. For example, when Open Source was just emerging as a research topic, most companies didn't show much interest primarily due to a misunderstanding of what Open Source was. It was also not clear how companies could benefit from Open Source. Over the years, this has changed dramatically due to many research studies on this topic, which are published in academic conferences, journals, and textbooks, which in turn are used in university courses. A similar thing is now happening with InnerSource. While it has been a topic of research for years, there are still many misconceptions about what InnerSource is, and how companies can benefit from adopting it. Klaas tries to close this gap by writing books like this one.