InnerSource Patterns for Collaboration

ERIN BANK, CA Technologies
GEORG GRÜTTER, Robert Bosch GmbH
ROBERT HANMER, Nokia
KLAAS-JAN STOL, Lero—the Irish Software Research Centre *and* University College Cork
PADMA SUDARSAN, Nokia
CEDRIC WILLIAMS, PayPal
TIM YAO, Nokia
NICK YEATES, Consultant

Open Source has changed the landscape for software development organizations. There are numerous very successful open source projects involving hundreds of developers dispersed worldwide, and many organizations are interested in adopting the principles and practices to build "open" communities within their organizations. This has been termed "InnerSource" and this paradigm is attracting considerable attention from the software industry. The InnerSource Commons is an industry-driven community that is actively pursuing an agenda to share knowledge, experiences, and lessons learned on adopting InnerSource. One means of doing this is to encode "best practices" as patterns. In this paper, we present a number of patterns that the InnerSource Commons community has distilled, which other organizations that are interested in InnerSource can adopt and tailor to their respective organizational settings.

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

General Terms: Human Factors

Additional Key Words and Phrases: Inner Source, Open Source development practices, software development, organizational culture, patterns

ACM Reference Format:

Bank, E. et al. 2017. InnerSource Patterns for Collaboration. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 24 (November 2017), 15 pages.

This work was supported, in part, by Science Foundation Ireland grant 15/SIRG/3293 and 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero—the Irish Software Research Centre (www.lero.ie).

Authors' addresses: Erin Bank, CA Technologies; email:Erin.Bank@ca.com; Georg Grütter, Robert Bosch GmbH; Robert-Bosch-Campus 1, 71272 Renningen, Germany; email:Georg.Gruetter@de.bosch.com. Robert Hanmer, Padma Sudarsan, and Tim Yao are with Nokia, IHN/9E521, 1960 Lucent Lane, Naperville, IL 60563-1594; email:Robert.Hanmer@nokia.com, Padma.Sudarsan@nokia.com,Tim.Yao@nokia.com; Klaas-Jan Stol, University College Cork, Dept. Computer Science; email:k.stol@cs.ucc.ie; Cedric Williams, PayPal, email:cewilliams@paypal.com; Nick Yeates, email:nyeates1@umbc.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 24th Conference on Pattern Languages of Programs (PLoP). PLoP'17, OCTOBER 22-25, Vancouver, Canada. PLoP'17, OCTOBER 22–25, Vancouver, Canada. Copyright 2017 is held by the author(s). HILLSIDE 978-1-941652-03-9

1. INTRODUCTION

Open Source Software (OSS) has had a tremendous impact on the software industry. Software organizations benefit in various ways from open source software, one of which is the use of the OSS development principles and practices. O'Reilly referred to this as "Inner Source" [O'Reilly 2000], and several authors have suggested that companies "steal and borrow" ideas from the open source domain to improve their software development capacity [O'Reilly 1999; Mockus and Herbsleb 2002; Fitzgerald 2011].

Some of the benefits that the open source development paradigm offers are ones that enterprise companies can leverage through InnerSource. These include setting up internal, thriving communities, leveraging the "wisdom" of the crowd (i.e. an organization's global developer workforce), transparency and visibility, software quality, and shorter time to market [Stol and Fitzgerald 2015; Capraro and Riehle 2017]. Many of the reasons companies pursue inner source are tied to the problems associated with closed development models that may be used within isolated business units—this is often referred to as "development silos."

The existence of 'silos' is due to the way large organizations are typically managed, which tends to be a 'divide and rule' approach. Large companies are usually divided into business units or divisions, each of which may be further divided into smaller units (e.g. departments, teams). While such an approach may make intuitively sense, in practice this leads to local optimization of resources. Business units, departments, and teams are each evaluated based on their efficiency and productivity, but this comes at the price of a number of drawbacks.

In a software development context, one key problem with silos is that of *duplicative development*—or building the same functionality many times and in a variety of ways. Instead, it would be much more cost-effective to write software once and have different divisions reuse, and make contributions to that software. Duplicative development has a number of clear drawbacks. Firstly, the mere fact that multiple teams work on similar functionality is wasteful. Furthermore, it can lead to products within the same portfolio that look and feel very different. Defects may have to be fixed multiple times. It also may lead to products do not work together seamlessly, if at all. Customer using multiple products in your portfolio expect a seamless and consistent experience.

Silos and duplicative development may also lead to a *longer time to market* [Stol et al. 2011; van der Linden 2009]. If a product is dependent on other components, a market release may be delayed if features are missing or defects must be fixed in the latter. Companies may miss the market opportunities and lose out to competitors.

Closed-development companies mean that projects are *limited to testers within a specific group*. This could also mean limitations in terms of resources for vulnerability response and issue resolution. How many issues could be found and resolved with a wider crowd that has diverse experience and perspective? Could more eyes on the code have a positive impact on its quality?

Silo-oriented organizations may also *limit their engineers* in that their collaborations tend to be limited to a smaller number of people that work within the same division. Building relationships, and perhaps more importantly, building trust with colleagues in other divisions of the same company is inhibited because this can only be achieved when collaborations take place. In addition, this inhibits learning and knowledge sharing—given that software development is inherently a knowledge-based activity, this is a very severe shortcoming in organizations that have separated their workforce into divisions. Opening up collaboration via inner source enables engineers to build relationships and work with a broader crowd, which enables them to learn skills beyond their domain, and provides opportunities for synergy and innovation. These opportunities are also attractive to engineering talent.

Inner-sourcing improves software reuse, delivers greater efficiency, inspires innovation and helps with talent acquisition and retention [Riehle et al. 2009; Stol and Fitzgerald 2015; Capraro and Riehle 2017]. The most important aspect of open source that is adopted in InnerSource is actually the *culture* [Neus and Scherf 2005]—while there is no definite set of features that characterizes the culture of open source, key aspects that contribute to successful open source projects include openness, transparency, voluntariness, self-organization, meritocracy, and mentorship. Because changing an organization's culture can be very challenging, effective implementation of

an InnerSource program is difficult, and due to the different organizational contexts there is no single solution to this challenge.

In recent years, an industry-led community called the “InnerSource Commons” has emerged. This community has gained significant momentum since it was founded in 2015 by Danese Cooper when she was hired by PayPal.[1] The InnerSource Commons (ISC) is a forum for sharing experiences and best practices to advance the InnerSource movement. As of November 2017, there are over 220 individual members of the ISC, representing more than 60 organizations across the globe. The ISC conducts two summits annually where members from the community gather to exchange knowledge and experiences. To maximize sharing of knowledge and experiences, all communication within the ISC is subject to the Chatham House Rule [Chatham 2002].

Within the ISC, a patterns subcommunity has emerged which aims to distill and structure best practices for adopting InnerSource and document these as InnerSource patterns. The ISC patterns community gathers regularly in online meetings to write and review patterns. This paper presents a number of patterns that this community has compiled thus far.

Because each organization has its own corporate culture, product architecture, and history of software development processes, these InnerSource patterns may require tailoring prior to adoption in a different context—the ‘context’ and ‘forces’ in each pattern provides insight as to whether a pattern might be appropriate given a certain setting. Notwithstanding the wide variety in organizational contexts, we have found that these patterns can provide useful guidance to other organizations.

The remainder of this paper is structured as follows. In Section 2, we present further background information on extant research on InnerSource and on the ISC patterns community. This is followed by Sections 3 to 7, which present five patterns that the patterns working group of the InnerSource Commons have distilled.

2. BACKGROUND AND RELATED WORK

One of the first studies on InnerSource dates from 2002 which reports efforts at Hewlett-Packard [Dinkelacker et al. 2002]. Since then, numerous other organizations have reported on their programs [Stol et al. 2014; Capraro and Riehle 2017], including Philips [Wesselius 2008], SAP [Riehle et al. 2009], Bell Laboratories [Gurbani et al. 2006; Gurbani et al. 2010], and PayPal [Oram 2015; Bonewald 2017].

Adopting InnerSource is different for each organization because of the different organizational contexts. Nevertheless, common “success factors” have been identified [Stol et al. 2014]. These factors can be used in assessing whether an organization is “fit” for InnerSource. However, they do not provide hands-on guidance for adopting InnerSource.

The InnerSource Patterns subcommunity within the ISC is actively pushing forward a better understanding of how InnerSource works. The subcommunity uses the concept of patterns to document proven solutions for known InnerSource problems [Yao and Sudarsan 2016], as well as to harness new problems and solutions from the larger community. Patterns were first proposed by Christopher Alexander as a way to describe common solutions to recurring design problems in architecture (buildings, cities)—in a similar fashion, the ISC captures “common solutions” to problems that relate to adopting InnerSource.

Because the InnerSource Patterns movement is still new, there are a number of problems for which no proven solution is known or documented. The InnerSource Patterns group has coined the term *Donut Pattern* for such ‘unproven’ patterns. Capturing Donut Patterns are an approach targeted brainstorming of solutions—we have visualized this concept of a Donut Pattern in Fig. 1. A donut pattern is one for which the problem, context, forces and even resulting context are known and specified. The solution, however, is unknown. Given the analysis of the context and forces and knowing the desired resulting context, members of the community can more effectively

[1] This community spells “Inner Source” using “camel case” removing the space, to make the term easier to find online, and this is also the spelling that we use in this paper.

come up with constituent parts of a solution to try. One could think of this as a type of *proto-pattern* and a process for encouraging collaboration.

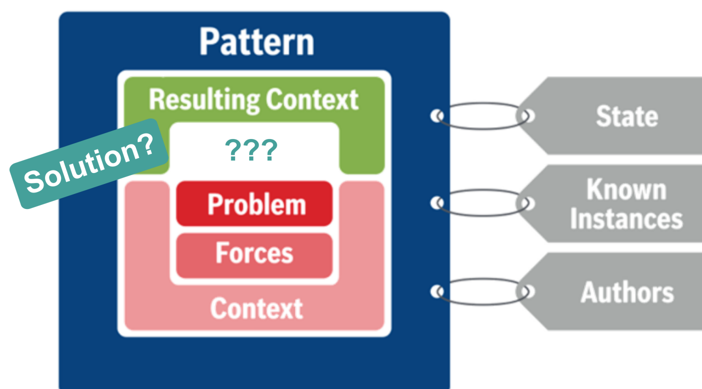

Fig. 1. A *Donut Pattern*, or *proto-pattern*, is one for which the solution is not yet known, but all other primary fields have been specified.

When the solution has thus been created, donut patterns can become "pattern ideas" that can be tried and evaluated. If successful, these pattern ideas can evolve to become proven, full-fledged InnerSource patterns. Note that there is an intention for patterns to be collectively owned and maintained as living patterns within the community. When patterns are used by members, the stories of their experiences are solicited to continue to hone and improve the patterns.

This paper presents four patterns that the community has found to be effective across organizations: 30 Day Warranty, Review Committee, and Common Requirements. In addition to these, a less well-proven idea is also included, which we have named Improve Findability.

Table I. Patterns presented in this paper

Pattern	Status	Summary
30 Day Warranty	Proven	How do you get a team that owns a widely used software component to accept major feature contributions from other internal teams, in spite of a history of poor quality contributions?
Dedicated Community Manager	Proven	How do you ensure that a new InnerSource initiative has the right community manager to grow its impact?
Review Committee	Proven	How do you convince middle management, who is unfamiliar with open source methods, to support a new InnerSource program without micromanaging it and causing it to fail?
Common Requirements	Proven	How do you resolve situations in which business lines sharing a common component have incompatible requirements for it?
Improve Findability	Pattern Idea	How do you resolve the findability issue resulting from poor naming conventions applied to InnerSource projects?

3. PATTERN 1: 30 DAY WARRANTY

3.1 Context

A development unit that uses a software component depends on the team owning the component to accept their contributions. The component-owner team does not have the resources, knowledge, permission, inclination to write the contributed component changes.

3.2 Problem

A team developing a component which is used throughout an organization is resisting acceptance or rejects contributions (feature requests) and as a result blocks progress or is disrupted by frequent escalations.

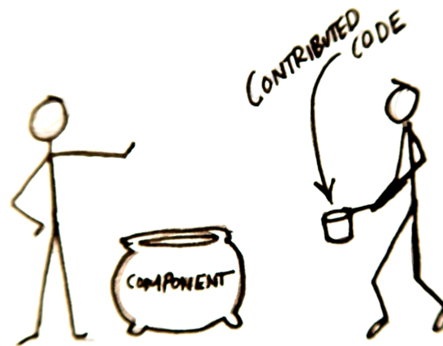

Fig. 2. Rejecting code contributions

3.3 Forces

- —There is distrust of contributions due to a past history of "cheating": development units submitted half-finished contributions and subsequently filed requests for fixes needed to made it ready for use in production.
- —If code is contributed from outside the component-owner team, the team has the natural suspicion that the contributing development unit does not know how to write code that would meet the component-owner team's expectations.
- —Each team looks first to help its own leaders achieve their own goals. This direction of loyalty can complicate resolution of this problem.
- —There is a natural aversion to taking responsibility for code not written by oneself.
- —Contributed code often has to be heavily rewritten before being accepted into the codebase. This can be due to lack of familiarity by the contributing development unit with the code base.
- —There is the fear of the contributors not being available to provide support and bug fixes after the time spent on contribution.
- —Teams fear code contributed by others will lead to high(er) maintenance costs but do not know how to control for that
- —Receiving/component-owner teams may fear that teaching others how to contribute code will expose technical debt in their system and that visibility may be damaging
- —Receiving/component-owner teams may not believe that they will get acceptable code no matter how much mentoring they provide

—Either team may not feel confident in measuring risks or certifying that they are mitigated in a contribution; the system itself is somewhat brittle (may not be ways to fully test and catch all problems).

3.4 Solution

Address the fears of both the receiving and the contributing teams by establishing a 30 day period starting with the time the contributed code goes into production, during which the contributing team consents to provide bug fixes to the receiving team.

Provide clear contribution guidelines spelling out the expectations of the receiving team and the contributing team.

Note that the warranty period could be 45, 60, or 100 days too. The duration may vary based upon the constraints of the project, the software life cycle of the project, commitments to customers, and other factors.

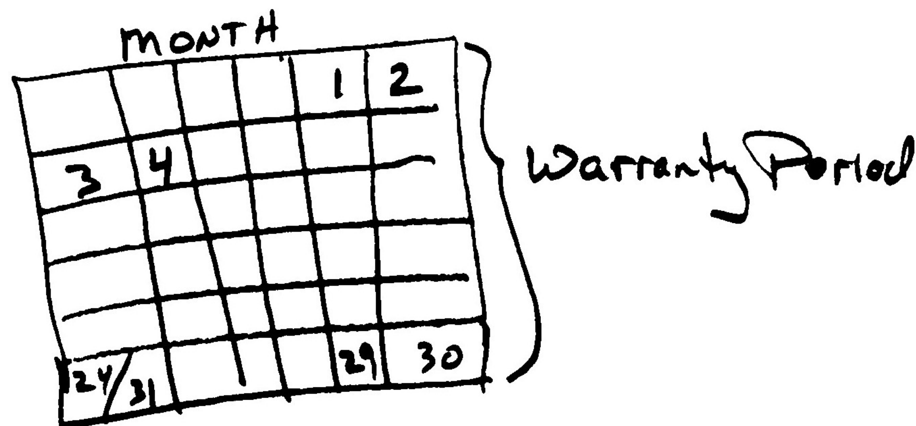

Fig. 3. Sketch of the 30-Day Warranty pattern

3.5 Resulting Context

—The receiving team is willing to accept contributions and able to share the workload of initial adaptations/fixes

—Increased transparency and fairness

—Keeps escalations from becoming too heavyweight

—Once the warranty period has concluded, full ownership of the contributed code is transitioned to the receiving team

3.6 Known Instances

This was tried and proven successful at PayPal.

3.7 Authors

Cedric Williams (PayPal)

3.8 Acknowledgments

Dirk-Willem van Gulik, Padma Sudarsan, Klaas-Jan Stol, Danese Cooper, and Georg Grütter.

3.9 Status

Drafted at the 2017 Spring InnerSource Summit; reviewed 18 July 2017.

3.10 Variants

Ensure cooperation of dependent teams by making them a community by having more than one, meritocratically appointed "Trusted Committers" (TCs) take responsibility.

4. PATTERN 2: DEDICATED COMMUNITY LEADER

4.1 Context

The company is a large and old company. It has no prior experience in Open Source or other, community based working models. The company culture is best characterized as a classical top-down management style—it is generally at odds with community culture. While there are supporters and a sponsor in top level management, middle management in the company is not yet sold on InnerSource. Management was not convinced to provide more than a limited budget to fund only a part time community leader.

The initially selected community leader has little or no prior experience with the Open Source working model and also does not have an extensive network within the company.

4.2 Story

Consider the following story. A company wants to start an InnerSource initiative in order to foster collaboration across organizational boundaries. They have decided to start with an experimental phase with limited scope. Management has selected a suitable pilot topic for the first InnerSource community and expects contributions from many business units across the organization. The company has nominated a new hire to head the community for 50% of his work time, because he was not yet 100% planned for. After 6 months, the community has received only a few contributions, most of which are from a single business unit. The company replaces the community leader with someone who has a longer history in the company, this time for only 30% of his time. After another 6 months, the number of contributions has picked up only marginally. The company is no longer convinced that InnerSource helps to achieve their goal of increased, cross divisional collaboration and abandons InnerSource.

4.3 Problem

How do you ensure that a new InnerSource initiative has the right community manager to grow it's impact? Selecting the wrong persons and/or not providing enough capacity for them risks wasted effort and ultimately the failure of a new InnerSource initiative.

4.4 Forces

If a company does not significantly invest in the initial InnerSource community in terms of budget and capacity for InnerSource, the credibility of its commitment to InnerSource might be perceived as questionable. A common impulse of a company with a traditional management culture to a project or initiative not performing as expected will be to replace its leader. Doing that without involving the community and following meritocratic principles will further undermine the companies commitment to InnerSource by highlighting the friction between the current company culture and the target culture—a community culture.

The value contribution of InnerSource projects will not be obvious for many managers which are steeped in traditional project management methods. Those managers are less likely to assign one of their top people, who are usually in high demand by non InnerSource-projects, to an InnerSource project for a significant percentage of their work time.

Communication takes up a significant percentage of a community managers daily work. At the same time, he or she will likely also have to spearhead the initial development, too. In the face of limited capacity, inexperienced

leaders will tend to focus on development and neglect communication. The barrier for potential contributors to make their first contribution and to commit to the community will be much higher if the community leader is hard to reach or is slow to respond to feedback and questions for lack of time. Furthermore, technically inexperienced leaders will most likely have a harder time to attract and retain highly experienced contributors than a top performer with a high degree of visibility within a company would have.

If a community can not grow fast enough and pick up enough speed, chances are they won't be able to convincingly demonstrate the potential of InnerSource.

If the company selects an experienced project or line manager steeped in traditional management methods to be the community leader, he or she is likely to focus on traditional management topics such as resource allocation, providing structure and reporting channels rather than leading by example through meritocratic principles. This will undermine the credibility of the InnerSource initiative in the eyes of developers.

4.5 Solution

Select a community leader who:

—is experienced in the Open Source working model or similar community based working models;

—has the required soft-skills to act as a natural leader;

—leads by example and thus justifies his position in the community meritocracy;

—is an excellent networker;

—inspires community members;

—can communicate effectively to both executive management and developers;

—is able to handle the managerial aspects of community work.

Empower the community leader to dedicate 100% of his time to community work including communication and development. Inform management of the need to be sensitive to the views of the community when engendering a change in community management. Ideally, empower the community to nominate a community leader themselves.

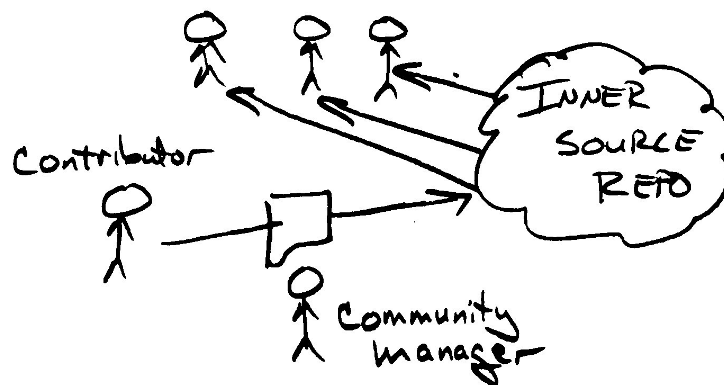

Fig. 4. Sketch of the Community Leader pattern

4.6 Resulting Context

A community leader with the properties described above will lend a face and embody the company's commitment to InnerSource. It will make it more likely that other associates in his network will follow his lead and contribute to InnerSource. Over time, he or she will be able to build up a stable core team of developers and hence increase the chances of success for the InnerSource project. By convincing a large enough audience within his company of the potential of InnerSource, he or she will make an important contribution to changing the company culture towards a community culture.

Having excellent and dedicated community leaders is a precondition for the success of InnerSource. It is, however, not a silver bullet. There are many challenges of InnerSource which are above and beyond what a community leader can tackle, such as budgetary, legal, fiscal or other organizational challenges.

4.7 Known Instances

Bosch Internal Open Source (BIOS) at Robert Bosch GmbH. Note that InnerSource at Bosch has, for the majority, aimed at increasing innovation and to a large degree dealt with internal facing products.

4.8 Status

This pattern was first drafted in Fall 2016, reviewed on 6 November 2016, and again on 6 April 2017.

4.9 Authors

Georg Grütter (Robert Bosch GmbH) and Diogo Fregonese (Robert Bosch GmbH)

4.10 Acknowledgements

Tim Yao, Padma Sudarsan, Nigel Green, Nick Yeates, Erin Bank, Daniel Izquierdo

5. PATTERN 3: REVIEW COMMITTEE

5.1 Context

A company wants to introduce its first InnerSource initiative. Most managers are not familiar with the Open Source working model and are instead accustomed to hierarchical, top-down control style management.

5.2 Problem

Managers perceive the InnerSource working model as a radical departure from the working models they are accustomed to and have experience with. As a consequence, it is likely that they will either reject or micro-manage the InnerSource initiative to try and minimize the perceived risk of this change. In both cases, the benefits of InnerSource cannot be realized. As a result, InnerSource is subsequently discredited.

5.3 Forces

- —The more perceived control a manager has over work done in the InnerSource space, the more likely it is that he or she will support the initiative without prior experience.
- —The less experience a manager has with the open source working model the more likely it is that he or she will want to tightly control the risk of the initiative.
- —The more heavy-handed and micro-managerial InnerSource initiatives are managed, the less likely it is that the open source working model can be adopted to the required extent. As a result, the benefits of InnerSource will not be realized.

5.4 Solution

Establish a review committee comprised of senior managers of all business units which participate in the InnerSource initiative.

- —The review committee members are given the authority to decide as a group which InnerSource projects will receive support in general and funding in particular.
- —Applicants can be elected by review committee members before meetings, to present their proposed InnerSource project for consideration during review committee meetings.
- —Leaders of InnerSource projects currently funded by the review committee are obliged to report on the status of their project during every review committee meeting.
- —Review committee members are obliged to provide constructive feedback to both new applicants and current project leaders during review committee meetings.
- —Every InnerSource project is to be given the chance to react to feedback in between review committee sessions, in order to avoid shutting down the project prematurely.
- —An InnerSource project leader can also present to the review committee the motion to shut down its own initiative. The review committee then has to decide whether or not the business units using the software require time to put measures in place which will ensure that development and/or maintenance of the codebase continues until an alternative solution to development by the InnerSource community is found (if business relevant or mission critical).
- —The review committee should convene regularly. A cadence of two meetings per year has proven successful.

5.5 Resulting Context

- —Managers apply a tool they are comfortable with to InnerSource in order to get the required amount of information about and control over the InnerSource initiative. This familiarity will make it more likely for them to sign off on the InnerSource initiative and grant the required degree of freedom for InnerSource projects.
- —Developers can still self organize to a sufficient degree. Micro-management does not happen because the review committee convenes rather infrequently.

5.6 Known instances

BIOS at Robert Bosch GmbH

5.7 Status

Finalized and Reviewed as of August 31, 2017.

5.8 Authors

Georg Grütter (Robert Bosch GmbH) and Diogo Fregonese (Robert Bosch GmbH)

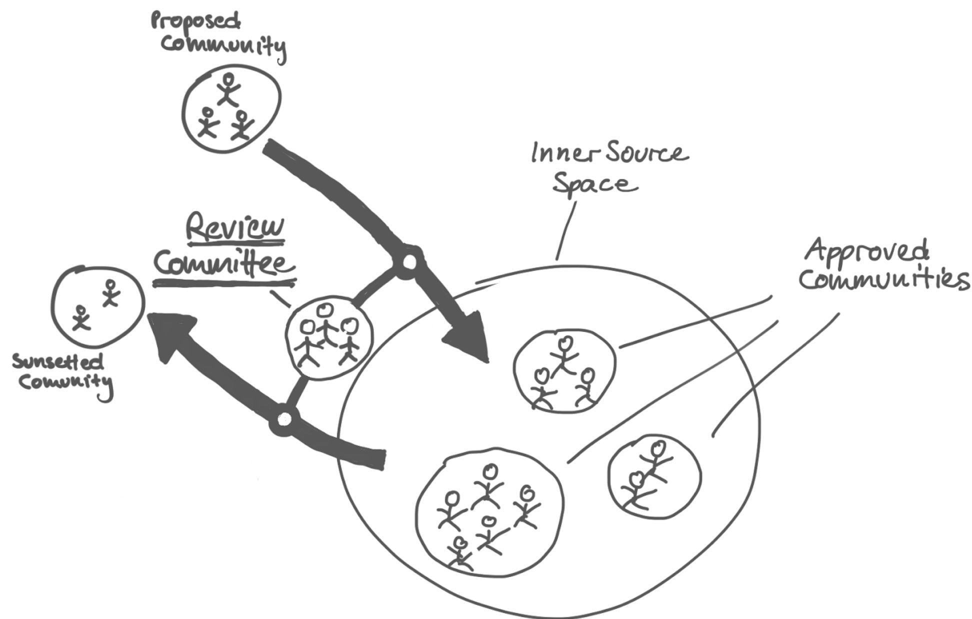

Fig. 5. Sketch of the Review Committee pattern

6. PATTERN 4: COMMON REQUIREMENTS

6.1 Context

Many projects are trying to use common code. There is a shared repository that all the projects access. This pattern applies if there is a Strong Code Owner [pattern to be written] or if there is weak code ownership, or no Benevolent Sponsor [pattern to be written]. Someone (or some project) wrote the code in the first place and contributed it to the repository. The common code is a small percentage of the overall deliverable from any of the projects. Each project has its own delivery schedule, set of deliverables and customers.

6.2 Problem

The common code in the shared repository isn't meeting the needs of all the projects that want to use it.

6.3 Forces

The project that made the code available has one set of needs. Its needs are similar to what some of the receiving organization wants, but not quite the same. Requirements on code should be derivable from real customer needs.

The needs of different customers are generally quite similar; however they might be expressed differently or weighted differently between customers. An example might be how some customers want some result presented in one way while others want it presented in the reverse order—it's simple to do the translation between them, but requires additional coding for one of the cases and the as a result the module that computes the result can't be reused by both customers.

Many customers want the supplier to help them know what they need. The company has many *Systems Engineers* writing requirements for the products. These requirements are supposed to be a distillation of customer needs to guide development of the product. Reusing code is an important goal to save the company time and money.

6.4 Solution

There are two aspects to solving this problem which should be done in parallel:

—Align the requirements of the projects so that the code that meets the requirements for one project also meets the needs for the other projects.

—Refactor the code into smaller pieces for which the many using projects can agree upon requirements.

Additionally, take advantage of customers expecting the supplier to help elucidate requirements. Bring about the alignment of requirements during the customer negotiations and influence the customers requirements rather than changing the component.

In the example presented above, the supplier helps both customers realize that they want the same thing, and it will save everyone effort (and money) if they agree to accept the result in the same format.

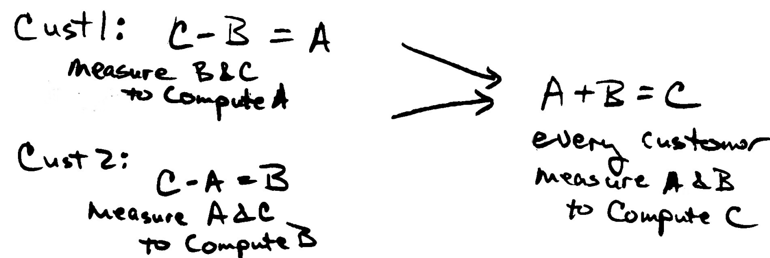

Fig. 6. Sketch of the Common Requirements pattern

6.5 Resulting Context

This might require negotiating requirements changes with the customer. The changes might also require involvement by the sales teams and product managers to get alignment on the requirements. The customer might need incentives, such as discounts, to agree to the changes.

A related pattern (to be written) is a circular story-writing exercise reported at one company employing Inner Sourcing. The developers write a story to solve a problem in one way. The program managers rewrite the story to better express their needs—keeping the essence the same. By the time it returns to developers though they don't recognize it as what they wanted to do in the first place and so balk at implementing it. The solution to this pattern is to have more seats around the planning table so that story modifications are understood across the project not just in the developer or program manager camps.

6.6 Author

Robert Hanmer (Nokia)

6.7 Status

Pattern reviewed on 22 August 2016 and again on 20 September 2016.

7. PATTERN 5: IMPROVE FINDABILITY

7.1 Also Known As

Badly Named Piles, Poor Naming Conventions

7.2 Context

Reusable software component(s) are available internally but users can't easily find them. This problem is more likely to occur in large, federated companies where different organizational units operate as silos. Historically, the company does not have a culture of sharing code across silos.

7.3 Problem

Reusable, internally-developed software component(s) are available, but users can't easily find them.

7.4 Forces

—The volume of contributions to inner source is impacting the ability to find components.

—The internal search engine is not robust, or is not connected to git repositories. (It can be difficult to change this).

—The company has disparate data sources, not all of which are indexed. (It can be difficult to change this).

—Cryptic naming conventions for projects and lack of keywords contribute to reduced findability.

—People may lose confidence in the integrity of inner source and become discouraged from engaging when they search and don't find what they need.

—Duplicative development occurs when people don't find the code that they'd like to reuse or leverage. This results in wasted time and added complexity.

7.5 Solution

To help improve findability for inner source projects:

—Provide guidelines for applying clear, meaningful naming conventions to projects, and reinforce the importance of avoiding cryptic code names.

—Include keywords in project descriptions.

—Apply tagging to repositories (validated).

—Use labels where possible.

—Provide incentives for folloing naming, tagging, and/or labeling conventions (consider gamifying).

—If possible, pull repository names, descriptions, and README.md files into the search engine (not the code itself).

—Instate a concierge service (guide) to help product people find stuff. (This approach might not scale, but could be helpful at the beginning of a program.)

—Consider instating a solution similar to Stack Overflow. (It will likely be difficult to make this happen.)

7.6 Known instances

None known as of yet—this is a pattern idea until it is proven.

7.7 Desired Resulting Context

—Internal components are visible and easily findable.

—Developers looking for code can search for it and find it quickly.

—Developers are now looking internally for software components.

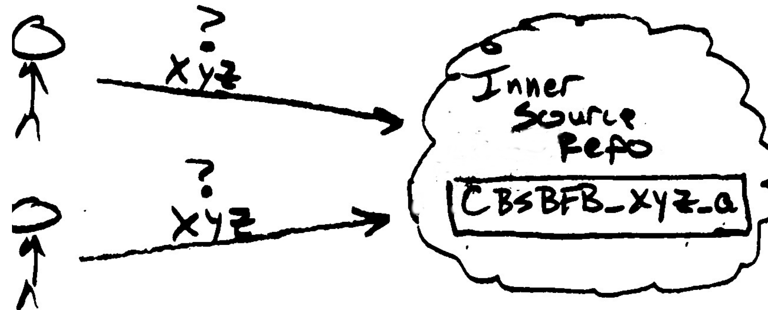

Fig. 7. Sketch of the Improve Findability pattern

—Increased reuse, faster time to market.
—Increased collaboration, because improved findability will lead to increased engagement in inner source practices.
—Higher quality code, because improved findability will lead to increased engagement in inner source practices that means more eyes finding and fixing bugs.
—Increased opportunities for innovation, because improved findability will lead to increased engagement in inner source practices and bring people who have problems to be solved together with others who have ideas on how to solve those problems.

7.8 Status

Brainstormed pattern idea reviewed 2017-03-11.

7.9 Authors

—Georg Grütter (Robert Bosch GmbH)
—Diogo Fregonese (Robert Bosch GmbH)
—Erin Bank (CA Technologies)
—Padma Sudarsan (Nokia)
—Tim Yao (Nokia)

ACKNOWLEDGMENTS

We thank our shepherd Shéhérazade Benzerga for insightful comments and feedback. Thanks also to the members of our PLoP 2017 Writer's Workshop group for their helpful feedback. This work was supported, in part, by Science Foundation Ireland grant 15/SIRG/3293 and 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero—the Irish Software Research Centre (www.lero.ie).

REFERENCES

S. Bonewald. 2017. *Understanding the InnerSource Checklist.* O'Reilly.
M. Capraro and D. Riehle. 2017. Inner Source Definition, Benefits, and Challenges. *Comput. Surveys* 49, 4 (2017).
Chatham. 2002. Chatham House Rule. (2002). https://www.chathamhouse.org/about/chatham-house-rule.
J. Dinkelacker, P. Garg, R. Miller, and D. Nelson. 2002. Progressive Open Source. In *Proc. International Conference on Software Engineering.*
B. Fitzgerald. 2011. Open Source Software: Lessons from and for Software Engineering. *Computer* 44, 10 (2011).

V.K. Gurbani, A. Garvert, and J.D. Herbsleb. 2006. A case study of a corporate open source development model. In *Proceedings of the 28th International Conference on Software Engineering*.

V.K. Gurbani, A. Garvert, and J.D. Herbsleb. 2010. Managing a Corporate Open Source Software Asset. *Commun. ACM* 53, 2 (2010).

A. Mockus and J.D. Herbsleb. 2002. Why not improve coordination in distributed software development by stealing good ideas from open source. In *The 2nd Workshop on Open Source Software Engineering: Meeting Challenges and Surviving Success*.

A. Neus and P. Scherf. 2005. Opening minds: Cultural change with the introduction of open-source collaboration methods. *IBM Systems Journal* 44, 2 (2005).

A. Oram. 2015. *Getting Started with InnerSource*. O'Reilly.

T. O'Reilly. 1999. Lessons from Open-Source Software Development. *Commun. ACM* 42, 4 (1999).

T. O'Reilly. 2000. In response to Matt Feinstein on "Open Source and OpenGL". (2000). http://archive.oreilly.com/pub/a/oreilly/ask_tim/2000/opengl_1200.html.

D. Riehle, J. Ellenberger, T. Menahem, B. Mikhailovski, Y. Natchetoi, Barak Naveh, and T. Odenwald. 2009. Open Collaboration within Corporations Using Software Forges. *IEEE Software* 26, 2 (2009).

K.J. Stol, P. Avgeriou, M. Babar, Y. Lucas, and B. Fitzgerald. 2014. Key Factors for Adopting Inner Source. *ACM Trans Software Engineering and Methodology* 23, 2 (2014).

K.J. Stol, A. Babar, P. Avgeriou, and B. Fitzgerald. 2011. A comparative study of challenges in integrating Open Source Software and Inner Source Software. *Information and Software Technology* 53, 12 (2011).

K.J. Stol and B. Fitzgerald. 2015. Inner Source—Adopting Open Source Development Practices in Organizations: A Tutorial. *IEEE Software* 32, 4 (2015).

Frank van der Linden. 2009. Applying open source software principles in product lines. *UPGRADE* X, 3 (2009), 32âĂŞ40.

J. Wesselius. 2008. The Bazaar inside the Cathedral: Business Models for Internal Markets. *IEEE Software* 25, 3 (2008).

T. Yao and P. Sudarsan. 2016. InnerSource Patterns: Establishing a new inner source patterns community. (2016). presentation at the InnerSource Commons Fall Summit 2016, https://drive.google.com/file/d/0B7_9iQb93uBQbnlkdHNuUGhpTXc/view.

PLoP'17, OCTOBER 22–25, Vancouver, Canada. Copyright 2017 is held by the author(s). HILLSIDE 978-1-941652-03-9