Inner Source in Platform-based Product Engineering

Dirk Riehle, Maximilian Capraro, Detlef Kips, Lars Horn

Abstract—Inner source is an approach to collaboration across intra-organizational boundaries for the creation of shared reusable assets. Prior project reports on inner source suggest improved code reuse and better knowledge sharing. Using a multiple-case case study research approach, we analyze the problems that three major software development organizations were facing in their product line engineering efforts. We find that a root cause, the separation of product units as profit centers from a platform organization as a cost center, leads to delayed deliveries, increased defect rates, and redundant software components. All three organizations assume that inner source can help solve these problems. The article analyzes the expectations that these companies were having towards inner source and the problems they were experiencing in its adoption. Finally, the article presents our conclusions on how these organizations should adapt their existing engineering efforts.

Index terms—Inner source, product line engineering, product families, platform-based product engineering, open source, open collaboration, case study research.

1. INTRODUCTION

Inner source software development is the use of open source best practices in firm-internal software development [26]. Thus, inner source is an approach to collaboration based on the *open collaboration principles* of egalitarian, meritocratic, and self-organizing work [63]. *Egalitarian work* means that software developers are free to contribute to projects that they have not been officially assigned to, *meritocratic work* means that decisions are made based on the merits of an argument and not based on the status of the involved people, and *self-organizing work* means that developers adjust their collaboration processes to the needs at hand rather than strictly following a predefined process [52].

In inner source, no open source software is being developed, but open source best practices are being used. Many engineering organizations expect that complementing existing top-down processes with such bottom-up self-organization will improve their productivity. This article focuses on software development within companies across intra-organizational boundaries, most notably profit center boundaries, that would otherwise hinder any such collaboration. In a nutshell, inner source is supposed to enable collaboration across development silos.

Dinkelacker et al. [26] of Hewlett Packard suggest improved quality, shared problem solutions, and more readily allocatable developer resources as a result of applying inner source. Gurbani et al. [36] [37] suggest that the contributions of many improve quality and that the free availability of a software component within the company reduces collaboration friction. Vitharana et al. [70] of IBM suggest improved reuse. Our experience is that inner source can improve access to resources, software quality and development speed, among other things [52].

Over the last five years, we have helped several software development organizations understand and adopt inner source. Many found it difficult to apply the lessons described in the aforementioned articles to their situation. What seemed to work on paper, did not work in practice.

This article presents case study research on the situation of three major software development organizations which were trying to apply inner source to platform-based product engineering. A *platform* is a set of shared reusable assets, including but not limited to software libraries, components, and frameworks [50]. We define *platform-based product engineering* to be the engineering of software products utilizing a shared common platform. *Product line engineering* [18] is a special but important case of platform-based product engineering.

Our case study companies expected inner source to help them overcome problems with lack of resources, lack of pertinent skills, and unclear requirements. Yet, they had problems putting inner source into practice. To this end, this article addresses the following research questions:

- *RQ1: What are current problems in platform-based product engineering (leading to inner source)?*
- *RQ2: What benefits do organizations expect from adopting inner source?*
- *RQ3: What problems did they experience when adopting inner source?*

The research method employed is *multiple-case case study research* [14] [28] [73] [9]. Data gathering and analysis was performed using workshops, formal interviews, and materials review. The process was incremental with learnings being pro-

- *Dirk Riehle is with the Computer Science Department, Friedrich-Alexander University Erlangen-Nürnberg, 91058 Erlangen, Germany. E-mail: dirk.riehle@fau.de.*
- *Maximilian Capraro is with the Computer Science Department, Friedrich-Alexander University Erlangen-Nürnberg, 91058 Erlangen, Germany. E-mail: maximilian.capraro@fau.de.*
- *Detlef Kips is with Develop Group, 91058 Erlangen, Germany. E-mail: kips@develop-group.de.*
- *Lars Horn is with e-solutions, 91058 Erlangen, Germany. E-mail: lars.horn@esolutions.de.*

vided back to the case study participants to receive validating feedback as to the theories being built ("member checking").

The contributions of this paper are the following:

- It presents three non-trivial case studies of platform-based product engineering.
- It provides three theories, each answering to one of the research questions, specifically:
  - a theory of key problems companies face in platform-based product engineering,
  - a theory of expected benefits of applying inner source to product engineering, and
  - a theory of experienced or expected problems of adopting inner source.

Similar to open source, which evolved from a volunteer-based ("free-for-all") development process to a foundation-based ("managed") software development process [53] [13], we find that for our case study organizations, inner source should move to a governed process beyond the definition given in the beginning of this section.

The paper is structured as follows. Section 2 reviews related work on inner source and product line engineering. Section 3 describes our research set-up, methods employed, and data sources. Section 4 presents the research results as a set of theories addressing the research questions. Section 5 discusses our findings and suggests hypotheses for theory validation. Section 6 discusses the limitations of this work, and Section 7 provides an outlook on future work and some concluding remarks.

2. RELATED WORK

The first research on inner source was reported about by Dinkelacker et al. [26] in 2002. A slow stream of case studies and examples has been reported about since then [36] [31] [52] [68] [66] [37] [70] [62]. Product line engineering [18] [50] has received most of the attention in platform-based product engineering so we focus on this.

2.1 Inner Source Software Development

Inner source is the use of open source practices in corporate software development [26] [62]. Other terms that have been used are hybrid open source [58], corporate open source [37], and firm-internal open source [52]. Inner source is not necessarily intended to replace an organization's development methods, but can be used to extend these methods [65] [67] [68].

DTE Energy [61], Ericsson [64], Hewlett-Packard [26] [47], IBM [48] [70], Kitware [45], Lucent [36] [37], Nokia [41] [42] [43], Philips [71] [67] [68] [69], and SAP [52] all report about inner source in corporate software development. These practitioner reports do not answer our research questions, but they do indicate the relevance of inner source research in general.

This paper focuses on inner source in platform-based product development. With the exception of Philips, none of the organizations reporting about inner source specifically addressed this situation. Consequently, it is not clear to which extent reported inner source problems and solutions apply to inner source in platform-based product development.

Philips applied an inner source approach to software product line engineering. Philips observed that inner source increased collaboration of geographically distributed developers, enabled collaboration across intra-organizational boundaries, enhanced knowledge management and information exchange, "helped to break the platform bottleneck, since using departments are able to create patches", and lead to improved software quality and more efficient development [71] [67] [68] [69]. Inner source adoption at Philips was challenged by process diversity among the organizational units [71]. When compared with the reports from Philips, we find similar benefits, but identify more challenges for the successful adoption of inner source in product line engineering.

In contrast to the practitioner reports from Philips and other organizations, we performed case study research using qualitative data analysis. Our cases cover three different mature development organizations, all of which are culturally and socially homogeneous. We reduced complexity by excluding cases of globally distributed software development. Thus, our resulting theories have a significantly higher validity and reliability than the practitioner reports.

In absolute numbers, there is still less research literature on inner source than there are practitioner reports.

Melian and Mähring [46] as well as Gaughan et al. [31] performed exploratory studies. They discuss benefits and challenges of inner source adoption. Stol et al. [63] introduce a model of nine key factors to support inner source adoption. Stol et al.'s model was synthesized from literature and evaluated in three case-studies. While some of the key factors are geared towards mitigating inner source adoption challenges, the model does not aim to present benefits and challenges of inner source adoption. None of the studies discusses specifics of inner source adoption in software product line engineering.

Stol et al. [62] present a case study of an undisclosed organization that is developing a software product line. They identified 13 challenges of integrating software developed using an inner source approach into their product line. We can confirm some of their findings, but also present theories and draw conclusions that go beyond the situation of integrating existing inner source components. We address the full life-cycle, from creation through development and use to maintenance of a component, and we do so using three large independent case studies rather than one. We therefore believe our results are more broadly applicable.

2.2 Platform-based Product Engineering

In our research, we investigate inner source as applied to platform-based product engineering. Most inner source reports discuss one-off projects where only one inner source component was being developed. In contrast, our work is about a group of products (case 1), a product family (case 2), and a product line (case 3) [5] [50], all three on top of a single shared platform that offers a large number of shared reusable assets.

A product line, according to Clements and Northrop is "a set of software-intensive systems sharing a common, managed set

of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [18]. This definition focuses on the artifacts. In addition, Schwanninger et al. state: "[...] the term 'software product line engineering' denotes a collection of engineering techniques and practices that supports the efficient development of such software-intensive systems (or products)" [57]. This second definition focuses on the processes rather than the artifacts and is more in line with our work here.

A significant part of research into product line engineering is about tools and mechanisms for managing product variability, for example, [2] [10] [11] [12] [21] [22] [24] [25] [49] [55] [56] [59] [60] [65]. Our work is orthogonal to this research, because we focus on the processes leading to the specification, development, and use of shared components, rather than the tools and mechanisms for managing it. Also, the specific challenges of product line variability are not a concern for this research, because the shared reusable assets from our case platforms not only supported a product line but also a product family and a product group.

In the established terminology of product line engineering, the domain engineering process governs the identification, definition, creation, and evolution of reusable assets, typically made available to applications as part of a platform the applications or products built on. The application engineering process then governs the selection, configuration, adaptation and eventual use of a reusable platform asset in the context of an application for a particular market segment [1] [3] [4] [6] [7] [8] [16] [17] [18] [50]. Our research is related to these processes if viewed more broadly (not just a product line but also a product family or group). Specifically, our research identifies problems that the case study companies had in their engineering efforts and which they believed could not be overcome using a product line approach but only using an inner source approach.

Problems with application and domain engineering processes in product line engineering have been identified, for example, by Berger et al. [10] and Jepsen et al [39]. Without calling it inner source, this research came to similar conclusions as the inner source research reviewed earlier.

The relationship between inner source and product line engineering has been recognized by industry. Most notably, van der Linden presented a tutorial at the 13th International Conference on Product Line Engineering about using inner source in product line engineering [67]. Like [66], this is a practitioner report. No research is known to us that specifically combines inner source with platform-based product engineering in general or software product line engineering in particular.

3. RESEARCH APPROACH

We performed multiple-case case study research. Case study research is a natural choice for dealing with phenomena for which no established theories exist [27]. Case study research is a well-established exploratory research method [9] [14] [19] [23] [28] [29] [73].

3.1 Case Selection (Sampling)

We acquired the case study companies from our industry network. We looked for cases that were similar along the following dimensions:

- Established long-running set of products on top of a shared platform (age > 10 years)
- Mature software development organization with established platform engineering practices in place
- Sufficiently large development organization (developer population size > 500 people)
- Culturally and socially homogeneous, with all developers located in one location or region

All our cases fulfill the properties. This makes them comparable along these dimensions, allowing us to draw cross-case conclusions and strengthen the breadth of our theory [9].

Please note that this focus also limits the generalizability of our results; most notably, and deliberately, we excluded problems and solutions of globally distributed software development in this research.

3.2 Cases and Companies

The three cases stem from three independent, large and diversified, internationally operating software product companies. Table 1 shows key properties of the products and their owning company.

- Company 1 provides multiple business software products. The case study (case 1) is about a particular product group.
- Company 2 provides a broad portfolio of products. The case study (case 2) is about a health-care software product family.
- Company 3 provides a broad portfolio of products. The case study (case 3) is about a telecommunications carrier software product line.

As mentioned, we avoided the complexity of globally distributed software development. Only during the course of analysis did we learn that case 2 collaborated with a remote party. When we inquired further, our case study partners confirmed that they thought this information was not relevant for the case.

We had not selected for this, but found that all three cases shared the same organizational setup:

- All products and the supporting platform are owned by a single *business unit* with a single overall *business owner* responsible for all products.
- The business unit is broken up into *product units*, each of which is a *profit center* of its own. A product unit manages the development of a particular product as sold to a particular market. A profit center is an organizational unit that is expected to directly contribute to the company's profit. The manager of a product unit has revenue responsibility for it.

		Case 1	Case 2	Case 3
Product group/family/line	**Product domain**	Business software	Health-care software	Telecommunications carrier software
	Type of platform-based product engineering	Group of products on shared platform	Product family on shared platform	Product line including shared platform
	Age of products	> 10 years	> 10 years	> 10 years
	Number of developers in product engineering	> 500 developers	> 500 developers	> 500 developers
	Is product engineering distributed?	No (same campus)	Yes, but within same metropolitan area	No (same campus)
	Developer population	Socially and culturally homogeneous	Socially and culturally homogeneous	Socially and culturally homogeneous
	How is product engineering organized?	Product = profit center Platform = cost center	Product = profit center Platform = cost center	Product = profit center Platform = cost center
Company information	**Age of company**	> 20 years	> 20 years	> 20 years
	Total number of developers in company	> 1.000 developers	> 10.000 developers	> 10.000 developers
	Is the company operating internationally?	Yes	Yes	Yes
	Case sponsor	Product group business owner	Platform organization	Internal consulting group

Table 1. Key properties of case study products and their companies

- The supporting *platform organization*, which provides the reusable assets, is a *cost center*, paid for jointly by the product units. A cost center is an organizational unit that supports other units, but is not expected to contribute to company profits directly.

We therefore distinguish three main and distinct organizational units of analysis: the overall business unit, the individual product units, and the platform organization.

3.3 Data Gathering

In each case, we were brought in by a case study sponsor. In case 1 and 2 we gained access to all units of analysis (product group business unit, selected product units, platform organization) as well as individuals from the corresponding organizational units. In case 3 we worked through an internal consulting group that mediated our access to the units of analysis.

In all cases, audio recordings of discussions were not permitted. In case 1 and 2 we went in as two researchers, with one researcher asking questions and the other researcher taking notes. In case 3 only one researcher was permitted, who also took the notes. In total, we conducted 21 semi-structured interviews of at least one hour or longer, see Table 2. We also collected product information, internal memos, software documentation, and organizational documentation, see Table 3.

We took on cases in the order of their numbering. Starting with case 1, we refined our questions and perspectives in the first set of interviews together with a representative of the case study sponsor and selected follow-on interviews accordingly.

Our key questions and interview guidelines remained stable after the first case and have been applied to case 2 and 3 like they had been applied to case 1. However, interviews of this type of research are exploratory, so we allowed for theoretical sensitivity, and followed discussion paths that had not been foreseen in our interview guidelines.

Case 2 already repeated most issues of case 1, but during case 3 we learned little new, so we concluded that we were nearing theoretical saturation and should stop [15] [19] [34].

3.4 Data Analysis

We performed iterative and incremental “qualitative” data analysis (QDA). We employed MaxQDA, a qualitative data analysis tool. Theory building using QDA consists of repeatedly working through existing and new materials, annotating (“coding”) text segments, and extracting a code system, the backbone of the theory under development [15] [19].

A code system consists of a hierarchy of codes, with so-called core categories at or near the root, and the most specific codes as the leaves. Different activities transform the hierarchy in different ways.

Case	Year	No. Interviews	Workshops	Supplemental Materials
1	2012	11	5	Yes
2	2013	6	None	Yes
3	2013	4	3	Yes

Table 2. Interview and materials information

	Case 1	Case 2	Case 3
Unit of analysis access	Direct access to all units of analysis	Direct access to all units of analysis	Mediated by sponsor
Subject access	Interview partners selected by consensus	Interview partners selected by consensus	Mediated by sponsor
Types of data collected	Collateral materials, interview notes	Collateral materials, interview notes	Collateral materials, interview notes
Researchers	Two researchers (one interviewer, one scribe)	Two researchers (one interviewer, one scribe)	Single researcher taking his own notes

Table 3. Access to units of analysis, data gathered, methods employed

- *Open coding* creates the basic set of codes from which the hierarchy is built. Open codes are straightforward annotations of the primary materials and directly link to them.
- *Axial coding* builds the code system by deriving more abstract concepts and categories from open codes, that is, the axes of the code system are being developed.
- *Selective coding* finally allows the coder to choose what is important and what is not. By dropping irrelevant aspects, the code system is being shaped into the theory backbone.

Concepts in the code system are cross-linked by memos to enrich concepts and relationships and the resulting theory with insights from the primary materials. We applied the constant comparative method, which ensures that the code system remains cohesive and focused on the questions to be answered [32].

The resulting code system and its memos are an abstract representation of the theory presented in this paper. The resulting theories are descriptive in nature. They have been derived using an inductive process and are therefore applicable to their original context, but not necessarily beyond [19] [15].

3.5 Quality Assurance

The empirical base of the work enhances the trustworthiness of the presented theories [33]. In general, the quality of the theories is maintained by following the methods properly.

We had a second coder code a second code system. This independently developed code system showed a high degree of agreement with the original code system [44]. For assessing this, we sat together and went through the codings one-by-one, finding little inconsistencies, supporting our conclusion of high rigor of our work and strengthening our theory's reliability.

The broad array of available materials supported data triangulation that increased the internal validity of our theories [35].

4. RESEARCH RESULTS

This section presents a theory of selected problems in platform-based product engineering, the expected benefits of applying inner source to such product engineering, and adoption problems when doing so.

4.1 Presentation of Results

This section presents results using *cause-and-effect diagrams* [38]. We found that this type of diagram brings out the theories better than a more traditional top-down concepts and relationships discussion.

In our diagrams, a rectangle represents a concept. A concept has a name. The numbers at the bottom of the box indicate the case in which they were mentioned. A category is represented as a rectangle with gray background. If a concept is positioned on top of a category, it belongs to that category.

Causes and effects flow from left to the right. An arrow between two concepts links the source to the target as one cause to one effect. An effect can also be a cause to other effects; cause and effect are roles of concepts. Cause and effects can have m:n relationships, and we capture these relationships as an acyclic graph. Cause and effect relationships are transitive. Links derived from other links by transitive concatenation are omitted in our diagrams.

In all three case studies, we were not allowed to make audio recordings. The quotations we provide in the following sections are derived from our notes and therefore summarize or paraphrase what we heard. In addition, we changed the terminology from company-specific terms to generic terminology.

4.2 Problems in Product Engineering

Figures 1 and 2 present cause-and-effect diagrams of problems and their effects on the engineering efforts of our case study

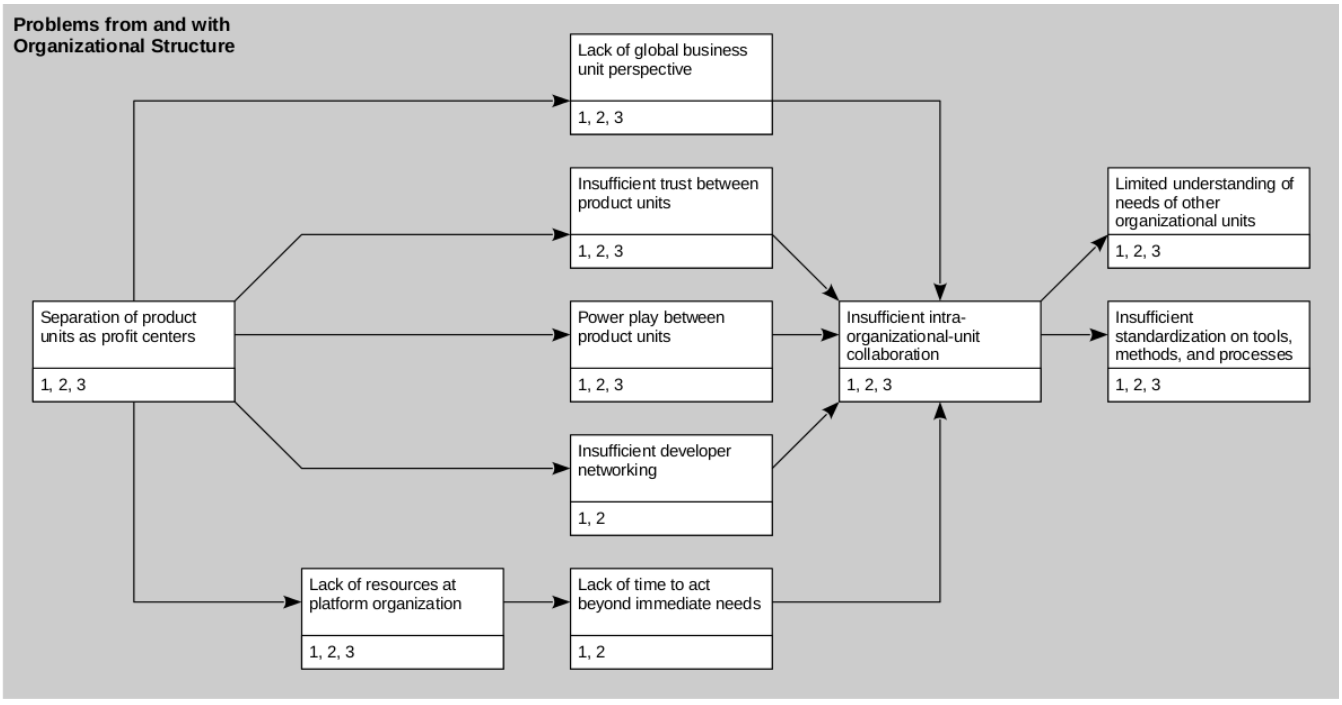

Figure 1. Problems in our case study companies resulting from their organizational structure

companies. Figure 1 shows problems with organizational structure, and Figure 2 shows how these problems affect the domain engineering process. Both figures share some of the same causes; they have been split up for readability purposes.

The key result is the following:

A root cause, the separation of product units as profit centers from a platform organization as a cost center, leads to delayed deliveries, increased defect rate, and redundant software components.

In our case studies, the business unit owns all products and the platform, a product unit develops a particular product, and a platform organization supports the product units in their work by providing shared reusable assets. The business units in our case studies are all structured into product units as profit centers and the platform organization as a cost center.

As Figure 1 shows, the problems encountered with the organizational structure are traced back to the "separation of product units as profit centers and platform organization as cost center", which makes them "silos" in the language of our interview partners, that is, organizational units that do not collaborate sufficiently. Specifically, the "separation of product units as profit centers" leads to

- "lack of global business unit perspective" where each product unit acts in their own interests irrespective of possible synergies from collaboration,
- "insufficient trust between product units" where other product units are viewed as threats or competitors rather than possible collaborators,
- "power play between product units" where managers in some or all of the product units are fighting to enforce their interests irrespective of other product unit needs,
- "insufficient developer networking" where developers do not find the time to talk to each other across the organizational unit boundaries.

In addition, the separation starves the platform organization for resources. This leads to

- "lack of resources at platform organization", because profit centers responsible for their own revenue are always in a stronger position to hire developers than any cost center.

Figures 1 and 2 do not show every cause and effect relationship, and discussing all interactions is beyond a reasonable length for this article. In the following subsections, we therefore focus on the following three central cause-and-effect chains:

1. Lack of resources at platform organization → Delayed domain artifact realization → Delayed product delivery
2. Power play between product units → Poorly prioritized domain requirements → Rework and wasted effort → Delayed product delivery
3. Insufficient intra-organizational-unit collaboration → Limited understanding of other organizational units → Unclear reusable assets requirements → Insufficient reusable asset quality → Increased defect rate

We chose these chains for presentation because they received the most mentions and interest in our interviews.

4.2.1 Chain 1: Lack of resources

In all three cases, the platform organization had a significantly higher workload than any of the product units, despite the more direct pressure on the product unit to deliver a product.

"All this work overload leads to lower code quality. I just finish up as quickly as I can and then move on." Developer (platform), case 1.

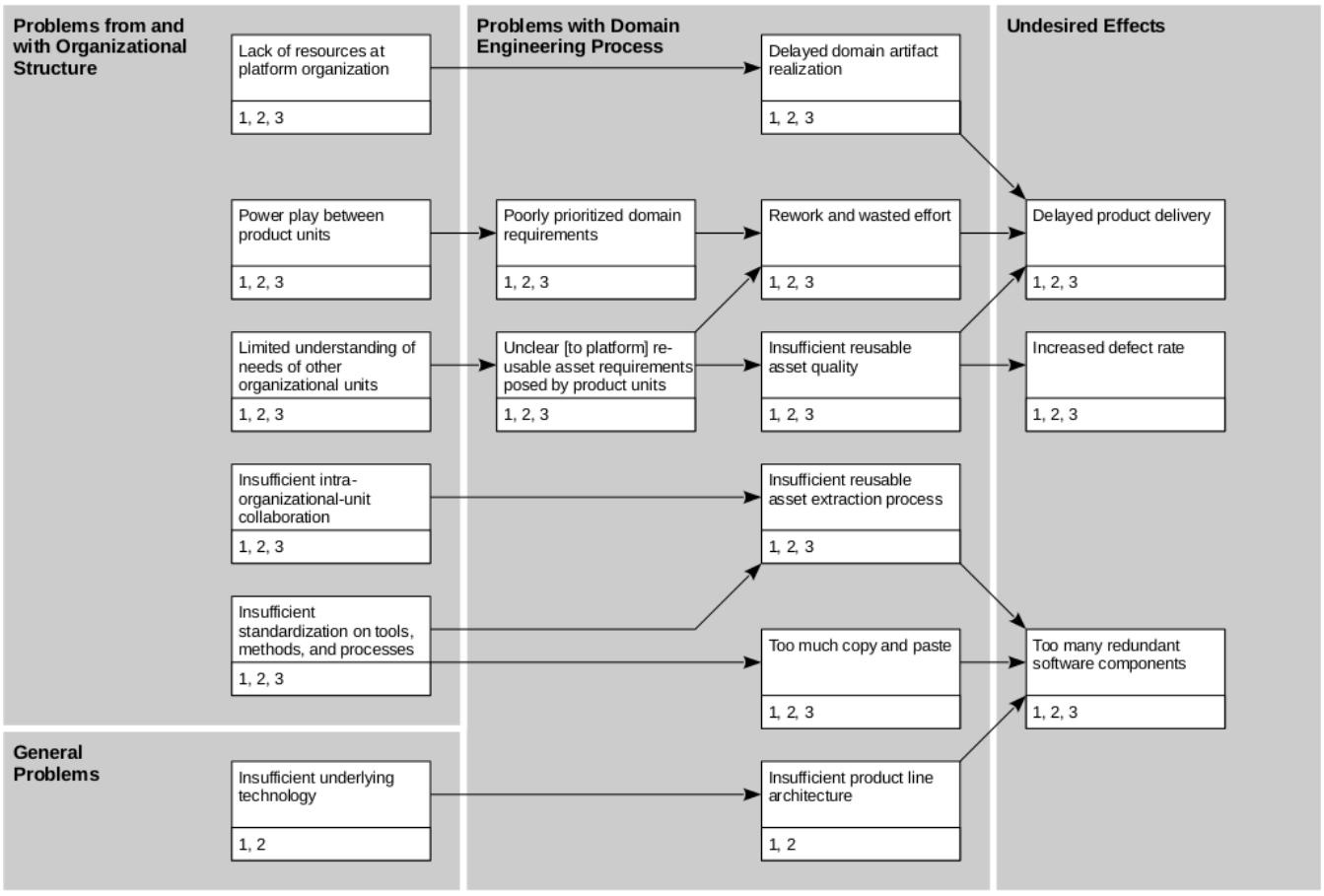

Figure 2. Resulting process and artifact problems in case study companies

"The platform often misses delivery deadlines for reusable assets, which keeps product units from delivering their own features in time." Developer (product unit), case 2.

"We fixed the bug ourselves, using a work-around. We cured the symptom, not the cause, but the platform organization had no time for this bug." Developer (product unit), case 2.

"The platform organization is completely overloaded by too many reusable asset requirements from product units. Most never get realized." Mediator, case 3.

The lack of resources of the platform organization has various consequences, including delayed delivery of products and lower quality of the code base.

All the products are mature and were bringing in substantial revenues. So why is the platform organization not as well staffed as the product units?

"The cost pressure is not high enough; time-to-market is more important. That is why we [product unit] get new developers more easily." Developer (product unit), case 1.

"We are moving the platform towards becoming a product of its own so that we can more easily hire developers ourselves." Manager (platform), case 2.

"Making a case for a new developer to save costs is much harder than making a case for a developer who will bring in more money." Mediator, case 3.

In all case studies, (product unit) profit centers find it easier to hire new developers than (platform organization) cost centers. In case 2, it had led management to contemplate turning the platform organization into a profit center itself (they were considering to turn the platform into a product of its own).

4.2.2 Chain 2: Product unit power play

In all three cases, the requirements engineering process for reusable assets suffered from poor prioritization of the requirements.

"A consequence of the power play between product units is that the platform drives the [domain engineering] process and involves product units only very late." Developer (platform), case 1.

"We [platform organization] don't know how to prioritize reusable asset requirements, and the product units are no help because each feature is most important." Manager (platform), case 2.

"Our feature requests often don't get prioritized highly enough, so we have to implement them ourselves. This leads to inefficient and ugly code." Developer (product unit), case 2.

Product units found it hard to participate effectively in the domain engineering process for the definition of shared reusable assets. This is a result of the power play between the product units: The power play led to disagreement and stalemates, leaving it to the platform organization to define and prioritize re-

quirements. The platform organization in turn does not know how to do this well because it is too far away from market requirements:

> *"We [product unit] often have to change requirements, and the platform does not prioritize these change requests highly enough. Generally speaking, the platform organization does not prioritize well, because it is too far away from the customer." Architect (product unit), case 2.*

While product units believe that the platform organization cannot prioritize well, the stalemate between product units to get their requirements prioritized highest has put the platform organization in charge of domain requirements prioritization—even though the product units believe that knowing their customers is important to prioritize these domain requirements right.

4.2.3 Chain 3: Insufficient collaboration

The lack of sufficient collaboration between product units and between product units and the platform organization led to limited understanding of what product units need and hence what the reusable asset requirements provided by them actually mean:

> *"Our silo culture and the lack of collaboration hinders the effectiveness of domain engineering. We seem to never agree on what would be an important reusable asset nor its specific features." Developer (product unit), case 1.*

> *"Reusable assets often don't work. They don't meet our [product unit] requirements." Manager (product unit), case 1.*

> *"I have too much to do to contribute code [to other projects]. We […] file change requests and that's it." Developer (product unit), case 2.*

> *"The lack of collaboration [across the product family] really hurts a unique look-and-feel." Manager (product unit), case 2.*

A main consequence of not understanding reusable asset requirements is rework and wasted effort until the product units are satisfied. This implies a higher defect rate than would have been necessary:

> *"We didn't understand the reusable asset requirements as communicated by the product units. […] Not much worked when we first delivered the component." Developer (platform), case 1.*

Summarizing, the case study companies felt that their existing domain engineering processes were not delivering reusable assets of sufficient quality fast enough.

4.3 Expected Benefits of Inner Source

The case study sponsors had read our early work at SAP on inner source (then called "firm internal open source") [52], which also provided their understanding of inner source. In that definition, we followed the basic pattern of explaining inner source as open-source-style software development within the company.

In this original understanding of inner source, the key idea is that all relevant software is laid open for everyone inside the company. As new requirements surface or problems with existing components are found, developers help themselves by contributing new features to components or fixing bugs of components that they are not necessarily responsible for.

In this subsection, we present what case study sponsors were expecting to achieve by applying inner source to their platform-based product engineering. In the next subsection we present the reservations they had and the problems they were experiencing in their inner source adoption efforts.

4.3.1 Overview of Expected Benefits of Applying Inner Source

Table 4 displays the expected benefits as taken from our analysis and the cause-and-effect concept linkage.

We separate general benefits that accrue to everyone from the benefits that accrue only to specific units of analysis. In addition, we add benefits that accrue to developers because of the high number of mentions.

- *General benefits* expected are improved innovation, collaboration, development efficiency and uniformity of processes. The largest subcategory is higher development efficiency, where improved code reuse and quality were key mentions. Finding and fixing bugs faster was particularly important. It was mentioned that a higher awareness of overall business unit goals was important and could be achieved. Finally, innovation was assumed to speed up.

- From the *business unit perspective*, inner source would get products to market faster. From a *product unit perspective*, product quality would improve, the platform would be easier to work with, and problems would be solved faster because of the product unit's broader understanding of the involved assets. From the *platform organization's perspective*, the benefits were complementary: A lower workload was assumed, because product units would be empowered to help themselves and requirements would become clearer and better prioritized.

- *Specific benefits* accruing to software developers were a higher job satisfaction and an improved reputation within the company.

Many of these benefits have already been reported about in the literature, see Section 2. Here, we'll first focus on the expected benefits as they relate to the problems of platform-based product engineering reported in the previous section, and then add selected expected benefits that are of interest to software development in general.

4.3.2 Expected Benefits towards Problems with Platforms

Inner source is expected to address the problem of "lack of resources" in the platform organization:

> *"The traditional processes are like a corset; we sometimes have to wait for a year to receive the features we need." Manager (product unit), case 2.*

> *"Rather than wait for the platform to add the new feature, we would like to do it ourselves to overcome the resource capacity problem." Manager (product unit), case 2.*

General benefits for everyone

- Improved global perspective
- Improved innovation
- Improved intra-organizational collaboration
- Improved intra-organizational knowledge sharing
- Improved development efficiency
  - Higher code reuse
  - Higher code quality
    - Earlier bug detection
  - More efficient use of tools
  - Improved resource management
  - Improved development speed
    - Faster bug fixing
      - because of Linus' law [51]
      - because of less administrative overhead
    - More efficient collaboration
  - Better clarified responsibilities
- Lower software complexity
- More uniform processes

Specific benefits to overall business unit

- Faster time-to-market
- Improved engineering process

Specific benefits to product unit

- Improved product quality
- Better requirements comprehension by platform
- Faster problem resolution by helping themselves

Specific benefits to platform organization

- Improved requirements
  - Clearer requirements because of better understanding of product unit
  - Better prioritized requirements by higher involvement of product unit
- Lower workload by enabling product unit developers to help themselves

Specific benefits to developers

- Higher job satisfaction
  - through improved relationships with colleagues
  - more meaningful work in inner source projects
- Improved reputation

Table 4. Overview of benefits expected of applying inner source to platform-based product engineering at case study companies

"Inner source helps allocate existing resources in a more efficient way [than existing approaches]." Mediator, case 3.

When discussing product unit power play and poorly prioritized and defined requirements, interview partners pointed out that inner source gives product units back some power and reintroduces a better understanding of the business value of features:

"Using inner source, we [product unit] can reclaim more say in feature prioritization, which we had lost to the platform organization." Manager (product unit), case 2.

"Inner source helps better determine the business value of requirements and prioritize them right." Mediator, case 3.

Finally, on the problem of "insufficient collaboration", inner source brings about more knowledge sharing, which was widely discussed as beneficial by our interview partners:

"Inner source helps product units gain the necessary knowledge for efficient use of platform components." Developer (platform), case 1.

"Inner source helps us better share knowledge to alleviate the effects of people leaving the company." Owner (business unit), case 1.

"We would like to broaden the capabilities of our developers beyond their immediate product, and inner source helps us do that." Manager (product unit), case 2.

4.3.3 Expected Benefits towards General Problems

Interview partners not only discussed how they expect inner source to help address the problems with platforms, but also how it helps improve development efficiency in general.

Inner source is expected to speed up development:

"By sharing best practices through inner source collaboration, I expect us to get more effective in using our tools." Developer (platform), case 1.

"Through inner source we'll get to know more developers which will help us fix problems faster in the future." Manager (product unit), case 2.

"Inner source improves time-to-market." Mediator, case 3.

Also, inner source is expected to improve code quality:

"Inner source leads to more uniformity and reduction of complexity [...]" Developer (platform), case 1.

"I expect a shared code base to be of higher quality." Developer (product unit), case 2.

"Inner source should help unify the quality assurance processes." Manager (product unit), case 2.

"Inner source encourages product units to find and fix defects in platform code. " Mediator, case 3.

Finally, inner source is expected to improve code reuse:

"Inner source [between product units] will make it easier to reduce redundant code, move components into the platform where they belong."Developer (product unit), case 2.

"We expect to see more code reuse." Mediator, case 3.

The general assumption is that inner source gets people to collaborate more and better across organizational unit boundaries and that this leads to more and better knowledge sharing and broader understanding of one's own and other people's work.

Our interview partners expect that, due to these changes, requirements are communicated more clearly, prioritized better, and understood more easily. Development efficiency improves because people understand the implications of their work better and can draw on broader support going about their work.

Several interview partners suggested that a well-organized inner source process would be a superior domain engineering process when compared with the traditional cross-functional teams that were responsible for new reusable asset definition and implementation.

Finally, interview partners suggested that inner source improves innovation processes:

"Inner source lets product units participate more in prioritizing and realizing platform requirements; this added flexibility creates more innovation." Architect (product unit), case 1.

"Most innovations take place outside the platform unit; using inner source we can more easily transfer them to the platform." Developer (product unit), case 2.

"Inner source takes the platform closer to the customer, makes it more relevant." Architect (platform), case 2.

Finally, inner source is expected to motivate developers and help them build a reputation.

"I expect developers to be more satisfied about their job." Owner (business unit), case 1.

"Inner source developers will see an increase of their internal 'market value'." Owner (business unit), case 1.

"Opening up assets and processes will increase the respect for platform development." Manager (platform), case 2.

Such added motivation and visibility was considered to be beneficial to the company as well.

4.4 Experienced Problems with Inner Source

While the interview partners from our case study companies expected that the benefits described in the previous subsection could be achieved, they also had questions as to how this could be done best.

All three case companies had already been applying inner source using the generic model laid out in the literature without much concern for the specifics of platform-based product engineering. In this subsection we present the problems they were experiencing.

Table 5 shows the relevant part of our code system. The main categories are

- "problems with developers" (both product unit and platform organization) and
- "problems with product unit managers" (only product unit, not platform).

Where it says "problems with [...]" in Table 5, the problems had already materialized. Where it says "expected problems with [...]", interview partners were only expecting these problems but had no actual experiences to back these up.

Expected but not empirically experienced problems are not necessarily irrelevant: They represent fears or misunderstandings, even if those fears may not be grounded in reality. For example, some worried about "degradation of code base due to uncontrolled contributions". Here, the (wrong) assumption is that projects are free for all to write to. Like in open source, any real inner source project will provide unlimited read access but will tightly control write access, typically employing a two-stage review process for quality assurance [30].

Two main subcategories emerged for both developers and product unit managers: Lack of engagement due to

- "boundary conditions not being right" and
- "active psychological resistance".

There were no concerns specifically attributed to engineering managers from the platform organization. It was assumed that they stood the most to gain since they were complaining about the lack of resources the loudest.

4.4.1 Problems with Developers

For developers, the worry was that they were generally too overloaded to contribute or would not know how to do it or would find the software too complex to make a contribution. Some of these problems can be remedied short-term by education (how to contribute) while others will remain long-term research topics (reducing software complexity). They are either manageable or out of scope from an inner source perspective.

In scope is the active psychological resistance that some developers and product unit managers showed. For developers, it boiled down to two subcategories:

- Dislike of performing quasi-public work and
- fear of follow-on and maintenance work.

4.4.1.1 Dislike of performing quasi-public work

With assets being more open and work being more transparent, a developer can build a reputation as well as lose one. Work is out in the (corporate) open, and mistakes are more visible than before. Achievements are more clearly attributable to individual developers.

"Many developers don't like to touch other people's code because they fear making mistakes." Manager (platform), case 1.

Problems with developers ◦ Lack of contributions due to ... ▪ boundary conditions not being right due to ... • general work overload • not knowing how to contribute • software being too complex ▪ active psychological resistance due to ... • dislike of debugging someone else's code • dislike of showing incomplete work • fear of making public mistakes • fear of maintenance work • fear of follow-on work ▪ lack of developer benefits	**Problems with product unit managers** ◦ Lack of contributions due to … ▪ boundary conditions not being right due to ... • lack of budget flexibility • lack of greater-good perspective ▪ active psychological resistance due to • lack of willingness to negotiate • fear of appearing unfocused to peers • fear of transparency, opening plans • fear of not meeting performance goals by ◦ loaning out of best developers ◦ losing resources • fear of loss of control
Expected problems with asset quality ◦ Degradation of code base due to ... ▪ uncontrolled contributions ▪ lack of contributor knowledge	**Expected problems with pilot projects** ◦ Cancellation of inner source initiative due to ... ▪ overly ambitious pilot that failed ▪ lack of metrics that show success
Expected problems with processes ◦ Wasted resources due to lack of coordination	**General problems** ◦ General resistance against change due to ... ▪ current insufficient communication ▪ current strong hierarchical organization

Table 5. Experienced or expected problems with inner source adoption

> *"Inner source leads to [public] mistakes, and [some] developers fear mistakes because they lead to reputation loss among colleagues." Manager (platform), case 1.*

> *"Some developers feel intimidated by inner source [development] and do not contribute because they feel they do not know how to do it." Mediator, case 3.*

Quasi-public work, however, is a two-edged sword: What worries some developers inspires others (see "reputation gain" under benefits in the previous subsection).

4.4.1.2 Fear of follow-on and maintenance work

With real or perceived workloads high in all case study companies, anything that suggested more work seemed problematic:

> *"Developers will show passive resistance because they fear inner source will add more work." Manager (platform), case 1.*

> *"Even if a developer has a good idea for an inner source project, they will not talk about it out of fear of having to perform the work themselves." Developer (product unit), case 1.*

> *"If the inner source project is large, developers will avoid contributing out of fear of being sucked in and not being able to leave." Mediator, case 3.*

Specifically, it was assumed that developers might not contribute, because they fear requests for continued or follow-on work, including maintenance work:

> *"Most developers hate maintenance of important components because it makes them responsible for fixing high-priority bugs; this creates too much stress." Developer (product unit), case 1.*

> *"Some developers show passive resistance, because they fear inner source will add more work." Developer (product unit), case 2.*

Worries about unwanted maintenance work were strong.

4.4.2 Problems with Product Unit Managers

For product unit managers, the worry was that they had not enough budget flexibility and typically were focused too much on their own career, that is, lacked a greater-good perspective (for the overall business unit). Active psychological resistance was assumed because of two effects:

- Fear of transparency and loss of control and
- fear of not meeting performance goals.

4.4.2.1 Fear of transparency and loss of control

The idea of exposing all project management artifacts from a road-map down to detailed task lists was frightening to some,

since it suggests exposure to public scrutiny and follow-on critique---a serious problem in highly political organizations.

> *"Most managers dislike showing their planning documents widely; it might open them up for critique." Manager (product unit), case 1.*

Similarly, letting developers participate in inner source projects and not knowing in detail what they were doing suggests loss of control, another unpleasant feeling:

> *"Allowing developers to contribute to inner source projects may feel like losing control to some managers." Developer (platform), case 1.*

4.4.2.2 Fear of not meeting performance goals

Another perception was that by letting developers contribute to inner source, middle managers would lose resources and hence may not be able to meet their performance goals. Thus, some disallowed any such engagement, and when forced, tried to keep their best developers to themselves.

> *"Negotiations between managers to allow their developers to contribute to inner source can be time-consuming." Manager (product unit), case 1.*

> *"Managers may disallow contribution to inner source if they feel their own product is not benefiting enough." Developer (platform), case 1.*

> *"A typical middle manager will try to keep their good developers to themselves and only let their low-performing developers contribute to inner source." Architect (product unit), case 2.*

4.4.3 Summary of Experienced Problems

From a "greater good" perspective, i.e. the efficiency of the overall business unit, inner source makes imminent sense. However, for middle managers of the product units and the developers in the trenches, real problems stand in their way.

As to developers, like in open source, we can assume that some will take to inner source and some won't. Some will want to build a company-wide reputation and further their career, while some will not.

As to middle managers, inner source initiatives are facing a tragedy of the commons problem. As others also observed [71], everyone wants to utilize the platform, but not everyone allows their developers to contribute to inner source projects.

5. DISCUSSION OF FINDINGS

Our case study companies found it difficult to successfully establish inner source projects; this article presents the reasons we found. The key problems are misaligned organizational incentives leading to local rather than global revenue optimization and psychological challenges of the involved middle managers and developers leading to the rejection of open collaborative behavior as necessary for inner source projects.

5.1 Discussion of Problems

The problems break down into organizational, psychological, and process challenges.

5.1.1 Organizational Challenges

We found that making product units profit centers leads middle managers to worry more about reaching their performance goals than overall engineering efficiency. Under such pressure, the relationship to the platform organization becomes more transactional rather than more relational: The platform is supposed to provide software for an agreed-upon specification for appropriate compensation.

However, formalizing the relationship and making it more transactional does not solve the underlying knowledge management problems: Understanding the requirements, implementing them properly, and knowing how to use the results are not capabilities that can be communicated well on paper.

5.1.2 Psychological Challenges

Some middle managers and developers feared the transparency that inner source brought to their projects: They disliked that all across the organization other managers and developers could see project and product artifacts, progress, and quality. Not wanting such openness of their work, they resisted inner source projects.

If we believe practitioner reports from large but comparatively young companies like Google [72] and Facebook [40], these psychological challenges may be a generational issue: Developers who have been exposed to open source during their education may find it easier to engage in open collaborative behavior than software developers to who open source still represents alien behavior. From this perspective, it may simply require time for inner source to establish itself.

5.1.3 Process Breakdown

The organizational and psychological challenges led to a domain engineering process in which most product units only wanted to provide requirements, but not be involved in their implementation. The product units found it hard to gain consensus on requirements, which ultimately led to the platform taking a more active role in requirements definition and prioritization than was appropriate for its position.

The consequence was increased hiring in product units for work that should be performed with and as part of the platform organization's work rather than redundantly in the product units. This represents a suboptimal use of resources that we can only explain with our findings described above, that is, the misalignment of organizational incentives and the psychological challenges faced by middle managers and developers alike.

5.2 Theory Generalization

We shortly review our recommendations to the case study companies and then present the main hypotheses that we see follow from our theories.

5.2.1 Proposed Solution

Our case study companies wanted to know how to overcome their problems with inner source. We believe that these mature organizations need a more structured process than just a well-spirited call to arms to take up inner source. Thus, next to general recommendations like establishing a software forge [52] and getting the overall business unit owner to create proper incentives, we also made more specific suggestions.

First, we suggested to establish a formal inner source incubation process. In this process, every manager or developer can make a suggestion for a new reusable component. All suggestions are public and are discussed publicly. In regular intervals, a council of architects makes a decision as to which suggestion will be turned into a project. The resulting inner source project will be staffed from all affected product units as well as the platform organization.

Furthermore, borrowing from open source foundations [54], we recommended to establish something we called an "inner source foundation", effectively a coaching organization like the Apache Software Foundation. This coaching organization has responsibility for the inner source process, but not the involved human resources. The inner source foundation helps inner source projects get instantiated and coaches them as to proper inner source practices.

5.2.2 Hypotheses and Predictions

Our case study companies have been continuing their efforts. However, it is too early to tell whether our recommendations have been beneficial to them.

The theories we present are only as good as the hypotheses that they generate and that can be validated in future work. Such confirmatory research will also allow for generalized conclusions that are not possible from pure case study research.

H1 Resistance and misunderstandings (like expected lower code quality of inner source components) can be addressed successfully by way of education and active participation in the practice of inner source software development.

This hypothesis is likely to evaluate to true, given the change in public opinion on the use of and participation in open source software projects from a negative to a positive stance.

H2 Psychological openness or resistance to inner source (i.e. desire or fear to work under quasi-public scrutiny) depends on manager and developer personalities and is not a function of organizational structure or process.

Resistance to quasi-public scrutiny had managers and developers holding back from inner source projects, despite apparent advantages. Many expressed that the observed resistance is rooted in the psychological make-up of people and unlikely to be remedied by organizational and process measures alone.

H3 As long as open source does not come natural to an organization, inner source will not come easy to it either. Until this has changed, an organization will need an explicitly governed inner source process.

The call for an explicitly governed process is standard corporate behavior. Such processes may be needed until inner source practices have become a mainstay of corporate culture.

Several managers remarked that building up inner source competence is a step towards open source competence, suggesting that both competencies build on the same base.

H4 Inner source and open source draw on the same competencies of people and a person who is good at one is likely to be good at the other.

Thus, we suggest that open source and inner source competencies are structurally similar, if not isomorphic. This is not surprising given that inner source has originally been motivated by open source. This hypothesized relationship then leads to our most potent hypothesis:

H5 While there is no doubt about the need of platform software and shared reusable assets, a platform development organization may not be needed any longer. It can be replaced by an inner source program.

This is an interesting though probably controversial hypothesis: If large companies can work together in an open source foundation to develop shared infrastructure components, why can't product units within an organization work together to create a platform of shared reusable assets without the need for a dedicated organizational unit that maintains this platform?

This does not deny the need for organizational support of inner source, specifically coaching and the adherence to good inner source governance practices, much like open source foundations coach projects and ensure good open source governance. However, none of the open source foundations that host these industry platforms are actually developing any of it: They are pure coaching, governance, and back-office organizations.

6. RESEARCH LIMITATIONS

This research faces a number of limitations. We first discuss the traditional quality criteria for empirical research and after that focus on specific criteria for exploratory (qualitative, yet empirical) research, followed by a general discussion.

6.1 Empirical Research Quality Criteria

We cannot generalize beyond our case studies and cannot draw statistical inferences. Still, we can discuss the traditional quality criteria of internal and external validity:

6.1.1 Internal validity

As discussed in Section 3, we had a second coder analyze the documents. The second coder derived a code system and linkage close to the original one. The high inter-coder agreement is leading us believe in an adequate quality of the code systems and the theories they represent.

In addition, we were able to draw on a rich and diverse body of materials, as discussed in Section 3. This diversity supported data triangulation in our analysis and increased the internal validity of our findings.

6.1.2 External validity

As also discussed in Section 3, we provided our findings back to the case study participants to learn about possible misunderstandings or omissions. The feedback was reinforcing and validated our analysis.

If there was any conformity bias, it would have been mitigated by the case study participants' desire to receive useful recommendations that actually helped them.

6.1.3 Generalizability

We review our work with respect to other findings and in particular open source to further assess external validity and generalizability.

As to expected benefits of inner source, our findings are well in line with prior work [36] [52] [31] [70] [37] [62], extending it in some places.

It is interesting to compare our work with the reports about inner source in product line engineering at Phillips [68] [69]. The work at Phillips appears to be practitioner reports only; no discernible research method was being applied. Still, we find agreement on the observation that platform engineers are often too far away from customers to define requirements well. We also learn that Phillips has been trying multiple engagement models to get inner source off the ground [71], suggesting they have yet to find a final answer to their inner source efforts.

With respect to the problems experienced when applying inner source, Section 5 provides some hypotheses about the relationship between open source and inner source. Inner source is motivated by open source and reflects its values and practices. Thus, any similarity between our inner source findings and what is known about open source makes those inner source findings more likely to be true:

- Empirical research on open source, surveyed e.g. in [20], finds that some developers simply resist participation in open source for psychological reasons.
- Also, practitioner handbooks, surveyed e.g. in [30], show that contributing to open source is often hindered by managers worrying about losing control over their developers.

These open source experiences reflect what we found happening in inner source at our case study companies, lending increased credibility to these results.

6.2 Exploratory Research Quality Criteria

While empirical, our research is primarily exploratory ("qualitative") in nature. For many years, researchers have argued that the traditional empirical validity criteria do not or should not apply to such research.

Guba and Lincoln, most notably, replace the traditional notion of validity with the concept of trustworthiness, which in turn is based on the more fine-grain concepts of credibility, transferability, dependability, and confirmability [33], all four derived from the traditional quality criteria. Thus, we review our work from this perspective as well.

Our copious notes from interviews and workshops comprise an as complete as possible record of what has been said. For case 1 and 2 we participated as two researchers, one actively engaged (asking questions), one observing and taking notes. We believe we fully captured most relevant events and information. After an interview we had our notes reviewed and corrected, where necessary, by the interviewees resp. participants.

We also evaluated our work by gathering feedback from our case study companies after we finished an analysis (so-called member checking). The responses showed clear agreement with our findings. Since the case studies were performed in sequence, findings from a prior case study informed our choices and foci for the next case study allowing us to address questions previously not addressed.

6.3 Other Generalizability Issues

Our interviewees expected inner source to help solve their engineering problems. One may argue that these problems could be remedied through traditional domain engineering processes as well. We did not question the decision of our case study companies to focus on inner source rather than more traditional practices, which they had been working on for many years already.

We performed three case studies, which may seem low. By case 3, however, interview partners were repeating themselves and others. Beyond these three cases, we have been involved with other inner source initiatives. Those varied on several dimensions that made the three cases of this article homogeneous, so we did not try to include them.

Our cases were chosen to investigate the homogeneous situation of successful mature platform-based product engineering without the problems of globally distributed software development. We do not know what geographical, temporal, and social diversity would do to our findings. Our findings only apply to co-located, culturally and socially homogeneous populations.

7. CONCLUSIONS

This article presents an analysis of three mature platform-based product engineering efforts. The respective business units expected that inner source, the cross-organizational-unit collaboration on software projects based on open source best practices, would improve productivity. Our analysis presents the problems these companies faced, the expectations they had, and the problems they experienced in the adoption of inner source.

We find that setting up product units as profit centers and platform organizations as cost centers leads to under-staffing platform organizations and hinders collaboration and knowledge sharing across organizational units. We also find that inner source benefits are most obvious to the overall business unit, while middle managers of product units and developers can be reluctant to contribute to inner source projects. To that end we make recommendations as to overcome this reluctance.

Finally, we draw conclusions from our theories and present hypotheses that will structure future research work.

Acknowledgments

We would like to thank Ann Barcomb, Christoph Elsner, Andreas Kaufmann, Daniel Lohmann, Klaus-Benedikt Schultis, Klaas-Jan Stol and the anonymous reviewers for feedback that helped us improve this article.

References

[1] Altintas, N. I., & Cetin, S. (2008). Managing large scale reuse across multiple software product lines. In High Confidence Software Reuse in Large Systems (pp. 166-177). Springer Berlin Heidelberg.

[2] Atkinson, C. (2002). Component-based product line engineering with UML. Pearson Education.

[3] Batory, D., Johnson, C., MacDonald, B., & Von Heeder, D. (2002). Achieving extensibility through product lines and domain-specific languages: A case study. ACM Transactions on Software Engineering and Methodology (TOSEM), 11(2), 191-214.

[4] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., ... & DeBaud, J. M. (1999, May). PuLSE: a methodology to develop software product lines. In Proceedings of the 1999 symposium on Software reusability (pp. 122-131). ACM.

[5] Bosch, J. (2000). Design and use of software architectures: adopting and evolving a product line approach. Pearson Education.

[6] Bosch, J. (2006). The challenges of broadening the scope of software product families. Communications of the ACM, 49(12), 41-44.

[7] Bosch, J. (2006). Expanding the scope of software product families: Problems and alternative approaches. Lecture Notes in Computer Science, 4034, 4.

[8] Bosch, J., & Bosch-Sijtsema, P. (2010). From integration to composition: On the impact of software product lines, global development and ecosystems. Journal of Systems and Software, 83(1), 67-76.

[9] Bourgeois III, L. J., & Eisenhardt, K. M. (1988). Strategic decision processes in high velocity environments: four cases in the microcomputer industry. Management science, 34(7), 816-835.

[10] Berger, T., Nair, D., Rublack, R., Atlee, J. M., Czarnecki, K., & Wąsowski, A. (2014). Three cases of feature-based variability modeling in industry. In *Model-Driven Engineering Languages and Systems* (pp. 302-319). Springer International Publishing.

[11] Berger, T., Pfeiffer, R. H., Tartler, R., Dienst, S., Czarnecki, K., Wąsowski, A., & She, S. (2014). Variability mechanisms in software ecosystems. *Information and Software Technology*, *56*(11), 1520-1535.

[12] Buhne, S., Lauenroth, K., & Pohl, K. (2005, August). Modelling requirements variability across product lines. In Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on (pp. 41-50). IEEE.

[13] Capra, E., & Wasserman, A. I. (2008). A framework for evaluating managerial styles in open source projects. In *Open Source Development, Communities and Quality* (pp. 1-14). Springer US.

[14] Cavaye, A. L. (1996). Case study research: a multi-faceted research approach for IS. *Information systems journal*, *6*(3), 227-242.

[15] Charmaz, K. (2014). Constructing grounded theory. Sage.

[16] Chastek, G., & McGregor, J. D. (2002). Guidelines for developing a product line production plan (No. CMU/SEI-2002-TR-006). Carnegie Mellon University, Software Engineering Institute.

[17] Chastek, G., Donohoe, P., & McGregor, J. D. (2004). A study of product production in software product lines (No. CMU/SEI-2004-TN-012). Carnegie Mellon University, Software Engineering Institute.

[18] Clements, P., & Northrop, L. (2002). Software product lines: practices and patterns.

[19] Corbin, J., & Strauss, A. (2014). Basics of qualitative research: Techniques and procedures for developing grounded theory. Sage publications.

[20] Crowston, K., Wei, K., Howison, J., & Wiggins, A. (2012). Free/Libre open-source software development: What we know and what we do not know. *ACM Computing Surveys (CSUR)*, *44*(2), 7.

[21] Czarnecki, K., & Eisenecker, U. W. (2000). Generative programming: Methods, tools, and applications. Addison-Wesley.

[22] Czarnecki, K., Helsen, S., & Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization. *Software process: Improvement and practice*, *10*(1), 7-29.

[23] Darke, P., Shanks, G., & Broadbent, M. (1998). Successfully completing case study research: combining rigour, relevance and pragmatism. *Information systems journal*, *8*(4), 273-289.

[24] Deelstra, S., Sinnema, M., & Bosch, J. (2005). Product derivation in software product families: a case study. Journal of Systems and Software, 74(2), 173-194.

[25] Dhungana, D., Grünbacher, P., Rabiser, R., & Neumayer, T. (2010). Structuring the modeling space and supporting evolution in software product line engineering. Journal of Systems and Software, 83(7), 1108-1122.

[26] Dinkelacker, J., Garg, P. K., Miller, R., & Nelson, D. (2002, May). Progressive open source. In Proceedings of the 24th International Conference on Software Engineering (pp. 177-184). ACM.

[27] Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering* (pp. 285-311). Springer London.

[28] Eisenhardt, K. M. (1989). Building theories from case study research. Academy of management review, 14(4), 532-550.

[29] Eisenhardt, K. M., & Graebner, M. E. (2007). Theory building from cases: opportunities and challenges. *Academy of management journal*, *50*(1), 25-32.

[30] Fogel, K. (2005). *Producing open source software: How to run a successful free software project*. O'Reilly Media, Inc.

[31] Gaughan, G., Fitzgerald, B., & Shaikh, M. (2009, August). An examination of the use of open source software processes as a global software development solution for commercial software engineering. In Software Engineering and Advanced Applications, 2009. SEAA'09. 35th Euromicro Conference on (pp. 20-27). IEEE.

[32] Glaser, B. G. (1965). The constant comparative method of qualitative analysis. Social problems, 436-445.

[33] Guba, E. G., & Lincoln, Y. S. (1989). *Fourth generation evaluation*. Sage.

[34] Guest, G., Bunce, A., & Johnson, L. (2006). How many interviews are enough? An experiment with data saturation and variability. *Field methods*, *18*(1), 59-82.

[35] Guion, L. A., Diehl, D. C., & McDonald, D. (2011). Triangulation: Establishing the validity of qualitative studies.

[36] Gurbani, V. K., Garvert, A., & Herbsleb, J. D. (2006, May). A case study of a corporate open source development model. In Proceedings of the 28th international conference on Software engineering (pp. 472-481). ACM.

[37] Gurbani, V. K., Garvert, A., & Herbsleb, J. D. (2010). Managing a corporate open source software asset. Communications of the ACM, 53(2), 155-159.

[38] Ishikawa, K. (1990). Introduction to quality control. Productivity Press.

[39] Jepsen, H. P., Dall, J. G., & Beuche, D. (2007, September). Minimally invasive migration to software product lines. In *Software Product Line Conference, 2007. SPLC 2007. 11th International* (pp. 203-211). IEEE.

[40] Keyani, P. (2008). The All-Night Hackathon Is Back! Retrieved March 12, 2015, from https://code.facebook.com/posts/573666012669084/the-all-night-hackathon-is-back-/

[41] Lindman, J., Rossi, M., & Marttiin, P. (2008). Applying open source development practices inside a company. In Open Source Development, Communities and Quality (pp. 381-387). Springer US.

[42] Lindman, J., Rossi, M., & Marttiin, P. (2010). Open Source Technology Changes Intra-Organizational Systems Development-A Tale of Two Companies. In ECIS.

[43] Lindman, J., Riepula, M., Rossi, M., & Marttiin, P. (2013). Open Source Technology in Intra-Organisational Software Development—Private Markets or Local Libraries. In Managing Open Innovation Technologies (pp. 107-121). Springer Berlin Heidelberg.

[44] Lombard, M., Snyder-Duch, J., & Bracken, C. C. (2002). Content analysis in mass communication: Assessment and reporting of intercoder reliability. *Human communication research*, *28*(4), 587-604.

[45] Martin, K., & Hoffman, B. (2007). An open source approach to developing software in a small organization. Ieee Software, (1), 46-53.

[46] Melian, C., & Mähring, M. (2008). Lost and gained in translation: Adoption of open source software development at Hewlett-Packard. In Open Source Development, Communities and Quality (pp. 93-104). Springer US.

[47] Melian, C., Ammirati, C. B., Garg, P., & Sevon, G. (2002). Building Networks of Software Communities in a Large Corporation. Technical Report. Hewlett Packard.

[48] Neus, A., & Scherf, P. (2005). Opening minds: Cultural change with the introduction of open-source collaboration methods. IBM Systems Journal, 44(2), 215-225.

[49] Passos, L., Czarnecki, K., Apel, S., Wąsowski, A., Kästner, C., & Guo, J. (2013, January). Feature-oriented software evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems* (p. 17). ACM.

[50] Pohl, K., Böckle, G., & van der Linden, F. J. (2005). Software product line engineering: foundations, principles and techniques. Springer Science & Business Media.

[51] Raymond, E. (1999). The cathedral and the bazaar. Knowledge, Technology & Policy, 12(3), 23-49.

[52] Riehle, D., Ellenberger, J., Menahem, T., Mikhailovski, B., Natchetoi, Y., Naveh, B., & Odenwald, T. (2009). Open collaboration within corporations using software forges. Software, IEEE, 26(2), 52-58.

[53] Riehle, D. (2010). The economic case for Open Source foundations. *Computer*, *1*(43), 86-90.

[54] Riehle, D., & Kips, D. (2012, May). Geplanter Inner Source: Ein Weg zur Profit-Center-übergreifenden Wiederverwendung. Technical Report CS-2012-05. Computer Science Department, Friedrich-Alexander-University Erlangen-Nürnberg.

[55] Sánchez, A. B., Segura, S., & Ruiz-Cortés, A. (2014, January). The Drupal framework: a case study to evaluate variability testing techniques. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems* (p. 11). ACM.

[56] Schmid, K. (2002, May). A comprehensive product line scoping approach and its validation. In Proceedings of the 24th International Conference on Software Engineering (pp. 593-603). ACM.

[57] Schwanninger, C., Groher, I., Elsner, C., & Lehofer, M. (2009). Variability modelling throughout the product line lifecycle. In Model Driven Engineering Languages and Systems (pp. 685-689). Springer.

[58] Sharma, S., Sugumaran, V., & Rajagopalan, B. (2002). A framework for creating hybrid-open source software communities. Information Systems Journal, 12(1), 7-25.

[59] She, S., Lotufo, R., Berger, T., Wasowski, A., & Czarnecki, K. (2010). The Variability Model of The Linux Kernel. *VaMoS*, *10*, 45-51.

[60] Sinnema, M., & Deelstra, S. (2007). Classifying variability modeling techniques. Information and Software Technology, 49(7), 717-739.

[61] Smith, P., & Garber-Brown, C. (2007, August). Traveling the open road: Using open source practices to transform our organization. In Agile Conference (AGILE), 2007 (pp. 156-161). IEEE.

[62] Stol, K. J., Babar, M. A., Avgeriou, P., & Fitzgerald, B. (2011). A comparative study of challenges in integrating Open Source Software and Inner Source Software. Information and Software Technology, 53(12), 1319-1336.

[63] Stol, K. J., Avgeriou, P., Babar, M. A., Lucas, Y., & Fitzgerald, B. (2014). Key factors for adopting inner source. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(2), 18.

[64] Torkar, R., Minoves, P., & Garrigós, J. (2011). Adopting free, libre, open source software practices, techniques and methods for industrial use. Journal of the Association for Information Systems, 12(1), 88-122.

[65] van der Linden, F. J., Schmid, K., & Rommes, E. (2007). Software product lines in action: the best industrial practice in product line engineering. Springer Science & Business Media.

[66] van der Linden, F. (2009). Applying open source software principles in product lines. Upgrade, 10, 32-41.

[67] van der Linden, F. (2009). Inner source product line development. In Proceedings of the 13th International Software Product Line Conference (p. 317). ACM.

[68] van der Linden, F., Lundell, B., & Marttiin, P. (2009). Commodification of industrial software: A case for open source. Software, IEEE, 26(4), 77-83.

[69] van der Linden, F. (2013). Open source practices in software product line engineering. In Software Engineering (pp. 216-235). Springer.

[70] Vitharana, P., King, J., & Chapman, H. S. (2010). Impact of internal open source development on reuse: participatory reuse in action. Journal of Management Information Systems, 27(2), 277-304.

[71] Wesselius, J. (2008). The bazaar inside the cathedral: business models for internal markets. Software, IEEE, 25(3), 60-66.

[72] Whittaker, J. A., Arbon, J., & Carollo, J. (2012). How Google tests software. Addison-Wesley.

[73] Yin, R. K. (2013). Case study research: Design and methods. Sage publications.