FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

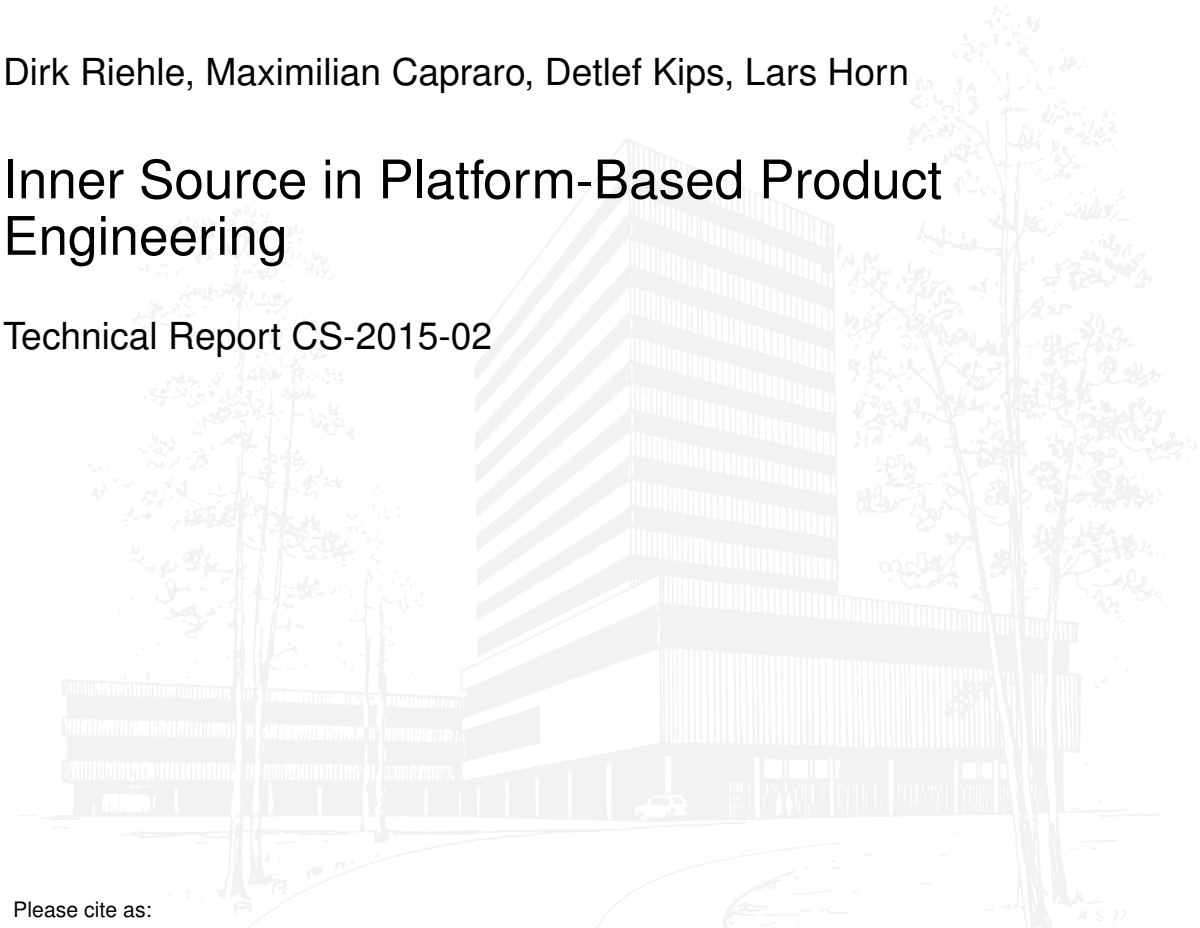
Department Informatik

Technical Reports / ISSN 2191-5008

Dirk Riehle, Maximilian Capraro, Detlef Kips, Lars Horn

Inner Source in Platform-Based Product Engineering

Technical Report CS-2015-02



Please cite as:
Dirk Riehle, Maximilian Capraro, Detlef Kips, Lars Horn, “Inner Source in Platform-Based Product Engineering,”
Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Technical Reports, CS-2015-02, .

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

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 platform-based product 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 or expecting 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 application of open source best practices to firm-internal software development [22]. Thus, inner source is an approach to collaboration based on the *open collaboration principles* of egalitarian, meritocratic, and self-organizing work [47]. *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 process to the needs at hand rather than strictly following a predefined process [38].

In inner source, no open source software is being developed, but open source practices are being adopted for internal software development processes. This article focuses on software development within companies across intra-organizational boundaries, most notably profit center boundaries, that would otherwise hinder such collaboration. In a nutshell, inner source is supposed to enable collaboration across development silos.

Dinkelacker et al. [22] 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. [27] [28] 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. [54] of IBM suggest improved reuse through inner source. Our experience is that inner source can improve access to resources, software quality and development speed, among other things [38].

Over the last five years, we have helped several software development organizations understand and adopt inner source. Most 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 multiple-case 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 [36]. We define *platform-based product engineering* to be the engineering of software products utilizing a share common platform. *Product line engineering* [16] is a special but important case of platform-based product engineering.

We find that the three case study companies expect inner source to help them overcome problems with lack of resources, lack of pertinent skills, and unclear requirements, among others, in the context of platform-based product engineering. Yet they had problems putting inner source into practice.

This article answers the following research questions:

- *RQ1: What are current problems in platform-based product engineering?*
- *RQ2: What benefits do organizations expect from adopting inner source?*
- *RQ3: What are experienced or expected problems in adopting inner source?*

The research method employed is *multiple-case case study research* [59] [23] [17] [10]. Data gathering was performed using workshops, formal interviews, and materials review. The process was incremental with learnings being provided back to the case study participants to receive validating feedback as to the theories being built.

The contributions of this paper are the following:

- *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.*

- 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 [39] [12], we find that for the studied organizations, inner source can and should move to a governed process beyond the definition given in the beginning of this section.

The paper is structured as follows. Section 2 provides a review of inner source as well as platform-based product 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 summarizes our proposed solution. 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 term inner source was coined by Dinkelacker et al. [22] in 2002. A slow stream of case studies and examples has been reported about since then [27] [54] [38] [28] [24] [46] [50] [52]. Product line engineering [36] [16] is an active area of research and development that has yet to be integrated with inner source development.

2.1 Inner Source Software Development

Dinkelacker et al. [22] describe Hewlett Packard's (HP) progressive open source strategy for using open source practices and methods. Progressive open source presents three different models of increasing openness: Inner source (within company), controlled source (with selected external partners) and open source (with the world). An inner source project establishes a project community accessible to anyone within the company. Melian and Mähring [34] report how an inner source project can evolve into a controlled source project where the project community includes selected other companies. Finally, a controlled source project can evolve into an open source project. Weiss also discusses collaboration between companies [55]. We use the terms inner source, partner source, and open source, respectively, to denote increasing degrees of openness in collaboration with partners.

Different researchers have used different terms to describe inner source and related phenomena. Other terms are hybrid open source [44], corporate open source [28], and firm-internal open source [38]. The term inner source has prevailed and most publications including the most recent ones are using it [53] [32] [47].

Most work defines inner source similar to [22] as the use of open source practices within the boundaries of one company. Van der Linden [50] does not differentiate among different scopes (as Dinkelacker et al. do with progressive open source) and also labels open source methods within consortia as inner source. He highlights using open source software engineering tools for collaborative and distributed development as an important characteristic of inner source software development.

The given definitions are broad and not precise. Stol et al. [50] note that due to these imprecisions inner source should be defined as "a development philosophy oriented around the open collaboration principles of egalitarianism, meritocracy, and self-organization" rather than a well-defined methodology.

Multiple organizations have tried to apply an inner source approach to aspects of their software development efforts. Motivation and actual benefits reported include higher software quality, independence of organizational units to each other, a generally more economical development, better knowledge, information and asset transfer as well as an enhanced utilization of available developer resources. Inner source helps to focus on company-wide success instead of local optimizations. Gaughan et al. [24] provide a list of motivations for inner source as well as a list of six benefits they observed.

Inner source adoption—the transition from an established development method to inner source—is not trivial. A lack of a company-wide perspective, misalignment with corporate culture, problems in dealing with a massive existing code portfolio, and resentments of middle management were reported [47].

Stol et al. [46] describe that inner source adoption cannot per se be supported by experiences or models from the open source world. They describe how, for example, Raymond's [37] preconditions for bazaar style software development do not necessarily apply to inner source. Sharma et al. [44] and Stol et al. [47] present models of inner source adoption.

At the time of Dinkelacker et al.'s publication, Sharma et al. developed a framework for the adoption of open source practices within organizations. They refer to the phenomenon as hybrid open source software communities. The framework describes multiple activities for community building, community governance and community infrastructure. For building a community, Sharma et al. suggest to establish electronic communication, an informal network among employees, a free flow of information as well as tailored community practices.

Also, Stol et al. present a framework of inner source adoption consisting of nine key adoption factors classified by the themes organization and community, practices and tools, and software product. While Sharma et al.'s framework is broader and more general, Stol et al.'s adoption factors are more specific and suggest actions like a standardized tool chain or a modular software design for inner source components.

Neither Stol et al. nor Sharma et al. explicitly consider the specifics of using inner source for platform-based product engineering, even though one of Stol et al.'s examples is about a product line engineering situation. Schultis et al. [42] studied collaboration in firm-internal software ecosystems using the example of two product lines at Siemens. They conclude that opening platform code can benefit collaboration but is more difficult to manage with increasing decoupling of organizational units. They do not present dimensions of openness.

Van der Linden [50] describes the use of inner source within the context of Philips' product line engineering. He observed an enhanced involvement of application engineers with domain engineering, which resulted in eased transfer of common application assets to the general platform. This finally lead to a reduced time to market. Also, Stol et al. [46] performed a case study of inner source in the context of product line engineering at a non disclosed organization and inferred a model of adoption challenges. Both, van der Linden and Stol et al. do not discuss specifics of the product-line engineering context to the extent that this paper does.

Van der Linden [50] describes the use of inner source within the context of Philip's product line engineering. He observed an enhanced involvement of application engineers with domain engineering, which resulted in eased transfer of common application assets to the general platform. This finally lead to a reduced time to market.

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) [6] [36], all three on top of a single shared platform that offers a large number of shared reusable assets.

The distinctions between product group, product family, and product line can be blurry. We define *product group* to be a set of products built on top of a shared platform, but with little else in common. A *product family* is a set of products with "a shared architecture on top of a shared platform" [57]. To the product family definition we add that the products in the family each focus on different business features and domains. While the terms product group and product family are in general use, most research on platform-based product engineering is being undertaken in the context of product line engineering, making these terms synonymous for some.

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" [16]. 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')" [43]. This second definition focuses more on the processes rather than the artifacts and is more in line with the work presented in this article.

A significant part of research into product line engineering is about tools and methods for managing product variability, for example, [41] [45] [19] [3] [49]. In contrast, we focus on the processes leading to the specification, development, and use of shared components.

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 build 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 [36] [16].

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 [51]. Like [50], this is a practitioner report. No research is known to us that specifically combines inner source with platform-based product engineering in general and software product line engineering in particular.

3. RESEARCH SET-UP

We take a multiple-case case-study-research approach. As an exploratory research method, case study research is a natural choice for dealing with not well-understood phenomena for which no established theories exist. The cases were chosen because they share key properties that answer our research questions, but otherwise are independent and distinct from each other. All case study products and platforms are mature and have established platform engineering practices in place. This allows us to draw cross-case conclusions and strengthen the breadth of our theory [10]. We follow Yin in setting up our multiple-case case study research [59].

3.1 Research Questions

We want to understand how software development organizations can improve their productivity using inner source. In this article, we answer the following three research questions:

- *RQ1: What are current problems in platform-based product engineering?*
- *RQ2: What benefits do organizations expect from adopting inner source?*
- *RQ3: What are experienced or expected problems in adopting inner source?*

We choose a multiple-case case-study-research design. In this design, we study three independent cases, but the three cases all share a set of key properties that makes them comparable to draw cross-case conclusions.

3.2 Case Study Set-up

The three cases stem from three independent, large and diversified, internationally operating companies with significant software development efforts.

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

Thus, we have the case of a group of products (case 1), a product family (case 2), and a product line (case 3). In the following, we use the term product group as a common name for products based on a shared platform, understanding that product families and product lines are more specific than a product

		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 product groups and their owning companies

group. We will focus on the specifics where necessary. All three cases share the situation of platform-based product engineering.

Table 1 shows key properties of the product groups and their owning company.

The case study product groups, their platforms, and the companies were chosen to avoid unrelated problems of globally distributed software development. The product groups of case 1 and case 3 are all being developed in the same location, and the developer populations are socially and culturally homogeneous. We thought that this also holds true for case 2, until we learned that they are collaborating with a remote party. This information surfaced only after data gathering and when asked about it, our case study partners confirmed that they thought this information was not relevant for the case.

These three platform-based product engineering efforts all have the same organizational set-up, allowing us to draw cross-case conclusions:

- The whole product group including the supporting platform is owned by a single *business unit* with a single overall *business owner* responsible for the product group.
- 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.
- 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.

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

3.3 Research Process

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. In all cases we received additional collateral in the form of product information, internal memos, software documentation, and organizational documentation, see Table 3.

Case 1 was the first case we took on. Taking a theoretical sampling approach, we refined our questions and perspectives in the first set of interviews together with a representative of the

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

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. Finally, 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 [13] [17].

Our analysis was performed using *MaxQDA*, a qualitative data analysis tool, which the first two authors used to construct two independent code systems.

A *code system* is a hierarchy of *codes* (data annotations), where codes are grouped into *concepts* and concepts are grouped into more abstract concepts and eventually so-called *core categories*, all of which can also be codes. In addition to the hierarchical relationship, concepts are linked by *memos*. The result is a model, or theory, structured from the most abstract (core categories) to the most specific (open codes linked to materials like stakeholder interview statements).

The two independently developed code systems showed a high degree of intercoder agreement [33], documenting a high rigor of our analysis and strengthening our theory's reliability.

In creating the code system, we followed a qualitative data analysis process of open, axial, and selective coding [13] [17]. We used the *constant comparative method* to keep the code system integrated [25]. Figure 1 depicts a screenshot of an intermediate state of the coding and analysis process. We used *data triangulation*, based on the broad array of materials (beyond interviews) made available to us, to increase the expected validity of our theories [26].

The resulting code system is a concept hierarchy, with links between concepts encoded either in the hierarchy or in memos for the codes. The categories, concepts, and their memos and linkage provide the backbone of the resulting theories.

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 it [17] [13]. This grounding in the data enhances the trustworthiness of the presented theories. Future work is to generate hypotheses and validate the theories by evaluating these hypotheses, that is, to follow up on this theory generation research with theory confirmation research [1] [60]. The quality of the theories is ensured by having followed the methods properly.

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 the experienced or expected adoption problems when doing so.

4.1 Presentation of Research Results

This section presents some of the theoretical results using *cause-and-effect diagrams* [29]. We found that this type of diagram brings out the theories better than a traditional generic top-down concepts and relationships discussion.

In our diagrams, a rectangle represents a concept. A concept has a name. The numbers in the box indicate the case in which they were mentioned. A category is represented as a gray background rectangle. 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 para-

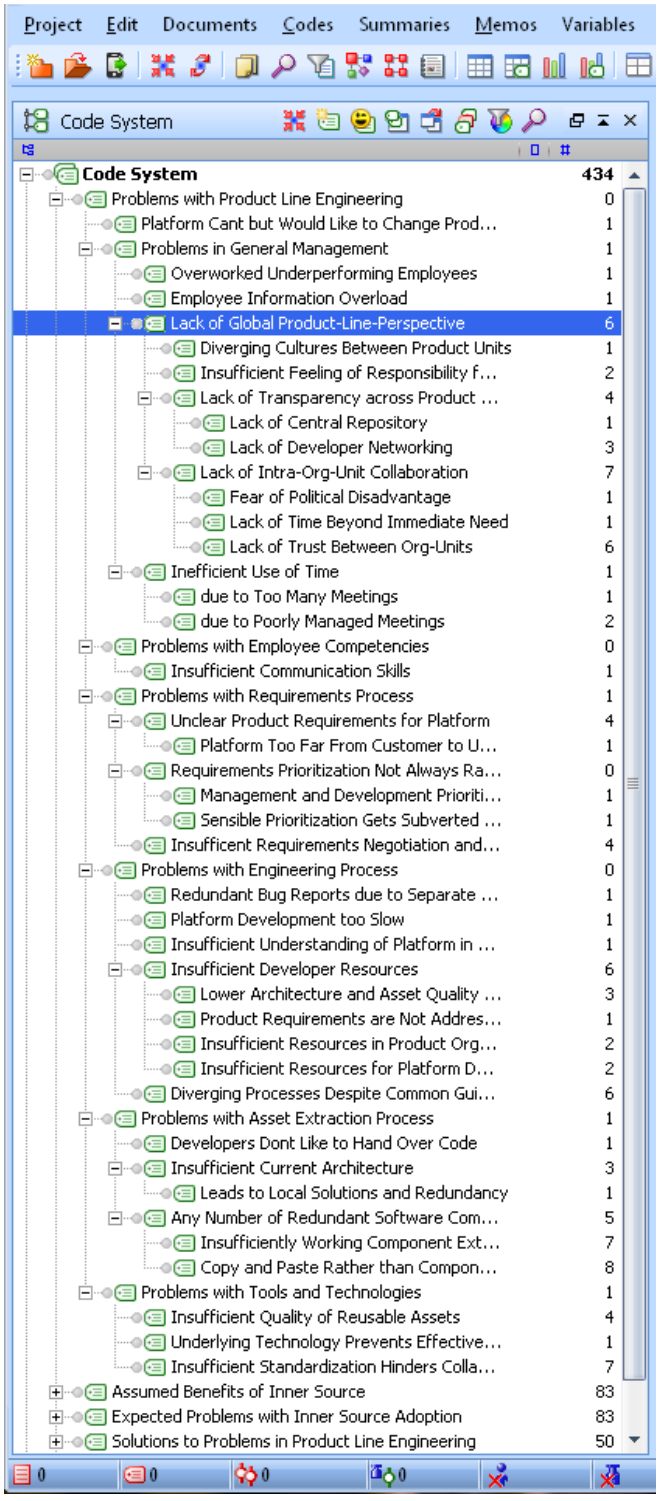


Figure 1. An intermediate snapshot of the code system

phrase what we heard. In addition, we changed the terminology from company-specific terms to generic terminology.

4.2 Problems in Platform-based Product Engineering

Figures 2 and 3 present cause-and-effect diagrams of problems and their effects on the product engineering efforts of our case study companies. 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.

We use a terminology where 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 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.

For readability purposes, the cause-and-effect chain has been split up into two figures. Figure 2 shows problems with organizational structure, and Figure 3 shows how these problems affect the domain engineering process.

As Figure 2 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

- a “lack of global business unit perspective” where each product unit acts in their own interests irrespective of possibly synergies from collaborating,
- “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 or will to talk with 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 managing their own revenue are always in a stronger position to hire developers than any cost center.

Figures 2 and 3 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—Delayed reusable asset realization—Delayed product delivery.
2. Product unit power play—Poorly prioritized requirements—Delayed product delivery.
3. Insufficient collaboration—Limited understanding—Increased defect rate.

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

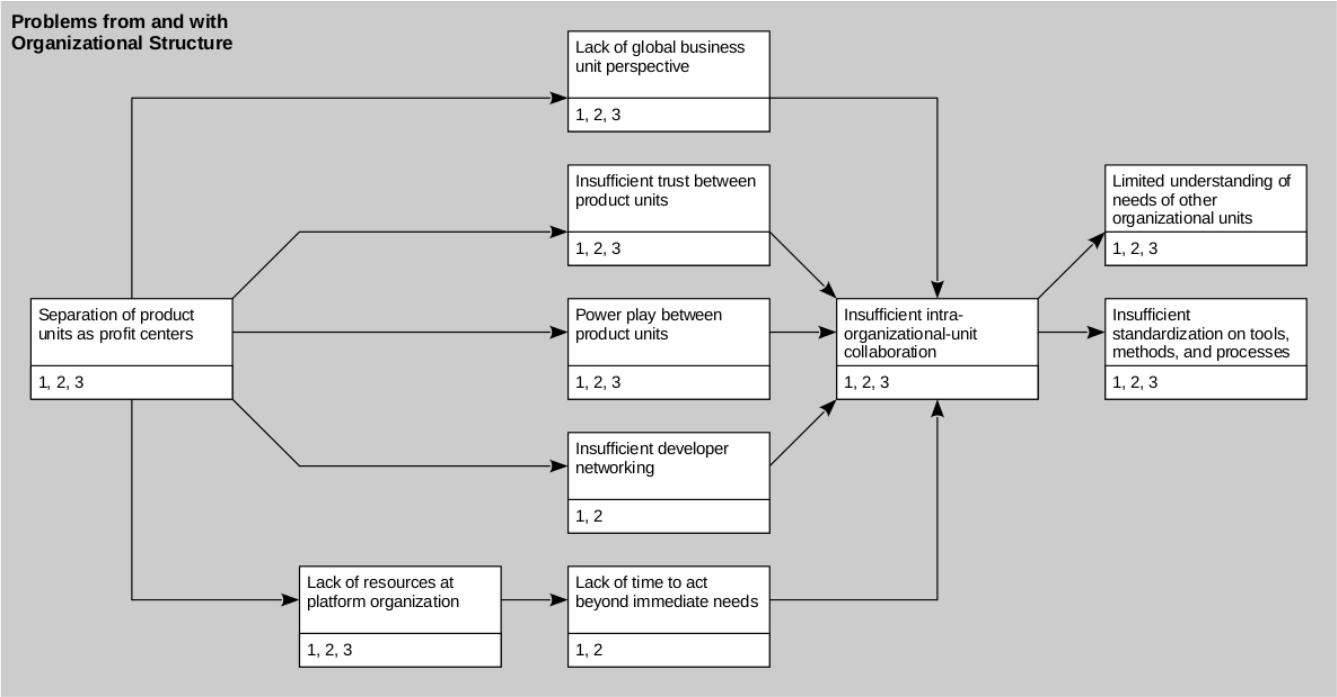


Figure 2. Problems in our case study companies resulting from their organizational structure

4.2.1 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.*

> *"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 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.

4.2.2 Poorly prioritized requirements

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 were not involved in a sufficiently structured way to help prioritize requirements for reusable assets. This is because product units cannot easily agree on the right priority of a feature and hence leave it to the platform organization to prioritize. 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 stale-mate between different 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

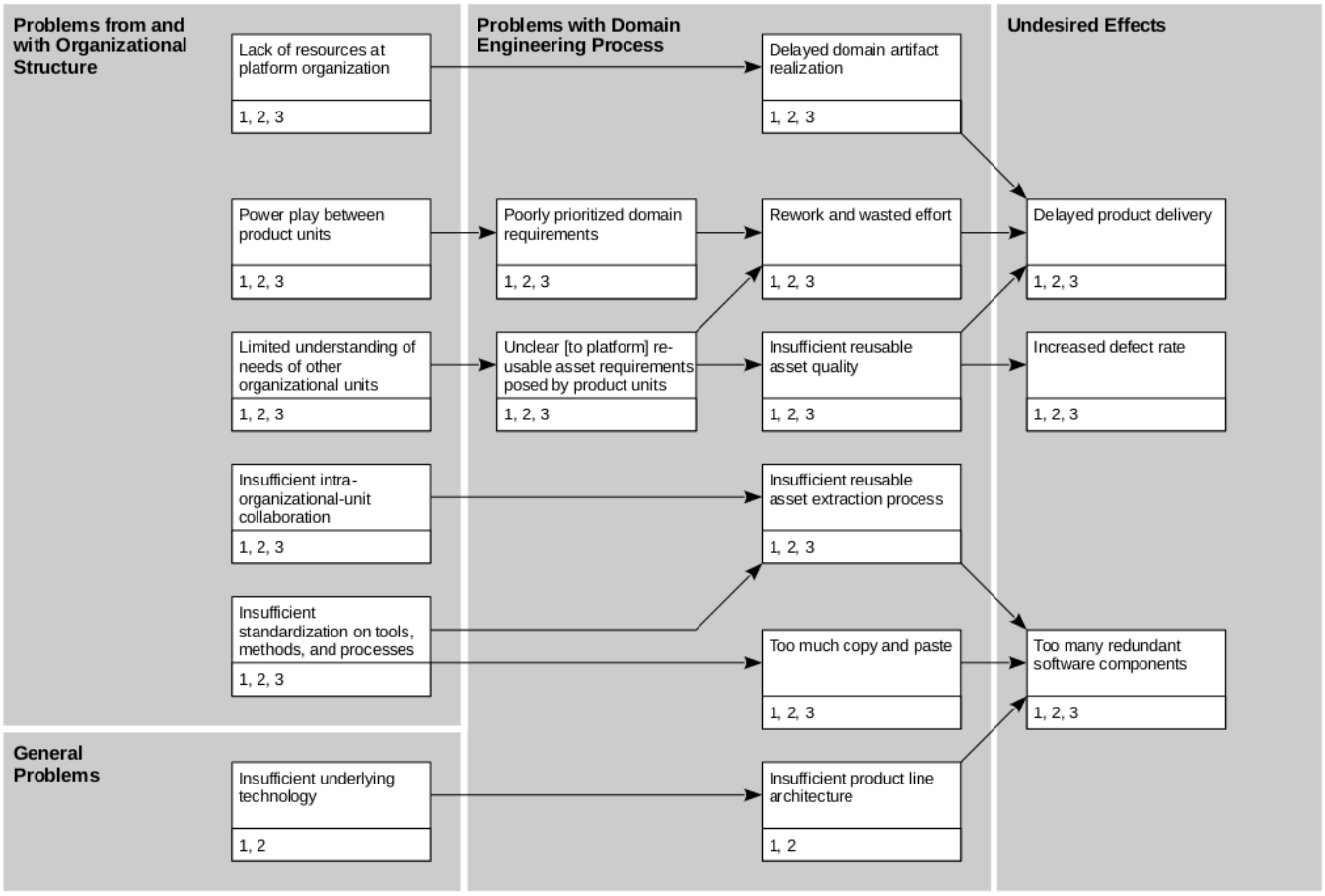


Figure 3. Resulting process and artifact problems in case study companies

their customers is important to prioritize these domain requirements right.

4.2.3 Insufficient collaboration

The lack of sufficient collaboration between product units and between product units and the platform organization leads to a 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 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") [38], 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. The key idea is that managers and developers from product units work jointly with each other and with managers and developers from the platform organization not only in defining, but also in developing reusable assets.

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 or expecting 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 Figure 4 shows their cause-and-effect relationships as derived from the concept linkage.

We separate general benefits that accrue to everyone from the benefits that accrue only to specific units of analysis, be it the overall business unit, the product units, or the platform or-

ganization. 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 platform-based product engineering problems 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 in Platform-based Product Engineering

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.*

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

When discussing the problem of 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 of Software Development

Interview partners not only discussed how they expect inner source to help address the problems we identified in the previous section, 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 across organizational unit boundaries and that this leads to 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 typically were responsible for new reusable asset definition and implementation.

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 [37]
      - 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 case study product engineering

More generally, 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 or Expected 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 taken steps towards adopting inner source following the generic model laid out in the literature without any concern for the specifics of platform-based product engineering. While company 1 was still planning its pilot, companies 2 and 3 were already actively pursuing inner source.

In this subsection we present the problems they were either already experiencing or still expecting.

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.

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

General concerns ranged from worries about "degradation of code base due to uncontrolled contributions" to "general resistance to change". These worries are either easy to mitigate (at least on a rational level: Why would anyone allow for uncontrolled contributions?) or out of scope for this research (general resistance to change). In the following, we focus on the specific

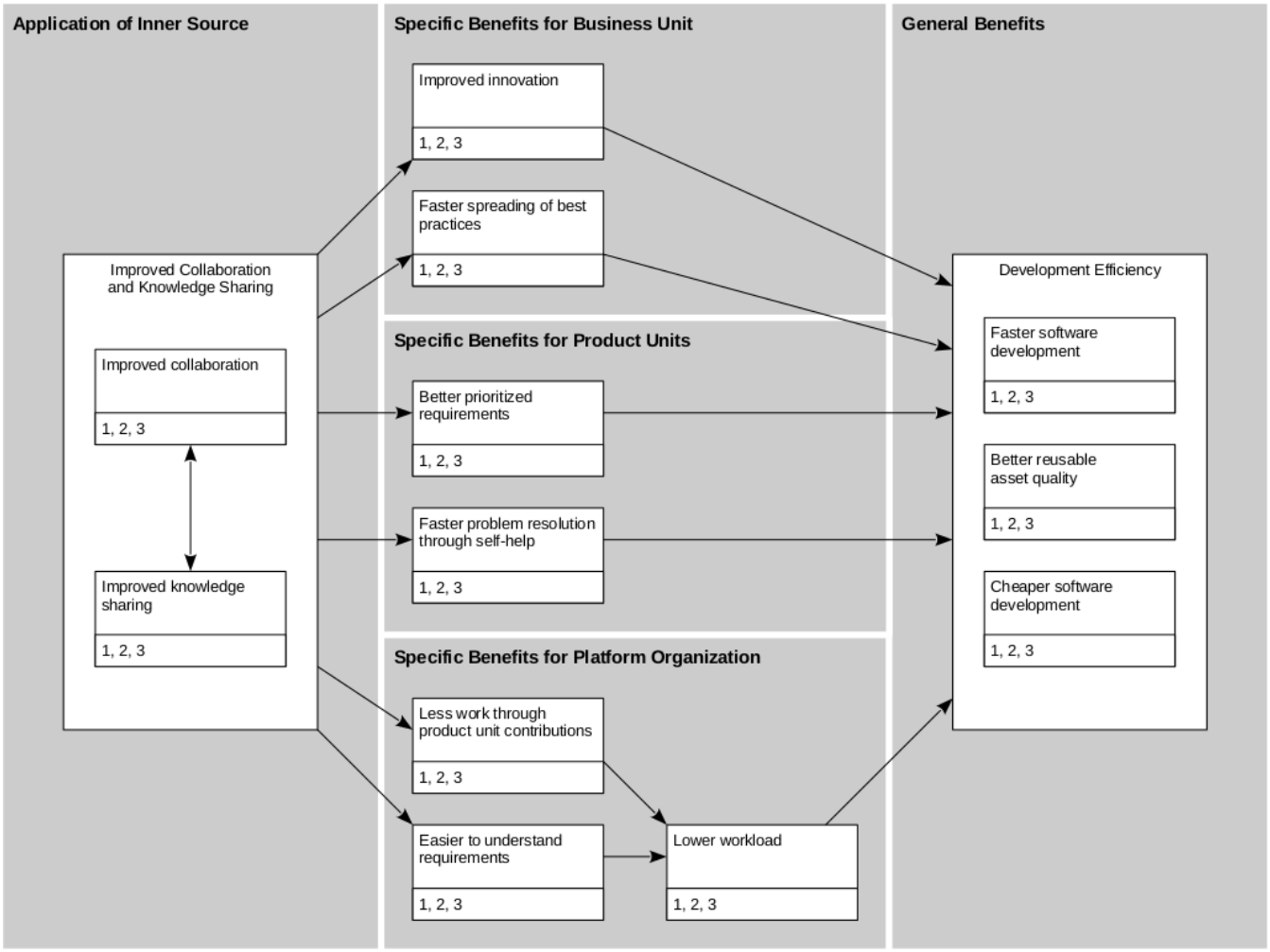


Figure 4. Key benefits expected from applying inner source to platform-based product engineering

problems experienced or expected by the key involved people, developers in general and product unit managers in particular.

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".

4.4.1 Problems with Developers

For developers, the worry was that they were generally too overloaded to contribute or wouldn't 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 or were expected to show. 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 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.*

> *"Inner source leads to mistakes, and 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 perceived or real workloads already 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 continued maintenance work were particularly strong.

4.4.2 Problems with Product Unit Managers

For product unit managers, the worry was that they would not have 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 suspected 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 are 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 or Expected Problems

From a "greater good" perspective, 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 would like to build a company-wide reputation and further their career, while some would not.

As to middle managers, inner source initiatives are facing a tragedy of the commons problem. As others also observed [56], everyone wants to utilize the platform, but not everyone wants to contribute by letting developers work on inner source projects.

5. DISCUSSION OF FINDINGS

Our case study companies found it difficult to 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 Organizational Challenges

We found that making product units profit centers leads to selfish behavior of middle managers worried about reaching their performance goals. 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. Some middle managers understood this, but they were not in the majority.

5.2 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 their artifacts and how they were doing. Not wanting this, they resisted or were expected to resist inner source projects.

If we believe practitioner reports from large but comparatively young companies like Google and Facebook [58] [31], 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.

5.3 Process Breakdown

The organizational and psychological challenges led to a domain engineering process in which most product units only

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

wanted to provide requirements but not be involved in their implementation. Given the consistent difficulties of such a hard separation, this behavior is surprising: Piling on more of what does not work is unlikely to help.

The consequence is 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.

5.4 Proposed Solution

Our case study companies wanted to know how to overcome these problems. We realized that these mature organizations needed 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 [38] and getting the overall business unit owner to create proper incentives, we also made several 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 [40], 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.

5.5 Inner Source Governance

Our case study companies are now continuing their inner source efforts and it is too early to tell whether our recommendations have been beneficial to them. Our hypothesis remains that as long as open source does not come natural to these organizations, inner source will not come easy to them either. Until this has changed, we expect that organizations will need an explicitly governed process like the one we suggested to them.

6. RESEARCH LIMITATIONS

This research faces a number of possible limitations.

- ***Interpretation.*** We validated our work by gathering feedback from our case study companies after we finished the analysis. The responses showed clear agreement with our findings. However, such feedback may naturally suffer from conformity bias. To alleviate some of this, a second coder analyzed the available data and we achieved broad inter-coder agreement on the analysis results.

- ***Linkage.*** Our interviewees suggested that inner source will help solve their problems in platform-based product engineering. 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 traditional engineering practices, which they had been working on for many years already.

- ***Completeness.*** We reached theoretical saturation after case 3, which may seem low. By then, interview partners were just 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 didn't try to include them.

- ***Scope.*** 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 companies behind the product groups expect that inner source, the cross-organizational unit collaboration on software projects based on open source best practices, will improve overall product engineering efficiency. Our analysis presents the key problems faced by these companies, the hopes they have for inner source, and the problems they experienced or expect in its adoption.

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. Our case study companies expect that inner source will improve collaboration and knowledge sharing to such an extent that overall product engineering efficiency will improve. However, we also find that the 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 such reluctance.

In future work, we intend to analyze to what extent our recommendations helped our case study companies achieve their goals. We also intend to perform more case study research expanding the scope of our current work to globally distributed software development, which we have not covered yet.

Acknowledgments

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

References

[1] Allan, G. (2006, July). The legitimacy of grounded theory. In Proceedings of the Fifth European Conference on Business Research Methods, Trinity College, Dublin (pp. 1-8).

[2] 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.

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

[4] 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.

[5] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

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

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

[19] Czarnecki, K., Helsen, S., & Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization. Software process: Improvement and pr

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

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

[27] 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.

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

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

[30] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study (No. CMU/SEI-90-TR-21). Carnegie Mellon University, Software Engineering Institute.

[31] 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-/

[32] 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.

[33] 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.

[34] 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.

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

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

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

[38] 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.

[39] Riehle, D. (2010). The economic case for open source foundations. Computer, 43(1), 86-90.

[40] 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.

[41] 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.

[42] Schultis, K. B., Elsner, C., & Lohmann, D. (2014, November). Architecture challenges for internal software ecosystems: a large-scale industry case study. In *Proc. of 22nd ACM SIGSOFT Int'l Symp. on the Foundations of Software Engineering (FSE'14)*.

[43] 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.

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

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

[46] 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.

[47] 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.

[48] Thompson, J. M., & Heimdahl, M. P. E. (2001). Extending the product family approach to support n-dimensional and hierarchical product lines. In Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on (pp. 56-64). IEEE.

[49] 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.

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

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

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

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

[54] 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.

[55] Weiss, M. (2011, August). Economics of collectives. In Proceedings of the 15th International Software Product Line Conference, Volume 2 (p. 39). ACM.

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

[57] Wijnstra, J. G. (2002). Critical factors for a successful platform-based product family approach. In Software ProductLines (pp. 68-89). Springer Berlin Heidelberg.

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

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

[60] Zikmund, W., Babin, B., Carr, J., & Griffin, M. (2012). Business research methods. Cengage Learning.