# Chapter 4
# Softwares Product Lines, Global Development and Ecosystems: Collaboration in Software Engineering

**Jan Bosch and Petra M. Bosch-Sijtsema**

**Abstract** Effective collaboration in software engineering is very important and yet increasingly complicated by trends that increase complexity of dependencies between software development teams and organizations. These trends include the increasing adoption of software product lines, the globalization of software engineering and the increasing use of and reliance on 3rd party developers in the context of software ecosystems. Based on action research, the paper discusses problems of in effective collaboration and success-factors of five approaches to collaboration in large-scale software engineering.

## 4.1 Introduction

Collaboration is perhaps the most important lever for achieving high quality, efficient and effective software engineering practices and results in virtually any software developing organization.[1] Achieving effective collaboration, however, has proven to be a major challenge in many organizations, resulting in failed or late projects, products or systems not aligned to customer requirements, clashes between the research and development (R and D) organization and the rest of the company, etc. Although significant progress has been made over time, through, among others, CMMI (Capability Maturity Model Integration), agile and iterative processes, explicit software architecture management, and effective collaboration in large-scale software development remain a challenge. For purposes in this paper we consider collaboration effective if it generates minimal overhead for the organization while avoiding the aforementioned problems.

J. Bosch (✉)
Intuit Inc, Mountain View, CA 94043, USA
e-mail: jan@janbosch.com

[1] Collaboration is defined as a recursive process where two or more people or organizations work together toward an intersection of common goals.

I. Mistrík et al. (eds.), *Collaborative Software Engineering*,
DOI 10.1007/978-3-642-10294-3_4, 

One can observe three trends that have surfaced over recent years that cause collaboration in software engineering to become significantly more complicated. The first trend is the increasingly broad adoption of software product lines [1, 2, 6, 21]. Software product lines have proven to be perhaps the most successful approach to improving productivity in software engineering; see, e.g., the product line hall of fame [10]. However, transitioning an organization that has traditionally worked in a product-centric fashion to a product line-centric way of working is a very complicated change process. The primary reason for the difficulty in changing the organization is because the product-line approach causes dependencies to be created between software assets, and between teams responsible for those assets, that did not exist earlier. In other words, an additional level of collaboration between software engineering teams and organizational units is required.

The second trend is the globalization of software development [4, 9, 19]. More and more global companies have either introduced several software development sites or engaged in strategic partnerships with remote companies, especially in India and China, due to several reasons; e.g., reduction of cycle time, reduction of travel cost, use of expertise when needed, entering new markets, and responsiveness to markets and customers [5]. Global development has many advantages but brings along its own set of challenges due to differences in culture, time zone, software engineering maturity and technical skills between teams in different parts of the world. Again, significant additional demands are placed on the collaboration between teams in the organization. When teams need to closely cooperate during iteration planning and have a need to exchange intermediate developer releases between teams during iterations in order to guarantee interoperability, the coordination cost starts to significantly affect the benefits normally associated with global development (cf. [11]).

A third important trend seen is the increasing adoption of ecosystems approaches [15]. We define software ecosystem as follows: a software ecosystem consists of a software platform, a set of internal and external developers and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs. Once a product or family of products has become successful in the market, a significant business opportunity appears in the form of third party developer and customer contributions to the product (family). This requires that the internal product (line) software is converted into a platform that is opened up to developers and development teams external to the organization. In addition, this requires that customers buying a product that is part of the software product line want to extend the functionality of the product with solutions available in the community or developed by 3rd party developers after the product has been deployed at the customer. Again, a significant additional demand is placed on the ability of the organization and the ecosystem as a whole to collaborate effectively as part of the software engineering process.

The trends discussed above have one important aspect in common: all increase the amount of coupling between software assets as well as between organizational unites. Below, we analyze the concept of decoupling in more detail. At the top level, coupling (defined as the absence of decoupling) can be broken into two main

categories, i.e., software asset coupling and organizational coupling. The former category is concerned with the dependencies that exist between technology assets, complicating their composition in planned and unplanned configurations. During the 1970s, this was already studied in the context of structured design [23] and during the last decade the research around software architecture has continued that tradition. Organizational coupling is a reflection of the dependencies between software assets in that the ability of teams to work independently is constrained because of dependencies between the software assets that the teams are responsible for.

For both types of coupling, in many contexts the term decoupling is used as the term of choice as it indicates that explicit steps have been taken to decrease dependencies between software assets that naturally are tightly connected. Based on our research, however, we take the position that the amount of coupling between software assets is a consequence of the beliefs of the software architects designing the system. Typically, the perceptions by the architects about what functionality is expected to vary versus which functionality is not, causes certain dependencies to be created without inhibitions whereas in other areas explicit decoupling techniques are applied.

Architects often are a product of the development organization they grew up in and, consequently, tend to assume a certain approach to large-scale software development. This approach assumes certain operating mechanisms to be present between different software development teams in order to govern their collaboration. The challenge, however, is that due to the three trends discussed above, the, often implicitly defined, approach to software development becomes increasingly inefficient.

We address that concern, by explicitly defining five approaches to inter-team collaboration, which are based on action research of several companies. We focus on which different collaborative approaches large-scale software development companies apply, when these approaches are most applicable and discuss some of their challenges we found in the case companies. The different models show how companies organize large-scale software development, ranging from a highly integrated to a fully decoupled, inter-organizational approach, i.e., integration-centric approach, release groupings, release trains, independent deployment and open (eco-) system development.

In the remainder of the paper we discuss these five approaches for collaboration in large-scale software as well as specific problems that arose within these approaches. We conclude the paper with discussing in which context these approaches would be most applicable.

## 4.2 Architecture, Process, Organization

Collaboration in software engineering is challenging and, as we discussed in the introduction, there are several trends that are complicating collaboration even further. In this section, we discuss the key challenges or problems that we have

identified in our research. The problem statement is organized according to three areas: (1) software architecture; (2) engineering processes and (3) the organization (mainly research and development). This is related to Herbsleb et al. [8] who perceives software architecture, plans (in our case organization) and processes as vital coordination mechanisms in software projects in order to have effective communication between software development teams. In the industrial reality, these areas are deeply interconnected, but we use this structure intentionally. Ideally, architecture and technology choices are derived from the business strategy and should drive process and tools choices. These, in turn, should drive the organizational structure of the R and D organization [13, 14]. In industry however, the three areas mentioned above are not always aligned. Often, the current organizational structure defines the processes and through that the architectural structure for the product or platforms and consequently constrains the set of business strategies that the company can aspire to implement. When companies define new growth strategies, the business strategy often collides with the existing organizational structure and consequently the process and architecture choices. The paradox is that the software development department still is responsible for releasing existing products and platforms while at the same time, needs to embark on new business strategy implementation. Typically, the architecture, process and organization approaches allow for too tight coupling and the problems discussed later can almost always be addressed by increasing the decoupling between architecture or organization elements.

Perhaps the key area for enabling effective collaboration in software engineering is software architecture. Collaboration often breaks down due to too many unnecessary dependencies between components and the teams responsible for those components. The dependencies not only need to be individually managed, but the overall system complexity grows exponentially with a growing number of dependencies.

The software architecture has a significant impact on the collaboration in the software development organization responsible for a system or platform. However, software architecture is only an enabler of effective collaboration; it does not define the collaboration itself. The engineering processes, both formal and informal, define the actual collaboration between teams and between individuals.

Next to the software architecture and processes, the organizational context and structure are important for effective collaboration in large-scale software development projects. Several aspects mentioned in literature are globalization [9, 19] co-ordination of interdependencies, knowledge management (transferring tacit knowledge into explicit knowledge for example [17]), and alignment of the architecture, processes and the organization. In the next section, we discuss five approaches found in industry according to the dimensions of architecture, process and organization.

The research and approach presented in this paper is based on an action research methodology applied by the authors in numerous software-intensive system companies as well as in other industries. The action research method seeks to bring together action and reflection, theory and practice, in participation with others, in the pursuit of practical solutions to issues of pressing concern to people, and more generally the flourishing of individual persons and their communities [20, 1].

**Table 4.1** Overview of case studies

| Cases | Company A | Company B | Company C |
|---|---|---|---|
| Product | Embedded systems | Consumer electronics | Software products |
| Market | Global | Global | North America, Asia |
| Type and size of teams | Component teams (between 10–30 team member)<br>Global teams (between 10–30 team member)<br>Platform organization (500+ members) | Division platform team (150+ members)<br>Product platform team (200+ members)<br>Product team (50+ members)<br>Global teams (30+ members) | Product platform team (200+ members)<br>Product team (25+ members) |
| Method and duration of study | Participant observer, 3 years | Participant observer, 3 years | Participant observer, 2 years |
| Data collection methods | Interviews, workshops | Interviews | Participant observation |

We studied several R and D (Research and Development) units and software development departments in three global companies (Fortune 100 and 500 companies), who developed embedded products and software and service products for different markets (European, US and Asian markets). In Table 4.1 we present an overview of the cases investigated. Data was collected with help of semi-structured and unstructured interviews (which were coded) and participant observatory methods. We applied a two-phase analysis method of first within-case analysis and later on cross-case analysis method.

## 4.3 Five Collaborative Approaches

From all the units and teams we studied, at least two cases reported one of the five approaches being applied for large-scale software development. These approaches are discussed below. We organize the discussion around three dimensions: architectural, process and organizational aspects of large scale software development and conclude with success factors of the different approaches. In Table 4.2 we present a summary of the five collaborative approaches.

### *4.3.1 Integration-Centric Development*

*Description*: We found several firms applying an integration-centric approach, in which the organization relies on the integration phase of the software development lifecycle. During the early stages of the lifecycle, there is allocation of requirements to the components. During the development phase, teams associated with each component implement the requirements allocated to the component. When the development of the components making up the system is finalized, the development

**Table 4.2** Collaboration models for large global software development

| Approach | Integration-centric | Release grouping | Release trains | Independent deployment | Open (eco-) system development |
|---|---|---|---|---|---|
| Description | Deep interconnections between the elements of the system. | Loosely coupled subsystems with high internal dependency | System components decoupled, but deployment coordinated | System components decoupled, deployment independent | Platform and 3rd part solutions decoupled and deployed independently |
| Architecture challenge | Strongly interconnected architecture – Tight interdependency and complexity challenge | High integration within release grouping, high decoupling between groupings – Management challenge of decoupling interfaces | High decoupling between components – Teams develop independently, while maintaining backward compatibility | High decoupling between components – Coordination and execution complicated | Highly decoupled with sand boxes for third party functionality – Security models in platform architecture challenge |
| Process challenge | Continuous coordination between teams – Lockstep evolution challenge | Continuous coordination within grouping – Variation challenge between and inside release groupings | Short iteration cycles; only coordination at start/end of cycle – Teams independent, but all teams need to release as same point in time | Each team selects length, frequency and time of iteration cycle – Challenge for high degree of automation and coverage of testing | Each team selects length of iteration cycle – Certification process possible |
| Organization challenge | High interdependency between teams – Mismatch architecture and organization structure | Teams responsible for different release groupings can be distributed – Coordination costs and completion time challenge | Distributed teams within organization – Reduction of coordination costs | Distributed teams within organization – Coordination performed by software architecture | Distributed teams across organizational boundaries – Challenge of misalignment business case of provider and external developers |
| Success factors | 1. Release cycle long.<br>2. Deep integration of components<br>3. Co-location of team | 1. Geographical distribution of teams aligned with release groupings<br>2. High integration within application domain | 1. Frequent releases beneficial for firm.<br>2. High level of maturity needed | 1. Different iteration cycles for different layers of the stack.<br>2. High level of maturity needed | 1. Market approach<br>2. Teams highly dispersed.<br>3. High level of maturity needed |

enters the integration phase in which the components are integrated into the overall system and system level testing takes place. During this stage, typically, many integration problems are found that need to be resolved by the component teams.

If the component teams have not tested their components together during the development phase, this phase may also uncover large numbers of problems that require analysis, allocation to component teams, co-ordination between teams and requiring continuous retesting of all functionality as fixing one problem may introduce others.

In response to the challenges discussed above, component teams often resort to sharing versions of their software even though it is under development. Although this offers a means of simplifying the integration phase, the challenge is that the untested nature of the components being shared between component teams causes significant inefficiency that could have been avoided if only more mature software assets would be shared. One approach discussed frequently in this context is continuous integration [12], but in our experience this often addresses the symptoms but not the root causes of decoupling.

*Architecture*: The architecture of the system or system family is typically not specified and if documentation exists, the documentation is often outdated and plays no role except for introducing new staff to the course grain design of the system. Because of this, the de-facto architecture often contains inappropriate dependencies between the components that increase the coupling in the system and cause unexpected problems during development.

In our cases, we found a typical architectural challenge that seems to be prevalent with this approach: the system architects failed to keep it simple. The key role of the software architect is to take the key software architecture design decisions [3] that decompose the system into consistent parts that can continue to evolve in relative independence. However, as has been studied by several researchers, (e.g., [22]) no architectural decomposition is perfect and each has crosscutting concerns as a consequence. These concerns cause additional dependencies between the components that, as discussed above, need to be managed and add to the complexity of the system. Techniques exist to decrease the "tightness" of dependencies, such as factoring out the crosscutting concerns and assigning them to a separate component or by introducing a level of indirection that allows for run-time management of version incompatibilities. In the initial design of the system, but especially during its evolution, achieving and maintaining the absolutely simplest architecture is frequently not sufficiently prioritized. In addition, although complexity can never be avoided completely for any non-trivial system, it can easily be exacerbated by architects and engineers in response to addressing symptoms rather than root causes, e.g., through overly elaborate version management solutions, heavy processes around interfaces or too effort consuming continuous integration approaches.

*Process*: Although most organizations employing this approach utilize techniques like continuous integration and inter-team sharing of code that is under development, the process tends to be organized around the integration phase. This often means a significant peak in terms of work hours and overtime during the weeks or sometimes months leading up to the next release of the product.

A challenge we found was lockstep evolution. When the system or platform can only evolve in a lockstep fashion, this is often caused by evolution of one asset having unpredictable effects on other, dependent assets. In the worst case, with the increasing amount of functionality in the assets, the cycle time at which the whole system is able to iterate may easily lengthen to the point where the product or platform turns from a competitive advantage to a liability. The root cause of the problem is the selection of interface techniques that do not sufficiently decouple components from each other. APIs may expose the internal design of the component or be too detailed that many change scenarios require changes to the API as well.

*Organization*: The development organization has a strong tendency to concentrate all-important work to one location. Even if the organization is distributed, there is often a constant push to concentrate development and the team members in remote locations tend to travel extensively.

One problem we found was a mismatch between architectural and organizational structure. In one of the organizations, we were involved in transitioning the company from a product-centric to a product-line centric approach to software development. This requires a shared platform that is used by all business units. The organization, however, was unwilling to adjust the organizational structure and instead asked each business unit to contribute a part of the platform. Each business unit had to prioritize between its own products and contributing to the shared platform and as a consequence the platform effort suffered greatly. Although the importance of aligning the organization with the architecture has been known for decades [7] in our case studies the organizations violate this principle frequently.

*Success factors*: Although the integration-oriented approach has its disadvantages, as discussed above, it is the approach of choice when two preconditions are met. First, if conditions exist that require a *very deep integration between the components* of a system or a family of systems, e.g., due to severe resource constraints or challenging quality requirements, the integration-oriented approach is, de-facto, the only viable option. Second, if the *release cycle* of a system or family of systems is *long*, e.g., 12–18 months, the amount of calendar time associated with the integration phase is acceptable.

### *4.3.2 Release Groupings*

*Description*: In this approach, the development organization aims to break the system into groups of components that are pre-integrated, i.e., a release group, whereas the composition of the release groups is performed using high decoupling techniques such as SOA-style (Service-Oriented-Architecture) interfaces [16]. At the level of a release group, the integration-centric approach is applied; whereas at the inter-release group level coordination of development is achieved using periodic releases of all release groups in the stack.

*Architecture*: In this approach, the architecture has been decomposed into its top-level components, which are aligned with the release groupings. Often, the

organization has run into the limits of the previously discussed approach and has taken the action to decouple the top-level parts of the system.

In the typical scenario, the organization evolves from an integration-centric to a release groupings approach. As the organization has allowed for many dependencies between components, the management of interfaces between release groupings often is insufficient. The definition of the APIs does not sufficiently decouple release groupings from each other. APIs may expose the internal design of the release grouping or are too detailed causing many change scenarios to require changes to the APIs.

*Process*: Similar to the architecture, the process is now also different between the release groupings, but the same as the previously discussed approach within the release grouping. The decoupling allows the release groupings to be composed, with relatively few issues. This is often achieved by more upfront work to design and publish the interface of each release group before the start of the development cycle.

In several of the cases that we studied, the organization failed to realize that processes needed to vary between and inside release groupings. This lead to several consequences, including features that cross release groupings tend to be underspecified before the start of development and need to be "worked out" during the development by close interaction between the involved teams. This defeats the purpose of release groupings and causes significant inefficiency in development.

*Organization*: As discussed in the description, the allocation of release groupings often mirrors the geographical location of teams and the definition of release grouping interfaces the level of the geographical boundaries significantly decreases the amount of communication and co-ordination that needs to take place and, consequently, efficiency is improved.

In our cases, we found that working geographically distributed increases the amount of time required to accomplish tasks due to cultural differences, time zone differences and engineers need to spend more time in co-ordinating their work across the globe. Engineers have to allocate more of their time for global coordination, which makes development less efficient. Although the release groupings approach addresses this concern to some extent, we found that the coordination cost still is quite significant.

*Success factors*: The release grouping approach is particularly useful in situations where teams responsible for different subsets of components are *geographically dispersed* . Aligning release groupings with location is, in that case, an effective approach to decreasing the inefficiencies associated with co-ordination over sites and time zones. A second context is where the architecture covers a number of application domains that require *high integration within the application domain, but much less integration between application domains*. For instance, a system consisting of video processing and video storage functionality may require high integration between the video processing components, but a relatively simple interface between the storage on processing parts of the system. In this case, making each domain a release grouping is a good design decision.

### 4.3.3 Release Trains

*Description*: In the third approach, the decoupling is extended from groups of components to every component in the system. All interfaces between components are decoupled to the extent possible and each component team can by and large work independently during each iteration. The key coordination mechanism between the teams is an engineering heartbeat that is common for the whole R and D organization. With each iteration, e.g., every month, a release train leaves with the latest releases of all production-quality components on the train. If a team is not able to finalize development and validation of its component, the release management team does not accept the component. Once the release team has collected all components that passed the component quality gates, the next step is to build all the integrations for the software product line. For those components that did not pass the component quality gates, the last validated version is used. The integration validation phase has two stages. During the first stage, each new release of each component is validated in a configuration consisting of the last verified versions of all other components. Component that do not pass this stage are excluded from the train. During the second stage, the new versions of all components that passed the first stage are integrated with the last verified versions of all other components and integration testing is performed for each of the configurations that are part of the product family. In the case where integration problems are found during this stage, the components at fault are removed from the release train. The release train approach concludes each iteration with a validated configuration of components, even though in the process a subset of the planned features may have been withdrawn due to integration issues between components. The release trains approach provides an excellent mechanism for organizational decoupling by providing a heartbeat to the engineering system that allows teams to synchronize on a frequent basis while working independently during the iterations.

*Architecture*: The architecture now needs to be fully specified at the component level, including its provided, required and configuration interfaces. No dependencies between components may exist outside the interfaces of the components.

In a web service-centric architecture inside an organization, the teams associated with components develop independently while maintaining backward compatibility for their provided interfaces. This allows each team to release at the end of the development cycle and, after a, typically automated, testing effort the new component versions are released at the same time.

*Process*: The key process challenges, as discussed above, are the predevelopment cycle work around interface specification and content commitment and the process around the acceptance or rejection of components at the end of the cycle. In addition, especially when the organization uses agile development approaches, sequencing the development of new features such that dependent, higher level features are developed in the cycle following the release of lower level features allows for significantly fewer ripple effects when components are rejected.

The release train approach allows team to work independently from each other during the development of the next release, but it still requires all teams to release at the same point in time. The process of testing the new version of components consists of two stages. First, each new version of a component is tested in the context of the released versions of all other components. This verifies backward compatibility. In the second stage, the new versions of all components are brought together to verify the newly released functionality across component boundaries.

*Organization*: As the need for co-ordination and communication between the teams has been reduced and is much more structured in terms of time and content, the organization can be distributed without many of the negative consequences found in the earlier approaches.

In one of the companies that we studied, this approach reduced the coordination cost quite considerably. Teams co-ordinated around the release of new versions of components to plan for the next release. However, limited centralized planning was necessary. Instead, teams co-ordinated with each other at the interface boundaries.

*Success factors*: The release train approach is particularly suited for organizations that are required to deliver *a continuous stream of new functionality in their products or platform*; either because new products are released with a high frequency or because existing products are released or upgraded frequently with new functionality. The organization has a business benefit from frequent releases of new functionality. Companies that provide web services provide a typical example of the latter category. Customers expect a continuous introduction of new functionality in their web services and expect a rapid turnaround on requests for new functionality. The release train approach does require a relatively *mature development organization and infrastructure*. For instance, the amount and complexity of validation and testing that is required demands a high degree of test automation. In addition, interface management and requirements allocation processes need to be mature in order to achieve sufficient decoupling, backward compatibility and independent deployment of components.

### *4.3.4 Independent Deployment*

*Description*: The independent deployment approach assumes an organizational maturity that does not require an engineering heartbeat (a heartbeat in the engineering system allows teams to synchronize on a frequent basis while working independently during iterations) including all the processes surrounding a release train [18]. In this approach, each team is free to release new versions of their component at their own iteration speed. The only requirement is that the component provides backward compatibility for all components dependent on it. In addition, the teams develop and commit to roadmaps and plans. The lack of an organization-wide heartbeat does not free any team from the obligation to keep

their promises. However, the validation of a component before being released is more complicated in this model as any component team, at any point in time, may decide to release its latest version.

*Architecture*: Similar to the release trains approach, the architecture needs to be fully specified at the component level. Architecture refactoring and evolution is becoming more complicated to co-ordinate and execute on.

In one of the cases, the business realities forced some fundamental architectural design decisions to be revoked and replaced with alternative solutions. This required the independent teams to resort to significantly more coordinated ways of working until the architecture had stabilized after several release iterations.

*Process*: The perception in the organization easily becomes that there no longer is an inter-team process for development as any team can develop and release at their leisure. In practice, this is caused because the process is no longer a straightjacket but more provides guardrails within which development takes place. The cultural aspects of the software development organization, especially commitment culture and never allowing deviations from backward compatibility requirements, needs to be deeply engrained and enforced appropriately.

As the process does not enforce joint releasing of components, any component team can release at their own frequency and time. This requires an even higher degree of automation and coverage of the testing framework in order to guarantee the continued functioning of the overall system.

*Organization*: Similar to the release trains approach, the organization can take many shapes and forms as long as the development teams associated with a component are not distributed themselves.

As the process and geographic co-location of the development organization is not longer something that one can rely on, the key organization principle is now centered on the software architecture. Co-ordination is no longer process and human-driven, but instead is performed via the software architecture. As a consequence, where as team leads and engineers talk very little to other teams, the architects in the organization typically increase their interaction to guide the evolution of the architecture.

*Success factors*: The independent deployment approach is particularly useful in cases where *different layers of the stack have very different "natural" iteration frequencies* . Typically, lower layers of the stack that are abstracting external infrastructure iterate at a significantly lower frequency. This is both because the release frequency of the external components typically is low, e.g. one or two releases per year, and because the functionality captured in those lower layers often is quite stable and evolves more slowly. The higher layers of the software stack, including the product-specific software, tend to iterate much more.

The key factor in the successful application of the independent deployment approach is the *maturity of the development organization*. The processes surrounding road mapping, planning, interface management and, especially, verification and validation, need to be mature and well supported by tools in order for the model to be effective.

### 4.3.5 Open Ecosystem

*Description*: The final approach discussed is an approach in which inter-organizational collaboration is strived after. Successful software product lines are likely to become platforms for external parties that aim to build their own solutions on top of the platform provided by the organization. Although this can, and should, be considered as a sign of success, the software product line typically has not been designed as a development platform and providing access to external parties without jeopardizing the qualities of the products in the product line is typically less than trivial. Even if the product line architecture has been well prepared for acting as a platform, the problem is that external developers often demand deeper access to the platform than the product line organization feels comfortable to provide.

The typical approach to address this is often twofold. First, external parties that require deep access to the platform are certified before access is given. Second, any software developed by the certified external parties needs to get validated in the context of the current version of the platform before being deployed and made accessible to customers.

Although the aforementioned approach works fine in the traditional model, modern software platforms increasingly rely on their community of users to provide solutions for market niches that the platform organization itself is unable to provide. The traditional certification approach is infeasible in this context, especially as the typical case will contain no financial incentive for the community contributor and the hurdles for offering contributions should be as low as possible. Consequently, a mechanism needs to be put in place that allows software to exist within the platform but to be sandboxed to an extent that minimizes or removes the risk of the community-offered software affecting the core problem to any significant extent.

The open ecosystem development model allows unconstrained releasing of components in the ecosystem not only by the organization owning the platform but by also by certified 3rd parties as well prosumers and other community members providing new functionality. Although few examples of this approach exist it is clear that a successful application of this approach requires run-time, automated solutions for maintaining system integrity for all different configurations in which the ecosystem is used.

*Architecture*: The main architectural focus when adopting this approach is to provide a platform interface that on the one hand opens up as much useful platform functionality for external developers and on the other hand provides an even higher level of quality and stability as the evolution of interfaces published to the ecosystem is very time and effort consuming as well as constraining. In addition, security precautions have to be embedded in the interface to provide the best defense mechanisms for accidental or intended harm to the customers in the ecosystem.

Especially in the case where external developers can release directly to customers without involvement of the platform company, the architecture has to be developed defensively at its external interfaces. In two of the cases that we studied, this

translated into the implementation of an elaborate security model in the platform architecture to control access of external code in the platform.

*Process*: As the ecosystem participants are independent organizations, no common process approach can be enforced, except for gateways, such as security validation of external applications. However, each limitation put in place causes hurdles for external developers that inhibit success of the ecosystem, so one has to be very careful to rely on such mechanisms.

In one of the cases that we studied, the platform company felt obliged to introduce a certification process for externally developed code as the risk for customers was considered to be too great.

*Organization*: The organization in this approach is best described as a networked organization, i.e., the platform providing organization has a rather central role, but the external developers provide important parts, often the most differentiating and valuable parts of the functionality.

The key difference that the two of the cases that we studied struggled with is that the business case for the platform organization is not necessarily aligned with the business case of external developers. Although the platform company should strive to achieve this situation, there is a natural tension in terms of monetization: the platform company has to leave sufficient value in the ecosystem for external developers to have an acceptable return on investment.

*Success factors*: The open ecosystem model is a natural evolution from the release train and independent deployment models when the organization decides to *open up the software product line to external parties*, either in response to demands by these parties or as a strategic direction taken by the company in order to *drive adoption by its customers*.

The key in this model, however, is the ability to provide proper architectural decoupling between the various parts of the ecosystem without losing integrity from a customer perspective. In certain architectures and domains, the demand for deep integration is such that, at this point in the evolution of the domain, achieving sufficient decoupling is impossible, either because quality attributes cannot be met or because the user experience becomes unacceptable in response to dynamic, run-time composition of functionality.

Two areas where this approach is less desirable are concerned with the platform *maturity* and the *business model*. Although the pull to open up any software product line that enjoys its initial success in the market place, the product line architecture typically goes through significant refactoring that can't be hidden from the products in the product line or the external parties developing on top of the platform defined by the architecture. Consequently, any dependents on the product line architecture are going to experience significant binary breaks and changes to the platform interface. Finally, the transition from a product to a platform company easily causes conflicts in the business models associated with both approaches. If the company is not sufficiently financially established or the platform approach not *deeply ingrained in the business strategy*, adopting the open ecosystem approach fail due to internal organizational conflicts and mismatches.

## 4.4 Conclusion

Collaboration can be viewed as the most important lever for achieving high quality, efficient and effective software engineering practices and results in virtually any software developing organization. Although collaboration has been complicated, several trends increase the complexity of managing dependencies between software development teams and organizations. These trends include the increasing adoption of software product lines, the globalization of software engineering and the increasing use of and reliance on 3rd party developers in the context of software ecosystems. The trends share as a common characteristic that the coupling between the software assets as well as between the organizational units is increased. Consequently, decoupling mechanisms need to be introduced to address the increase in coupling.

In this paper, we have discussed the challenges of decoupling approaches for large-scale software collaboration from an architecture, process and organization perspective. From extensive action research involving several cases, we found five different approaches on a continuum ranging from low to high decoupling. We illustrated the challenges of these approaches in specific instances from the case study examples. Our experience shows that these challenges are caused due to the application of a collaboration model that is not applicable for a specific situation. In most cases that we studied, significant problems were caused by the application of a collaboration approach that did not provide sufficient decoupling and could or were addressed by the introduction of a more decoupled approach to collaboration.

The contribution of the paper is that it presents a clear overview of possible collaboration approaches for large-scale software development and their particular challenges where surprisingly little literature exists in this area. With this paper we give an insight in different decoupling approaches, their specific challenges and their success factors (applicability).

## References

1. Bosch J (2000) Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education. London: Addison-Wesley & ACM Press.
2. Bosch J (2002) Maturity and evolution in software product lines: Approaches, artifacts and organization. Proceedings of the 2nd Software Product Line Conference (SPLC).
3. Bosch J (2004) Software architecture: The next step. Proceedings of the First European Workshop on Software Architecture (EWSA 2004), Springer LNCS.
4. Carmel E, Agarwal R (2001) Tactical approaches for alleviating distance in global software development. IEEE Software 1(2: 22–29.
5. Cascio F, Wayne S, Shurygailo S (2003) E-leadership and virtual teams. Organizational Dynamics 31(4): 362–376.
6. Clements P, Northrop L (2001) Software Product Lines: Practices and Patterns. Reading, MA: Addisson-Wesley.
7. Conway ME (1968) How do committees invent. Datamation 14(5): 28–31.
8. Herbsleb JD, Grinter RE (1999) Architectures, co-ordination and distance: Conway's law and beyond. IEEE Software 16(5): 63–70.
9. Herbsleb JD, Moitra D (2001) Global software development. IEEE Software 18(2): 16–20.

10. HOF http://www.sei.cmu.edu/productlines/plp_hof.html.
11. Kraut R, Steinfield C, Chan AP, Butler B, Hoag A (1999) Co-ordination and virtualization: The role of electronic networks and personal relationships. Organization Science 19(6): 722–740.
12. Larman C (2004) Agile and Iterative Development: A Manager's Guide. Reading, MA: Addison-Wesley.
13. Linden F van der, Bosch J, Kamsties E, Kansala K, Obbink H (2004) Software product family evaluation. Proceedings of the Third Conference Software Product Line Conference (SPLC 2004), Springer Verlag LNCS 3154, pp. 110–129.
14. Linden F van der, Schmid K, Rommes E (2007) Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Berlin Heidelberg: Springer Verlag.
15. Messerschmitt DG, Szyperski C (2003) Software Ecosystem: Understanding an Indispensable Technology and Industry. Cambridge, MA: MIT press.
16. Newcomer E, Lomow G (2005) Understanding SOA with Web Services. Upper Saddle River, NJ: Addison Wesley.
17. Nonaka I (1994) The Knowledge Creating Company. How Japanese Companies Create the Dynamics of Innovation. New York: Oxford University Press.
18. Ommering R van (2001) Techniques for independent deployment to build product populations. Proceedings of WICSA 2001, pp. 55–64.
19. Sanwan R, Bass M, Mullick N, Paulish DJ, Kazmeier J (2006) Global Software Development Handbook. Boca Raton, FL: CRC Press.
20. Reason P, Bradbury H (2001) Handbook of Action Research. (Eds.) Thousand Oaks, CA: Sage Publishing.
21. SPLC http://www.splc.net/.
22. Tarr P, Ossher H, Harrison W, Sutton SM Jr (1999) N degrees of separation: Multi-dimensional separation of concerns. Proceedings 21st International Conference Software Engineering (ICSE'1999), IEEE Computer Society Press, pp. 107–119.
23. Yourdon E, Constantine LL (1979) Structured Design. Englewood Cliffs, NJ: Prentice-Hall.