

From Integration to Composition: On the Impact of Software Product Lines, Global Development and Ecosystems

Jan Bosch¹ and Petra Bosch-Sijtsema²

¹. Intuit Inc, Mountain View, CA, USA. Jan@JanBosch.com

². Helsinki University of Technology, Finland and visiting scholar Stanford University, Stanford, CA, USA.
Pbosch@stanford.edu

Abstract

Three trends accelerate the increase in complexity of large-scale software development, i.e. software product lines, global development and software ecosystems. For the case study companies we studied, these trends caused several problems, which are organized around architecture, process and organization, and the problems are related to the efficiency and effectiveness of software development as these companies used too integration-centric approaches. We present five approaches to software development, organized from integration-centric to composition-oriented and describe the areas of applicability.

Keywords: software product lines, software ecosystems, global development, software integration, software composition

Introduction

For most software systems companies, large scale software development is complicated, expensive, slow and unpredictable. Four decades of software engineering research have resulted in a wide range of techniques to manage the complexity of software systems development. However, the growth in size of modern software systems and the scale of the R&D organizations responsible for these systems is such that we constantly need new approaches to manage the complexity.

Now, however, we can identify three trends that further accelerate the complexity of software development and we need to analyze and understand the resulting challenge even better than earlier. The first trend is the widespread adoption of software product lines. Over the last decade, the structured approach to intra-organizational reuse of software assets has reached a new level of sophistication. However, the adoption of software product lines also brings a new level of dependency in the organization that did not exist in a product-centric approach, causing added complexity. The second trend is the broad globalization of software development in many organizations. Although especially global companies have always had distributed development, the industry-wide adoption of development centers in Asia, e.g. India, and other continents has elevated the complexity of dependency management to a new level. Finally, especially on the internet, but also in other domains, the importance of building a software ecosystem, i.e. a community of 3rd party application developers, around a successful product, is becoming increasingly popular and central to the strategy of many

companies. The consequence, however, is that the complexity of software development is increased due to the dependencies between the platform company and the 3rd party developers.

Based on empirical studies that we performed at several case study companies, mostly through action research, we have come to the conclusion that many of the challenges that exist in large-scale software development are the consequence of the organization applying an overly integration-centric approach. A significant simplification of software development can be achieved by transitioning from an integration-centric approach to composition-oriented approach to software development. However, achieving this transition requires deep understanding because of the close interconnection between architecture, process and organization. Consequently, we discuss all these factors and their relationship in this paper.

The contribution of the paper is threefold. First, we provide a detailed description of the three main trends that are driving and even accelerating complexity in modern large-scale software development. Second, we analyze the architectural, process and organizational problems that an organization may experience when it applies the wrong approach to its software development. Finally, we present five approaches to large-scale software development, organized along a continuum ranging from integration-centric to fully composition-oriented. Although the approaches in themselves are not necessarily novel, our framework in which we present them as well as our discussion of applicability and the implications on architecture, process and organizations is.

The remainder of the paper is organized as follows. The next section presents and discusses the three trends that, in our opinion, are driving the transition from an integration-centric to an increasingly compositional approach of software engineering. In the subsequent section, we introduce the case study companies that, in part, underlie our research. Then, we present the problems we observed in our case studies, in which we capture the consequences of companies not transitioning to more compositional architectures, processes and organizational approaches. The next section presents a taxonomy of five approaches to large scale software development organized from integration-centric to increasingly composition-oriented. Finally, we end the paper with a conclusion section.

Three Trends

Large scale software development is a very complex, effort consuming and expensive activity. Although decades of software engineering research have resulted in innovation and improvements, including increasing the abstraction level at which software development takes place, new software development processes and novel approaches to architecting systems, the fact is that large scale software development still is largely unpredictable, inefficient and error-prone.

Although the companies that we have studied as part of our research have found ways to manage their software engineering to the best of their abilities, these approaches tend to be rather integration-centric, i.e. significant effort is placed on the last stage of the software development lifecycle where the independently developed parts are manually integrated and validated. However, there are three trends that have started to require software companies to make significant changes to their approaches as the cost and complexity of an integration-centric approach is increasingly rapidly. In this paper, we describe

the approaches companies can apply to transition to a composition-oriented approach. First, however, we need to present the three trends that drive this need to transition, i.e. software product lines, global development and software ecosystems.

Software Product Lines

Over the last decade, software product lines (Bosch 2000; Bosch 2002; Clements & Northrop 01; SPLC) have enjoyed major adoption in the industry. Companies have adopted the technique for a variety of reasons, e.g. to decrease cost of development, to reduce time to market, to expand the product portfolio or to achieve commonality in user experience between different products. Although a variety of alternatives exist, a software product line, in its basic form, consists of a software platform shared by a set of products. Each product can typically select and configure components in the platform for its own purposes and extend the platform with product specific functionality.

Organizationally, teams exist for each of the products as well as for the platform. In many cases, the platform consists of several components with associated teams. Software development is coordinated between the teams in several different ways. During the front-end of the development cycle, plans and roadmaps are shared, discussed and aligned between the teams. During development, often interaction takes place to coordinate interfaces and to verify that next versions of components are still compatible, even though these are under development. At the end of the development cycle, each product team aims to integrate the platform and product-specific components into a reliable and consistent product (see, e.g. Bosch 2000 for details).

Software product lines, when applied successfully, can provide enormous benefits. In (HOF), cases are described where adopting software product lines allowed for reducing development expenses with 50% to 75%, decrease defect density with similar numbers and increases the size of product portfolio with up to an order of magnitude.

Global Development

For many companies, software development is going global (Carmel & Agarwal 2001; Herbsleb & Moitra, 2001; Sanwan et al. 2006). 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, or entering new markets, responsiveness to market and customer needs (Cascio 2003). 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.

From an organizational perspective, geographically dispersed teams that develop components that are part of a system or family of systems have significantly more difficulty in implementing the necessary coordination. The challenge is that companies tend to copy their development processes and ways of working when changing development from local to global and then work to address the symptoms by technology and coordination processes. As a consequence, the coordination cost may start to significantly affect the benefits normally associated with global development (cf. Kraut et al 99).

Software Ecosystems

The most recent trend, especially in the domain of Web 2.0 companies but also in other domains, is the adoption of a software ecosystem strategy (Messerschmitt & Szyperski 03). 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.

Taking a close look at the definition, one can see a clear path from software product lines to software ecosystems, i.e. there is a platform and a set of internal developers building products on top of the platform. However, in addition to that, there is a set, or community, of external developers that build on top of the same platform as well as extend the products released by the company. A software ecosystem approach takes a community perspective, including external developers, domain experts and users, and hence requires a community-centric way of collaborating and coordinating. In this sense, whereas the scope of a software product line is intra-organizational, the scope of a software ecosystem is much broader, including external developers and the extensions that they provide as well as other parties providing contributions.

Similar to the transition from a product-centric to a product line centric approach to software development, software ecosystems build dependencies between components and their associated organizations that did not exist earlier. In this case, not only does the evolution of the platform and products developed on top of it need to be coordinated with the teams within the company, but also the external developers and development teams need to be involved in the coordination. This coordination process affects all phases of the lifecycle. During the road mapping and planning process, external developers often have strong views on the priorities and sequencing of platform functionality. During development, the architecture, specifically the interfaces between the platform and the products, should be developed in collaboration with the external developer community, not just published. Finally, during the validation of the new release of the platform, the external developer community needs to be involved in order to minimize the unintended breaks in functionality when rolling out the next platform release.

Case Companies

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 (Reason and Bradbury, 2001: 1). The basis for this paper is formed by three cases that we have worked with in detail during the last years, but the conclusions and premise of the paper is influenced by tens of other companies that the authors have worked with during their careers. Although we describe the case study companies holistically, our focus is on the software R&D part of each of the companies.

Case Company A

Company A is a Fortune 100 company developing embedded products, i.e. products that include mechanical, hardware and software parts. The company releases several new products per year and uses a software product line approach to decrease the per-product software R&D expenditure. As a consequence a significant part, i.e. more than half, of the software R&D is performed in the central platform organization. The size of the software ranges in the 7-15 million lines of code range.

The company, being global, has development sites in several locations in Europe, the Americas and Asia, specifically India. The software platform organization is, consequently, also distributed across the world. The company has aligned teams with the software architecture of the platform and initially organized teams in a distributed fashion where the head for the component team, the lead architect as well as key members of the development team were located in the company's home country and the rest of the team was located at remote sites.

The aforementioned approach resulted in very low productivity in the distributed teams due to major communication overhead, cultural differences and motivational issues for team members at the remote sites. The organizational setup was changed and full component responsibility was assigned to a geographically local team when it became clear that no learning effect would compensate for the perceived inefficiencies. The head of the team and lead architect still needed to coordinate over geographical and architectural boundaries, but the amount of communication was significantly less due to the smaller number of people involved and the smaller number of issues on architectural boundaries.

The company also adopted a software ecosystem approach on top of its platform, allowing external developers to develop applications and extensions on the platform that can be deployed on the products that it ships. In several cases, decisions had to be taken that forced management to choose between the responsibilities towards the external developers in the ecosystem and the needs for differentiation and time to market of new products or new releases of existing products.

Case Company B

Company B is a Fortune 500 company developing software products and services operating, primarily, on personal computers. The company's products address both consumer and business markets and the company releases several products per year, including new releases of existing products and completely new products.

The products developed by the company range in the multi- to tens of millions lines of code and tend to contain very complex components that implement national and international regulations. Although significant opportunities for sharing between different products exist, the company has organized its development based on a product-centric approach, i.e. teams are organized around a product and tend to be geographically local. Consequently, little or no sharing takes place between teams.

The company is predominantly present in a few western countries, but is in the process of expanding its business to global markets, specifically focusing on emerging (BRIC) markets. As part of its strategy to globalize, the company has established a major development center in Asia. Since it has a limited history of inter-team dependencies, the main approach to achieving the benefits of global development was to

move the responsibility for complete legacy products to the development center in Asia. The developers at the western development site were, consequently, freed up and these developers were assigned to work on the development of new products and the evolution of rapidly growing recently released products.

Case Company C

The third company is a Fortune 100 company that builds a wide variety of embedded systems for different markets. We mainly focus on the division that develops products for the global consumer market, basically servicing all continents.

The business strategy of the company is focused on having a rich set of consumer products in the market, while minimizing the development effort through the application of software product line principles.

The size of the software in the products ranges in the several million lines of code. The development teams are distributed across three continents, resulting in global development that requires careful coordination as the company employs a product line approach. Although each product is built from a standard platform, the development of the platform is not centralized, but rather the platform components are owned by distributed teams, but can still be used, extended and changed by product teams in other locations. This approach provides significant independence for product teams, but it does risk inefficiency in the system due to duplicate extensions to components and too product-specific extensions to the shared components.

Problems Observed at Case Study Companies

Despite decades of research, large-scale software engineering is challenging and, as we discussed earlier in the paper, 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. These problems are based on the research at the three case study companies, but we have identified the same problems at case companies not presented in the previous section.

The problem statement is organized according to three areas, i.e. software architecture, engineering processes and the R&D organization. The industrial reality is that 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&D organization (Linden 2004). In industrial practice, 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 this 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 R&D department still is responsible for releasing existing products and platforms while at the same time, needs to embark on the implementation of the new business strategy. This factor contributes to the problems discussed in this section, which we have collected and generalized

empirically through action research. In our experience, the main cause for the problems discussed below is that the architecture, process and organization approaches allow for too tight coupling and the problems discussed later in this section can almost always be addressed by increasing the decoupling between architecture or organization elements.

For each of the areas, we present a set of key problems associated with that area. Problems discussed in later areas may have bearing on earlier areas, but we have structured the section such that each problem is discussed in the area that it is predominantly related to.

Software Architecture

As discussed in the introduction, the premise of the paper is that a significant transition is necessary from a process-centric, manual integration approach to an architecture-centric automated composition approach. This stresses the importance of software architecture for achieving the necessary decoupling. Composition often breaks down due to too many unnecessary dependencies between components and the teams responsible for those components. Each dependency requires effort to manage it, but the primary consequence is that overall system complexity grows exponentially with a growing number of dependencies, reducing productivity further.

One may see several symptoms in cases where the architecture does not provide sufficient decoupling for the R&D organization, and potentially for the external partners or ecosystem participants. Examples include ripple effects in cases where one team's delays spread throughout the software stack, high integration cost and lock-step evolution. Although these examples may be part of the process, the inefficiency is clearly caused by the software architecture. When the architecture does not allow for sufficient decoupling between teams, the overall efficiency of product and platform development is significantly affected in a negative fashion.

More precisely, we identify below the primary software architecture problems as observed in the case study companies and present some examples found from the cases.

- **Failing to keep it simple:** We found that all case study companies suffered from this problem, but especially company B suffered from this in its legacy products. The key role of the software architect is to take the key software architecture design decisions 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. (Tarr et al. 1999), no architectural decomposition is perfect and each has cross-cutting 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 cross-cutting 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.

- **Lock-step evolution:** Case study companies A and B suffered from the lock-step evolution problem, i.e. all software assets have to move to the next iteration or release simultaneously otherwise the system or platform breaks down. 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. The cross-cutting concerns discussed under the previous point, obviously exacerbate the lockstep evolution problem.
- **Insufficient quality attribute management:** Even in a case where the functional aspect of the API has been designed properly, the quality attributes and the design decisions to achieve certain quality attributes, may cause implicit inter-component dependencies. For instance, subsequent versions of a component, having different resource requirements and timing behavior may cause other components to fail. Although quality attributes are inherently cross cutting, one can use mechanisms, such as containers, to minimize the impact of these kinds of changes. Especially the case study companies developing embedded systems used solutions that bypassed the architectural design and principles. Often these solutions were introduced as a temporary, product-specific extension to solve a particular quality attribute problem but over time these solutions became part of the platform architecture as they were “sanctioned” as the exception tends to be turned into a rule.

Although the problems discussed above are real concerns for the case study companies, it is important to note that architects need to optimize for a conflicting set of functional, quality and business requirements, sometimes causing them to sacrifice simplicity and decoupling in favor of other requirements. When this occurs in an objective and explicit fashion, the higher integration and evolution cost associated with the architecture are a justifiable consequence. In our experience, however, most architectures suffer from the problems discussed above due to a lack of objective and explicit decision making. Typically, design evolution is reactive and symptom oriented and there is a severe lack of architecture refactoring efforts. As a consequence, these architectures require a manual, integration-oriented approach instead of allowing for an automated, compositional approach during product creation and evolution.

Engineering Processes

The software architecture is central in providing the decoupling in the system that allows for a compositional approach. However, software architecture is only an enabler of compositionality and

actual compositionality can be jeopardized by implementation and process. The engineering processes, both formal and informal, define the concrete collaboration between teams and between individuals.

Engineering process seems to have lost in popularity over recent years, both as a backlash against CMM and CMMI initiatives in many companies, including the case companies reported upon in this paper, and in response to the wide-spread adoption of agile development processes. Therefore, it is important to realize that analogous to any system having an architecture, any individual, team and organization employs a process, be it implicit or explicit, ingrained in the culture or read from a cookbook-like recipe. The observations from the case study companies show that the largest challenges around processes typically exist at the inter-team level and often in response to addressing the symptoms of inter-components and/or inter-team issues, causing complexities and inefficiencies.

Below, we present the key engineering processes problems that we have identified at the case study companies. It is important to note that these symptoms may occur even if the architecture is optimal from the perspective of decoupling. The inefficiencies occur because the engineering processes are not properly aligned with the needs of system or platform:

- **Insufficient pre-iteration cycle work:** In some of the teams in case company B that we studied, features that cross component boundaries were underspecified before the development cycle started and were “worked out” during the development. In practice, this requires close interaction between the involved teams and causes significant overhead that could easily be avoided by more upfront design and interface specification. A consequence of this approach is that it builds an “addiction” between teams in that there is a need for frequent (daily) developer-to-developer drops of code that is under development in order to avoid integration problems later on. This, in turn, often results in largely manual testing of new functionality because requirements solidify during the development cycle and automated tests could not be developed in time.
- **Unintended resource allocation:** Resource allocation is a tool used by companies to align resources with the business strategy. In practice, however, at two of the case study companies, i.e. A and B, teams frequently assign part of their resources to other software components and their associated teams. The reason is that they are dependent on the other components to be able to get their own functionality developed and released. One can view this as a lack of road mapping activities and in that sense, it supports the previous point. The consequence is again, that the coordination costs between teams easily become excessive, resulting in a general perception in the organization that significant inefficiencies exist.
- **High and unpredictable product integration cost:** A third problem, observed in all case study companies and to a significant extent caused by the earlier discussed problems is that during product integration, incompatibilities between components are detected during system test and quality attributes break down in end-to-end test scenarios. This causes a costly and unpredictable integration process that, being at the end of the development cycle, causes major difficulties at the affected companies.

- **Lack of process discipline:** We found that some teams in our case study organizations, especially case company B, suffer from a lack of process discipline. Although processes have been defined, we observed that engineers and teams sometimes simply ignore them. For example, in one organization, case company A, engineers in different teams worked out additional, detailed interfaces between their respective components, despite the fact that the designed architecture did not allow for any connections between these components. As a result, independent validation of the components became impossible. The architectural integrity was violated and during the evolution of the system, these implicit dependencies caused major inefficiencies in the development process. Although engineers and teams are aware of and understand the defined processes, the typical argument is that in this particular case the process is not applicable and that following the process would have negative consequences, e.g. cause delays. Once the difficulties begin to mount, this behavior is reinforced resulting in an increasingly lax process attitude.
- **Mismatched engineering processes:** Another case is when the engineering processes are defined and followed, but the processes are simply not suitable for the type of system or platform and the organizational setup. For instance, processes relying on face-to-face communication or processes that assume interaction between teams during iterations cause significant inefficiencies in large R&D organizations and especially global organizations. For example in case company A, the delivery of the next version of a component to the integration team required verbal communication between the integration team and the component team who, after the company adopted global development, worked in a distributed setting with 12 hours time difference. The ability to have verbal communication was obviously highly impacted by the time difference.
- **Disconnected business and engineering processes:** Finally, even when the proper engineering processes have been defined and used, the connection to business processes, especially product management, may be missing. Especially while implementing a new business strategy, not only the R&D organization, but also the rest of the organization has to change significantly. We found two major problems: (a) the business strategy is defined at a level of abstraction that still requires significant interpretation that cannot be performed by the R&D organization, especially with respect to prioritization and sequencing of functionality. (b) In the mature existing products, the product management has grown accustomed to defining the next release in terms of features to be added to the existing product. The new business strategy requires the definition of a completely new product or platform that has to be defined as an entirety and not as a delta of last year's product and the product management function is unable to provide necessary guidance. For instance, in case company B, the new platform offered significant new business opportunities, but product management was unable or unwilling to exploit these for fundamentally new products.

R&D Organization

During our research at the case study companies, including those reported on in this paper, we found that the organizational structure and context are important for the architecture and the software process. We observed problems in the following scopes:

- **Globalization:** Having software engineers located at different geographical locations or even crossing country boundaries has significant implications for the communication and collaboration of the engineer teams. 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 coordinating their work across the globe (Powell et al. 2004). Engineers have to shift their time for valuable work and global coordination, which makes development less efficient. Cultural and language differences are evident in globalization and cultural differences appear to lead to coordination difficulties and create obstacles to effective communication (cf. Kraut et al 1999). For example, in one organization that we worked with, case company A, teams were geographically split, with the team lead architect and senior engineers located at the main site of the organization in Europe and the remaining engineers in a remote site in India. This required significant communication taking place over geographical boundaries resulting in very inefficient development processes as well as a de-motivated team at the remote site, due to a lack of autonomy and responsibility of the remote site.
- **Tacit knowledge:** Tacit knowledge is implicit and deeply rooted (Nonaka 1994), and is an important aspect for working in global teams. Work practices, awareness of cultural differences as well as contextual awareness of a location can be defined as tacit knowledge of the local team members, which is difficult to share or transfer to remote teams. Another aspect in which tacit knowledge is important to take into account is in organizational change processes. Tacit knowledge in this respect is a good match for the existing legacy of products and platforms. When a new business strategy is implemented, part of the existing tacit knowledge needs to be changed. However, as it is implicit knowledge it becomes difficult to implement the necessary changes. For example, in case company C, the culture allowed teams to add product specific code in all areas of the architecture, which initially caused inefficiencies when the company transitioned to a product line approach. It took significant effort, especially by the lead architects, to convert the implicit, tacit assumptions into a formalized description of behavior, explain the negative consequences and to enforce a different approach to engineering.
- **Mismatch** between architectural and organizational structure. One of the organizations, i.e. case company B, transitioned 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. Now 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 (Conway 1968), in practice many organizations violate this principle constantly.

- **Coordination cost.** A problem observed in all case study companies is that when decoupling between shared software assets is insufficiently achieved is excessive coordination cost between teams. One might expect that alignment is needed at the road mapping level and to a certain extent at the planning level. 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 of shared asset teams is starting to significantly affect efficiency.

Concluding Remarks

Our research with the case study companies concludes that most often a too integration-centric approach is applied to software development, i.e. architects and engineers insufficiently address decoupling of assets that would allow teams to work more independently. This results in the problems and associated inefficiencies discussed in this section. We have organized these issues around architecture, process and organization, but as is clear from the examples that we provided in the section, there are deep interconnections and interdependencies between each of the areas. This is the reason for addressing these in an integrated fashion in this paper, as most of the existing work related to this area addresses only a slice or segment of the overall problem area.

From Integration to Composition

The main thesis of this paper is that software development organizations need to transition from a more integration-centric to a more composition-oriented approach because of the trends discussed earlier in the paper affecting them. Although we believe, based on our research, most companies fit this category; it does not automatically mean that applying the most compositional approach is the best approach for all companies. There are good reasons for using an approach that contains more integration-oriented elements, as we discuss below. In this section, we outline five approaches to organizing large-scale software development, i.e. integration-centric development, release groupings, release trains, independent deployment and open (eco-) system development. These approaches are, as the names indicate, organized from more integration-oriented to more composition-oriented. For each approach, we first describe the approach and then analyze when this approach could be applied.

A. Integration-centric development

Description. When applying an integration-centric approach, the organization relies almost exclusively 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 enters the integration phase in which the components are integrated to form 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, in collaboration with the integration team.

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, coordination between teams and requiring continuous retesting of all functionality as fixing one

problem may introduce others. Consequently, several organizations use continuous integration (Larman 2004) in combination with automated regression testing to address the symptoms of this approach. Although this does alleviate the symptoms, it does not address the root cause: the organization is using the wrong approach for its software development, i.e. an approach that provides insufficient decoupling.

A second technique component teams often resort to, in response to the challenges discussed above, is 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.

When to apply. 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 inter-dependencies 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 to 18 months, the amount of calendar time associated with the integration phase may be acceptable. Finally, it has great benefits if the R&D organization is local rather than distributed, because of the amount of communication and coordination that needs to take place between the teams.

Summary.

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 coarse 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.

Process: Although most organizations employing this approach employ 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.

Organization: The R&D 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.

B. 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 interfaces (Newcomer & Lomow 2005). Within the context 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.

When to apply. 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 coordination 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 and processing parts of the system. In this case, making each domain a release grouping is a good decision.

Summary.

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.

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.

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 coordination that needs to take place and, consequently, efficiency is improved.

C. 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&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 component is not accepted by the release management team. 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 system level 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. Components that do not pass this stage are excluded from the train as the backward compatibility requirements are not met. 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 for the engineering system that allows teams to synchronize on a frequent basis while working independently during the iterations.

One of the main changes that need to be implemented when moving from release groupings to release trains is the discipline to perform the necessary upfront work before the start of the development cycle. Each component needs to commit to the features that it will develop and announce the changes to its interface. The second change is to enforce the discipline to “throw a component off the train” when it causes problems during the composition stage. Although it is acceptable to allow for a few hours or days during the composition stage to work out minor mismatches, any component that causes major issues needs to be rejected. Especially during the transition process, this may cause significant tension in the organization while the new rules are being enforced.

When to apply. 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 and, consequently, derives a competitive advantage. 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.

Summary.

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

Process: The key process challenges, as discussed above, are the pre-development 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.

Organization: As the need for coordination 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.

D. 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. The notion of product populations as studied by van Ommering (Ommering 2001) is a illustrative example of the independent deployment approach. 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 and that any deprecation of interfaces is handled in a structured way over a significant amount of time. 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.

When to apply. 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 frequently and require a shorter development cycle.

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.

Summary.

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 coordinate and execute on.

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.

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.

E. 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, in these cases, 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. Illustrative examples in the web 2.0 space include Salesforce.com¹ and Facebook². However, 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, especially as customer often have significant possibilities to match and mix, i.e. compose their own configuration of functionality.

When to apply. 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

¹ www.salesforce.com

² www.facebook.com

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 may fail due to internal organizational conflicts and mismatches.

Summary.

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.

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.

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.

Conclusion

Large-scale software development is complex, effort consuming and unpredictable and after decades of software engineering research we still have problems managing the constantly evolving complexity. Three trends are driving an acceleration of the complexity, i.e. software product lines, global development and software ecosystems. In this paper, we discuss the resulting problems, organized

around architecture, process and organization. The conclusion is that the case study companies described in this paper, as well as others that are not, typically employ a too integration-centric approach that causes significant inefficiencies in the system.

In this paper, we outlined five approaches to organizing large-scale software development, i.e. integration-centric development, release groupings, release trains, independent deployment and open ecosystem development. These approaches are, as the names indicate, organized from more integration-oriented to more composition-oriented. The collaboration models are summarized in table 1. Selecting the right inter-team software development approach is perhaps the most important lever for achieving high-quality, efficient and effective software engineering practices and results in virtually any software developing organization. Although our research suggests that most companies use an approach that is too much integration-oriented, one should not conclude that a more composition-oriented approach is always preferable. As discussed earlier in the paper, there are cases where a more integration-oriented approach is preferable.

Table 1 about here

The contribution of the paper is threefold: (1) we provided a detailed description of the three main trends that are driving and even accelerating complexity in modern large-scale software development. (2) We analyzed the architectural, process and organizational problems that an organization may experience based on intense data collection from action research when it applies the wrong approach to its software development. (3) We presented five approaches to large-scale software development, organized along a continuum ranging from integration-centric to fully composition-oriented. Although our research shows that the case study companies applied an approach that was too integration-oriented and would have benefitted from adopting a more compositional approach, this is not an absolute truth. Our recommendation is that each software development organization takes a conscious and informed decision to adopt the approach that is optimal for the particularities of the organization. When in doubt, however, typically the more composition oriented approach should be preferred as it offers better abilities to assign small teams to parts of the system and to adhere to short release cycles, allowing one to experiment more rapidly in the market place with real customers.

References

- Bosch, J. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, May 2000.
- Bosch, J. Maturity and evolution in software product lines: Approaches, artifacts and organization, In Proceedings of the 2nd software product line conference (SPLC), 2002.
- Bosch, J. Software architecture: The next step, in Proceedings of the first European workshop on Software Architecture (EWSA 2004). Springer LNCS, May 2004.

- Carmel, E. & Agarwal, R. Tactical Approaches for alleviating distance in Global Software Development. *IEEE Software*, 1(2), 22-29 (2001).
- Cascio, F. Wayne, S. Shurygailo, E-leadership and virtual teams, in *Organizational Dynamics*, vol. 31, issue 4, pages 362-376, (2003).
- Clements, P., L. Northrop, *Software product lines: practices and patterns*, Addison-Wesley. 2001.
- Conway M.E. (1968) How do Committees Invent. *Datamation*, 14 (5): pp 28-31.
- Herbsleb, J. D. & Moitra, D. (2001) Global Software Development. *IEEE Software*, 18(2), 16-20.
(HOF) http://www.sei.cmu.edu/productlines/plp_hof.html
- Kraut, R., C. Steinfield, A.P. Chan, B. Butler, A. Hoag. Coordination and Virtualization: The Role of Electronic Networks and Personal Relationships, in *Organization Science*, vol. 19, issue 6, pages 722-740, 1999.
- Larman, C. *Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2004.
- Linden, F. van der, J. Bosch, E. Kamsties, K. Känsälä, L. Krzanik and H. Obbink, *Software Product Family Evaluation*, Proceedings of the Third Conference Software Product Line Conference (SPLC 2004), Springer Verlag LNCS 3154, pp. 110-129, September 2004.
- Linden, F. van der, K. Schmid, E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, Springer Verlag 2007, ISBN 978-3-540-71436-1.
- Messerschmitt, D. G., C. Szyperski, *Software Ecosystem: Understanding an Indispensable Technology and Industry*, MIT press, 2003.
- Newcomer, E. & G. Lomow, *Understanding SOA with Web Services*. Addison Wesley, 2005. ISBN 0-321-18086-0.
- Nonaka, I. *The knowledge creating company. How Japanese companies create the dynamics of innovation*, Oxford University Press, New York, 1994.
- Ommerring, R. van, Techniques for independent deployment to build product populations, Proceedings of WICSA 2001, pp. 55 – 64, 2001.
- Powell, A., G. Piccoli & B. Ives, Virtual teams: A review of current literature and directions for future research, in *The Database for advances in information systems*, 35, 1, pp. 6-36, 2004.
- Randell, B. & J.N. Buxton, (Eds.), *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO (1970).
- Reason, P & H. Bradbury (Eds.), *Handbook of Action Research*. Sage Publishing, Thousand Oaks (2001).
- Sanwan, R., M. Bass, N. Mullick, D. J. Paulish & J. Kazmeier, *Global software development handbook*, CRC Press, 2006
- Service: http://en.wikipedia.org/wiki/Software_as_a_Service
- [SPLC] <http://www.splc.net/>
- Tarr, P., H. Ossher, W. Harrison & S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In Proc. 21st Int'l Conf. Software Engineering (ICSE'1999), pages 107 – 119. IEEE Computer Society Press, May 1999.
- Yourdon, E. & L.L. Constantine, *Structured Design*, Prentice-Hall, 1979.

VITAE

Jan Bosch (corresponding author) is VP, Engineering Process at Intuit Inc. Earlier, he was head of the Software and Application Technologies Laboratory at Nokia Research Center, Finland. Before joining Nokia, he headed the software engineering research group at the University of Groningen, The Netherlands, where he holds a professorship in software engineering. He received a MSc degree from

the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden. His research activities include software architecture design, software product families, software variability management and component-oriented programming. He is the author of a book "Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach" published by Pearson Education (Addison-Wesley & ACM Press), (co-)editor of several books and volumes in, among others, the Springer LNCS series and (co-)author of a significant number of research articles. He is editor for Science of Computer Programming, has been guest editor for several journal issues, chaired several conferences as general and program chair, served on many program committees and organized numerous workshops.

As a consultant, as a professor and as an employee, Jan has worked with and for many companies on strategic reuse in general and software product lines specifically, including Philips, Thales Naval Netherlands, Robert Bosch GmbH, Siemens, Nokia, Ericsson, Tellabs, Avaya, Tieto Enator and Det Norske Veritas. Around software product lines, he has published on, advised and implemented specific techniques and methods around, among others, software architecture, software variability management, the link to business strategy, organizational models, assessment frameworks, adoption frameworks and quality attributes. More information about his background can be found at his website: www.janbosch.com.

Petra Bosch-Sijtsema is currently a visiting scholar at Stanford University, USA at the Project Based Learning Lab, and a researcher at Helsinki University of Technology, Laboratory of Work Psychology and Leadership in Finland. She received her licentiate from Lund University (Sweden) and her PhD from the University of Groningen (The Netherlands) in Management and Organization. Petra has worked at universities in Sweden, the Netherlands, Canada, Finland and the US. Her research focuses on global distributed organizations and teams.

Tables

Table 1: Collaboration models for large global software development, from integration-centric to fully composition-oriented approaches.

Approaches	Integration-centric	Release grouping	Release trains	Independent deployment	Open (eco-) system development
1. Architecture	Strongly interconnected architecture	High integration within release grouping, high decoupling between groupings	High decoupling between components	High decoupling between components	Highly decoupled with sand boxes for third party functionality
2. Process	Continuous coordination between teams	Continuous coordination within grouping	Short iteration cycles; only coordination at	Each team selects length of iteration cycle	Each team selects length of iteration cycle

			start/end of cycle		
3. Organization	High interdependency teams	Teams responsible for different release groupings can be distributed	Distributed teams within organization	Distributed teams within organization	Distributed teams across organizational boundaries
4. Applicability	1. Release cycle long 2. Co-location of team	1. Geographical distribution of teams aligned with release groupings	1. Frequent releases beneficial for firm 2. High level of maturity needed	1. Different iteration frequencies for different parts of system 2. High level of maturity needed	1. Market approach 2. Teams highly dispersed 3. High level of maturity needed