

Chapter 15

Coordination Between Global Agile Teams: From Process to Architecture

Jan Bosch and Petra Bosch-Sijtsema

Abstract Traditional process-centric software development has served software-intensive companies well for decades. During recent years, however, the trends of increased adoption of software product lines, software ecosystems and in particular global software engineering have lead to unmanageable complexity and unacceptable overhead. In this paper we present research performed at three global companies in which we studied the relation between large-scale and agile approaches to software development as well as current problems. In addition, by integrating the best practices adopted at the case study companies, we present an alternative approach: architecture-centric software engineering. This approach largely removes inter-team dependencies and provides much higher efficiency and productivity in global software development contexts.

15.1 Introduction

For four decades now, software engineering continues to be a fascinating field. With Moore's law, the network law and the storage law doubling capacity every 18, 12 and 9 months, respectively, the size of the software systems on top of the hardware and communication networks is growing at similar rates. One can find examples of this within large Internet companies, e.g. around search engines, the IT systems supporting Fortune 100 companies and in the software ecosystems surrounding large platforms, ranging from PC operating systems to mobile devices. As a consequence, the

P. Bosch-Sijtsema is visiting scholar at Stanford University, Stanford, CA, USA.

J. Bosch (✉)
Intuit, Mountain View, CA, USA
e-mail: Jan@JanBosch.com

P. Bosch-Sijtsema
Aalto University School of Science and Technology, Espoo, Finland
e-mail: Petra@PetraBosch.com

D. Šmite et al. (eds.), *Agility Across Time and Space*,
DOI [10.1007/978-3-642-12442-6_15](https://doi.org/10.1007/978-3-642-12442-6_15), © Springer-Verlag Berlin Heidelberg 2010

scale of software systems increases with an order of magnitude about every decade and the architectural, tools, processes and organizational approaches need, to a large extent, be reinvented at the same frequency.

Over the last decade, we can see three main trends drive the increasing complexity of software development [6]. First, the widespread adoption of software product lines [3, 4, 8] causes increasing dependencies between different organizational units that earlier were independently developing their software products and services. Second, the increasing use of global software development teams, where the development of a large software system is spread over two or more continents. This is causing informal or more formal approaches to software process to become significantly less productive due to the inefficiencies of coordination over geographical, cultural and time zone boundaries [see e.g., 7, 11, 12, 16]. Third, there is an increasing popularity of software ecosystems [6, 13], i.e. a company providing a software platform and group of 3rd party developers that provide functionality on top of the platform. The factors complicating software development in this context include the lack of process mechanisms over corporate boundaries and the inherent tension between the interests of the platform company and the 3rd party developers. The focus of this paper is on the second trend, i.e. distributed and global development.

During the 1990s and the early 2000s, the complexity of software development was addressed through large software process efforts such as the Capability Maturity Model [SEI@CMI] that tried to formalize and standardize the development process to increase predictability of resources usage, time and quality. The negative implications of heavyweight process approaches were identified and acted upon by the agile software development community. Over the last decade, several agile software development process approaches have been developed, including XP [1], lean software development [14] and scrum [15, 17]. In the context of smaller scale software development projects, agile development projects have shown significant success. Inspired by the agile approaches, especially for web applications and services, software teams now focus on small team size, short release cycles, ranging from weeks to several times per day, and experimentation in the market place, i.e. the notion of perpetual beta.

Agile development has been widely documented [1, 2] as working well for small (<10 developers) co-located teams. From Agile software literature it becomes clear that agile teams work mainly co-located, have frequent face-to-face contact and highly motivated team members work in self-organized teams. Techniques such as pair-wise programming, daily standup meetings and sprint planning meetings are relying to a large extent on the team being co-located. As a consequence, agile development has shown success especially in small software development projects.

The key topic we address in this paper is the relation between large-scale and agile approaches to software development. All process approaches discussed so far assume what we refer to as an *integration-oriented* approach [6] to software development, i.e. system integration is a major and effort consuming part of the software development cycle as all system components need to be perfectly aligned with each other in order to provide the required system functionality. As a consequence, release cycles, size of software teams, the process overhead, etc. are increasing dramatically over time and grow exponentially with increasing system size. Although

there are application domains where systems need to be highly integrated and the consequences outlined above do not represent a competitive disadvantage, in most domains this is not the case.

The premise that we put forward is that although both traditional software process approaches and agile approaches propose mechanisms to deal with increasing scale of software systems, the fundamental problem is that the coordination cost of taking an integration-oriented, process-centric approach to software development is fundamentally flawed. Process-centric assumes people performing certain tasks as part of the process definition. The inherent assumption is that by formalizing the interactions within and between teams, the pitfalls found in less mature project organizations, e.g. unpredictability, major mismatches between components late in the lifecycle, etc. can be avoided. Experience shows that this is indeed the case, but the price that the organization pays for this is a degree of inefficiency that grows exponentially with increasing system size. The root cause of this inefficiency associated with large-scale software development is the coordination cost between all the teams and individuals involved in the overall software system. Whereas process approaches aim to structure and optimize these interactions and coordination efforts, the consequence is that the symptoms are addressed and not the root cause.

Of the three trends complicating software development that we discussed earlier, we believe that global software development, i.e. distributed development crossing geographic, cultural and time zone boundaries, are particularly affected by the issues discussed so far. This is because coordination efforts, in the end performed by humans, are even more costly in cases characterized by geographic distance, minimal overlap in working hours and cultural differences. Several examples exist where the coordination cost in a global context were a, if not the, major factor in the failure of a major software development effort.

The contribution of this paper is that we propose an alternative: rather than relying on process-centric coordination, we propose the use of the system architecture as a mechanism for coordination and outline how to achieve inter-team coordination. By basing software development on a software architecture that provides decoupling and simplicity, large-scale software development can provide the same efficiencies as small-scale development by providing individual teams, typically associated with a system component, ease of development, independent releasing of components of the system as well as allowing for easy incorporation of external developers and the components developed by them.

The remainder of the paper is organized as follows. In the next section, we discuss large-scale software development as well as a number of definitions. After that we present the case study companies in which we, primarily through participant-observer case study research, studied the challenges of large-scale software development. Subsequently, we discuss the problems of coordination in integration-centric software development approaches. In the next section, we define the architecture-centric approach to coordinating development teams. Finally, we conclude the paper.

15.2 Large-Scale Software Development

Although many development projects are small scale, many if not the majority of software engineers work in the context of large-scale software development. We define large-scale software development along three dimensions, i.e. size, team distribution and specialization. *Size* we define in terms of the number of individuals and teams. The number of individuals ranges from tens at the low end to hundreds or even thousands of engineers. Similarly, the number of teams ranges from a handful to tens or more than a hundred.

The second dimension is *distribution* of teams. We define three levels of distribution, i.e. local, distributed and global. We consider software development local if all teams are located at the same site and could, potentially, meet daily for face-to-face meetings. Distributed teams do not have the ability to frequently meet personally, but can compensate through technological means, e.g. telephone meetings, video conferencing, etc. to have synchronous (same-time, different place) communication. Global teams are located, as the name indicates, around the globe and have very few overlapping working hours during the day. Communication tends to occur primarily through asynchronous means such as email and file sharing. To illustrate the latter, the time difference between California and India is 11.5 or 12.5 hours, depending on the daylight savings schedule. As a consequence, global teams working on the same system have no overlapping regular working hours. Inter-team communication tends to be asynchronous, complemented with individuals at both sides organizing telephone or video meetings during early mornings and late evenings. In Fig. 15.1, we visualize the three types of team distribution.

The third dimension is the degree of *specialization*. In small-scale development, each team member, independent of the job title, is aware of virtually everything

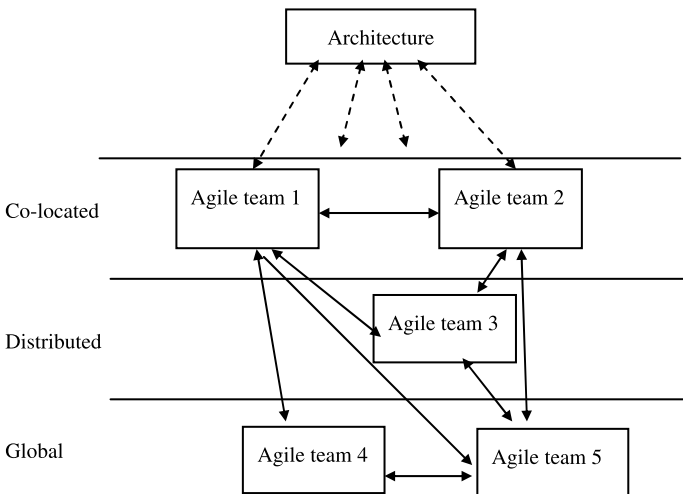


Fig. 15.1 Illustrating local, distributed and global teams

that is going on just by virtue of being part of the team. That, however, does not scale to large-scale software development. Consequently, individuals within the organization need to specialize into specific tasks associated with specific subsystems and information sharing becomes a formal activity with dedicated operating mechanisms associated with it.

Finally, throughout the paper, we use a number of concepts that require a more precise definition. **Coordination** is a consciously organized relation between activities and forces [10], work tasks are divided over actors and the act of is making different people or things work together for a goal or effect. For coordination a number of coordination mechanisms or instruments can be applied like direct supervision, standardizations and interaction or communication. Communication (synchronous, asynchronous and face-to-face) is an important mechanism used for coordination, but other mechanisms for coordination exist, including the use of the software architecture as a coordination mechanism that requires minimal communication between teams. We define **integration** as the *manual* process of combining the components into a working whole. We define **composition** as the *automated* process of combining components into a working system.

15.3 Case Study Companies

The research and approach presented in this paper is based on a participant observation methodology applied by the authors in numerous software-intensive system companies as well as in other industries. The participant observation techniques were applied per case study and individual case study analysis was performed. As a second step, the case study data were compared with help of comparative case study analysis methods [9]. Data was collected in three global organizations by participant observation, interviews and workshops over a period of 3 years per case company (see Table 15.1 for an overview).

15.3.1 Case Company GLOembed

Case company GLOembed 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. The case company does not work with agile teams as such, but has subsystem teams (building components) and product teams (who build products out of subsystems). The teams are primarily co-located, although some are global, and intra-team coordination is mainly performed through same-site and same-place communication and mostly through informal means. Coordination within the team is a relatively simple tasks shared by all team members. Inter-team coordination is performed through architects in whom the lead architect communicates all strategy related aspects to the globally distributed teams.

15.3.2 Case Company GLOtelcom

Company GLOtelcom 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 to 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 organization is transitioning to work more with agile teams (currently 30%). In these agile teams full component responsibility was assigned to a geographically local team in Asia. Development takes place in 2-week cycles; teams consist of 10–20 members and coordinate development efforts mostly through informal means. The head of the team and lead architect coordinate over geographical and architectural boundaries. The team has bi-weekly integration processes with HQ through central architecture teams, integration teams and product management teams. The inter team coordination involves a large amount of communication between many different teams and organization members and units.

15.3.3 Case Company GLOsoftware

Company GLOsoftware 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 million 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 works for 50% with agile teams and 50% with TSP/PSP teams, which are fully local (and co-located). It has new product development teams (who have no interdependency with other teams) and component teams in large established products in both Northern America and Asia. The teams are fully co-located in either the US or in Asia and have a local leader. Intra team coordination is performed by 4-week sprints and the normal agile coordination mechanisms such as daily stand-up meetings, product backlog, etc. Coordination between teams is performed centrally by the product management organization.

Table 15.1 Summary of the case studies

Summary of cases	GLOembed	GLOtelcom	GLOsoftware
Number of developers	> 1000	> 1000	> 1000
Domain	Consumer electronics	Telecommunication	Software development
Software development approach	No agile teams. Top down approach – Teams building sets of components – Product teams (set of sub systems of components build into product)	Transition to agile teams (+/-30%). Local teams in Asia with one remote team lead at Head quarters.	50% agile teams and 50% TSP/PSP teams – New product development teams (no inter team coordination) – Component teams in large established products
Size of development teams	20–40 team members	10–20 team members (agile)	5–10 team members (agile)
Location	Primarily co-located development teams Teams all over the world	Main development team co-located with remote team lead. Teams mainly in Europe and Asia	Primarily co-located development teams Teams mainly US and Asia
Coordination within team (intra team)	Teams primarily co-located, but some global. Coordination mainly through informal mechanisms	Co-located teams in Asia, 2-week development and informal coordination. Much contact with lead at HQ in Europe	Teams fully local (co-located) in either US or Asia, with local leader. Sprints of 4 weeks periods, daily stand-up meetings, product back-log, etc.
Coordination between teams (inter team)	Coordination and communication through architects. Lead architect communicated to all teams on strategy related aspects	Bi-weekly integration process. – Central architecture team – Integration team – Product management teams Many people involved.	Central coordination between teams by product management organization.

15.4 Coordination and Integration Inter-team Challenges

From our cases we found that the smaller (local) teams were able to coordinate their work rather efficiently and effectively as is confirmed by agile software development literature. However, the main problems we found in the case studies were challenges between inter-team communication and inter-team coordination especially for large-scale software development. These challenges can be placed on a continuum on which on one side local inter-team coordination is placed and on the other side of the continuum the global inter-team coordination is situated. The inter-team coordination challenges increase when teams have to coordinate over different time zones, cultures and countries (global).

Below we discuss the main problems we found from the case studies.

1. Top-down approach challenges or process-centric approach problems related to inter-team interaction.
2. Interaction problems.

15.4.1 Top-Down Approach Challenges

Process-Centric Coordination All three cases applied a process-centric approach for inter-team coordination for all phases of the software development lifecycle, including road mapping, requirements, dependency management during development, API evolution, integration and release management. Case study GLOembed applied a model in which only architects between the teams communicated with each other and a lead architect traveled to all the different team sites to communicate about the strategic plans and road maps. Case study GLOtelcom had local teams in India, but the lead architects were at headquarters. Furthermore, road mapping, product management and integration were done by numerous meetings that either took place in person at the headquarters, requiring all remote team representatives to travel, or through teleconferencing, requiring remote team members to attend outside work hours. Case GLOsoftware had a central organized inter-team coordination process lead by a central product management department who communicated to all the different component teams. All these teams were dependent on a central and top-down unit for inter-team coordination, which implied challenges in amount of communication (case GLOtelcom and GLOsoftware) and coordination needed for integration, and high dependency on one lead architect (GLOembed). In all cases, the amount of effort that was spent on non-value adding activities was very high and increasing over time as more and more items were identified that required collaboration between teams.

Integration Costs All three cases applied some sort of process-centric approach for coordinating work between teams for large-scale software development. However, we found that all cases had high and unpredictable product integration cost.

We observed in all case study companies that during product integration, incompatibilities between components are detected during system tests 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.

Coordination and Communication Costs Between Teams A problem observed in all case study companies is that when decoupling between shared software assets is insufficiently achieved, excessive coordination cost between teams are one outcome. 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. Case study GLOtelcom showed an example where communication and coordination costs were very high due to a large amount of integration meetings between all the different involved units for large-scale software development.

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. GLOtelcom and GLOsoftware, 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 inter-team coordination. 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.

Insufficient Pre-iteration Cycle Work In some of the teams in case company GLOsoftware, 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.

15.4.2 Interaction Problems

Global Interaction Problems Between Teams Interaction between global teams implies more challenges due to time zone differences, cultural and language differences and, often, different work practices. For example, in one organization that we

worked with, case company GLOtelcom, 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. Another example is case company GLOsoftware in which teams from the US cooperate with teams from Asia with a 12.5 hour time difference. Inter-team communication and coordination can only happen asynchronously or by traveling to the different locations to meet face-to-face. In GLOembed the lead architect travelled to all the global sites to visit the teams in person to discuss road mapping and strategic decisions.

Maintaining Motivation in Remote Teams In all case companies, we observed behavior at the main site of the organization that would keep the most interesting and strategic work at the main site and outsource the routine and less strategic work. In addition, there was a strong desire to maintain control over work that took place at the remote sites and to exercise that control through direct supervision of remote individuals and teams. This was caused both by a sense of protectionism at the main site, where work at the remote site was considered threatening. It also was a consequence of applying the same operating mechanisms that are applied locally, where frequent face to face contact is not experienced as supervision, in a global context where the interaction tends to become much more formal. The consequence was significantly reduced motivation and retention in the remote sites. This may turn into a self reinforcing system if work performed at the remote sites is of insufficient quality, or at least perceived to be, which further reduced trust in the main site to delegate work to the remote site.


Low Productivity In case study company GLOembed and GLOtelcom, the productivity of teams as well as of the overall system integration was very low in the cases where teams were internally distributed and where the coordination between teams was very process-centric with extensive coordination taking place during every phase of the lifecycle. Especially during systems integration, where the software assets from the various teams are brought together, many incoherencies were identified, despite the coordination efforts during the development process.

Table 15.2 presents a summary of the observed problems on inter-team coordination of both local teams compared to organizations with global agile teams that need integration between the teams.

15.5 Coordination Through Architecture

Throughout the chapter, we have presented the viewpoint that the root cause of the inefficiency associated with large-scale software development is concerned with the amount of coordination that is required between teams. The problems discussed earlier in the chapter are either a direct consequence of that root cause or can be traced

Table 15.2 Observed problems with process-centric coordination approaches between agile teams

Observed problems in inter-team coordination	Local		Global
Process-centric coordination	Relatively inexpensive due to largely informal, face-to-face communication. Broad interfaces between teams		High costs – Dependency on architects/central units for inter-team coordination tasks
Integration cost	Lower to medium cost. Productivity and outcome higher (faster)		High cost Low productivity
Communication & coordination cost	Lower communication and coordination costs. – Daily face-to-face or synchronous mediated interaction		High communication and coordination costs. Very costly – Inconvenience due to time differences – Quality of interaction lower – Technology solutions
Interaction problems	Interaction between teams easier because of close proximity, same time zone and similar language, culture and work practices		Problems with time zones, cultural and language differences, differences in work practice. Influence coordination and communication cost

back to it. Addressing this root cause is conceptually very simple: remove all need for inter-team coordination. That would allow small, agile teams to develop and re-release independently and increase efficiency of software development tremendously. However, the teams are still building solutions that are part of a larger system and therefore cannot be completely independent. The approach that we, based on our experience with Web 2.0 companies and software ecosystems, describe here is to move any remaining coordination needs from the process level to the architecture. This, in effect, replaces manual work with an automated solution.

The cost associated with process-centric coordination is much higher in a global context than in a local context due to the communication inefficiencies. Development approaches that rely on significant inter-team communication perform poorly in global and distributed contexts. The amount of coordination between teams can be reduced to a quite significant extent compared to what traditional software development approaches dictate. Below, we discuss the coordination needs for each stage of a traditional software development lifecycle.

15.5.1 Road Mapping

Traditionally, the road mapping process outlines high-level features and assigns these to releases of a large system. Assuming a release frequency of 6 to 12 months,

every release contains several new features. The road mapping process requires the organization to decide on the relative priority of the things that it could build. In order to decide on this, the effort associated with each high-level feature needs to be estimated. The effort estimations are naturally rather coarse and lack accuracy, which often affects the latter stages quite significantly.

The importance of an accurate ROI (return on investment) and effort estimation for each high level feature causes most organizations to involve people from virtually every function and team involved in the development, sales and deployment of the system. Especially for large systems, this often means that several tens of people are involved.

In the architecture-centric approach the organization translates its business strategy into a number of domains of functionality where it wants to see significant improvement. The teams take these domains as input for determining what to build in the next iteration. However, as discussed in the next section, the organization does not plan and order the exact functionality to be built but instead relies on the teams to optimize.

For the organization, it means giving up control and predictability in terms of the functionality delivered. However, it is important to realize that the notion of control and predictability tends to be an illusion in most companies.

15.5.2 Requirements

In traditional development, at the start of every iteration the high-level features assigned to this iteration are translated to more detailed system level requirements. These requirements are, in turn, translated to component level requirements. At this stage the overlapping with other activities starts in earnest as the process of translating system level requirements to component level requirements requires active involvement of the architects and team leads to make sure that the requirements allocation is appropriate and that the effort estimations are supported by the teams.

In the architecture-centric approach, there is no centralized requirement management process. Each team, which is associated with a component in the system, evaluates the domains in which progress is desired, complements that with its own customer understanding and announces to the organization what it intends to release at the end of the iteration. There is no coordination of requirements and there is a risk that more than one component team attacks related or similar functionality. On the other hand, because there is no coordination between teams, no effort was lost on non-value adding activities.

15.5.3 Architecture

The next activity in development is to determine the impact of the new requirements on the architecture and to design the changes to the architecture. This typically results in added and removed components, but the primary area of concern is often the impact on interfaces between existing components and, by extension, the teams responsible for these components.

As we discussed earlier in the chapter, in traditional software development, the architecture is often underspecified and teams are at liberty to develop interfaces between their components during development in mutual discussion. This may seem efficient as it allows for working in a decentralized fashion, our research at the case study companies as well as with other companies shows that architecture is the one area where discipline needs to be enforced. For every problem not handled by the architecture, a process coordination mechanism needs to be put in place to allow teams to release the system.

In architecture-centric development, component teams not only announce the requirements but also the changes to their component from an external perspective, including interfaces to be added, deprecated and removed by the end of iteration. A separate team manages the architecture, with a focus on compositionality and backward compatibility.

15.5.4 Development

The fourth activity is development. The case study companies had, to a significant extent, adopted agile development methods with four to six week development cycles. Ideally, development takes place in isolation from other teams so that each team can be as effective as possible. In practice, the teams need to spend a lot of time aligning their development effort with other development teams, test teams and the integration team.

The high coordination cost was caused by several of the issues discussed earlier in the chapter, but two of the key drivers were the lack of architectural specification and concurrent development of functionality. Teams spent too little time during the preparation of the iteration on analyzing and designing detailed changes to the component interfaces with the intention to “work it out” during the development cycle. Especially in global development this is particularly inefficient. The second main cause of coordination overhead is concurrent development. System-level features often require changes in multiple components and these changes typically have dependencies on each other. Concurrent development requires teams to interact during the development stage to work out compatibility issues and detailed assignment of responsibilities.

Architecture-centric development is concerned with facilitating independent development by component teams and to minimize the number of unproductive hours spent on coordination while maximizing the amount of productive hours. As the team has announced the interface changes, knows what backward compatibility is required, knows what functionality it wants to build and the other component interfaces to develop against, this stage should allow the team to focus solely on development. One of the principles that need to be enforced in this context is that no team can initiate development on functionality that is dependent on functionality that is under development by another team. Although this at first may seem to slow development as the implementation of a system level feature requires multiple iterations depending on the number of dependencies, in practice the removal of coordination cost and the short cycles for most agile teams outweighs any benefits that may be achieved by concurrent development.

15.5.5 Integration or Composition

In traditional development, the development of the next version of the components is followed by an integration phase. Here the fruits of the work of the various development teams are brought together and integrated in a product or platform release. As discussed in the problems statement, in the case study companies, the integration stage is very effort consuming and unpredictable. All case study companies used forms of continuous or frequent integration. However, the SCM (source control management) and test infrastructure did not allow for full coverage and hence the companies still used an explicit integration and validation phase before releasing the new product system to market.

The integration phase is especially painful in global software development as there is enormous need for interaction between the integration team and all of the component teams. During system testing, many issues are found that require collaborative resolution between teams. Although the amount of interaction needed may be limited, in global contexts there often are significant delays due to time zone differences, causing many issues that could be resolved in minutes or a few hours to become part of a daily rhythm instead.

In architecture-centric development, there is no integration phase, but instead the system is focused on composition. Each component team releases frequently, but uncoordinated with other teams. When a component team releases, its component has to pass the automated SCM and test system. The automated test system is improved in response to any problem that manages

to get through the system and is only surfaced after deployment. As a consequence, over time the quality of the validation reaches a very high level. The traditional approach is to put process steps in place to avoid problems to occur, but this requires coordination and manual effort. This additional focus on the automated SCM, test system and deployment infrastructure removes the need for an unpredictable and effort consuming integration phase and allows teams to release their components independently.

15.5.6 Architecture-Centric Software Engineering

Architecture-centric software engineering focuses on minimizing the inefficiencies associated with traditional process-centric development. The approach adopts a set of principles that is different and often initially uncomfortable in corporate contexts. However, there is of course a clear parallel to the development approaches found in the open-source software communities.

The key enabler for architecture-centric software engineering is to minimize dependencies between components. Although this central to architecture design, architects often de-prioritize decoupling to achieve other attributes. In [5], we present the notion of software ecosystems where architecture decoupling is paramount for its success. The principles it introduces are valuable in this context as well.

The concerns in a corporate context are often related to the loss of control over R&D investment, resource allocation and product roadmaps. Our experience from the case study companies as well as other organizations is that the perception of control often is an illusion. Either the R&D organization operates at such a low expectation level that any organization can meet it, or plans and milestones are frequently missed in unpredictable ways.

Architecture-centric software engineering removes so many inefficiencies from the software development process that the output of the organization is much higher, even if senior management has less visibility into the operational issues in the R&D organization.

Although none of the case study companies has implemented all aspects of the architecture-centric software engineering approach, each employs some of the practices. The consequences of globalizing their software development while interested in adopting more agile development approaches necessitated each of the case study companies to change some of their, initially process and integration-oriented, practices and adopt a more architecture-centric approach. Based on our research at these companies, Web 2.0 companies and in the context of software ecosystems, we are convinced that the presented approach provides enormous benefit to organizations that adopt it.

Practical Tip: Illustrate the lack of predictability in large-scale software development by collecting and analyzing historical data. In most companies, there is a significant gap between plan and outcome. This data can then be used to break the illusion of control and to create an opening for experimenting with a new approach. Once the experiment is approved, make sure to deliver real business value as soon as humanly possible and collect data on relevant metrics, e.g. productivity or time to customer of new functionality. Select comparable development efforts using the traditional approach to support the transition from a belief that the new approach is better to a quantitatively supported position that the new approach is superior.

15.6 Conclusions

Over the last four decades, software engineering has continuously evolved to address the continuous and enormous increase in complexity due to sheer system size, the complexity of the application domains and level of interaction required with other embedded and IT systems. The case study companies reported on in this paper have been very successful in applying traditional software development approaches to their product development and have, as a consequence, seen significant growth.

With increased globalization of software development and the increasing popularity of agile software development approaches, it has become blindingly obvious that a process-centric approach to large-scale software development over time results in unmanageable complexity and unacceptable inefficiency. In the paper, we discuss several problems, categorized in four categories, i.e. process-centric coordination, integration cost, communication and coordination cost and interaction problems. These problems can largely be attributed to one root cause: dependencies between components in the architecture and the teams responsible for these dependencies.

Based on our research with the case study companies, but also with a several other companies as well as software ecosystems, we propose an alternative: rather than relying on process-centric coordination, we propose the use of the system architecture as a mechanism for coordination and outline how to achieve inter-team coordination. By basing software development on a software architecture that provides decoupling and simplicity, large-scale software development can provide the same efficiencies as small-scale development by providing individual teams, typically associated with a system component, ease of development, independent releasing of components of the system as well as allowing for easy incorporation of external developers and the components developed by them.

The contribution of the paper is twofold. First, it presents the results of a case study into the implications of applying process-centric, integration- approaches in large-scale oriented software development based on longitudinal case studies at

three large organizations. Second, it presents architecture-centric software engineering as a novel approach that combines the best practices from these companies, as well as from companies in the Web 2.0 and software ecosystem industries.

References

1. Beck, K. (1999). *Extreme programming explained: Embrace change*. Boston: Addison-Wesley.
2. Boehm, B., & Turner, R. (2004). *Balancing agility and discipline: A guide for the perplexed*. Boston: Addison-Wesley.
3. Bosch, J. (2000). *Design and use of software architectures: Adopting and evolving a product line approach*. London: Pearson Education (Addison-Wesley & ACM Press).
4. Bosch, J. (2002). Maturity and evolution in software product lines: Approaches, artifacts and organization. In *Proceedings of the 2nd software product line conference (SPLC)* (pp. 257–271), San Diego, USA, 19–22 August 2002.
5. Bosch, J. (2009). From software product lines to software ecosystems. In: *Proceedings of the 13th international software product line conference (SPLC 2009)*, August 2009.
6. Bosch, J., & Bosch-Sijtsema, P. M. (2010). From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, 83, 67–76.
7. Carmel, E., & Agarwal, R. (2001). Tactical approaches for alleviating distance in global software development. *IEEE Software*, 1(2), 22–29.
8. Clements, P., & Northrop, L. (2001). *Software product lines: Practices and patterns*. Boston: Addison-Wesley.
9. Eisenhardt, K. M. (1989). Building theories from case study research. *Academy of Management Review*, 14(4), 532–550.
10. Hatchuel, A. (2001). Coordination and control. In A. Sorge & M. Warner (Eds.), *The IEBM handbook of organizational behavior* (pp. 320–339). London: Thompson Business Press.
11. Herbsleb, J. D., & Moitra, D. (2001). Global software development. *IEEE Software*, 18(2), 16–20.
12. Kraut, R., Steinfield, C., Chan, A. P., Butler, B., & Hoag, A. (1999). Coordination and virtualization: The role of electronic networks and personal relationships. *Organisation Scientifique*, 19(6), 722–740.
13. Messerschmitt, D. G., & Szyperski, C. (2003). *Software ecosystem: Understanding an indispensable technology and industry*. Cambridge: MIT Press.
14. Poppendieck, M., & Poppendieck, T. (2003). *Lean software development: An agile toolkit*. Boston: Addison-Wesley.
15. Rising, L., & Janoff, N. S. (2000). The Scrum software development process for small teams. *IEEE Software*, 17(4), 26–32.
16. Sanwan, R., Bass, M., Mullick, N., Paulish, D. J., & Kazmeier, J. (2006). *Global software development handbook*. New York: CRC Press.
17. Schwaber, K. (2001). *Agile software development with Scrum*. New York: Prentice Hall.