Global Software Development: Are Architectural Rules the Answer?

Viktor Clerc, Patricia Lago, Hans van Vliet
Department of Computer Science
VU University Amsterdam, The Netherlands
{viktor, patricia, hans}@cs.vu.nl

Abstract

Global software development (GSD) faces additional challenges as compared to single-site software development. Some of the better known challenges include temporal, geographical, and socio-cultural differences. To overcome these challenges, organizations need to revert to measures in order to deliver software in time in a distributed setting. Some of these measures may exist in the form of architectural rules: principles and statements on the software architecture that must be complied with throughout the organization. From the GSD literature we distilled four main GSD challenges and seven sub-challenges, or issues. For each issue, we list possible solutions and observe that solutions to GSD challenges may be obtained by adhering to architectural rules. We present a study on how two organizations involved in GSD solve the GSD challenges and issues. One of the organizations mainly uses rules regulating the architecture of the product. The other organization does not emphasize these architectural rules but rather focuses on the joint team effort in establishing and committing to measures that mainly pertain to the architecture process. We conclude that rules regulating a combination of both proves valuable in handling GSD challenges.

1. Introduction

Global software development (GSD) faces challenges additional to those of single-site development. The primary obvious difference is that software engineering practices are performed at geographically separate locations. This difference introduces challenges that have been reported in the literature [17, 20], such as temporal, strategic, and socio-cultural challenges [1]. Some promising solutions to address these challenges have been identified, e.g. in [18, 23], but their real contribution cannot yet be fully distilled from practice because empirical evidence is lacking.

Some of the challenges in GSD originate from the architecture of the software. The architecture can induce dependencies between software engineering tasks such as managing synchronization and meeting a release schedule (e.g. [4, 8]). As with single-site development (e.g. [6]), architectural rules can help to overcome some of these challenges in GSD. *Architectural rules* are principles and statements about the software architecture that must be complied with throughout the organization [7]. Where architectural rules concern the architecture as a product, additional measures pertaining to the processes may be necessary for successfully implementing that architecture.

Although using architectural rules seems beneficial for a distributed setting, we lack detailed insight into what kind of architectural rules are generally applied in a GSD environment. In addition, we do not know in what way these architectural rules contribute to overcome challenges specific to GSD.

In this paper, we provide an overview of the major GSD challenges as mentioned in the literature, and refine them into seven issues. For each issue, we list possible solutions and describe to what extent these solutions can be expressed as architectural rules. Next, we use this overview to study two organizations involved in GSD. We determine what practices are used to overcome the various issues. Our study reveals that architectural rules pertaining to the product (i.e. the structure of the software) and additional measures on the process are relevant. For example, allowed dependencies between subsystems may be defined in architectural rules, but so need the processes to verify compliance with these rules.

This paper is structured as follows. Section 2 provides an overview of related work in the field of software architecture, architectural rules, and global software development. Section 3 provides an overview of the challenges that exist in GSD. This section also addresses possible solutions to overcome these challenges as identified in the literature. Section 4 shows two organizations overcome these challenges in practice. In this discussion, we focus on the contribution of architectural rules to the handling of GSD issues. Finally, Section 5 lists our conclusions and provides directions for future work.

2. Related Work

This section provides related work in the field of software architecture, the concept of architectural rules, and general challenges identified in GSD literature.

Recently, software architecture is regarded as the set of architectural design decisions [21, 22]. Examples of these decisions include preferred or mandatory standards on communication between layers of the architecture or datastructure conventions. Those architectural decisions that must be complied with throughout the organization are defined as *architectural rules* [6, 7]. Architectural rules can help to overcome challenges in software engineering and software architecture [6]. However, we lack insight into *what* rules are necessary to guide GSD.

A software architecture imposes coordination and communication efforts between development teams that are necessary to successfully implement the architecture. To minimize these efforts, the software architecture often mimics the structure of the organization: a certain modular architectural design is used as the basis to distribute development work across different teams [8]. With GSD, the challenge to distribute work becomes even bigger. Development work is spread across development sites, hence demanding more coordination and communication efforts [16].

As [7] shows, having a description of the software architecture and rules on the architecture alone is not enough to ensure successful compliance in GSD. These rules should be accompanied by additional process guidelines and activities to let them sink in properly. These additional guidelines and activities pertain to the use and *personalization* of architectural rules as opposed to *codification* that often receives most emphasis [10].

Substantial work has been done to provide insight in the problems or challenges that occur when spreading development efforts across multiple development sites [14, 18]. Several dimensions of the problem with physical separation of software engineering practitioners are listed by [17]: technical problems, project management problems, knowledge management problems, and communication problems. Socio-cultural, temporal, and geographical issues are further exemplified in [1, 20].

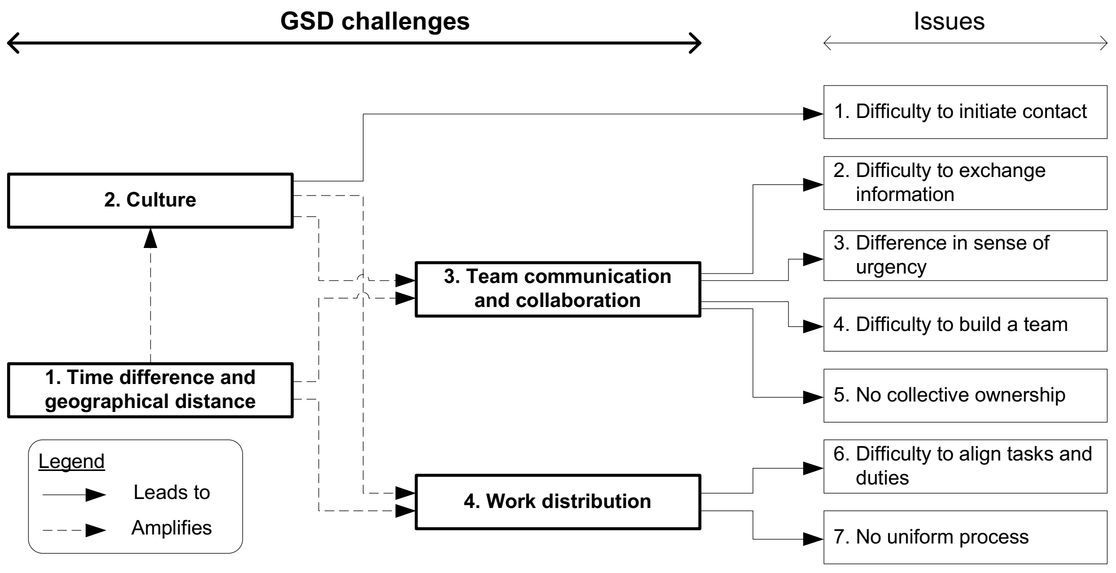
Some solutions to overcome the challenges are also addressed in the literature. Herbsleb and Grinter [14] report on several solutions to overcome the distance innate with GSD, such as attending to Conway's Law [8], travelling at the start of projects to get to know relevant individuals from multiple sites up-front, and recording architectural design decisions including their rationale. Corry et al. [9] describe a technique in which architects frequently travel between development sites to engage developers in software architecture work. Bass [3] proposes an approach in which collaborating practitioners first acquire a mental model of the software engineering task they need to perform ([11]), and only then pursue completion of the task. The significance of a shift in perspective is signalled in [3]: project management should pay more attention to coordination problems and applied synchronization efforts. Other work of Bass et al. ([4]) relates the required communication and coordination efforts in software engineering ([16]) to the organization's capabilities. Problems occur when the required communication and coordination efforts exceed the organization's capabilities. This results in misalignment between the architectural design decisions that have been taken and the organizational structure [4]. Since communication and coordination efforts are more important at an organization involved in GSD, these efforts place additional requirements to the organization's capabilities.

3. Challenges in Global Software Development

In this section, we list four GSD challenges we have distilled from the literature (see sections 3.1 through 3.4). We took proceedings from relevant conferences and special issues of magazines, both on the topic of GSD, as a basis. We next identified the issues described in these contributions and grouped and related them when possible. Figure 1 provides an overview. For some of the challenges, we make a further subdivision, to finally end up with a list of seven *issues*. As the figure shows, challenges can amplify other challenges. The remainder of this section describes these challenges, issues and interrelations. For each issue, examples of possible solutions are given. We primarily focus on solutions that pertain to the architecture, but mention other solutions when these are not available.

3.1 Challenge 1: Time Difference and Geographical Distance

The first challenge in GSD originates from the innate difference with GSD as compared to single-site software development: working at geographical distant sites and the time difference incurred. Geographical distance and time difference burden software engineering activities. Software engineering activities are knowledge-intensive tasks and therefore require communication during various phases in the lifecycle [16]. Time difference leads to delays because of less overlapping working hours [14]. Furthermore, geographical distance leads to delays because it increases the unresponsiveness of practitioners. This challenge amplifies that of culture (*Challenge 2*), as empirically supported by [16].



GSD challenges
Issues
2. Culture
1. Time difference and geographical distance
3. Team communication and collaboration
4. Work distribution
1. Difficulty to initiate contact
2. Difficulty to exchange information
3. Difference in sense of urgency
4. Difficulty to build a team
5. No collective ownership
6. Difficulty to align tasks and duties
7. No uniform process
Legend
Leads to
Amplifies

Figure 1. Overview of challenges and their interrelations.

3.2 Challenge 2: Culture

GSD involves software engineers, software architects, and other practitioners, possibly from all over the globe. People with different cultural backgrounds are required to cooperate in order to deliver working software in time. As shown by [19], people with different cultural backgrounds behave differently. We have examined literature on cultural challenges and identified the issue that is regarded as key for supporting software engineering practices in a GSD setting:

1. Difficulty to initiate contact

Establishing contact between practitioners of different cultures faces additional challenges as compared to establishing contact between practitioners of the same culture [19]. Moreover, the challenge of culture is amplified when project members located at different development sites want to initiate contact [14]. Examples of cultural problems include limitations in the vocabulary of practitioners [20] and difference in communication style: reluctance to ask questions [18], direct versus indirect communication [15], or giving preference to sending an e-mail over establishing contact directly by phone [14].

The main solution to overcome this cultural issue may be to establish face-to-face contact and to get to know "who is who" in the project [13, 18]. Suggestions that can help to overcome this issue are *1)* establish a directory listing all practitioners involved in the development of certain parts of the architecure and *2)* start all projects with a kick-off meeting. Establishing contact is easier when practitioners are located at the same development site. Once contact has been initiated, practitioners are more willing to overcome their cultural differences in order to communicate effectively.

The cultural challenge described in this section influences or *amplifies* other GSD challenges, such as communication challenges, collaboration challenges, and work distribution challenges.

3.3 Challenge 3: Team Communication and Collaboration

The nature of GSD results in increased difficulties to communicate and collaborate within teams and between teams involved in software development. We distilled the following issues from literature addressing this challenge:

2. Difficulty to exchange information
3. Difference in sense of urgency
4. Difficulty to build a team
5. No collective ownership

Difficulty to exchange information
Having teams at geographically separated sites results in a lack of unplanned, informal contact during which seemingly irrelevant but in fact highly valuable information is exchanged [14, 16]. The information exchanged during unplanned social meetings is not necessarily *architectural* in nature, but can help to more easily obtain architectural information. Moreover, unplanned social meetings help people to build awareness of what will happen before a formal decision is made. In particular, awareness of architectural design decisions is important, since these decisions

may have a high impact across development sites. In addition, the difficulty to exchange information arises at an early stage in the project, because practitioners just do not know whom to contact at a remote site; they simply don't know the practitioners at a remote site (see *Challenge 2*).

A number of suggestions to lower the threshold to exchange information across different development sites have been provided in the literature. These suggestions include providing incentives to improve collaboration and knowledge sharing. To improve collaboration, teams of experts or communities of practice [18], in which *gurus* can flourish, can be established. Incentives for architectural knowledge sharing include the establishment of social ties and knowledge internalization [12]. Other suggestions to address the difficulty of exchanging information include traveling to different development sites at an early stage in the project and meeting various practitioners at these sites. As a result of the face-to-face contact, practitioners experience that there actually is "a person behind an e-mail address" [18]. This effect can be achieved by processes and policies regulating the frequency, location, and participants of the required inter-site meetings.

Difference in sense of urgency
Multi-site software development can result in a difference in the sense of urgency across development sites to handle specific requests. A difference in establishing contact on a certain topic (e.g. practitioners prefer to send someone an e-mail over calling that person) influences the sense of urgency to handle that topic. As with the previous issue, getting to know practitioners from other sites is key to overcoming this issue. Once a key individual is already known to practitioners at another site, these practitioners are more willing to pose a question to that person. This person can even become a *liaison*, or first contact point for practitioners at the other site [17]. Communication is sped up, activities are agreed upon, and a collective sense of urgency towards the activities that need to be performed is achieved. This collective sense of urgency, in turn, will shorten development time. Measures that focus on the development process can be of value in this case. An architectural rule could make explicit that every project, site, or subsystem should have a liaison. Additional information on how, when, and where to contact this liaison could be provided as well.

Difficulty to build a team
Now that we have seen possible solutions to address the difficulties of exchanging information and the difference in sense of urgency, we face another difficulty: it is difficult to build up a single, virtual team across development sites because this is hampered by the geographic distance and time difference [16] (see *Challenge 1*). A single, virtual team is characterized by low communication thresholds and high collaboration possibilities. Not having such a team results in possible mismatches in terminology and definitions and consequently, communication overhead and delays [17].

A possible solution to the issue of building a team is to develop architectural rules that contain conventions and procedures on collaboration and the use of a single, shared environment (e.g. when to check in source code, how to build and name a release). These rules then are disseminated across the involved development sites.

In order to build a single, virtual team the effectiveness of communication across development sites should be high. Two possible approaches to improve effectiveness are discussed below:

- First, as a preventive measure, it is possible to reduce the amount of communication across development sites by aligning the organizational structure with the software architecture [8]. This requires that the architecture is fixed (or, *sufficiently* fixed) to distribute the work and have development sites operate in parallel. Section 3.4 will further delve into this subject.

- Second, the use of collaboration tooling can help to create a single, shared environment across development sites. A single, shared environment can improve the efficiency of the communication between different groups that is anyhow needed, despite possible communication reduction efforts. Such an environment should form the entry point to revert to when communication across development teams is required. The environment should provide an overview of "who is who" in the teams at the different development sites. At a minimum, contact information, role and responsibilities, and a mugshot [18] should be included in the overview. In addition, these tools need to provide awareness of the availability of practitioners at other development sites. Collaboration technologies such as instant messaging, a wiki, or message-boards can lower the threshold of actually establishing contact and communicating with another site. The most appropriate tooling for a given situation depends foremost on the time difference that exists between development sites. The tools should allow architects and other practitioners to record design decisions [14] in an easy and accessible way.

  One word of warning, though. Although the use of collaboration tooling adds value, they are not a panacea. Face-to-face communication (ad hoc) remains essential for successful global software development [14].

Architectural rules can help to overcome the challenge of building a team. First of all, a list of all employees involved in the project can be provided, along with their responsibility towards parts of the architecture. Second, the architectural rules can provide conventions on the use of collaboration tooling, by addressing what information is stored, where it can be found, and who is responsible for keeping the information up to date.

No collective ownership

When GSD organizations have dedicated owners of parts of the source code, this introduces delays in corrective maintenance. When it is not possible for practitioners to modify the source code that is not owned by them, additional (formal) communication with the owner becomes necessary to acquire a common understanding. In addition, it becomes necessary that the practitioner is able to transfer the sense of urgency he has to the owner of the source code.

The problems described above can be addressed by introducing collective ownership of (parts of) the source code and other documents. Collective ownership of source code and other documents means that all individuals collaborating in a software project (and possibly located at multiple development sites) can work on any model or artifact in the project [2]. Often, collective ownership is supported by having one shared configuration management system with a single source code tree that is accessible [18]. As such, collective ownership reduces the view of "them against us" often experienced in multiple teams [16]. For large software development projects this could lead to collective ownership within sub-teams (consisting of employees of all development sites) that work on a designated part of the system.

The possibility of changing source code freely should be accompanied by communication guidelines, a sound test set, and frequent builds. Communication guidelines at a minimum should involve contacting other known users or developers of the source code and informing them that a change is being made. The test set ensures that a change does not introduce bugs in other parts of the source code or introduces conflicts with architectural rules. Frequent builds address the possible delay between carrying out a change and verifying the correct behavior of the resulting source code [18]. Aforementioned topics need to be communicated uniformly across the GSD organization. Architectural rules can help to do that by defining the mandatory policy on these topics.

3.4 Challenge 4: Work Distribution

In GSD, different teams from across the globe need to deliver working software in time. In addition to the communication challenges experienced by these teams (see *Challenge 2*), the organization of the teams itself and the variety of software development activities they need to perform play a significant role. We identified the following two issues in the literature:

6. Difficulty to align tasks and duties

7. No uniform process

Difficulty to align tasks and duties

The software architecture serves as a basis for distributing work. Ambiguities in the software architecture or frequent replanning of tasks leads to a lack of insight into the interdependence of tasks of the teams involved [23]. In addition, the distribution of work is hampered when engineering tasks are not understood. This all results in long discussions that add up to the delays already experienced in GSD [16]. Furthermore, [16] shows that the amount of discussions generally does not decrease in the course of the project.

A possible solution to this lack of understanding of tasks is to distribute tasks and work only when the architecture is stable enough. Architectural rules could specify when such is the case. For example, architectural rules could describe that the architecture description should elucidate how all 'must-have' requirements are addressed and that an independent verification of the architecture (by the quality assurance team) should have taken place.

No uniform process

It is essential to have a uniform strategy towards software engineering processes and team processes. A development organization that has different strategies towards these processes experiences delays [23]. An example of different strategies towards software engineering processes can occur during integration. Different approaches towards this integration phase exist: *1)* a dedicated integration team accepts all subsystems and is responsible for the integration, or *2)* integration is the responsibility of all teams involved in delivering the subsystems that need to be integrated. When an organization chooses to have a dedicated integration team that integrates the subsystems, the integration team needs to assure itself of the correct working of the subsystems supplied. Furthermore, when issues arise, members of the development teams should stand by to provide assistance. Although this solution is less resource-consumptive, it is hampered by time differences and causes delays because practitioners located at different sites communicate less effectively [16]. Especially at integration time, the need is high for fast-paced interactions to quickly solve bugs once they are discovered [18]. Consequently, it can prove beneficial to have a dedicated team consisting of representatives from the different development teams at a single location integrate the subsystems into the desired release.

The example above shows that it is of paramount importance to have a uniform description of how integration should occur in projects. A non-uniform description of the integration process leads to delays because of ambiguity in task description. Similarly, process guidelines should exist for other software development processes, such as build processes and change management processes. Mullick et al. [23], for instance, make note of broken builds because of a lack of a uniform configuration management strategy. Examples on this topic are also given by [14], reporting on problems when not having single-branched [18] distributed change management across development sites.

Processes are an effective means to organize teams and their interrelationships and to distribute and communicate work that needs to be done [14]. Organizations need to put emphasis on team organization and communication processes in addition to software engineering processes. The team organization and communication processes need to focus on a communication structure (including a description of the roles and responsibilities of the various teams) and interactions between various types of teams (including the knowledge they exchange, at what stages or milestones in the development process they interact, and the frequency of interaction [23]). Especially, the tasks and role of the architecture team should be exemplified so that all practitioners understand this pivotal role. Although these process are not specifically aimed at catering for unplanned and informal communication, they can provide insight in the responsibilities of teams and the communication strategy across teams.

In conclusion, organizations involved in GSD need to focus on a set of processes aiding team organization and work distribution. These processes can have a technical topic, such as integration, branching, and releasing, as well as topics like work distribution and communication structure. Having these processes for software engineering and team organization and communication alone is not enough, though. In addition, it is necessary to communicate the processes consistently [23] and to ensure that the processes are understood and followed.

4. Organizations' Solutions to GSD Issues

Using the list of issues presented in Section 3, we studied two organizations involved in GSD. The aim of this study is to see how these organizations use architecture and architectural rules to overcome the issues. We conducted semi-structured interviews with two architects at each organization to obtain responses to the issues. In addition, we used the information collected in a previous study [7]. We verified our interpretation of the answers with the interviewees. For each organization, we first give a general impression of that organization. Next, we delve into the seven issues and show how the organization addresses each issue.

4.1. Organization A

Organization A uses five sites located on separate continents to develop software for different consumer electronics products. The development of each product is done in a project, in which a number of subsystems need to be integrated. Each of these subsystems is developed by a subsystem team located at a single development site. The subsystem team owns the source code of that subsystem. Subsystem teams consist of a subsystem architect, a configuration manager, an event manager, and several software engineers. Integration of the subsystems into the final product is done by a dedicated team located at the main development site. In total, about 300 employees work in the various teams.

The organization uses a product-line architecture to support various projects in which the consumer electronics products are developed. The architecture is maintained by a central architecture team located at a single development site. The architecture team maintains the architecture, by a.o. describing architectural rules in small, text-based documents. These rules only cover issues that exceed the individual subsystems. Issues pertaining to individual subsystems need to be addressed by subsystem architects. This results in a certain degree of freedom for subsystem teams.

1. Difficulty to initiate contact – As a general policy of Organization A, architects visit the largest remote development site one week each month. Other key individuals, such as senior software engineers and members of the integration team, do not travel that much. Practitioners indicated that this travel policy did not overcome the lack of trust as part of cultural challenges completely. Recently, Organization A was conducting a transfer of software development activities to the largest remote development site and retained the architecting activities at the main development site. This resulted in a lack of motivation of some of the employees at the remote development site because they felt this would decrease their influence on the major design decisions.

2. Difficulty to exchange information – Organization A uses a collaboration infrastructure that provides a detailed overview of the teams involved in a software development project, including the roles, responsibilities, and mugshots of each team. The collaboration infrastructure is linked with the configuration management infrastructure: information on builds, releases, and problems is extracted from the configuration management system and published on each subsystem team's website. Together with a description of the members of the team and the team organization, this website is one of the primary sources of information for other subsystem teams. Besides using the website to obtain information, the organization uses e-mail communication and instant messaging technology regularly to allow discussions between the subsystem teams and between the architecture team and subsystem teams. As a matter of fact, Organization A regards team composition and team contact details as the most important information. Aforementioned collaboration infrastructure lowers the threshold for exchanging information at Organization A. However, other issues with GSD still burden fully effective exchange of information.

3. Difference in sense of urgency – The organization experiences difficulty in maintaining a shared sense of urgency, judging by an example given: *"when programming* *errors or compiler warnings occurred, it was difficult to get practitioners from the involved development site to react on this warning. When this succeeded, practitioners from that development site did nearly everything to just remove the warning message, hardly paying attention to the actual semantics of the implemented solution. Practitioners from the main development site actually dug into the warning message, tried to understand the root cause of this message, and removed the root cause."* Although the latter approach took much time, Organization A was much more confident that this approach, once used throughout the organization, would be preferable.

4. Difficulty to build a team – Organization A has a nearly exact mapping of the organizational structure to the architecture. For all subsystems defined in the architecture, a subsystem team exists. In addition, architectural rules are the responsibility of the architecture team. Some of the architectural rules of Organization A define allowed dependencies between subsystems; these dependencies closely resemble the communication processes in place. In conclusion, the architecture determines the team structure so that the definition of teams is fixed. This leads to increased formality in the communication between these teams.

5. No collective ownership – The configuration management infrastructure in place at Organization A supports distributed change management. Each subsystem team is owner of the source code of that subsystem. Consequently, the ownership of the source code is distributed. Communication across subsystem teams on source code topics is structured via change requests and problem reports, and formalized in architectural rules. The organization experiences this as highly formal and inadequate; the major issues were discussed using other channels than change requests. As a result, change requests underwent delays and did not always represent reality.

6. Difficulty to align tasks and duties – Alignment of tasks and responsibilities is mainly done via the architecture. Requirements are assigned to subsystems, which have dedicated resources assigned. Therefore, it is clear what activities will be performed by what subsystem teams, even by what practitioners within those teams. Because these tasks and responsibilities are defined up-front, it is necessary to conduct formal verifications to determine whether reality is in line with the rules. Nevertheless, the tasks performed by the subsystem teams are not reviewed regularly by the architecture team nor the quality assurance team.

7. No uniform process – When we observed the architectural rules at Organization A [7], we noticed that Organization A has a strong emphasis on architectural rules from a configuration management perspective: naming conventions, coding guidelines, and process guidelines on releasing, integrating, and deploying source code are regarded as important by the organization. These guidelines pertain to the use of a single configuration management system. In addition, we observed much emphasis on software engineering processes and communication infrastructure. What lacks, are detailed guidelines on inter-team communication and collaboration and a sound mechanism to disseminate the most important architectural decisions. Especially when unplanned, social contact is hard, it is essential to establish and announce communication guidelines between teams to disseminate the essential (architectural) information, such as the major architectural decisions. This prevents subsystem teams from feeling put aside and trailing on reality, as was the case at Organization A.

Although Organization A has guidelines pertaining to the software engineering processes as described above, the organization often deviated from these guidelines. This results in e.g. ambiguity on the integration process; the guidelines describe that all subsystems were delivered to a dedicated integration team, responsible for the full integration to the final software product. Practice shows a more staged approach towards integration. Furthermore, the visibility of the architecture team is decreasing because of a lack of resources and the perception that the team was trailing on projects. Verification of compliance with the architectural rules is not implemented at Organization A. As a result of this, multiple processes co-exist in practice.

4.2. Organization B

Organization B develops business administration systems for various customers. Software development occurs by one development team spread across two sites at different continents. Organization B does not have dedicated architects – rather, architecting is a joint effort by the complete development team. For over one year, Organization B uses these two development sites. In total, about 100 employees work in projects at these two development sites.

Although Organization B acknowledges that the differences between its customers may result in different architectural solutions, experiences have led to a certain convergence on the architecture that can be used most often. The software architecture consists of four layers which are highly decoupled by using principles like inversion of control (see e.g. [26]), and is supported by an extensive toolset aiding Organization's B development approach.

The development approach of Organization B uses several practices from the agile development domain [2]. Examples of these practices include pair programming and test-driven development. Test-driven development [5] serves as a good basis for performing integration activities. Integration is done continuously by running all tests automatically and providing a response to the team.

1. Difficulty to initiate contact – In setting up the activities at the remote development site, specific attention was paid to selecting employees with corresponding work ethics and attitude. Furthermore, the management of the main development site often traveled to the remote site with several key individuals. This traveling resulted in getting to know the people at the other site, and establishing a shared approach using exactly the same methodology and tooling for software engineering. Every employee hired at the remote development site visits the main development site within two months. Finally, employees from the remote development site regularly visit the practitioners at the main development site to exchange ideas and have face-to-face discussions on project-specific matters.

2. Difficulty to exchange information – The work of Organization B is supported by a configuration management system and an integrated issue-tracking system. Furthermore, Organization B uses a wiki as a collaboration tool to capture discussions, provide documents, and relate information on configuration management and issue-tracking uniformly. The use of the wiki started as a shared initiative and now is an established practice. All important documents are stored on the wiki and all issues are reported uniformly. All software engineers are involved in daily stand-up meetings with video-conferencing facilities.

3. Difference in sense of urgency – Organization B is not really bothered by this issue. Frequent traveling across development sites (including informal activities) helps to established a shared sense of urgency. In addition, all employees spend time at the main development site to get to know the customer's context. This enables the sites to relate communication efforts back to requirements of that customer.

4. Difficulty to build a team – First of all, the sites of Organization B regard each other fully as "peers". This contributes to a shared team understanding. Furthermore, having highly communicative meetings across the development sites throughout the course of the project ensures that a single, uniform team exists across development sites.

5. No collective ownership – Organization B uses a single configuration management system in which all software engineers are allowed to view and modify all parts of the source code. Programming occurs in pairs that are often changed. In addition, no distinction between software engineering tasks is made between the development sites. Consequently, the organization has a high degree of collective ownership. In case a practitioner makes a change to a part of the source code that is relative unknown, the practitioner contacts other software engineers on the change that is to be made. Correct workout of the change is verified by the different kinds of tests that run automatically.

6. Difficulty to align tasks and duties – Given the nature of the projects and the high pressure put on them, Organization B uses a very light-weight approach for capturing architectural knowledge (including architectural design decisions) and making this knowledge explicit. This knowledge is used to introduce new practitioners into the architecture and the rules that apply. Since no distinction between software engineering tasks is made, all practitioners within the team use a single task list for dividing the work.

7. No uniform process – Organization B uses SCRUM [25] as a software development process. As mentioned at *Issue 2*, frequent communication between practitioners from the development sites results in common understanding of the development process. Furthermore, an organization-wide shared vision on software development gives support for following that process. Since a while, releases are made from both the main development site and the remote development site, pointing out the similarity in the processes. Finally, we refer to a quotation of a software engineer of Organization B: *"It does not matter whether I work at the remote development site or at the main development site."*

5. Conclusions and Future Work

This paper reports on the use of architectural rules to overcome GSD issues. We have structured the challenges and solutions as identified in the literature. As a result, we have defined four challenges in GSD: time difference and geographical distance, culture, team communication and collaboration, and work distribution. These challenges cannot be fully regarded in isolation; they influence each other.

We observe that the solutions that are proposed by literature pertain to both the product (i.e. the structure of the software) and the process. Furthermore, we have exemplified how the solutions can be implemented by using architectural rules or process measures.

We used the list of GSD issues when interviewing two organizations to identify what solutions the organizations use to overcome these issues. We learned that some of these solutions can be described by using principles and statements (i.e. *architectural rules*) on the software architecture that must be complied with throughout the organization.

Table 1 summarizes the results of our analysis. Organization A has a clear focus on software architecture throughout the organization. Mechanisms such as an architecture team and architectural rules on different topics exist to remain "in control" of the architecture. The organization adheres to a "waterfall" approach in defining and disseminating architecture throughout the organization. With this type of approach, it is easier to check and assure that the architectural rules are obeyed. In addition, it is possible to e.g. measure to what extent architectural rules help in GSD. Nevertheless, this type of approach also requires to verify backward compliance. However, this verification does not take place at Organization A. Organization B has a more

Table 1. Overview of organizations' solutions to GSD issues. Solutions in the form of architectural rules are marked with [R]. Process measures are marked [C].

Issue	Organization A	Organization B
1. Difficulty to initiate contact	Many trips of architects to software engineers [C] Not all key individuals travel [C]	Periodic traveling of management and key individuals [C] Developers travel when hired [C]
2. Difficulty to exchange information	Yellow-pages, subsystem website [R] [C]	Use of a wiki [C] Frequent highly communicative meetings [C]
3. Difference in sense of urgency	Local bug fixing [C]	Shared view of customer context across development sites [C]
4. Difficulty to build a team	Dedicated subsystem teams, communication through architecture [R]	Frequent highly communicative meetings [C]
5. No collective ownership	Distributed code responsibility/ownership [R] [C]	Shared code responsibility/ownership [C] Frequent builds [C]
6. Difficulty to align tasks and sites	Alignment via architecture [R] Formal compliance-checking needed [C]	Joint planning [C] Frequent communication [C] Informal, continuous compliance-checking [C]
7. No uniform process	CM-tooling is present [R] Clear "official" process, multiple "real" processes in practice [C]	CM-tooling is present [R] SCRUM development process with test-driven development [C]

agile approach towards architecture. Although the organization has several architectural rules in place, the organization uses a number of practices and measures in the development process as solutions to overcome the GSD issues.

Architectural rules prove valuable in handling some of the the issues in GSD. Organization A mainly uses architectural rules that pertain to the product as well as some additional process measures. Sometimes, architectural rules pertaining to the product are meant to induce the necessary processes (issues 4 and 6 in Table 1). However, we also observe that architectural rules pertaining to the product may induce difficulties to addressing GSD issues for Organization A, such as maintaining a uniform process. Organization B does not so much focus on architectural rules but rather reverts to measures on the development process to tackle the issues related to GSD and ensure compliance with architectural rules. Organization A, on the other hand, leaves the teams a certain degree of freedom in these processes and does not verify compliance with these processes nor the architectural rules.

Our study shows that architectural rules are not a solution for all issues identified: especially cultural challenges and team collaboration challenges are not addressed by using architectural rules in the organizations.

Further, we have to take into account the differences between the two organizations, such as differences in complexity and size [24]. Nevertheless, we feel that a number of light-weight practices in place at Organization B can be transferred successfully to Organization A. For example, two-way traveling greatly helps in building a team and, consequently, in having development sites regard each other fully as "peers". In addition, to build a collective sense of urgency, the proportion of architect's work and developer's work (regardless of what development site is their stand) could be distributed more uniformly across development sites. The other way round, Organization B may need to place more emphasis on architectural rules in addition to process measures when development scales up.

This study sheds light onto the solutions that organizations use to overcome GSD challenges. Specifically, it focuses on the contribution of architectural rules to do so. In future work, we intend to compare the two organizations to propose a tailored solution in terms of roles and processes that are needed. Furthermore, we will try to obtain insight into what kind of architectural knowledge should be made explicit to address these GSD challenges.

Acknowledgment

We thank the practitioners from the participating organizations for their valuable input.

This research has been partially sponsored by the Dutch

Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge.

References

[1] P. J. Ågerfalk, B. Fitzgerald, H. Holmström, B. Lings, B. Lundell, and E. O. Conchúir. A Framework for Considering Opportunities and Threats in Distributed Software Development. In *International Workshop on Distributed Software Development*, pages 47–61, Paris, 2005. Austrian Computer Society.

[2] S. W. Ambler and R. E. Jeffries. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, 1st edition, 2002.

[3] M. Bass. Monitoring GSD Projects via Shared Mental Models: a Suggested Approach. In *GSD '06: Proceedings of the 2006 International Workshop on Global Software Development for the Practitioner*, pages 34–37, Shanghai, China, 2006. ACM Press.

[4] M. Bass, V. Mikulovic, L. Bass, J. D. Herbsleb, and M. Cataldo. Architectural Misalignment: An Experience Report. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA 2007)*, Mumbai, India, 2007. IEEE Computer Society.

[5] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Signature Series. Addison-Wesley Professional, 1st edition, 2002.

[6] M. Boasson. The Artistry of Software Architecture. *IEEE Software*, 12(6):13–16, 1995.

[7] V. Clerc, P. Lago, and H. Van Vliet. Assessing a Multi-Site Development Organization for Architectural Compliance. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA 2007)*, Mumbai, India, 2007. IEEE Computer Society.

[8] M. E. Conway. How Do Committees Invent? *Datamation*, 14(4):28–31, 1968.

[9] A. V. Corry, K. M. Hansen, and D. Svensson. Traveling Architects - A New Way of Herding Cats. In *Second International Conference on the Quality of Software Architectures (QoSA 2006)*, volume 4214 of *Lecture Notes in Computer Science*, pages 111–126, Västerås, Sweden, 2006. Springer Berlin / Heidelberg.

[10] K. C. Desouza, Y. Awazu, and P. Baloh. Managing Knowledge in Global Software Development Efforts: Issues and Practices. *IEEE Software*, 3(5):30–37, 2006.

[11] J. A. Espinoza, R. E. Kraut, S. A. Slaughter, J. F. Lerch, J. D. Herbsleb, and A. Mockus. Shared Mental Models, Familiarity, and Coordination: A Multi-Method Study of Distributed Software Teams. In *23rd International Conference in Information Systems (ICIS'02)*, pages 425–433, Barcelona, Spain, 2002.

[12] R. Farenhorst, P. Lago, and H. Van Vliet. Prerequisites for Successful Architectural Knowledge Sharing. In *ASWEC'07: 18th Australian Conference on Software Engineering*, pages 27–38, Melbourne, Australia, 2007. IEEE Computer Society.

[13] R. E. Grinter, J. D. Herbsleb, and D. E. Perry. The Geography of Coordination: Dealing with Distance in R&D Work. In *The International ACM SIGGROUP Conference on Supporting Group Work*, pages 306–315, Phoenix, Arizona, United States, 1999. ACM Press.

[14] J. D. Herbsleb and R. E. Grinter. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, 16(5):63–70, 1999.

[15] J. D. Herbsleb and R. E. Grinter. Splitting the Organization and Integrating the Code: Conway's Law Revisited. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 85–95, Los Angeles, California, United States, 1999. IEEE Computer Society Press.

[16] J. D. Herbsleb and A. Mockus. An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering*, 29(3):1–14, 2003.

[17] J. D. Herbsleb and D. Moitra. Guest Editors' Introduction: Global Software Development. *IEEE Software*, 18(2):16–20, 2001.

[18] J. D. Herbsleb, D. J. Paulish, and M. Bass. Global Software Development at Siemens: Experience from Nine Projects. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 524–533, St. Louis, Missouri, USA, 2005. ACM Press.

[19] G. Hofstede. *Cultures Consequences: International Differences in Work-Related Values, second edition*. Sage Publications Inc., 2001.

[20] H. Holmström, E. O. Conchúir, P. J. Ågerfalk, and B. Fitzgerald. Global Software Development Challenges: A Case Study on Temporal, Geographical, and Socio-Cultural Distance. In *The IEEE International Conference on Global Software Engineering (ICGSE'06)*, pages 3–11, Florianopolis, Brazil, 2006. IEEE Computer Society.

[21] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120, Pittsburgh, Pennsylvania, 2005.

[22] P. Kruchten, P. Lago, and H. Van Vliet. Building up and Reasoning about Architectural Knowledge. In *Second International Conference on the Quality of Software Architectures (QoSA 2006)*, volume 4214 of *Lecture Notes in Computer Science*, pages 43–58, Västerås, Sweden, 2006. Springer Berlin / Heidelberg.

[23] N. Mullick, M. Bass, Z. El Houda, D. J. Paulish, M. Cataldo, J. D. Herbsleb, R. Sangwan, and L. Bass. Siemens Global Studio Project: Experiences Adopting an Integrated GSD Infrastructure. In *The IEEE International Conference on Global Software Engineering (ICGSE'06)*, pages 203–212, Florianopolis, Brazil, 2006. IEEE Computer Society.

[24] D. E. Perry and G. E. Kaiser. Models of Software Development Environments. *IEEE Transactions on Software Engineering*, 17(3):283–295, 1991.

[25] K. Schwaber and M. Beedle. *Agile Software Development with SCRUM*. Prentice Hall, 1st edition, 2001.

[26] C. Walls and R. Breidenbach. *Spring in Action*. In Action Series. Manning Publications Co., 2005.