Global Software Engineering:

The Future of Socio-technical Coordination

James D. Herbsleb

James D. Herbsleb is an Associate Professor of Computer Science and the Director of the Software Industry Center at Carnegie Mellon University. His research interests are in globally-distributed and open source software engineering, as well as coordination in collaborative work more generally. Prior to joining the faculty of Carnegie Mellon University, he initiated the Bell Labs Collaboratory project, leading a research team which designed, implemented, and deployed solutions for global development, including tools, practices, and organizational models. He has also published papers describing how open source software development actually works, its limitations, and the extent to which open source practices can be applied in industrial settings.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Global Software Engineering: The Future of Socio-technical Coordination

James D. Herbsleb
School of Computer Science
Carnegie Mellon University
jdh@cs.cmu.edu

Abstract

Globally-distributed projects are rapidly becoming the norm for large software systems, even as it becomes clear that global distribution of a project seriously impairs critical coordination mechanisms. In this paper, I describe a desired future for global development and the problems that stand in the way of achieving that vision. I review research and lay out research challenges in four critical areas: software architecture, eliciting and communicating requirements, environments and tools, and orchestrating global development. I conclude by noting the need for a systematic understanding of what drives the need to coordinate and effective mechanisms for bringing it about.

1. Introduction

It is no longer unusual for a large software project to have teams in more than one location, often on more than one continent. Many forces have conspired to bring about this situation, including concern for cost, the need to tap global pools to acquire highly skilled resources, finding an appropriate mix of expertise for a project, satisfying investment requirements imposed by governments in foreign markets, and mergers and acquisitions. There is little reason to expect these factors to diminish in the future. Rather, it appears that we face increasing globalization of markets and production, increasing the pressure to distribute projects globally. In this paper, I assume that this direction will continue or even accelerate.

While global software development (GSD) is becoming a way of life, such work takes much longer than co-located work [39], and suffers from a wide range of problems (see, e.g., [53]). At the same time, we have accumulated considerable knowledge and experience, which is beginning to appear in the form of comprehensive practitioner-oriented books [10, 60], and several special sections or special issues on this topic [22, 40].

The vision of the desired future of global development, shared by many, would to be to have the following capabilities. For any given project, be able to

- use available resources independently of geographic location
- plan practices and technology to support the level of coordination accurately anticipated to be required among sites
- achieve shared understanding of requirements
- measure the "fit" of a software architecture with the organization that will build the system, and have a set of known, effective tactics for improving the fit
- effectively manage change

The structure of this paper is to identify the problems that stand in the way of achieving this vision, and for each problem, review the current state of research and future challenges.

1.1 Scope

The paper is focused on technical coordination in geographically distributed projects. By *coordination*, I mean managing dependencies among tasks [47]. If tasks carried out at different sites shared no dependencies, global projects would not pose significant challenges. People at any given site would not need to communicate with or even know of the existence of other sites. Thus, *the key phenomenon of GSD is coordination over distance*. This paper focuses on those features of software projects that influence the need to coordinate, and activities designed to achieve technical coordination, which includes communication, tools, processes, and practices.

The paper does not consider a number of other related questions, such as business decisions about whether to outsource, legal arrangements among collaborating organizations, or evaluation of providers

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

of outsourcing services. Nor does it consider collaboration in software engineering more generally (see [68]). These issues, though important, are beyond the scope of this paper.

To keep the number of citations tractable, I chose to focus on conference and journal papers rather than workshop papers, and to give preference to more recent work. I also gave preference to work that focuses specifically on software engineering, and to work that appears in software engineering or computer science publications, although I make occasional citations to other literature as appropriate.

In the remainder of this introduction, I present a view of global development which will serve to identify the problem areas.

1.2 What is different about global development?

In a (highly idealized) traditional, co-located project, teams with a history of working together have naturally built up a number of ways of coordinating their work. They have a shared view of how the work will proceed, either because of a shared, defined process or just by acquiring a common set of habits and vocabulary over time. Through frequent interactions, both formal and informal, team members have a clear idea of who has what sort of expertise and how responsibilities are allocated. Information flows freely through the network during the many informal interactions that happen in the hallway, over meals, before and after formal meetings. There is relatively little miscommunication as teams share a common native language as well as national and corporate culture. People are generally aware of what others are working on, know if and how their work affects other people, and know day to day the level of urgency and stress experienced across the project. Prior collaborations have produced long-standing professional and social relationships that provide a context and history within which problems and misunderstandings can be resolved.

The fundamental problem of GSD is that many of the mechanisms that function to coordinate the work in a co-located setting are absent or disrupted in a distributed project. Geographic distance profoundly affects the ability to collaborate [53]. Even relatively small distances can have major effects. Evidence indicates, for example, that spontaneous communication and collaboration declines as a function of distance between offices, and the asymptote is near a surprisingly short distance of about 30 meters [1]. Radical co-location, i.e., putting whole teams in a shared area rather than individual offices, can have a surprisingly large effect on development speed and efficiency [67].

As geographic and temporal distance increases, a wide variety of impediments to coordination are introduced (see generally [10, 12, 22, 40, 41, 53]. GSD projects are diverse, of course, and experience various disruptions to different degrees. But the following are among the more common ways in which coordination mechanisms are disrupted:

Much less communication, less effective communication. In GSD projects, people communicate with many fewer people at distant sites that at their own site, and the communication is much less frequent [39]. There are many reasons for this difference [11, 43], including temporal distance (different time zones), socio-cultural distance (language and culture), and geographic distance (making travel difficult). Even the communication that occurs is less effective [53]. Few overlapping hours means replies often don't come until the next day. These communication issues have many effects, including a lack of information about who is expert in what, and who is responsible for what.

Lack of awareness. Because people at different sites share relatively little context, they tend to have little knowledge of what people at other sites are doing day to day, if they are available for communication, and what their immediate concerns are. This lack of contextual information makes it difficult to initiate contact, and often leads to misunderstandings of communication content and motivations. Perhaps most importantly, it hinders a project's ability to keep track of the effects of change as they propagate across sites.

Incompatibilities. Sites often differ in development tools, processes, practices, informal work habits, corporate culture, and in many other ways. These differences may often be incompatible, leading to a wide variety of problems, such as errors found at one site that cannot be duplicated elsewhere, different processes lead to confusion and misunderstandings about how the work is done and its current status, habits like freely sharing bad news that may not be appreciated or understood correctly elsewhere.

In the sections that follow, I focus on four areas of research that have made substantial progress in addressing these issues, and describe the most significant obstacles that remain.

2. Software architecture

As with most types of complex systems, software is generally designed as sets of interacting components [58, 63]. Software architectures not only influence quality attributes of software, but they are also an important means of coordinating software projects.

Adopting an architecture tends to guide developers toward compatible decisions, assuming there is effective communication and a common understanding of what the architecture actually is [37, 54].

2.1 Research summary

It is widely believed that the extent and nature of task dependencies in development work are a product of dependencies in the software architecture. Conway [16] observed that the structure of a product tends to resemble the structure of the organization that designed it. The reason is that once interfaces have been specified, the components themselves can be designed relatively independently of one another [4, 57]. The underlying assumption is that modularity in product structure (i.e., relatively few, and well-specified interactions among components) results in modularity in subsequent design tasks (i.e., relatively few dependencies between design decisions concerning different components). Thus, if development of each component is assigned to a single team, there is relatively little need for coordination across teams. As Conway [16] pointed out, this is likely to result in a homomorphic relation between the architecture and the organization, i.e., each component is assigned to exactly one team, although one team may build more than one component.

Research has also revealed some of the mechanisms by means of which organizations adjust to the coordination needs imposed by product architecture. It is reflected in communication patterns, in the ways in which people choose what information to pay attention to, and in collaborative problems-solving strategies [36]. The particular ways in which the components interact tends to dictate the precise decisions on which the teams to which the components have been assigned must coordinate. Software architects, therefore, design not just the structure of the software, but also have a major role in shaping the task dependencies among the teams designing and building the system.

A study of how application programmer interfaces (APIs) are used in development organizations show not only the advantages of these architectural constructs in enhancing coordination among groups, but also their tendency to promote isolation and reduce information sharing [65]. This can be a disadvantage if coordination becomes necessary among groups, as can happen when interfaces are unstable [38, 54, 65], or if there are important semantic dependencies.

Interestingly, software architects are aware of the organizational consequences of their technical work, and spend considerable time trying to ensure their designs have favorable organizational outcomes [29]. Currently, research provides very little guidance for architects struggling with the organizational implications of architectural decisions. One exception is the work of Mockus and Weiss [50], who developed ways of using change histories to identify what parts of the code can be maintained at separate sites, heuristically minimizing the need for cross-site interaction. Cataldo and colleagues [13], also using change history data, developed a way of computing coordination requirements, i.e., who needs to coordinate with whom in order to accomplish a unit of development work. They found that when coordination behaviors were congruent with these requirements, code changes were accomplished more quickly.

2.2 Research challenges

Unraveling the complex relationship between software dependencies and task dependencies. It is generally assumed that creating modular software, i.e., minimizing the dependencies among static modules and among run-time components, will have the effect of reducing the dependencies among the tasks involved in designing and constructing the components. While there is little reason to doubt that this is generally true, we need to know much more about the relationship of software dependencies and task dependencies in order to design software architectures that are better suited to GSD. Decoupling components, e.g., by adding an intermediary, may or may not decouple tasks. Adding an intermediary where the most difficult dependencies are semantic rather than syntactic may in fact make the task coordination problem harder. Understanding task coupling would, for example, let us predict if the need for coordination is so intense that teams need to be co-located. Is it important that we only assign a pair of components to teams that have a shared work history? We know very little at this point about the coordination requirements that architectural decisions impose on teams.

Measuring architectural/organizational fit. Beyond just understanding how architectural decisions impose task dependencies, we need to be able to determine how well equipped a given organization is to carry out the design and implementation of a system with a particular architecture. With the exception of informal and ad hoc attempts by software architects to accommodate the development organization, we do not know yet how to assess the fit of architecture and organization proactively. It is especially important to do so early in the project, so appropriate and timely adjustments can be made.

The functional dependencies in the software are certainly not the whole story. For example, stringent nonfunctional requirements like performance, security,

or reliability have the tendency to bind development tasks together more tightly around these global properties, since decisions within a module may have far-reaching effects. Similarly, stability of an interface, i.e., the extent to which the initial design will hold up, may be as important or more important than the precise nature of the technical dependency.

Tactics for improving architecture/organization fit. Even if we can assess the fit of an architecture and a development organization, in order to act effectively on this information, we need a set of tactics that will allow us to better adjust the organization to the architecture, or the architecture to the organization. If we determine that the need for coordination between two distributed teams designing two components will exceed their ability to coordinate, what do we do?

It is likely that we will develop both organizational and architectural tactics. Organizational tactics might include such measures as increasing communication, providing additional communication technology, or reassigning work to co-located groups. Technical tactics might favor solutions that sacrifice some cost or quality attribute in order to reduce the need to coordinate. To provide a real world example, two groups at different locations used a single shared memory chip. Chip size and therefore cost could be reduced by being very clever about memory use, but this would likely require substantial coordination between teams. Alternatively, the memory could simply be partitioned, with each team allocated sufficient space for its needs. This will be less efficient and more costly, but reduces inter-team coordination needs. We need a broad collection of effective tactics, along with the knowledge of how to choose which to use.

3. Eliciting, communicating requirements

Getting the requirements right, and dealing with unstable requirements, are notoriously difficult problems (e.g., [19]). By "requirements" in this case, I mean not just end user requirements, but anything that specifies what a team should deliver. In the global development context, the inherent difficulty of achieving a shared understanding of the requirements is amplified, both because of loss of context and loss of communication bandwidth.

3.1 Research summary

Requirements elicitation and communication presents several specific challenges in the GSD context [21, 23] – indeed, Cheng and Atlee consider globalization to be one of the major research challenges in requirements engineering [14]. A global context makes it more difficult to seek out and to integrate the necessary knowledge. Process mismatches, differing technical and domain vocabularies, incompatible environments, and conflicting assumptions can be particularly problematic in a GSD context [8]. Cultural differences can pose formidable challenges for achieving a shared understanding of the requirements [44].

Recommended practices include ways of improving communication, easing mismatches with frequent deliveries, and making organizational responsibilities more transparent [21]. Other authors have suggested a number of practices in each dimension of people, process, and technology, for addressing GSD problems [8]. While we are still a long way from a clear understanding of what kinds of requirements elicitation techniques are most likely to succeed in what circumstances, some researchers have begun to approach these questions of "fit" [3].

In some cases, it is possible to use more formal approaches to ensure that what is built satisfies its requirements. An excellent illustrative example of the successful application of a formal approach is the SLAM toolkit developed by Microsoft Research to ensure that writers of device drivers did not violate the complex kernel API [6]. While not usually considered in the context of requirements or GSD, this sort of analysis has in fact been effective in conveying a very complex API to a number of vendors in different organizations. This, of course, requires a particular kind of analysis tool, i.e., one that needs no input and has an acceptably low rate of false positives [5].

3.2 Research challenges

Much of the research in requirements engineering focuses on the requirements themselves, how to elicit them, analyze them, manage them, and to recognize and resolve conflicts. For effective requirements engineering in GSD, we need to tackle additional issues that take on increased urgency. In particular, we need significantly to improve our ability to support the ongoing negotiation processes that are prevalent throughout the project lifecycle [19, 23]. Negotiation is much more difficult in a GSD context, because of the diversity of backgrounds, communication problems, and the difficulty of responding to change [21].

Anticipating the need to support negotiation. Just as we need to understand how characteristics of software architectures drive task dependencies in software design and construction, we need to understand the trade-offs that lead to the need for negotiation in the requirements engineering process. Can we predict, for example, how unstable the

requirements will be for a given project? What are the economic, social and organizational factors that make the search for a single unified view of requirements particularly difficult? Which are the relevant stakeholders that need to be involved for effective expectation management? This would go a long way toward understanding the intensity of negotiation and information exchange that will be required. Can we provide novel views or notification services linked with the evolving requirements that will effectively alert distributed stakeholders to changes that affect their interests? This could allow misunderstandings and diversity of interests to be revealed earlier, before options for resolving such conflicts become more limited. Can we build in more effective reviews and synchronization points to ensure that the stakeholders are moving toward a genuine common understanding?

Media for requirements communication. While there is now some research on the effects of various communication media involved in the requirements elicitation and negotiation process [26], we are still a long way from understanding in a detailed way what media are suitable for all of the different kinds of communication among all the business stakeholders, analysts and developers. Most development organizations seem to be placing large bets that the correct configuration is to have an analyst and marketing group physically near the customer, but are also willing to locate one or more development groups remotely. As yet we have litte evidence of the effectiveness of this arrangement. It would also be very useful to have a careful analysis of the communication needs for various kinds of interactions around requirements in terms of communication media affordances for things like common ground [16].

4. Environments and tools

Software engineering has long focused on developing and deploying tools to assist with coordination of large software projects. Version control and change management are largely taken for granted, and integrated development environments with extensible framework architectures are becoming quite common. GSD research on environments and tools often focuses on extending these coordination capabilities, building on the functionality or actually integrating with, these standard tools.

4.1 Research summary

Many of the tools commonly used for co-located development lend themselves quite well, perhaps with some enhancements, to global projects. Standard tools for version control and change management can be used in distributed fashion, with a single centralized server, and as long as the wide area network is sufficiently reliable and sufficient bandwidth is available, there is little difference in a co-located and distributed tools. Where networks are not adequate for a central server, commercial tools with replicated databases and automated synchronization are available.

Since tool-building has been such a focus in GSD, there is a large volume of research to try to summarize. I will organize this section under two subtopics: tools for awareness and communication, and for exploring project memories.

4.1.1 Awareness and communication. Given the reduced level of communication in GSD, the problem of understanding what other project members are doing, in order to coordinate effectively with them, is much more difficult. In addition to providing awareness and communication capabilities, many tools exploit the advantages of integrating collaborative features into the development environment. For example, this approach takes advantage of capabilities and data already in tools such as version control systems, it allows easy pasting and linking of other development artifacts into messages, and permits the remote sharing of existing screens in the integrated development environment (IDE) [15].

Several studies have introduced chat capabilities into development environments to explore their utility. Handel et al [34] found that while use varied considerably, many teams made use of a stand-alone chat tool, and used it primarily for work-related discussions, with a smattering of non-work use. Fitzpatrick and colleagues [27] integrated chat and notification services into a version management tool, and studied how these facilities were used. The findings again were rather positive, noting the utility of chat for e.g., generating discussion and supplementing information contained in the logs.

Several research projects have explored incorporating diverse awareness and collaboration tools into the Eclipse IDE. The Jazz project (e.g., [15]) has incorporated IM capabilities, including the ability to observe who is currently logged in, discussion forums, audio (VOIP) capabilities, and awareness icons that allow developers to see what other developers are editing in their local environments. Sarma et al [61] built a tool called Palantir, which provides developers with a visualization of who is changing what artifacts in their local environments, and provides a measure of the severity of the changes. They have also considered how to use information abstraction and analysis approaches to provide

awareness at the larger, global project level, rather than just the team-oriented facilities of Palintir [62].

Other work has focused on trying to provide collaborative facilities for specific parts of the life cycle. Sinha et al [64] built a tool for collaborative requirements engineering on top of the jazz environment, including collaborative functionality designed to address common requirements engineering issues such as resolving requirements ambiguities and managing and notifying stakeholders about requirements changes. In order to support the kind of intensive communication required to support planning for an agile-style development in a distributed context, Morgan and Maurer [51] designed a tool to support intense interaction needed for agile-style planning in a GSD context, using a tabletop and virtual note card metaphor. Spanjers et al [66] built a specialized tool for GSD build and test, which allows remote sites to observe execution, access reports and work products, and makes the test procedures available and visible. Lanubile et al [46] designed a tool to support distributed, largely asynchronous inspections.

4.1.3. Exploring project memory. Typical software development tools tend to create a potentially very rich "project memory" [17] with versioned files, change histories, and documents, some small fraction of which are highly relevant to current work. Such histories have especially great potential in communication-starved contexts like GSD.

Open source development provides a specialized example of use of tools for global development. Such environments typically include relatively simple tools such as version control, change management, and mailing lists, and sometimes chat. Dinkelacker et al [24] report on the successful use of open source style tools in an industrial setting. Gutwin et al [32] focused on understanding how open source developers use these tools to maintain both a broad awareness of the people and activities on the project as well as find specific information such as experts on a given topic.

Two research projects in Murphy's research group have tackled the task of performing computations over artifacts in project histories to provide assistance for finding relevant artifacts. Cubranic et al [18] describe the design and evaluation of Hipikat, a tool that draws on information retrieval techniques to help developers identify artifacts that are related to an artifact used to generate a query. Their evaluation shows that the tool finds useful starting points for exploring the code. Kersten et al [45] design a tool, Mylar, that computes a degree of interest for project artifacts, using task context and project history, and provides a filter function to help identify useful artifacts. Bruegge et al [9] take a different tack with their tool, Sisyphus. Rather than selecting or filtering artifacts, Sisyphus supports the creation and subsequent browsing of a graph created by linking artifacts, as well as annotations and comments on those artifacts.

Other work has focused on finding relevant people, rather than artifacts. Several papers have specifically tackled the question of locating people with the right expertise in a distributed setting. Erlich and Chang [26] performed a social network analysis of information seeking in order to understand where people currently go to find different kinds of expertise such as technical information, administrative information, and innovative ideas. McDonald and Ackerman [48] and Mockus and Herbsleb [49] have designed and deployed tools for locating expertise in software development projects, using data from the version control system about which developers have contributed to what parts of the code.

Others have focused on tools to facilitate or to diagnose particular kinds of interactions in projects. Anvik and colleagues [2] use machine-learning techniques in the design of a tool that uses past history of bug fixes to assist open source bug triagers identify appropriate developers for fixing an incoming bug. Halverson et al [33] identified common problems in change management in software projects, and developed two visualizations using change management data to assist in identification of specific problems and overall project health. Froehlich et al [28] developed Augur, which uses linked views of code and activities to promote awareness, e.g., by seeing who changed what lines, and what lines were checked in together. Finally, Souza and colleagues [65] proposed a social call graph, i.e., a graph in which the nodes, and the edges are drawn between developers whose code calls the code of another developer. They suggest such a representation could be useful for finding people whose code is likely to be impacted by a change.

4.2 Research challenges

Virtual co-location. Current research on collaborative IDEs is adding impressive capabilities, and tackling some important integration issues. Yet we do not know, at this point, how close these environments are to creating the natural, effortless kind of awareness and communication that happens in co-located settings. We should focus on creating an environment that puts people in virtual proximity so that those who do in fact need to coordinate and communicate can do so as naturally through the environment as they could if they had offices in the same hallway. This will require a better knowledge of who actually needs to coordinate with whom, as

research has shown that these coordination requirements are volatile and apparently non-obvious to all but the most proficient developers [13]. It will also require a detailed understanding of what kinds of task dependencies exist, and what kinds of facilities, such as different communication media, are needed to resolve them.

Continuing to exploit project memory. While we have now accumulated a substantial body of research on how to identify relevant artifacts and people in project memories, it is not clear that winning strategies and approaches have yet been identified. While many existing approaches seem promising, we have much to learn about how specific approaches can be used to support particular tasks, and when the recommendations made by tools are sufficiently accurate and precise that they outweigh the potential distraction and neglect of non-recommended choices. We need to move beyond a proof-of-concept approach, in which validation is primarily a few plausible examples of good recommendations or a demonstration that the recommendations are at better-than-chance levels. This approach has served us well to this point, but we need to understand the practical realities of bringing this functionality to bear on day-to-day tasks, figuring out when the cost is worth the benefit, and how to mitigate any ill-effects that might arise because of biases built into the recommendation algorithms.

Enriching project memories. Project memories consist primarily of data accumulated almost inadvertently, for other reasons, e.g., for version control or to keep track of change requests. One of the real challenges for the future is to think about such repositories from the point of view of creating a project memory, and designing an environment that collects a richer set of data, for example, by more information about the original context. Many actions by individual developers that might have value are not typically tracked, for example, recording all the documents a developer examined around the time a change was made in order to better understand a rationale for the decision. Just accumulating the data is not sufficient, of course, it must be stored in tractable form, capable of being exploited, without undue space and performance penalties. There is also the question of privacy concerns. Collecting more and more data, while potentially useful, created the risk that the data can be used in illegitimate and punitive ways. One of the real challenges is to provide a rich data set that will support search and exploration to accomplish project work, but cannot readily be exploited for surveillance purposes.

Project history and collaborative tool infrastructure. As more kinds of data are accumulated, and as more collaborative functionality is added to IDEs, the infrastructural requirements become more demanding. Notification services that can readily be customized by users and are sensitive to tasks being executed will be important. Strategies for caching, storing, and consolidating data form disparate sources will be a critical need. The need for interoperable tools with standard data formats and interaction protocols will become increasingly important as we move toward more integration of collaborative capabilities into the environment.

5. Orchestrating global development

The practices, organizational structures, and methods used for co-located development are often not adequate for GSD projects. Typical ways of orchestrating projects often rely heavily on the sorts of frequent communication, shared knowledge, and common history that are absent in distributed projects.

5.1 Research summary

In addition to the relationship between software architecture and the form of the development organization discussed in Section 2, several kinds of organizational models have been proposed for GSD projects. Evidence suggests that unlike co-located projects, distributed projects suffer fewer coordination problems when they exhibit informal hierarchies rather than a network organizational form [42], presumably because of the high cost of communication in distributed teams. Other work has examined a variety of organizational models used in industry, which for example, assign different components to teams at different sites, or assign the work associated with different process steps to different sites [30].

Other authors [35], based on qualitative observations, have suggest the importance of the match on several dimensions, of organizations that are collaborating on a software project. Offshore teams have other options as well, for example adopting a matrix form rather than mirroring the structure of the client team [52]. Under appropriate circumstances, such as when a number of different product groups want to enhance and maintain a common resource, an open source style of organizing development, in which various users of a resource develop the functionality they need, can be effective [31].

In addition to research on overall organizational models for GSD, other research has focused on how agile methods can be adapted to GSD, even though on the surface, the intense use of informal communication to coordinate work makes agile seem a poor match for GSD [56]. While experience with agile in GSD is still

fairly sparse, Ramesh and colleagues [59] report successful experiences when agile is adjusted appropriately for a GSD context, e.g., to facilitate communication and knowledge management.

Finally, there have been a number of qualitative studies of individual practices or collections of practices to support GSD. Paasivaara and colleagues [55] identify a number of practices used and deemed successful in GSD projects, including frequent deliveries and establishing links among peers. Cusick and Prasad [20] suggest a collection of practices, some of which are adaptations of traditional practices (e.g., issue tracking, short phases, small deliverables), while some are unique to GSD (e.g. ensuring that domain expertise is retained both onshore and offshore).

Other qualitative studies suggest the value of particular practices for addressing specific problems. Battin and colleagues [7] associate practices with specific GSD problems that these practices address. For example, they propose an incremental integration strategy to deal with cross-site integration. Carmel and Agarwal [11] advocate several ways of reducing the intensity of needed interactions among sites, and for reducing cultural and temporal distance between sites. Ebert et al [25] describe successful practices for distributed validation, including things like co-locating code inspection, and introducing continuous builds.

5.2 Research challenges

What practices are effective when? We do not know much at this point about when various GSD practices are effective, and when they are not. When is it advisable to have a "cultural liaison" [11]? When is it advisable to introduce agile approaches into GSD projects? Many authors propose plausible rules of thumb for such questions, but we need research to test these views. We also need to identify the range of applicability for practices. For example, what kinds of practices are appropriate for outsourcing where there are legal and organizational boundaries, as opposed to projects distributed within a single company?

Interactions among practices. Our understanding of how various practices play together takes two forms. First, sets of compatible or complementary practices are bundled together and recommended as a whole [20, 60]. The other approach is to address the practices in a purely atomic manner, matching them for example to problems they address [7]. The truth likely is somewhere in between. Bundles of practices are rarely so bound together that no other practices could be substituted. Nor is it the case that the decision to adopt a practice can be considered in total isolation from all other practices. Yet we don't really know much about the conflicts and complementarities among practices, knowledge which is critical for an intelligent consideration of tradeoffs.

6. Conclusion

I argued at the beginning of this paper that the key phenomenon in GSD is coordination at a distance. The need to manage a variety of dependencies across sites drives the essential problems of GSD. It follows that the biggest need to make substantial progress in GSD is to achieve a deeper understanding of the kinds of coordination that are required, the factors that drive these needs so they can be predicted for a given project, and the principles governing how the coordination mechanisms available to a development organization can best be deployed against these needs.

We currently have a number of individual solutions, such as tools, practices, and methods, but we understand as yet very little about the tradeoffs among them, and the conditions of their applicability. If we work toward compatible processes across sites, can we reduce the amount of communication? If we carefully design architectures to isolate work at different sites, can we get away with incompatible processes? We currently have very little to go on for addressing these crucial questions. We have a pressing need for good theories that will provide a sound basis for reasoning about tradeoffs and predicting outcomes.

7. Acknowledgements

The author gratefully acknowledges support by NSF grant IIS-0534656, as well as support from the Software Industry Center and its sponsors, particularly the Alfred P. Sloan Foundation and Siemens Corporate Research. He also thanks Leonard Bass, Matthew Bass, Marcelo Cataldo, and Daniela Damian for their insightful comments on an earlier draft.

8. References

[1] Allen, T.J., *Managing the Flow of Technology*. 1977, Cambridge, MA: MIT Press.

[2] Anvik, J., Hiew, L., and Murphy, G.C., *Who should fix this bug?* in *Proceeding of the 28th international conference on Software engineering*. 2006, ACM Press: Shanghai, China.

[3] Aranda, G.N., et al. Technology Selection to Improve Global Collaboration. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.

[4] Baldwin, C.Y. and Clark, K.B., *Design Rules: The Power of Modularity*. Vol. 1. 2000, Cambridge, MA: The MIT Press.

[5] Ball, T., et al. Thorough Static Analysis of Device Drivers. in *EuroSys*. 2006. Leuven, Belgium.

[6] Ball, T. and Rajamani, S.K. The SLAM Toolkit. in *Computer Aided Verification: 13th International Conference*. 2001. Paris, France.
[7] Battin, R.D., et al., Leveraging Resources in Global Software Development. *IEEE Software*, *18,* 2 (Mar/Apr 2001), p. 70-77.
[8] Bhat, J.M., Gupta, M., and Murthy, S.N., Overcoming Requirements Engineering Challenges: Lessons from Offshore Outsourcing. *IEEE Software*, *23,* 5 (2006), p. 38-44.
[9] Bruegge, B., Dutoit, A.H., and Wolf, T. Sysiphus: Enabling informal collaboration in global software development. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[10] Carmel, E., *Global Software Teams*. 1999, Upper Saddle River, NJ: Prentice-Hall.
[11] Carmel, E. and Agarwal, R., Tactical Approaches for Alleviating Distance in Global Software Development. *IEEE Software*, *March/April,* (2001), p. 22-29.
[12] Casey, V., Richardson, I. Project Management within Virtual Software Teams. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[13] Cataldo, M., et al., *Identification of coordination requirements: implications for the Design of collaboration and awareness tools*, in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 2006, ACM Press: Banff, Alberta, Canada.
[14] Cheng, B. and Atlee, J., *Research Directions in Requirements Engineering*, in *Future of Software Engineering 2007*, L. Briand and A. Wolf, Editors. 2007, IEEE-CS Press.
[15] Cheng, L.-T., et al., Building Collaboration into IDEs. *Queue*, *1,* 9 (2004), p. 40-50.
[16] Conway, M.E., How Do Committees Invent? *Datamation*, *14,* 4 (1968), p. 28-31.
[17] Cubranic, D. and Murphy, G. Hipikat: Recommending Pertinent Software Development Artifacts. in *International Conference on Software Engineering*. 2003. Portland, OR.
[18] Cubranic, D., et al., Hipikat: a project memory for software development. *31,* 6 (2005), p. 446.
[19] Curtis, B., Krasner, H., and Iscoe, N., A field study of the software design process for large systems. *Communications of the ACM.*, *31,* 11 (1988), p. 1268-1287.
[20] Cusick, J. and Prasad, A., A Practical Management and Engineering Approach to Offshore Collaboration. *IEEE Software*, *23,* 6 (2006), p. 20-29.
[21] Damian, D., Stakeholders in Global RE: Lessons learned from practice. *IEEE Software,* (2007), p.
[22] Damian, D. and Moitra, D., Global Software Development: How Far Have We Come? *IEEE Software*, *23,* 5 (2006), p. 17-19.
[23] Damian, D.E. and Zowghi, D., Requirements Engineering challenges in multi-site software development organizations. *Requirements Engineering Journal*, *8,* (2003), p. 149-160.
[24] Dinkelacker, J., et al. Progressive open source. in *International Conference on Software Engineering*. 2002. Orlando, Florida.
[25] Ebert, C., et al. Improving validation activities in a global software development. in *International Conference on Software Engineering*. 2001. Toronto, Canada.
[26] Ehrlich, K. and Chang, K. Leveraging expertise in global software teams: Going outside boundaries. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[27] Fitzpatrick, G., Marshall, P., and Phillips, A., *CVS integration with notification and chat: lightweight software team collaboration*, in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 2006, ACM Press: Banff, Alberta, Canada.
[28] Froehlich, J. and Dourish, P., *Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams*, in *Proceedings of the 26th International Conference on Software Engineering*. 2004, IEEE Computer Society.
[29] Grinter, R.E. Systems Architecture: Product Designing and Social Engineering. in *International Joint Conference on Work Activities, Coordination, and Collaboration*. 1999. San Francisco, CA.
[30] Grinter, R.E., Herbsleb, J.D., and Perry, D.E. The Geography of Coordination: Dealing with Distance in R&D Work. in *GROUP '99*. 1999. Phoenix, AZ: ACM Press.
[31] Gurbani, V.K., Garvert, A., and Herbsleb, J.D., *A case study of a corporate open source development model*, in *Proceeding of the 28th international conference on Software engineering*. 2006, ACM Press: Shanghai, China.
[32] Gutwin, C., Penner, R., and Schneider, K., *Group awareness in distributed software development*, in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. 2004, ACM Press: Chicago, Illinois, USA.
[33] Halverson, C.A., et al., Designing task visualizations to support the coordination of work in software development. *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work,* (2006), p. 39-48.
[34] Handel, M. and Herbsleb, J.D. What is Chat Doing in the Workplace? in *Conference on Computer-Supported Cooperative Work*. 2002. New Orleans, LA.
[35] Heeks, R., et al., Synching or Sinking: Global Software Outsourcing Relationships. *IEEE Software*, *March/April,* (2001), p. 54-60.
[36] Henderson, R.M. and Clark, K.B., Architectural innovation: The reconfiguration of existing product technologies and the failure of established firms. *Administrative Science Quarterly*, *35,* 1 (1990), p. 9-30.
[37] Herbsleb, J.D. and Grinter, R.E., Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, *Sept./Oct.,* (1999), p. 63-70.
[38] Herbsleb, J.D. and Grinter, R.E. Splitting the Organization and Integrating the Code: Conway's Law Revisited. in *21st International Conference on Software Engineering (ICSE 99)*. 1999. Los Angeles, CA: ACM Press.
[39] Herbsleb, J.D. and Mockus, A., An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions on Software Engineering*, *29,* 3 (2003), p. 1-14.
[40] Herbsleb, J.D. and Moitra, D., Global Software Development. *IEEE Software*, *March/April,* (2001), p. 16-20.
[41] Herbsleb, J.D., Paulish, D.J., and Bass, M. Global Software Development at Siemens: Experience from Nine

Projects. in *International Conference on Software Engineering*. 2005. St. Louis, MO.
[42] Hinds, P. and McGrath, C., *Structures that work: social structure, work structure and coordination ease in geographically distributed teams*, in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 2006, ACM Press: Banff, Alberta, Canada.
[43] Holmstrom, H.C., E.O. Agerfalk, P.J. Fitzgerald, B. Global Software Development Challenges: A Case Study on Temporal, Geographical and Socio-Cultural Distance. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[44] Hsieh, Y. Culture and Shared Understanding in Distributed Requirements Engineering. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[45] Kersten, M. and Murphy, G.C., *Using task context to improve programmer productivity*, in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 2006, ACM Press: Portland, Oregon, USA.
[46] Lanubile, F., Mallardo, T., and Calefato, F., Tool support for geographically dispersed inspection teams. *Software Process: Improvement and Practice*, *8,* 4 (2003), p. 217-231.
[47] Malone, T.W. and Crowston, K., The interdisciplinary theory of coordination. *ACM Computing Surveys*, *26,* 1 (1994), p. 87-119.
[48] McDonald, D.W. and Ackerman, M.S. Expertise Recommender: A Flexible Recommendation System and Architecture. in *ACM Conference on Computer Supported Cooperative Work*. 2000. Philadelphia, PA: ACM Press.
[49] Mockus, A. and Herbsleb, J.D. Expertise Browser: A Quantitative Approach to Identifying Expertise. in *International Conference on Software Engineering*. 2002. Orlando, FL.
[50] Mockus, A. and Weiss, D.M., Globalization by Chunking: A Quantitative Approach. *IEEE Software*, *January - March,* (2001), p.
[51] Morgan, R. and Maurer, F. MasePlanner: A Card-Based Distributed Planning Tool for Agile Teams. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[52] Narayanan, S., Mazumder, S., and R, R. Success of Offshore Relationships: Engineering team structures. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[53] Olson, G.M. and Olson, J.S., Distance Matters. *Human-Computer Interaction*, *15,* (2000), p. 139-178.
[54] Ovaska, P., Rossi, M., and Marttiin, P., Architecture as a coordination tool in multi-site software development. *Software Process: Improvement and Practice*, *8,* 4 (2003), p. 233-247.
[55] Paasivaara, M. and Lassenius, C., Collaboration Practices in Global Inter-organizational Software Development Projects. *Sofware Process Improvement and Practice*, *8,* (2000), p. 183-199.
[56] Paasivaara, M. and Lassenius, C. Could Global Software Development Benefit from Agile Methods? in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[57] Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, *15,* 12 (1972), p. 1053-1058.
[58] Perry, D.E. and Wolf, A.L., Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, *17,* 4 (1992), p. 40-52.
[59] Ramesh, B., et al., Can distributed software development be agile? *Commun. ACM*, *49,* 10 (2006), p. 41-46.
[60] Sangwan, R., et al., *Global Software Development Handbook*. 2006, Boca Raton, FL: Auerbach Publications.
[61] Sarma, A., Noroozi, Z., and Hoek, A.v.d. Palantír: raising awareness among configuration management workspaces. in *International Conference on Software Engineering*. 2003. Portland, Oregon.
[62] Sarma, A. and van der Hoek, A. Towards Awareness in the Large. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[63] Shaw, M. and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*. 1996, Upper Saddle River, NJ: Prentice Hall.
[64] Sinha, V., Sengupta, B., and Chandra, S., Enabling Collaboration in Distributed Requirements Management. *IEEE Software*, *23,* 5 (2006), p. 52- 61.
[65] Souza, C.R.B.d., et al., *Sometimes you need to see through walls: a field study of application programming interfaces*, in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. 2004, ACM Press: Chicago, Illinois, USA.
[66] Spanjers, H., et al. Tool Support for Distributed Software Engineering. in *International Conference on Global Software Engineering*. 2006. Florianopolis, Brazil.
[67] Teasley, S.D., et al., Rapid Software Development through Team Collocation. *IEEE Transactions on Software Engineering*, *28,* 7 (2002), p. 671-683.
[68] Whitehead, J., *Collaboration in Software Engineering: A Roadmap*, in *Future of Software Engineering 2007*, L. Briand and A. Wolf, Editors. 2007, IEEE-CS Press.