By PÄR J. ÅGERFALK *and* BRIAN FITZGERALD, *Guest Editors*

FLEXIBLE AND DISTRIBUTED SOFTWARE PROCESSES: OLD PETUNIAS IN NEW BOWLS?

Globalization and turbulent business environments are two factors that create significant challenges for software organizations today. In the wake of the IT downturn, many organizations have turned toward globally distributed software development (GSD) in their quest for the silver bullet of high-quality software delivered cheaply and quickly. At the same time, the increasingly volatile requirements in the business environment and the general trend toward leanness have led to a focus on more flexible, agile approaches as a potential solution.

Despite 50 years of software development experience, the perception of the so-called "software crisis" persists in many quarters, with continued instances of software projects exceeding budgets, and development schedules, and exhibiting poor levels of quality when completed—if completed at all. In recent years, agile methods have been proposed as a new, practice-led, paradigm that potentially addresses these problems by promoting communication, flexibility, innovation, and teamwork. Such methods differ significantly from the traditional plan-based approaches emphasizing development productivity rather than

Illustration *by* Paul Zwolak

process rigor. Thus, they seek to accomplish only those development tasks that deliver business value quickly, while accommodating changing user requirements.

Practice is ahead of research in this area, and the use of agile approaches appears to be growing rapidly, but the fundamental underpinnings of agile methods need to be better conceptualized and theorized. However, agile methods are not the only result of the quest for flexible software processes. A similar aim can be found in the process tailoring and method engineering literature [3, 4, 7, 8]. Here, the focus is on adapting development processes to changing circumstances, albeit usually in the context of rigorous plan-based methods. Ironically, the suggestion by agile method advocates that agile methods must be applied in their entirety, as the benefits only arise through the synergistic combination of individual practices, is somewhat at odds with the spirit of flexibility, and not borne out in recent research that suggests agile methods should be flexibly tailored to the particular development context to achieve maximum effect [6].

Against this background, the software development environment is changing. The general trend toward globalization has particular implications for software development [2]. There are many *potential* benefits from GSD, including reduced development costs; reduced cycle time from follow-the-sun software development across multiple time zones; cross-site modularization of development work; access to a larger and better skilled developer pool; innovation and shared best practices; and closer proximity to customers.

However, as illustrated in the accompanying table, GSD also surfaces significant challenges in relation to communication, coordination, and control issues. While geographical distance by itself may induce a number of problems, increased geographical distance also often increases temporal distance and sociocultural distance. When people are not co-located they often must rely on asynchronous communication channels, such as email, and when working in different time zones, they cannot always expect to find the right person at the right time—an example of the emergence of a temporal distance. Similarly, when people from different countries and with different backgrounds collaborate, their frames of references, work habits, and language may differ, which can often lead to great frustration—an example of sociocultural distance. Interestingly, most GSD research seems to take the assumed benefits more or less for granted and focuses primarily on the problems associated with GSD. Our own ongoing research, on the other hand, indicates that many of the proclaimed benefits are simply not realized.

Although there is a steadily increasing body of literature on each of these two phenomena—process flexibility and agile methods on the one hand and GSD on the other—their combination is poorly understood, although expected to be beneficial. Interestingly, agile methods and GSD appear to be largely incommensurable. Due to physical separation of development teams in GSD, many of the key concepts within agile development, such as pair-programming, face-to-face interaction, and onsite customers, are difficult to apply. Also, given the risks inherent in GSD, the natural tendency is probably to favor plan-based approaches.

	Temporal Distance	**Geographical Distance**	**Sociocultural Distance**
Communication	+ Improved record of communications - Reduced opportunities for synchronous communication	+ Closer proximity to market + Access to remote skilled work forces - Face-to-face meetings difficult	+ Innovation and sharing best practice - Cultural misunderstandings
Coordination	+ Coordination needs can be minimized - Typically increased coordination costs	+ More flexible coordination planning - Reduced informal contact can lead to lack of critical task awareness	+ Greater learning and richer skill set - Inconsistent work practices can impinge on effective coordination - Reduced cooperation arising from misunderstanding
Control	+ Time zone effectiveness can be utilized for gaining efficient 24x7 working - Management of project artifacts may be subject to delays	+ Communication channels can leave an audit trail - Difficult to convey vision and strategy - Perceived threat from training low-cost "rivals"	+ Proactiveness inherent in certain cultures - Different perceptions of authority can undermine morale - Managers must adapt to local regulations

Issues in GSD with examples of potential opportunities (+) and threats (-) [1].

Given the complex nature of these topics, we solicited short commentaries from a number of domain experts to set the scene for this special section through a virtual panel debate. We begin with David Parnas, our distinguished colleague at the University of Limerick, who is more qualified than most to assess the relative merits of the issues at stake. He suggests, with healthy skepticism, that while agile methods may correctly diagnose the problems, they are not the right solution. He contends the problems associated with GSD are not significantly new and different from traditional software development problems. His message is that the root of the problem is poor documentation and poor understanding of the role of documentation. Agile methods that try to avoid documentation as far as possible is not the right way to go, he says.

DAVID PARNAS

AGILE METHODS AND GSD: THE WRONG SOLUTION TO AN OLD BUT REAL PROBLEM

I have been hearing the term "software crisis" for more than 40 years. Clearly, it is not a crisis; it is a chronic problem. Each time someone uses the term "crisis," it is a preface to the announcement of a new miracle cure for the problem. Examinations of the cure usually reveal new words for ideas that have been tried before without much effect. This decade's crisis is global software development (GSD); this decade's miracle cure (for all the ills of the industry) seems to be a collection of methods known as "agile."

I do not believe the problems associated with GSD are new problems and I do not believe agile methods are the right solution. The symptoms of the problems are as old as the industry. They include:

- Software is rarely delivered on time, schedules slip repeatedly.
- The ultimate slip—cancellation, with nothing to show for the effort.
- Software so poor that system cannot be used.
- System solves the wrong problem (requirements not met).
- System out of date before it is in use (requirements change).
- "Too many frills, not enough lifting power."
- Staff burn-out (software not maintainable by new staff).
- Seemingly small changes cause huge effort.

These problems were experienced when software was developed by co-located programmers; they are now experienced when the programmers are far apart. When one looks behind these symptoms, one finds they are caused by communication problems:

- There is poor communication between users and developers.
- There is poor communication between architect and programmers.
- There is poor communication among the programmers.

When this was first observed (as early as the 1960s), it was recognized that documentation was a key problem. As a development manager told me in 1969, "We do not know how to write specifications for components, or to write requirements specifications—that's why we have these problems." That manager, and many others, asked researchers to figure out how to write better specifications and other kinds of documentation.

Computer scientists did respond by doing what we all do best—the things we have long been doing. Philosophers and logicians wrote specifications that looked just like axiom systems. Algebraists wrote specifications that defined algebras and argued about what type of algebra was needed. Developers, who were looking for real solutions, looked and then turned away. This led to two schools of thought among developers: the first developed process standards requiring huge amounts of wordy documentation; the second said, "Code is a document and all the documentation we need."

The pendulum began to swing between the two extremes. The huge wordy standards produced documentation that was vague, poorly organized, and difficult to use. Those who tried to skip to the code, continued to produce code that was poorly organized and needed constant refactoring. Their maxim was that it is code we must produce so let's always produce code. Documents and meetings do not produce code. Unlike code, a document is not testable, so why bother? To avoid the poor communication about requirements, they advocated keeping the customer involved in code development. Essentially, they wanted to keep coders in contact with customers and bypass non-productive architects or others that write documents.

What could be wrong with this no nonsense approach? All the things that were wrong when no documentation was written!

GSD has exacerbated the communication problems. If it is difficult to communicate with the person at the next desk, it is more difficult to communicate with someone half a world away. However, this is not really new. I remember software developed more than 30 years ago by two groups, one in Washington D.C. and the other in San Diego. Today, it is a lot easier to communicate data with Sydney than it was to communicate between the two coasts of the U.S in 1975. We knew then, and should still know, that you cannot communicate orally the detailed information needed to produce good software. The solution is neither to add more documentation nor to abandon documentation—it is to get better documentation. Just as the huge and burdensome documentation standards were not a solution, neither is returning to the days of no documentation.

It is fashionable today to speak of "grand challenges." The real grand challenge is not to find ways to avoid documenting, but to find ways to produce useful documents—documents that take time but save more time. We will find that real agility comes from good design that is well documented in precise, lean documentation.

Parnas's piece certainly brings to mind the image of petunias in Douglas Adams's *The Hitchhiker's Guide to the Galaxy:* "Curiously enough, the only thing that went through the mind of the bowl of petunias as it fell was 'Oh no, not again.'" To open up the debate, we asked another founding father of software engineering, Barry Boehm, to respond to Parnas's comments. From his remarks, we gather it is, in fact, not a question of to document or not to document. Rather, it is a question of not falling into the trap of believing there is such a thing as a one-size-fits-all method. Obviously, this then brings us back to the question of tailoring. It appears agile methods per se may not be the answer to the required development process flexibility. It is how these methods are tailored and enacted that is central.

BARRY BOEHM

ONE-SIZE-FITS-ALL METHODS: THE WRONG SOLUTION TO NEW PROBLEMS

David Parnas is right when he says that distributed development is not new. What is new is the radical speedup of global communications and computing. This enables distributed software teams to capitalize on labor cost differentials, multiple time zones, and local knowledge to adapt to rapid changes in the marketplace, in the competition, in IT infrastructure, and in technology more quickly and cost-effectively than their competitors. Some good references documenting these trends are Thomas Friedman's *The World Is Flat* and the recent ACM Task Force Report on *Globalization and Offshoring of Software.*

In producing software cost estimates with our COCOMO II cost model, we are seeing significant changes in its Requirements Volatility cost driver ratings, especially for applications needing to adapt to rapid competitive change. Requirements Volatility ratings for stable embedded devices are still in the 0.1% to 0.3% per-month range, but the counterpart ratings for rapidly changing competition-driven applications are often in the 10% to 30% per-month range.

As a good example of thorough documentation of a relatively stable, medium-size application, Parnas's methods were used to produce a 523-page document providing the software requirements for the A-7E aircraft's operational flight program. With a requirements volatility of 0.1% to 0.3% per month, this would require only 0.5–1.5 pages per month to be changed, to first order. However, a similarly sized application performing the A-7E's electronic warfare function or its commercial equivalent during a period of rapid countermeasure/counter-countermeasure interaction could easily have a change rate of 10% to 30% per month, requiring the developers to renegotiate and rewrite on the order of 50–150 pages of documentation per month. Such a situation would leave the project spending more time in rewriting documents than in changing the software to stay ahead of the competition.

Fortunately, Parnas provided us with a great method for handling a lot of this change. His 1979 paper, "Designing Software for Ease of Extension and Contraction" still contains the best strategy available for anticipating and reducing the cost of change. This is to identify the major sources for requirements change and to encapsulate them as information-hiding module secrets. Then when an anticipated change comes, you only need to change one module to adapt your application.

Unfortunately, though, not every change is foreseeable, particularly in competitive or unprecedented situations. It is in these situations that the agile methods people have come up with good strategies: to engage a committed customer representative as part of the development team, and to invest in team-building activities such as planning games, daily stand-up meetings, pair programming, collective code ownership, and continuous integration that create shared tacit interpersonal knowledge rather than explicit documented knowledge. Then when an unanticipated change comes, your team members can rely on their shared tacit knowledge and team cohesion to rapidly adapt the application to accommodate the change. In many situations, this approach works better than Parnas's one-size-fits-all "always document" approach.

Again, unfortunately, many agile advocates treat their solution approach as a one-size-fits-all solution as well, with such slogans as "avoid Big Design Up Front (BDUF), because you aren't going to need it (YAGNI)." Even agile leader Kent Beck acknowledges that relying on pure tacit, undocumented knowledge on large projects is not workable. On the other hand, even on large projects, there are likely to be portions of the application with requirements volatility rates sufficiently high to make operating on tacit knowledge preferable to spending a lot of time updating documentation. For such projects, some kind of hybrid agile and document-driven approach appears best. How can you organize such a hybrid approach?

There are no one-size-fits-all solutions. The best way I have been able to find is to use risk as a way to determine where to go agile and where to go document-driven. Thus, for example, if you are developing a graphic user interface (GUI) for an unprecedented decision support system and want to document its requirements, the most frequent answer you will get from users is, "I can't tell you in advance, but I'll know it when I see it (IKIWISI)."

In such a case, it is a high risk to try to document the GUI in advance, and with a GUI builder tool, it is a low risk not to document it.

On the other hand, when you are outsourcing a relatively stable piece of business logic to a contractor 10 time zones away, it is a high risk not to invest in a significant amount of thorough documentation of the interfaces and protocols connecting the outsourced software to the rest of your software. And it will be important to encapsulate any agile portions of the software within information-hiding modules, following Parnas's guidelines.

Thus, in many competitive 21st century applications, it will be important to avoid one-size-fits-all solutions, and to use risk considerations to determine which parts of an application are best handled by explicit documented knowledge, and which parts are best handled by tacit interpersonal knowledge.

We also asked an academic with a specific interest in the area of agile methods, Giancarlo Succi (author of *Extreme Programming Examined)* to respond. Here, he invokes what he terms the Kantian categorical imperative of doing good and avoiding evil to characterize the debate. This is essentially the problem of means-ends inversion, whereby the endeavor to do good things, such as documentation, is always blindly followed to the expense of those real value-added development activities suited to the needs of the particular context.

GIANCARLO SUCCI

AGILE METHODS: BETWEEN CATEGORICAL IMPERATIVES AND LEAN PRODUCTION

Parnas uses the term "chronic" to refer to the problems of software development—and as one of the founders of software engineering, he is a trustworthy source. He correctly identifies communication among key stakeholders—developers, managers, customers, and users—as one of the greatest problems in software development. Moreover, he is also completely correct in claiming that not writing any documentation when developing software in a geographically and temporally distributed fashion exacerbates the problem. Ultimately, he acknowledges that the grand challenge for today's software engineers is to write useful documentation.

And here we are. We know that building any system, including software systems, involves several phases—analyzing what we want to do, planning and designing it, constructing it, and so on. It also involves documenting such phases so that others will be able to maintain the system over its lifetime of perhaps up to 20 years.

The fact is that when people have (correctly) realized it is better to structure and to document the process of building software systems, they have often taken a very dogmatic approach to these two tasks. It is as if they are driven by the fear of not having enough structure and documentation.

Kant, the famous German philosopher, claims that all ethics are guided by a categorical imperative: "Do good, avoid the bad!" Such a Kantian categorical imperative seems to have been applied to software engineering: "Do good structures and documentations, avoid the bad!" as the focus has been for decades on developing structure and documentation, while the true reasons for which they have been developed has slowly been forgotten, transforming them from means to ends.

Altogether, methodologies and formalisms have been built to have ever stronger structure and ever more accurate documentation—but the meanings of "stronger" and "more accurate" have not been properly defined, nor has enough emphasis been placed on the fact that different application domains have different needs. It is as if building control software for a nuclear power plant is the same as developing a system to book tennis courts. We have lacked understanding of time and effort dimensions when structuring and documenting, as if "better" always meant "harder" and "more complex" to do and understand, regardless of the associated time, effort, and cost.

And here we encounter the lean revolution. The lean revolution is not new—it dates from manufacturing in the early 1950s. The lean approach does not advocate ignoring any structure and documentation. Rather, it aims at something totally different: the separation of the activities that bring value to the user from those that do not, and the consequent elimination of such useless activities called "muda" (garbage in Japanese).

In software we know the live system gives value to the user, as does the source code, and the former is automatically derived from the latter. We cannot do without them. Everything else is questionable. This is the lean revolution. Questionable does not mean useless. Rather, it means "subject to research" and this is what we do!

No one thinks that documentation is useless. But consider a system developed by a team of smart programmers in Smalltalk to run the payroll of a small company. Would it be better for such a team to document a sound selection of variables, methods, and class names, or a lot of comments and associated reports. Which approach is more understandable for users, more likely to be written (and read!), less likely not to contain mistakes, more robust to code evolution, more cost effective?

No one thinks that analysis and design are useless. But consider a system to dispatch tracing messages and other information to a group of trucks. This domain is likely to be alien to most software developers. In such a case, would it be better to first spend a lot of time analyzing the system requirements, then a lot of time doing the upfront design, and eventually writing the code, or to work incrementally, involving the end customer, interleaving some analysis, design, and even coding, so that developers grow their knowledge of the system domain? Would it be better to use sound, comprehensive, formal languages for analysis, design, and code, or to use a single, unique language for analysis, design, and code—the language used to write the final system, the ultimate desire of the customer?

While I have not seen a better ethical approach than the Kantian categorical imperative, such an approach should not be blindly applied to software development. Lean production, and its correlate in software engineering agile methods, reminds us of this.

To shed some light on the tailoring issue raised by both Boehm and Succi, we asked one of the leading practitioners in agile GSD to comment. Matthew Simons, Managing Director of ThoughtWorks India and a prolific writer on the topic, confirms that agile methods must indeed be tailored for GSD. As he describes here, their tailored story cards clearly illustrate the need for appropriate documentation, while still experiencing the benefits of agile development—even in a GSD context.

MATTHEW SIMONS

GLOBAL SOFTWARE DEVELOPMENT: A HARD PROBLEM REQUIRING A HOST OF SOLUTIONS

Globally Distributed Software Development (GSD) is one of the megatrends shaping our industry. It presents a special challenge not because it introduces new ways for software projects to fail, but because it drastically complicates communication. As David Parnas rightly points out, the root cause of most software failures is ineffective communication. So it follows that as communication becomes more difficult the risk of project failure escalates.

Parnas comes out against agile methods, which he feels are being promoted as a silver bullet to address the challenges of GSD. He focuses specifically on the avoidance of documentation that some practitioners of some agile methods espouse and proposes that this will never suffice in a distributed context. He closes with the argument that there is value to be gained from investments in producing better documentation.

My experiences working with globally distributed teams over the past five years lend some support to Parnas's advocacy for effective documentation. We have found that in the distributed context, pure agile development with little or no documentation beyond code is impractical and inefficient. However, we have also found the full set of agile practices, which encompass much more than just an approach toward documentation, address the challenges of communication in distributed teams better than anything else we've come across.

For context, I work in India with teams distributed mainly between India and the U.S. or U.K. Most of our teams follow an approach close to Extreme Programming, where the standard artifact is the story card. The idea is that a few brief sentences written on an index card can serve as a placeholder for a discussion that will later take place between a developer and a customer. That conversation is where all the detail required to develop the feature will come out, without the overhead of documenting everything more formally.

While the idea of story cards appeals to those with no desire for reading or writing technical documentation, the reality of the situation is that such scanty artifacts are rarely sufficient for development of anything beyond simple systems for highly accessible individuals or small groups of users. This scenario rarely applies to GSD. As the picture becomes more complex we have found the story card-only approach quickly becomes inadequate.

Rather than abandon lightweight documentation entirely, we supplement our story cards with concise documents that capture additional useful details. Typically this includes a crisp textual description of the feature, context on the business driver, a screen shot (if appropriate), a list of likely impacts on other parts of the system and, most importantly, a set of functional test cases. These tests clearly define what automated tests must be written and shown to pass before the feature is considered complete.

The best examples of these supplementary documents are no more than about two pages long. Anything longer and you risk investing so much effort in system documentation that it starts to become a hindrance. If you find people on your team who are spending hours getting every detail in their documents just right or updating reams of documentation instead of testing the system and clarifying developer queries, you have probably passed beyond the limits of lightweight documentation. Two pages are about the upper limit beyond which the majority of developers are likely to get bogged down and start ignoring the documents (sad, but true).

Breaking your system down into small pieces and describing those pieces precisely is an advanced skill that takes time to master and deliver. I like to think that if Parnas had a chance to review the documents we are using to support our agile process he might consider them to be good examples of what he is looking for.

While writing effective documents is a good place to start, we have found it takes a lot more than that to deliver complex systems with distributed teams. Without the discipline that comes from the remaining agile practices (such as Test-Driven Development, Continuous Integration, Pair Programming, among others) good documents are nothing more than a step on the long and perilous path toward successful delivery.

Disregarding agile methods as an effective response to the challenges of GSD on the basis of their stance toward documentation alone doesn't do them justice. Doing so could prevent you from gaining access to the significant benefits in risk reduction and quality that properly implemented agile methods have to offer to distributed development.

Unfortunately, and to the undoubted chagrin of all, we did not have the time and space to let these virtual panelists respond further to each other's remarks. We expect this brief debate will stimulate some thoughts, and hopefully there will be a forum for continued discussion elsewhere.

In addition to these distinguished virtual panelists' reflections on agility, flexibility, and globalization, we also solicited two types of research submissions: feature articles and commentaries. Having received more than 60 submissions, we are pleased to present the five pieces selected for publication after a comprehensive peer-review process.

Gwanhoo Lee, William DeLone, and Alberto Espinosa explore the tension between flexibility and rigor and suggests that successful GSD requires both agility/flexibility and rigor/discipline. They then derive a set of ambidextrous coping strategies to be used in GSD projects for achieving the required balance between the two extremes.

Balasubramaniam Ramesh, Lan Cao, Kannan Mohan, and Peng Xu focus on how distributed software development can be agile. The challenges to agile GSD they identify—communication, control, and trust—echo Parnas' message about the importance of documentation. However, the strategy they suggest is more in line with the agile method sentiment and aims to create formal arenas for informal communication and knowledge sharing.

One-Ki (Daniel) Lee, Probir Banerjee, Kai Lim, Kuldeep Kumar, Jos van Hillegersberg, and Kwok Kee Wei consider how agility can be achieved in GSD projects by the proper alignment of IT strategy, IT infrastructure, and IT project management. The framework they present points out many important issues beyond those typically addressed by software process models and agile methods.

Clearly, achieving agility and flexibility in GSD is not only about software process management but also about business strategy and IT infrastructure: global business objectives are what should drive GSD, not specific development techniques or software engineering fads.

All three articles are based on solid case-based empirical research and as such reflect contemporary GSD practice quite well. Interestingly, they arrive at somewhat contradictory conclusions, for example, regarding the viability of achieving follow-the-sun software development. While Lee et al. found that minimization of task dependencies facilitated the

FLEXIBILITY AND THE ABILITY TO ADAPT TO DIFFERENT CIRCUMSTANCES ARE NECESSARY FOR SUCCESSFUL SOFTWARE DEVELOPMENT—BE IT GLOBALLY DISTRIBUTED OR NOT.

implementation of follow-the-sun development, Ramesh et al. found this model to be far from reality in most cases. Although this latter conclusion accords with our own experience in the area, the inconclusiveness seems to indicate a need for further in-depth studies to understand this complex issue better in line with Carmel's study of Infosys [5].

In their commentary, Patrick Wagstrom and James Herbsleb present an application that can be used to elicit source code dependencies, and based on those predict future communication needs between developers. As highlighted by our virtual panel debate and the three articles here, communication is indeed a key concern in software development, and is emphasized more than ever in the context of GSD and agile methods. The kind of tool presented is thus likely to play an increasingly important role in planning and managing GSD projects.

Nick Flor, author of the first academic paper on qualitative benefits of pair programming, explores the future of pair programming in the GSD context—remote pair programming. His commentary concludes by noting the seven properties that make traditional pair programming successful can also be achieved remotely with a proper cross-workspace information infrastructure.

The articles and commentaries in this special section provide a snapshot of an increasingly important area both theoretically and practically. Our virtual panel debate indicates we are dealing with what are largely subjective views and what works in one context may not work in another. In a sense this reinforces the whole idea of this special section: Flexibility and the ability to adapt to different circumstances are necessary for successful software development—be it globally distributed or not. The articles then go on to present a number of useful insights for how to achieve the required flexibility in GSD particularly.

Returning to Douglas Adams's petunias, who having experiencing a few moments of existence, quickly concluded that it was all just a repetition of previous experience. Whether or not flexible and distributed software processes offer an improved future for software development or simply represent old petunias in new bowls is yet to be seen. However, the associated challenges addressed in this special section are real and increasingly important. We hope you enjoy reading this section as much as we enjoyed putting it together. C

REFERENCES

1. Ågerfalk, P.J., Fitzgerald, B., Holmström, H., Lings, B., Lundell, B., and Ó Conchúir, E. A framework for considering opportunities and threats in distributed software development. In *Proceedings of the International Workshop on Distributed Software Development* (Paris, Aug. 29, 2005). Austrian Computer Society, 47–61.
2. Aspray, W., Mayadas, F., and Vardi, M.Y., Eds. *Globalization and Offshoring of Software: A Report of the ACM Job Migration Task Force.* ACM, 2006; www.acm.org/globalizationreport.
3. Basili, V.R. and Rombach, H.D. Tailoring the software process to project goals and environments. In *Proceedings of the 9th International Conference on Software Engineering.* (Los Alamitos, CA, 1987). IEEE Computer Society Press, 345–357.
4. Cameron, J. Configurable development processes. *Commun. ACM 45*, 4 (Apr. 2002), 72–77.
5. Carmel. E. Building your information systems from the other side of the world: How Infosys manages time zone differences. *MIS Q Executive 5*, 1 (Mar. 2006), 43–53.
6. Fitzgerald, B., Hartnett, G., and Conboy, K. Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems 15*, 2 (2006), 197–210.
7. Fitzgerald, B., Russo, N.L., and O'Kane, T. Software development method tailoring at Motorola. *Commun. ACM 46*, 4 (Apr. 2003), 65–70.
8. Kumar, K. and Welke, R.J. Methodology engineering: A proposal for situation specific methodology construction. *Challenges and Strategies for Research in Systems Development.* W.W. Cotterman and J.A. Senn, Eds. John Wiley, Washington, DC, 1992, 257–269.

Pär J. Ågerfalk (par.agerfalk@ul.ie) is an assistant professor at Örebro Univesity and a researcher at Lero—The Irish Software Engineering Research Centre, University of Limerick.

Brian Fitzgerald (bf@ul.ie) is the Frederick A. Krehbiel II Chair in Innovation in Global Business and Technology at Lero—The Irish Software Engineering Research Centre, University of Limerick.