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.

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