What Do We Know about Scientific Software Development's Agile Practices?

The development of scientific software has similarities with processes that follow the software engineering "agile manifesto": responsiveness to change and collaboration are of utmost importance. But how well do current scientific software-development processes match the practices found in agile development methods, and what are the effects of using agile practices in such processes?

Over the years, software engineering (SE) practice and research have focused on techniques and concepts intended to be generally applicable to software development. However, SE best practices and research have rarely been adopted in scientific software development.[1] Here, we use *scientific software* to refer to software developed by scientists for scientists. Such software implements complex algorithms to solve systems of mathematical equations, provide simulations, and so on.

Because employing such practices might aid in the scientific software-development process, in this article we survey and analyze how much and how well current scientists employ best SE practices for scientific computing, taking into consideration why they approach the development process in various ways. In particular, we're interested in the use of *agile practices* for scientific software development—that is, in the use of lightweight, incremental processes that involve the customer's continuous feedback.

1521-9615/12/$31.00 © 2012 IEEE
Copublished by the IEEE CS and the AIP

Magnus Thorstein Sletholt
Distribution Innovation
Jo Erskine Hannay
Simula Research Laboratory
Dietmar Pfahl
Lund University
Hans Petter Langtangen
University of Oslo and Simula Research Laboratory

Contextualizing to Define Our Research

In most aspects of scientific software development, the urge to conduct science is the primary motivation and goal. Unlike software engineers, the scientist's mindset is to perform science, not to write software.[2] Development methods that emerge usually are based on local experience.[3] Also, the variation in domains, maturity of the science, and motivation in scientific software projects influence development methods, and thus we should expect large variations both across and within domains.

Nevertheless, some common ground can be found. Scientists use their software to perform complex calculations or simulations. In some scientific projects, scientists use the software to test a scientific theory. These characteristics of scientific software entail that, in contrast to the development of, say, administrative or business-enterprise software, the scientific software writer can't determine what an application's correct output should be in the traditional sense. Also, the software might evolve through the combined effort of many scientists over the course of many years, continuously adding new system functionalities.[4] This poses particular challenges from the software engineering viewpoint: First, the *requirements* elicitation and specification will be highly dynamic. Because of the exploratory nature of many scientific projects, requirements elicitation and specification is difficult because they might be unclear, or even unknown, up front. Second, because requirements are so volatile, testing the software with regards to such requirements is often problematic.

In fact, the lack of knowledge about requirements and testing principles has been identified as a problem area in several studies.[3,5,6] One survey article noted that requirements-related activities are perceived as problematic in scientific software projects.[5] We also identified that scientists perceive the definition of test cases for software validation and verification as challenging. For example, it's often not obvious to stipulate whether an error lies within a scientific theory or in that theory's implementation (numerical approximation). Moreover, technical testing skills seemed to be a clear weak point for scientists developing software.

The challenges with determining requirements up front and the subsequent testing have been addressed explicitly in SE agile practices. Could scientific software development lend itself more to agile-oriented practices than to traditional plan-driven practices? Rebecca Sanders supports this possibility by stating that most projects under investigation in her study took an iterative, rather than a plan-oriented, approach to development.[6]

Adopting an explicit process model should enable projects to benefit from SE best practices and research. However, it's well known that technology adoption relies on a sufficient number of elements shared between the technology and the problem domain. It's therefore worth investigating the extent to which more appropriate process models harmonize with scientific software development. Thus, we defined two research questions:

- How well do practices in current scientific software-development processes match the practices found in agile development methods?
- How does the use of agile practices influence the handling of common challenges in scientific software-development projects?

Regarding the first question, we were interested to find out which, if any, agile practices were used. Related to the second question, we investigated whether using relevant agile practices yields a better handling of testing-related and requirements-related activities.

Agile Practices: Literature Review

Agile practices are currently being adopted by more and more projects, including large projects with complex architectures. For the purpose of our analyses, we identified 35 agile practices (see Table 1). The first 12 practices in Table 1 originate from the Scrum methodology.[7] The remaining 23 elements are Extreme Programming (XP)[8] practices (see the "Agile Software Development" sidebar). The elements marked with an asterisk are XP practices that are also recommended practices in the Scrum methodology. We discuss how we selected these agile practices elsewhere.[9]

To review evidence of agile practices in scientific software projects, we performed a literature review that extracted and critically appraised available literature on the subject. We conducted the literature review in a similar fashion to the method described by Tore Dybå and his colleagues,[10] searching multiple literature databases in a systematic manner.[9]

The practice numbers in Table 2 refer to the agile practices (with the same numbers) in Table 1. An "×" in a cell of Table 2 indicates that we found evidence that the practice wasn't present. A check indicates that we found evidence that the practice was present. Blank fields indicate that we were unable to determine whether a practice was followed from the available information.

Relevant Articles and Our Initial Findings

Although there are more than 100 publications reporting on scientific software-development projects, our literature search (and subsequent filtering) left us with only five articles that addressed the possible use of agile practices in such projects:

1. "Engineering the Software for Understanding Climate Change";[11]
2. "Chaste: Using Agile Programming Techniques to Develop Computational Biology Software";[12]
3. "Agile Methods in Biomedical Software Development: A Multi-Site Experience Report";[13]
4. "Exploring XP for Scientific Research";[14] and
5. "Introducing Agile Development into Bioinformatics: An Experience Report."[15]

Table 1. List of agile practices.

Practice number	Agile practices
1	Priorities (product backlog) maintained by a dedicated role (product owner)
2	Development process and practices facilitated by a dedicated role (Scrum master)
3	Sprint planning meeting to create sprint backlog
4	Planning poker to estimate tasks during sprint planning
5	Time-boxed sprints producing potentially shippable output
6	Mutual commitment to sprint backlog between product owner and team
7	Short daily meeting to resolve current issues
8	Team members volunteer for tasks (self-organizing team)
9	Burn down chart to monitor sprint progress
10	Sprint review meeting to present completed work
11	Sprint retrospective to learn from previous sprint
12	Release planning to release product increments
13	User stories are written*
14	Give the team a dedicated open work space*
15	Set a sustainable pace*
16	The project velocity is measured*
17	Move people around*
18	The customer is always available*
19	Code written to agreed standards*
20	Code the unit test first
21	All production code is pair programmed
22	Only one pair integrates code at a time
23	Integrate often
24	Set up a dedicated integration computer
25	Use collective ownership*
26	Simplicity in design*
27	Choose a system metaphor
28	Use class-responsibility-collaboration (CRC) cards for design sessions
29	Create spike solutions to reduce risk*
30	No functionality is added early
31	Refactor whenever and wherever possible
32	All code must have unit tests
33	All code must pass all unit tests before it can be released
34	When a bug is found tests are created
35	Acceptance tests are run often and the score is published

** Denotes Extreme Programming (XP) practices that are also recommended practices in the Scrum methodology.*

To see what information the reviewed publications revealed concerning the first research question, we mapped the agile practices listed in Table 1 to development practices used in the projects described in the five selected articles. Table 2 shows the result of this mapping. Articles 1, 2, 4, and 5 describe exactly one project, while article 3 describes six projects (labeled as 3.1, 3.2, …, and 3.6, respectively, in Table 2).

Regarding our second research question, all five articles indicated positive effects of agile practices in scientific software development. A tentative conclusion is that agile methods can effectively handle the special characteristics of requirements and testing in scientific software development. The evidence in favor of such a conclusion is stronger for small projects with relatively few team members.

Agile Software Development

Agile practices emerged in the mid-1990s as an alternative to the traditional, plan-driven approach to software development. The practices are intended to address the problems in meeting customer requirements when the requirements were specified and locked early—the observation being that requirements will change over time as the customer and software developer become aware of further needs and constraints. Agile practices imply lightweight, incremental processes that fully and continuously involve the customer and that are adaptable to changing requirements. In 2001, a group of software engineers formulated the "agile manifesto" (http://agilemanifesto.org), outlining profound principles of agile development. Many elaborations and specializations of agile practices exist. Two agreed-upon elaborations that capture agile development comprehensively are Scrum and Extreme Programming (XP).

Scrum[1] is an organizational process model that defines roles in a development project, as well as the activities that Scrum teams will perform. Each team is largely autonomous and works in two-to-four-week iterative increments (sprints). Scrum teams consist of a Scrum master, a product owner, and regular team members (developers and testers). The Scrum master's primary objective is to facilitate communication and to keep the team's productivity (velocity) on a satisfactory level. The product owner ensures customer involvement and communication. Scrum scales up (for example, by "Scrum of Scrums"—meaning that many first-order Scrum teams can work in parallel while coordination of these Scrum teams is done via a second-order Scrum team to which each first-order Scrum team sends a representative). Sprints are time boxed, meaning that they have a fixed time and flexible scope, but every sprint should produce a functioning part of the system (potentially shippable code). Sprint planning and estimation is based on the team's recorded historical productivity, thus providing reliable and constantly updated scope-time estimates.

XP[2] also focuses on close customer–developer relationships and communication in short-time iterations. Rather than an organizational framework, XP describes work practices in some detail. Among the most central practices are pair programming, continuous code review, testing and refactoring, and distributed competence among developers.

References

1. M. Cohn, *Succeeding with Agile: Software Development Using Scrum*, Addison-Wesley, 2009.
2. D. Wells, *The Rules of Extreme Programming*, 2009; www.extremeprogramming.org/rules.html.

The testing approaches' rigor seemed to satisfy the need for having reproducible, correct results.[15] For requirements activities, we identified a degree of mismatch between scientific software projects and the agile-assumed context of a clear customer–developer relationship. However, the agile methods' responsiveness and flexibility proved valuable for the requirements activities. Elicitation and specification of tasks were perceived as easier and more focused with agile methods.[12,14] Good practices regarding requirements prioritization were also observed.[11]

Thus, the literature review indicated that projects using agile practices better handle testing-related activities. The review also supports the assumption that projects using agile practices are better at handling requirements activities, but the findings aren't as substantial as for testing.

Agile Practices: Case Study

To complement our questionnaire-based survey[5] and literature review,[9] we conducted a multiple-case study (see the "Emprical Research Methods" sidebar) that compared three large scientific software projects: Finite Elements in Computational Science (FEniCS), Dalton, and Olga (see Table 3). The case study added some dimension to the results found in the projects investigated in the literature review. We selected these three projects because they represent different types of scientific software than the projects investigated in the review, as they're much larger in terms of size, duration, and participants. These cases also extended the range of scientific domains beyond that of bioinformatics. In addition, we had easy access to scientists involved in software-development activities. Thus, the selection of cases was to a certain degree opportunistic.

The case study's purpose was to

- analyze and conceptualize core product and development-process elements in the three projects;
- investigate the extent to which these elements map to agile practices (as listed in Table 1); and
- investigate the effects of agile practices on testing and requirements handling, as perceived by project participants.

The following sections characterize each of the case study projects and provide information related to two research questions. Table 4 shows the agile mapping chart of all three projects. As you can see, most of the agile practices weren't

Table 2. Mapping chart of agile practices from the literature review.

Practice number	Projects									
	1	2	3.1	3.2	3.3	3.4	3.5	3.6	4	5
1	✗								✗	
2	✗								✗	
3									✗	
4	✗								✗	
5		✓	✓	✓	✓	✓	✓	✓	✓	✓
6	✗								✗	
7	✗	✓	✓			✓	✓	✓	✓	✓
8	✓		✓	✓	✗	✓	✓	✓	✓	
9	✗	✓							✓	
10									✗	
11									✓	
12	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
13	✗	✓	✓	✓	✓	✓	✓	✓		✓
14	✓		✓	✓	✓	✓	✓	✓	✓	
15									✓	
16	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
17	✗	✓								✓
18	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
19	✓								✓	✓
20		✓								
21		✗	✗	✗	✗	✗	✗	✗	✓	
22									✓	
23	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
24										
25	✓	✓	✓	✓		✓	✓	✓	✓	✓
26									✓	
27									✓	
28										
29										
30	✓									
31		✓	✓	✓	✓	✓	✓	✓	✓	✓
32		✓							✓	
33										
34		✓								
35	✓	✓						✓		

present in cases 1 and 2, but many were detected in case 3.

Case 1: FEniCS

FEniCS is a project with participants from several universities and research institutions. The aim is to automate solving differential equations. The program is open source, free, and distributed through software managers in Ubuntu and Debian.

FEniCS isn't a traditional software application; it's a collection of separate packages that form a framework for the automated solution of differential equations. Scientists then write applications, typically relating to a specific scientific problem, on top of the FEniCS framework/interface. The components are written in C++ and Python. An international, geographically distributed community of developers

Empirical Research Methods

We used several empirical research methods in our line of investigation. Here we provide a brief summary of each method.

Survey

For our preliminary investigations, we used a Web-based survey.[1] Surveys are designed to collect large amounts of subjective data in a schematic form that lends itself to statistical analysis.

Literature Review

In the next stage,[2] we followed guidelines for systematic literature reviews in software engineering.[3] Strict adherence implies that we defined a review protocol that specified our literature-selection criteria, variable extraction procedures, secondary-analysis methods, quality assessment, and inter-reviewer agreement scores. The purpose is to ensure a complete and replicable account of available literature on a topic. For our purposes, we followed a simpler review process,[4] which omitted the formal protocol, multiple reviewers, and secondary analysis.

Case Study Method

The third stage involved a multiple-case study. A case study is an in-depth examination of a selection of contemporary phenomena (agile practices) within a real-life context (actual scientific software-development projects).[5] The first part of our case study was exploratory: We investigated the software-development processes in the three projects "as is." The second part was confirmatory: We mapped and evaluated these processes from an agile viewpoint, according to the agile mapping chart. In addition to examining project documentation, we interviewed between two and four key developers in each project over several sessions, following ethical guidelines. The interviews covered a set of high-level topics, and we later analyzed the content by categories of information that appear under each case in the main text.

Further Methods

To find out more about agile practices' causal effects, future studies might deploy controlled field experiments and further comparative case studies. In addition, mappings from studies on agile practices in other domains could offer insights, if sufficient generalized arguments are viable.

References

1. J.E. Hannay et al., "How Do Scientists Develop and Use Scientific Software?" *Proc. 2nd Int'l Workshop on Software Eng. for Computational Science and Eng.*, IEEE CS Press, 2009, pp. 1–8.
2. M.T. Sletholt et al., "A Literature Review of Agile Practices and Their Effects in Scientific Software Development," *Proc 4th Int'l Workshop on Software Eng. for Computational Science and Eng.*, ACM Press, 2011, pp. 1–9.
3. B.A. Kitchenham, *Procedures for Undertaking Systematic Reviews*, joint tech. report, (TR/SE-0401) Computer Science Dept., Keele Univ., and (0400011T.1) Nat'l Information and Comm. Technology of Australia (NICTA), 2004.
4. T. Dybå, T. Dingsøyr, and G.K. Hanssen, "Applying Systematic Reviews to Diverse Study Types: An Experience Report," *Proc. 1st Int'l Symp. Empirical Software Eng. and Measurement*, IEEE CS Press, 2007, pp. 225–234.
5. R.K. Yin, *Case Study Research: Design and Methods*, Sage Publications, 2003.

Table 3. Characteristics of the case study projects.

Characteristics	FEniCS*	Dalton	Dalton
Scientific domain	Mathematical (automated solution of differential equations)	Chemistry (molecular electronic structures)	Physics (flow modeling of oil, gas, and natural water)
Number of contributors	>10	40	50
Duration	10 years	30 years	30 years
Programming languages	C++, Python	Fortran77/90, C, C++	Fortran, C++, C#
Chosen process method	No specific	No specific	Scrum
Distributed development	Yes	Yes	Yes
Availability	Free, open source	Free, licensed	Proprietary

** FEniCS stands for Finite Elements in Computational Science.*

contributes to the coding and documentation efforts. As with any project with distributed development, collaboration, coordination, and communication are key aspects that must be handled appropriately for the project to be effective and successful.

Table 4. Mapping chart of agile practices from the case study.

Practice number	Projects		
	FEniCS	Dalton	Olga
1	✖	✖	✔
2	✖	✖	✔
3	✖	✖	✔
4	✖	✖	✔
5	✖	✖	✔
6	✖	✖	✔
7	✖	✖	✔
8	✔	✔	✔
9	✖	✖	✖
10	✖	✖	✔
11	✖	✖	✔
12	✖	✖	✖
13	✖	✖	✔
14	✖	✖	✖
15	✖	✖	✖
16	✖	✖	✖
17	✖	✖	✔
18	✔	✔	✔
19	✔	✖	✔
20	✖	✖	✖
21	✖	✖	✖
22	✖	✖	✖
23	✔	✖	✔
24	✖	✖	✖
25	✖	✖	✔
26	✔	✖	✖
27	✖	✖	✖
28	✖	✖	✖
29	✖	✖	✖
30	✖	✔	✔
31	✔	✖	✖
32	✖	✖	✖
33	✔	✔	✔
34	✖	✔	✖
35	✔	✖	✖

Teams and roles. Most of the developers are equal peers, much like open source development. Developers choose how much effort they devote to the project at a given time. Although there are no project leaders in the traditional sense, some implicit project roles exist. Developers who are heavily involved and who contribute more are generally more influential. A core team reviews code changes. Membership in the core team is granted based on a constant level of contributions to the project.

Development process. The development process in FEniCS has been formed over the course of several years. Although the process is undefined and doesn't match any established model, certain practices have been established; for example, code is developed incrementally. It was hard to identify any detailed process activities (and hence, transitions between them), because the development is mostly based on personal initiative and commitment. This could imply several vastly different approaches to development.

Coding is clearly perceived as the most important and time-consuming phase/activity. It's difficult to precisely define tasks beforehand, as they are so closely connected to research. There are multiple aspects that can influence a specific task during both planning and coding, such as changes to the original requirements or specifications and technical difficulties. Because these types of challenges arise often, there's no focus on specifying tasks or estimating efforts for tasks. The overall feel of it is that developing the project is much like conducting research (the output isn't necessarily known), which means that you have to accommodate dynamic requirements.

Requirements and testing. The level of self-organization is apparent in all activities related to requirements handling. Although developers use LaunchPad to coordinate tasks ("blueprints") and to track their progress, the individual developers requesting a specific functionality usually specify and define the tasks themselves. Minor tasks don't have a blueprint or specification. No uniform pattern, such as user stories, is used for specifying tasks.

The project has a dedicated tester, who updates a Buildbot—that is, a system to automate the compile/test cycle to validate code changes. With the help of the Buildbot, the tester checks that the system compiles and that there are no build errors, and he/she also runs regression tests. Apart from that, testing isn't prioritized and it's left up to the individual developer.

General challenges. Obvious challenges for FEniCS relate to collaborating and coordinating the project work. Sometimes, problems can arise when people have different requirements related to the same functionality.

Agile practices. Few agile practices were identified in the FEniCS project.

Case 2: Dalton

The Dalton project is an older scientific software project in the molecular electronic structures' subdomain of chemistry. The aim is to automate computation of such molecular properties. The software was first released in 1997, with several versions in the years to follow; the latest version dates to the first quarter of 2010. An international community of scientists is involved in the program's development.

The software is written in Fortran 77 and C, and the authors recommend a Unix platform. The program consists of seven components, with more or less independent development cycles. The program is distributed free of charge, as long as the user signs a personal license agreement.

Perhaps because of the project's age, no tools for managing code were used initially; source code was exchanged via email. As the number of code lines increased, the need for a source-code revision tool became more apparent.

Teams and roles. It was hard to identify roles. Supervisors to masters and doctoral students assume roles akin to project leaders for their students, but normally the scientists don't assume any specific roles. A board has been established to map out the project's general, future directions, as well as to make decisions on significant matters. However, the board's responsibilities are as much related to scientific research as to the software-development project. There's also a role for users who have signed the license agreement. Such users can request functionality or suggest changes, but there are no guarantees that the suggestions will be implemented.

Development process. No explicit development model exists, but a culture has been established over the years, yielding guidelines for how to attend to certain aspects of the development. Nevertheless, most of the development is performed individually. Aspects requiring collaboration—for instance, integrating code or planning a release—are handled in an ad hoc manner rather than systematically.

None of the interviewees were able to identify any transitions between the activities, as most of these activities (such as coding, analysis, design, and testing) are carried out more or less simultaneously. The development bears certain resemblances to iterative development, but the activities in the Dalton development process aren't formalized at all.

Requirements and testing. The developers are located at various research facilities, most of them in Scandinavia. Occasionally meetings occur to discuss requirements and ongoing activities. The nature of specific scientific research plays a part; but most of the time the full requirements aren't known until far into the implementation phase.

Requirements are handled individually and all the developers have their own private to-do lists. The level of self-organization is high; tasks are both defined and chosen by the developers themselves. Often development is motivated by (personal) research needs. There's seldom any gathering of the requirements on a plenary level; this happens only when a release is imminent. As there are relatively few people involved in the project at a given time, the most active developers seem to have some idea of what other members are currently doing.

There are some regression tests that must be passed before the developer can commit his or her code to the main repository. It's unclear whether there are also unit tests; their absence was pointed out by some but others said that they had written unit tests. Perhaps some parts of the regression-test suite also target single functions in the code.

Challenges. It's challenging to coordinate new software-version releases. It's hard to get all of the scientists to deliver on time, and deadlines have occasionally been postponed. It's also difficult to manage the code and integrate all the different code branches. Requirements are sometimes impossible to stipulate before well into the implementation stage; many tasks are explorative and the correct output might be difficult, if not impossible, to predetermine. This also complicates testing matters, as errors could be in either the theory or implementation.

Agile practices. As with FEniCS, only a few agile practices were identified in the Dalton project.

Case 3: Olga

The third case is Olga. Contrary to the other cases, this is a commercial project, developed by the SPT Group. Olga is a simulator tool for accurate flow modeling of oil, water, and gas in wells and pipelines. Being a commercial system that must stay competitive, Olga has a more traditional type of developer–customer relationship. Another difference between Olga and the other two projects in the case study is the immaturity of the underlying theory on which the software development is based. Although in FEniCS and Dalton the maturity of the underlying scientific theory is high, in Olga the software can be considered as a documentation of the underlying theory.

Teams and roles. In terms of roles, the developers belong to one of three main company departments: R&D, maintenance, and GUI. GUI developers seem to be dedicated solely to their department, while other developers might engage in the other departments' projects. Occasionally, they also work with support (which also might be viewed as a kind of maintenance effort). In the various projects, however, regular Scrum roles are used; every project has a dedicated product owner and a Scrum master. There are also other developer roles in the company that don't pertain specifically to the Scrum methodology, such as a dedicated tester.

Development process. Because of the company's size, the Olga project is organized via multiple subprojects of various sizes and durations. Most of these use Scrum or many of the Scrum practices. The projects' lifecycles aren't synchronized; each project has its own deadlines and backlogs to consider. Project iterations vary from two to four weeks. Releases of the complete software package occur approximately twice a year. Of course, as the projects differ in various aspects, different Scrum practices are variably applied; some of the practices are followed throughout, while others might be absent or carried out unsystematically or incompletely according to the Scrum methodology.

Requirements and testing. Requirements are handled in a tracking system, where tasks must follow a series of states before ultimately being ready for the main repository. Scrum also provides some guidelines on how to deal with the tasks, more specifically the sprint backlog and the sprint-planning meetings. The tasks are planned, broken down into smaller, more manageable parts, and then estimated. Each project can choose its own way of estimating the tasks; some projects use planning poker.

The company has a test suite that covers close to 50 percent of the production code. The tests consist of use cases that check the results of executing the software. This test suite is run quite often (at least once a day). There are also some unit tests in the project, but unit testing is a relatively new aspect for the project, which means that there are no specific or established guidelines related to such tests. The developers' perception is that it's much easier to write unit tests while writing new code, compared to creating or updating unit tests for already existing code. A limited part of the code is currently being addressed by unit tests. As the software is scientifically explorative, the output can't always be verified until someone possesses real, observed data. Comparisons between results from the software and scientific data are conducted on a regular basis.

Challenges. It's difficult to establish testing routines to be performed systematically in the project. For example, although unit testing is regarded as important, not all new code is unit tested.

Whether the output matches observed data is hard to assess prior to collecting the scientific data. This also complicates effort estimation, because the workload associated with a task is so uncertain.

Agile practices. In contrast to the FEniCS and Dalton projects, several agile practices could be identified in the Olga project. Most of the Scrum practices are present, as the project explicitly uses this process methodology.

Discussion

Now that we've provided some background on our multiple-case study, let's discuss the results from the literature review and case study in light of the two proposed research questions.

Presence of Agile Practices

As Tables 2 and 4 show, both the literature review and case study indicate that agile practices are indeed present in projects developing scientific software. However, with the exception of Olga, which deliberately uses the Scrum method, for most of the agile practices listed in Table 1, we couldn't find clear positive evidence as to their application.

Overall, we found that practices 5 (time-boxed sprints), 7 (short daily meetings), 8 (self-organizing team), 12 (release planning), 13 (user stories), 14 (dedicated open work space for team), 16 (project velocity is measured), 18 (customer is always available), 23 (integrate often), 25 (collective ownership), and 31 (refactor whenever and wherever possible) are present in most of the projects. In addition, practice 19 (code written to agreed standards) is present in five out of 13 projects, and we found negative evidence about its usage in only one project. Finally, the use of practice 33 (all code must pass all unit tests before release) was evident in three projects, and we found no project with evidence that it wasn't used. Thus, our literature review and case study indicates that 13 out of 35 agile practices are used in projects developing scientific software.

On the other hand, we found that only practice 21 (all production code is pair programmed) had clear evidence that it's not used in most of the projects. In addition, we found that practices 24 (set up a dedicated integration computer), 28 (use class-responsibility-collaboration, or CRC, cards for design sessions), and 29 (create spike solutions to reduce risk) either weren't used or there was no evidence about their use.

For the remaining 18 agile practices, the picture is unclear. With each of these practices, we found projects that use them and projects that don't. In addition, the number of projects where we found positive or negative evidence for the use of these practices is small.

Impact on Challenging Aspects (Requirements and Testing)

Most of the projects in the literature review reported effective handling of testing and requirements. In some projects, these aspects were emphasized as being especially successful when using an agile development approach. However, not all publications reported clearly which specific agile practices the authors applied. In those cases, it was difficult to assess which agile practices affected testing and requirements handling, and how much of the reported positive effects could be attributed to the use of these practices. Two of the projects in the case study employed virtually no agile practices, while the third one explicitly followed Scrum practices. In the following sections, we'll discuss the extent to which certain agile practices could be said to benefit requirements handling and testing in the projects under study (from both the literature review and case study).

Impact on requirements handling. All projects indicated that the customer is always available (practice 18). This is presumably a consequence of the fact that developers of scientific software are potential or actual users of their software. Therefore, issues related to requirements are less likely to be caused by misunderstandings between developers and customers and might instead relate to other difficulties with capturing the functional and nonfunctional requirements—for example, because of the immaturity of the scientific theory on which the software is based.

Nearly all projects from the literature review used short time-boxed iterations (practice 5), self-organizing teams (practice 8), and release planning (practice 12). The presence of these practices might have facilitated that requirements are discussed or refined quite often, making eventual changes easier to deal with. User stories (practice 13) were also used in most of the projects, which could have further promoted deliberate handling and refinement of requirements and tasks.

From the case study, the Olga project aligns well with these observations. Olga developers used most of the Scrum practices, and they regarded requirements activities to be dynamic and proper for the project.

The two noncommercial projects from the case study didn't use any of the agile practices related to requirements. However, it's interesting to observe that the people involved in these projects didn't perceive any particular problems with requirements, even though they didn't use the agile practices. This might be explained by the fact that the development is based on personal motivation, and that the individuals define and write their own requirements.

Impact on testing. Testing activities in software-development projects are directly related to five agile practices: 20 (code the unit test first), 32 (all code must have unit tests), 33 (all code must pass all unit tests before release), 34 (when a bug is found tests are created), and 35 (acceptance tests are run often and the score is published). Because we couldn't find clear evidence for the presence (or absence) of most of the test-related agile practices, their impact on testing activities in the various projects wasn't easy to identify precisely. However, in all projects where we could identify the presence of one or more test-related agile practices, problems with testing were less-frequently reported than in the other projects. For example, project 2 from our literature review (discussed elsewhere[12]) used at least four test-related practices (20, 32, 34, and 35) and reported that the agile approach to testing was a valuable asset, both in testing new functionalities and regression testing existing functionalities.

The FEniCS project in our case study uses two test-related agile practices (33 and 35). In interviews, we learned that testing isn't considered a problematic development activity in this project. Thus, the FEniCS project is better off than most of the projects we surveyed in a previous study.[5] Although this could indicate that the presence of test-related agile practices contributed to this comfortable situation, we can't provide clear support for a causal relationship between the presence of these practices and a lack of testing problems.

Similar to FEniCS, the Dalton project didn't emphasize particular problems with testing. In Dalton, we found that two test-related agile practices (33 and 35) are in use. In addition, we found that at least some developers used agile practice 32. Again, we couldn't establish a clear causal relationship between using test-related agile practices and the lack of problems with testing, but we have some indication that these factors might be related.

Of those projects where we found evidence of one test-related agile practice, project 4 from our literature review was the one that indicated most explicitly positive effects on testing activities. Among the observed effects were more focus and more deliberate handling of testing (an activity that hadn't been prioritized before using agile practices).

Limitations

Every empirical investigation has natural threats to validity that should be reported and minimized. The validity of our literature review was to some degree influenced by the quality of the articles that we selected for our in-depth analyses. In addition, we had to address the following typical threats to validity: *reliability threats*, because of single-reviewer assessment; *publication bias*, because of articles possibly being submitted and published more readily when they report positive findings; and *selection bias*, because of reviewer reliability threats and search engine mechanics. (We discuss more details about validity threats related to the literature review elsewhere.[9])

Regarding our case study, as Mike Cohn pointed out, there are a few common risks associated with interviews as a source for evidence.[7] In addition to a selection bias for opportunistic case selections, here we describe the most relevant risks, along with the precautions taken to limit risk factors.

Bias caused by poorly asked questions. To avoid unclear questions as much as possible, we had the interview guide reviewed by experienced scientists. However, misunderstandings and misconceptions might have arisen because of technical terms and unfamiliar concepts from software engineering. During the interview sessions, we made an effort to clarify and explain unknown concepts.

Response bias. This bias relates to the possibility that questions are formulated in a way that they influence the response. We tried to mitigate this risk in two ways. First, the interview guide containing the questions had been approved by other researchers. Second, the questions in the guide were relatively open and general. Therefore, the probability that the questions influenced the responses was small.

Inaccuracies caused by poor recall. We recorded all of the interviews. This eliminated the risk of losing important details in the interviewees' responses.

Reflexivity. Reflexivity refers to the possibility that an interviewee responds according to a perception of what the interviewer wants to hear. We took precautions that the interviewer expressed neutrality with regards to software engineering practices and agile practices in particular.

Agile practices, as defined and observed in our studies, are used both explicitly and implicitly in scientific software-development projects. However, in the projects that we investigated in more detail (in the case study), most of these practices were, in fact, not present or used only occasionally by some developers. A select few practices were present throughout. One of those was the practice of self-organizing teams. The other practice used was that the software must pass all unit tests for the code to be released. However, it turned out that the code coverage achieved by such tests was low.

The results indicated that agile practices aren't used across the board. Only in the exceptional case of Olga, a commercial project, was the deliberate decision made to use agile practices associated with the Scrum methodology. In the other projects, we found evidence only of those agile practices that lend themselves naturally to scientific software projects, which are characterized by frequent code alterations due to changing requirements, tight collaboration in small teams, and short planning horizons. Although the frequently used practices 18 (customer is always available), 23 (integrate often), 25 (collective ownership), and 31 (refactor whenever and wherever possible) correspond well with the conditions under which scientists develop scientific software, other practices such as pair programming (practice 21) don't lend themselves to those conditions. We might tentatively conclude that contemporary scientific-software-development projects embrace the agile spirit in their focus on flexibility and communication, but otherwise are selective in using specific agile practices according to the book. Apart from that, some of the more technology-driven practices simply might not be known to scientists who aren't professional software developers. Nevertheless, the literature review indicated that agile techniques generally had positive effects in the projects investigated. None of the studies displayed any particular negative side effects of using agile practices.

To be conclusive on the pros and cons of agile practices in scientific software, more research is needed. A substantial challenge is that the level of process awareness is low (a characteristic shared with software development in many domains). For example, one of our case study projects switched to Scrum recently and is therefore an opportune case for assessing the effects of introducing agile practices in a scientific software project. However, project members couldn't recall what development approach was used prior to Scrum, or if there were significant changes in handling requirements or testing as a result of its introduction. Nevertheless, the initial results from our combined studies are promising. A preliminary conclusion might be that the agile approach can be valuable to scientific software development, especially for smaller-sized teams and projects. CiSE

Acknowledgments

We thank the members of the FEniCS, Dalton, and Olga projects for participating in the case study. Parts of this work were funded by the Excellence Center at Linköping–Lund in Information Technology (ELLIIT) and the Simula School of Research and Innovation.

References

1. D.F. Kelly, "A Software Chasm: Software Engineering and Scientific Computing," *IEEE Software*, vol. 24, no. 6, 2007, pp. 118–120.
2. V.K. Decyk, C.D. Norton, and H.J. Gardner, "Why Fortran?" *Computing in Science and Eng.*, vol. 9, no. 4, 2007, pp. 68–71.
3. J.C. Carver et al., "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies," *Proc. 29th Int'l Conf. Software Eng.*, IEEE CS Press, 2007, pp. 550–559.
4. R. Sanders and D. Kelly, "Dealing with Risk in Scientific Software Development," *IEEE Software*, vol. 25, no. 4, 2008, pp. 21–28.
5. J.E. Hannay et al., "How Do Scientists Develop and Use Scientific Software?" *Proc. 2nd Int'l Workshop on Software Eng. for Computational Science and Eng.*, IEEE CS Press, 2009, pp. 1–8.
6. R. Sanders, *The Development and Use of Scientific Software*, master's thesis, School of Computing, Queen's University, Kingston, Ontario, Canada, 2008.
7. M. Cohn, *Succeeding with Agile: Software Development Using Scrum*, Addison-Wesley, 2009.
8. D. Wells, *The Rules of Extreme Programming*, 2009; www.extremeprogramming.org/rules.html.
9. M.T. Sletholt et al., "A Literature Review of Agile Practices and Their Effects in Scientific Software Development," *Proc 4th Int'l Workshop on Software Eng. for Computational Science and Eng.*, ACM Press, 2011, pp. 1–9.
10. T. Dybå, T. Dingsøyr, and G.K. Hanssen, "Applying Systematic Reviews to Diverse Study Types: An Experience Report," *Proc. 1st Int'l Symp. Empirical Software Eng. and Measurement,* IEEE CS Press, 2007, pp. 225–234.
11. S.M. Easterbrook and T.C. Johns, "Engineering the Software for Understanding Climate Change," *Computing in Science & Eng.*, vol. 11, no. 6, 2009, pp. 64–74.
12. J. Pitt-Francis et al., "Chaste: Using Agile Programming Techniques to Develop Computational Biology Software," *Philosophical Trans. Royal Society—Series A: Mathematical, Physical and Eng. Sciences*, vol. 366, no. 1878, 2008, pp. 3111–3136.
13. D.W. Kane et al., "Agile Methods in Biomedical Software Development: A Multi-Site Experience Report," *BMS Bioinformatics,* vol. 7, no. 273, 2006, pp. 1–12.
14. W.A. Wood and W.L. Kleb, "Exploring XP for Scientific Research," *IEEE Software*, vol. 20, no. 3, 2003, pp. 30–36; doi:10.1109/MS.2003.1196317.
15. D. Kane, "Introducing Agile Development into Bioinformatics: An Experience Report," *Proc. Agile Development Conf.,* IEEE CS Press, 2003, pp. 132–139.

Magnus Thorstein Sletholt *is a software developer at Distribution Innovation. His research interests include agile development and testing. Sletholt has an MSc in computer science from the University of Oslo. Contact him at magnus.sletholt@di.no.*

Jo Erskine Hannay *is a senior researcher at Simula Research Laboratory. His research interests include challenges in large agile projects, software quality management, and effort estimation. Hannay has a PhD in computer science from the University of Edinburgh. Contact him at jo@simula.no.*

Dietmar Pfahl *is an associate professor at Lund University. His research interests include scientific computing, empirical software engineering, requirements engineering, testing, and software process improvement.*

Pfahl has a PhD in computer science from the Technical University of Kaiserslautern, Germany. Contact him at dietmar.pfahl@cs.lth.se.

***Hans Petter Langtangen** is a professor at the University of Oslo and a department head at Simula Research Laboratory. His research interests include implementation techniques for scientific software, numerical solutions for partial differential equations, biomechanics, and stochastic mechanics. Langtangen has a PhD in computer science from the University of Oslo. Contact him at hpl@simula.no.*