HBR.ORG

Harvard Business Review

MAY 2016
REPRINT R1605B

THE BIG IDEA

Embracing Agile

How to master the process that's transforming management

by Darrell K. Rigby, Jeff Sutherland, and Hirotaka Takeuchi

THE BIG IDEA

TODD ST. JOHN

Darrell K. Rigby is a partner in the Boston office of Bain & Company. He heads the firm's global innovation and retail practices. **Jeff Sutherland** is a cocreator of the scrum form of agile innovation and the CEO of Scrum Inc., a consulting and training firm. **Hirotaka Takeuchi** is a professor in the strategy unit of Harvard Business School.

embracing agile

How to master the process that's transforming management

BY DARRELL K. RIGBY, JEFF SUTHERLAND, AND HIROTAKA TAKEUCHI

Agile innovation methods have revolutionized information technology. Over the past 25 to 30 years they have greatly increased success rates in software development, improved quality and speed to market, and boosted the motivation and productivity of IT teams.

Now agile methodologies—which involve new values, principles, practices, and benefits and are a radical alternative to command-and-control-style management—are spreading across a broad range of industries and functions and even into the C-suite. National Public Radio employs agile methods to create new programming. John Deere uses them to develop new machines, and Saab to produce new fighter jets. Intronis, a leader in cloud backup services, uses them in marketing. C.H. Robinson, a global third-party logistics provider, applies them in human resources. Mission Bell Winery uses them for everything from wine production to warehousing to running its senior leadership group. And GE relies on them to speed a much-publicized transition from 20th-century conglomerate to 21st-century "digital industrial company." By taking people out of their functional silos and putting them in self-managed and customer-focused multidisciplinary teams, the agile approach is not only accelerating profitable growth but also helping to create a new generation of skilled general managers.

The spread of agile raises intriguing possibilities. What if a company could achieve positive returns with 50% more of its new-product introductions? What if marketing programs could generate 40% more customer inquiries? What if human resources could recruit 60% more of its highest-priority targets? What if twice as many workers were emotionally engaged in their jobs? Agile has brought these levels of improvement to IT. The opportunity in other parts of the company is substantial.

But a serious impediment exists. When we ask executives what they know about agile, the response is usually an uneasy smile and a quip such as "Just enough to be dangerous." They may throw around agile-related terms ("sprints," "time boxes") and claim that their companies are becoming more and more nimble. But because they haven't gone through training, they don't really understand the approach. Consequently, they unwittingly continue to manage in ways that run counter to agile principles and practices, undermining the effectiveness of agile teams in units that report to them.

These executives launch countless initiatives with urgent deadlines rather than assign the highest priority to two or three. They spread themselves and their best people across too many projects. They schedule frequent meetings with members of agile teams, forcing them to skip working sessions or send substitutes. Many of them become overly involved in the work of individual teams. They talk more than listen. They promote marginal ideas that a team has previously considered and back-burnered. They routinely overturn team decisions and add review layers and controls to ensure that mistakes aren't repeated. With the best of intentions, they erode the benefits that agile innovation can deliver.

Innovation is what agile is all about. Although the method is less useful in routine operations and processes, these days most companies operate in highly dynamic environments. They need not just new products and services but also innovation in functional processes, particularly given the rapid spread of new software tools. Companies that create an environment in which agile flourishes find that teams can churn out innovations faster in both those categories.

From our work advising and studying such companies, we have discerned six crucial practices that leaders should adopt if they want to capitalize on agile's potential.

1 Learn How Agile Really Works

Some executives seem to associate agile with anarchy (everybody does what he or she wants to), whereas others take it to mean "doing what I say, only faster." But agile is neither. (See the sidebar "Agile Values and Principles.") It comes in several varieties, which have much in common but emphasize slightly different things. They include *scrum,* which emphasizes creative and adaptive teamwork in solving complex problems; *lean development,* which focuses on the continual elimination of waste; and *kanban,* which concentrates on reducing lead times and the amount of work in process. One of us (Jeff Sutherland) helped develop the scrum methodology and was inspired to do so in part by "The New New Product Development Game," a 1986 HBR article coauthored by another of us (Hirotaka Takeuchi). Because scrum and its derivatives are employed at least five times as often as the other techniques, we will use its methodologies to illustrate agile practices.

The fundamentals of scrum are relatively simple. To tackle an opportunity, the organization forms and empowers a small team, usually three to nine people, most of whom are assigned full-time. The team is cross-functional and includes all the skills necessary to complete its tasks. It manages itself and is strictly accountable for every aspect of the work.

Idea in Brief

THE PROBLEM
Agile methods such as *scrum*, *kanban*, and *lean development* are spreading beyond IT to other functions. Although some companies are scoring big improvements in productivity, speed to market, and customer and employee satisfaction, others are struggling.

THE ROOT CAUSE
Leaders don't really understand agile. As a result, they unwittingly continue to employ conventional management practices that undermine agile projects.

THE SOLUTION
Learn the basics of agile. Understand the conditions in which it does or doesn't work. Start small and let it spread organically. Allow "master" teams to customize it. Employ agile at the top. Destroy the barriers to agile behaviors.

some executives seem to associate agile with anarchy.

The team's "initiative owner" (also known as a product owner) is ultimately responsible for delivering value to customers (including internal customers and future users) and to the business. The person in this role usually comes from a business function and divides his or her time between working with the team and coordinating with key stakeholders: customers, senior executives, and business managers. The initiative owner may use a technique such as design thinking or crowdsourcing to build a comprehensive "portfolio backlog" of promising opportunities. Then he or she continually and ruthlessly rank-orders that list according to the latest estimates of value to internal or external customers and to the company.

The initiative owner doesn't tell the team who should do what or how long tasks will take. Rather, the team creates a simple road map and plans in detail only those activities that won't change before execution. Its members break the highest-ranked tasks into small modules, decide how much work the team will take on and how to accomplish it, develop a clear definition of "done," and then start building working versions of the product in short cycles (less than a month) known as *sprints.* A process facilitator (often a trained scrum master) guides the process. This person protects the team from distractions and helps it put its collective intelligence to work.

The process is transparent to everyone. Team members hold brief daily "stand-up" meetings to review progress and identify roadblocks. They resolve disagreements through experimentation and feedback rather than endless debates or appeals to authority. They test small working prototypes of part or all of the offering with a few customers for short periods of time. If customers get excited, a prototype may be released immediately, even if some senior executive isn't a fan, or others think it needs more bells and whistles. The team then brainstorms ways to improve future cycles and prepares to attack the next top priority.

Compared with traditional management approaches, agile offers a number of major benefits, all of which have been studied and documented. It increases team productivity and employee satisfaction. It minimizes the waste inherent in redundant meetings, repetitive planning, excessive documentation, quality defects, and low-value product features. By improving visibility and continually adapting to customers' changing priorities, agile improves customer engagement and satisfaction, brings the most valuable products and features to market faster and more predictably, and reduces risk. By engaging team members from multiple disciplines as collaborative peers, it broadens organizational experience and builds mutual trust and respect. Finally, by dramatically reducing the time squandered on micromanaging functional projects, it allows senior managers to devote themselves more fully to higher-value work

that only they can do: creating and adjusting the corporate vision; prioritizing strategic initiatives; simplifying and focusing work; assigning the right people to tasks; increasing cross-functional collaboration; and removing impediments to progress.

2 Understand Where Agile Does or Does Not Work

Agile is not a panacea. It is most effective and easiest to implement under conditions commonly found in software innovation: The problem to be solved is complex; solutions are initially unknown, and product requirements will most likely change; the work can be modularized; close collaboration with end users (and rapid feedback from them) is feasible; and creative teams will typically outperform command-and-control groups.

In our experience, these conditions exist for many product development functions, marketing projects, strategic-planning activities, supply-chain challenges, and resource allocation decisions. They are less common in routine operations such as plant maintenance, purchasing, sales calls, and accounting. (See the exhibit "The Right Conditions for Agile.") And because agile requires training, behavioral change, and often new information technologies, executives must decide whether the anticipated payoffs will justify the effort and expense of a transition.

Agile innovation also depends on having a cadre of eager participants. One of its core principles is "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done." When the majority of a company, a function, or a team chooses to adopt agile methodologies, leaders may need to press the holdouts to follow suit or even replace them. But it's better to enlist passionate volunteers than to coerce resisters.

OpenView Venture Partners, a firm that has invested in about 30 companies, took this path. Having learned about agile from some of the companies in its portfolio, Scott Maxwell, the firm's founder, began using its methodologies at the firm itself. He found that they fit some activities more easily than others. Agile worked well for strategic planning and marketing, for instance, where complex problems can often be broken into modules and cracked by creative multidisciplinary teams. That wasn't the case for selling: Any sales call can change a representative's to-do list on the spot, and it would be too complicated and time-consuming to reassemble the sales team, change the portfolio backlog, and reassign accounts every hour.

Maxwell provided the companies in OpenView's portfolio with training in agile principles and practices and let them decide whether to adopt the approach. Some of them immediately loved the idea of implementing it; others had different priorities and decided to hold off. Intronis was one fan. Its marketing unit at the time relied on an annual plan that focused primarily on trade shows. Its sales department complained that marketing was too conservative and not delivering results. So the company hired

Richard Delahaye, a web developer turned marketer, to implement agile. Under his guidance the marketing team learned, for example, how to produce a topical webinar in a few days rather than several weeks. (A swiftly prepared session on CryptoLocker malware attracted 600 registrants—still a company record.) Team members today continue to create calendars and budgets for the digital marketing unit, but with far less line-item detail and greater flexibility for serendipitous developments. The sales team is much happier.

3 Start Small and Let the Word Spread

Large companies typically launch change programs as massive efforts. But the most successful introductions of agile usually start small. They often begin in IT, where software developers are likely to be familiar with the principles. Then agile might spread to another function, with the original practitioners acting as coaches. Each success seems to create a group of passionate evangelists who can hardly wait to tell others in the organization how well agile works.

The adoption and expansion of agile at John Deere, the farm equipment company, provides an example. George Tome, a software engineer who had become a project manager within Deere's corporate IT group, began applying agile principles in 2004 on a low-key basis. Gradually, over several years, software development units in other parts of Deere began using them as well. This growing interest made it easier to introduce the methodology to the company's business development and marketing organizations.

In 2012 Tome was working as a manager in the Enterprise Advanced Marketing unit of the R&D group responsible for discovering technologies that could revolutionize Deere's offerings. Jason Brantley, the unit head, was concerned that traditional project management techniques were slowing innovation, and the two men decided to see whether agile could speed things up. Tome invited two other unit managers to agile training classes. But all the terminology and examples came from software, and to one of the managers, who had no software background, they sounded like gibberish. Tome realized that others would react the same way, so he tracked down an agile coach who knew how to work with people without a software background. In the past few years he and the coach have trained teams in all five of the R&D group's centers. Tome also began publishing weekly one-page articles about agile principles and practices, which were e-mailed to anyone interested and later posted on Deere's Yammer site. Hundreds of Deere employees joined the discussion group. "I wanted to develop a knowledge base about agile that was specific to Deere so that anyone within the organization could understand it," Tome says. "This would lay the foundation for moving agile into any part of the company."

Agile Values and Principles

In 2001, 17 rebellious software developers (including Jeff Sutherland) met in Snowbird, Utah, to share ideas for improving traditional "waterfall" development, in which detailed requirements and execution plans are created up front and then passed sequentially from function to function. This approach worked fine in stable environments, but not when software markets began to change rapidly and unpredictably. In that scenario, product specifications were outdated by the time the software was delivered to customers, and developers felt oppressed by bureaucratic procedures.

The rebels proposed four new values for developing software, described principles to guide adherence to those values, and dubbed their call to arms "The Agile Manifesto." To this day, development frameworks that follow these values and principles are known as agile techniques.

Here is an adapted version of the manifesto:

PEOPLE OVER PROCESSES AND TOOLS
Projects should be built around motivated individuals who are given the support they need and trusted to get the job done. Teams should abandon the assembly-line mentality in favor of a fun, creative environment for problem solving, and should maintain a sustainable pace. Employees should talk face-to-face and suggest ways to improve their work environment. Management should remove impediments to easier, more fruitful collaboration.

RESPOND TO CHANGE RATHER THAN FOLLOW A PLAN
Most detailed predictions and plans of conventional project management are a waste of time and money. Although teams should create a vision and plan, they should plan only those tasks that won't have changed by the time they get to them. And people should be happy to learn things that alter their direction, even late in the development process. That will put them closer to the customer and make for better results.

WORKING PROTOTYPES OVER EXCESSIVE DOCUMENTATION
Innovators who can see their results in real market conditions will learn faster, be happier, stay longer, and do more-valuable work. Teams should experiment on small parts of the product with a few customers for short periods, and if customers like them, keep them. If customers don't like them, teams should figure out fixes or move on to the next thing. Team members should resolve arguments with experiments rather than endless debates or appeals to authority.

CUSTOMER COLLABORATION OVER RIGID CONTRACTS
Time to market and cost are paramount, and specifications should evolve throughout the project, because customers can seldom predict what they will actually want. Rapid prototyping, frequent market tests, and constant collaboration keep work focused on what they will ultimately value.

The Right Conditions for Agile

CONDITIONS	FAVORABLE	UNFAVORABLE
MARKET ENVIRONMENT	Customer preferences and solution options change frequently.	Market conditions are stable and predictable.
CUSTOMER INVOLVEMENT	Close collaboration and rapid feedback are feasible. Customers know better what they want as the process progresses.	Requirements are clear at the outset and will remain stable. Customers are unavailable for constant collaboration.
INNOVATION TYPE	Problems are complex, solutions are unknown, and the scope isn't clearly defined. Product specifications may change. Creative breakthroughs and time to market are important. Cross-functional collaboration is vital.	Similar work has been done before, and innovators believe the solutions are clear. Detailed specifications and work plans can be forecast with confidence and should be adhered to. Problems can be solved sequentially in functional silos.
MODULARITY OF WORK	Incremental developments have value, and customers can use them. Work can be broken into parts and conducted in rapid, iterative cycles. Late changes are manageable.	Customers cannot start testing parts of the product until everything is complete. Late changes are expensive or impossible.
IMPACT OF INTERIM MISTAKES	They provide valuable learning.	They may be catastrophic.

SOURCE BAIN & COMPANY

Using agile techniques, Enterprise Advanced Marketing has significantly compressed innovation project cycle times—in some cases by more than 75%. One example is the development in about eight months of a working prototype of a new "machine form" that Deere has not yet disclosed. "If everything went perfectly in a traditional process," Brantley says, "it would be a year and a half at best, and it could be as much as two and a half or three years." Agile generated other improvements as well. Team engagement and happiness in the unit quickly shot from the bottom third of companywide scores to the top third. Quality improved. Velocity (as measured by the amount of work accomplished in each sprint) increased, on average, by more than 200%; some teams achieved an increase of more than 400%, and one team soared 800%.

Success like this attracts attention. Today, according to Tome, in almost every area at John Deere someone is either starting to use agile or thinking about how it could be used.

4 Allow "Master" Teams to Customize Their Practices

Japanese martial arts students, especially those studying aikido, often learn a process called *shu-ha-ri*. In the *shu* state they study proven disciplines. Once they've mastered those, they enter the *ha* state, where they branch out and begin to modify traditional forms. Eventually they advance to *ri*, where they have so thoroughly absorbed the laws and principles that they are free to improvise as they choose.

Mastering agile innovation is similar. Before beginning to modify or customize agile, a person or team will benefit from practicing the widely used methodologies that have delivered success in thousands of companies. For instance, it's wise to avoid beginning with part-time assignment to teams or with rotating membership. Empirical data shows that stable teams are 60% more productive and 60% more responsive to customer input than teams that rotate members.

Over time, experienced practitioners should be permitted to customize agile practices. For example, one principle holds that teams should keep their progress and impediments constantly visible. Originally, the most popular way of doing this was by manually advancing colored sticky notes from the "to-do" column to "doing" to "done" on large whiteboards (known as kanban boards). Many teams are still devoted to this practice and enjoy having nonmembers visit their team rooms to view and discuss progress. But others are turning to software programs and computer screens to minimize input time and allow the information to be shared simultaneously in multiple locations.

A key principle guides this type of improvisation: If a team wants to modify particular practices, it should experiment and track the results to make sure that the changes are improving rather than reducing customer satisfaction, work velocity, and team morale.

Spotify, the music-streaming company, exemplifies an experienced adapter. Founded in 2006, the company was agile from birth, and its entire business model, from product development to marketing and general management, is geared to deliver better customer experiences through agile innovation. But senior leaders no longer dictate specific practices; on the contrary, they encourage experimentation and flexibility as long as changes are consistent with agile principles and can be shown to improve outcomes. As a result, practices vary across the company's 70 "squads" (Spotify's name for agile innovation teams) and its "chapters" (the company term for functional competencies such as user interface development and quality testing). Although nearly every squad consists of a small cross-functional team and uses some form of visual progress tracking, ranked priorities, adaptive planning, and brainstorming sessions on how to improve the work process, many teams omit the "burndown" charts (which show work performed and work remaining) that are a common feature of agile teams. Nor do they always measure velocity, keep progress reports, or employ the same techniques for estimating the time required for a given task. These squads have tested their modifications and found that they improve results.

5 Practice Agile at the Top

Some C-suite activities are not suited to agile methodologies. (Routine and predictable tasks—such as performance assessments, press interviews, and visits to plants, customers, and suppliers—fall into this category.) But many, and arguably the most important, are. They include strategy development and resource allocation, cultivating breakthrough innovations, and improving organizational collaboration. Senior executives who come together as an agile team and learn to apply the discipline to these activities achieve far-reaching benefits. Their own productivity and morale improve. They speak the language of the teams they are empowering. They experience common challenges and learn how to overcome them. They recognize and stop behaviors that impede agile teams. They learn to simplify and focus work. Results improve, increasing confidence and engagement throughout the organization.

A number of companies have reallocated 25% or more of selected leaders' time from functional silos to agile leadership teams. These teams rank-order enterprisewide portfolio backlogs, establish and coordinate agile teams elsewhere in the organization to address the highest priorities, and systematically eliminate barriers to their success. Here are three examples of C-suites that took up agile:

1. Catching up with the troops. Systematic, a 525-employee software company, began applying agile methodologies in 2005. As they spread to all its software development teams, Michael Holm, the company's CEO and cofounder, began to worry that his leadership team was hindering progress. "I had this feeling that I was saying, 'Follow me—I'm just behind you,'" he told us. "The development teams were using scrum and were doing things differently, while the management team was stuck doing things the same old-fashioned way"—moving too slowly and relying on too many written reports that always seemed out-of-date. So in 2010 Holm decided to run his nine-member executive group as an agile team.

The team reprioritized management activities, eliminating more than half of recurring reports and converting others to real-time systems while increasing attention to business-critical items such as sales proposals and customer satisfaction. The group started by meeting every Monday for an hour or two but found the pace of decision making too slow. So it began having daily 20-minute stand-ups at 8:40 AM to discuss what members had done the day before, what they would do that day, and where they needed help. More recently the senior team began to use physical boards to track its own actions

FURTHER RESOURCES

From HBR.org:

"Lean Knowledge Work"
Bradley R. Staats and David M. Upton

"Decoding the DNA of the Toyota Production System"
Steven Spear and H. Kent Bowen

"Beyond Toyota: How to Root Out Waste and Pursue Perfection"
James P. Womack and Daniel T. Jones

Other:

Agile Alliance
For guides to agile practices, links to "The Agile Manifesto," and training videos

Scrum Alliance
For a "Scrum Guide," conference presentations and videos, and the "State of Scrum" research report

ScrumLab Open
For training presentations, videos, webinars, and published papers

Annual State of Agile Survey
For key statistics such as usage rates, customer benefits, barriers to adoption and success, and specific practices used

scrum "takes the mystery out of what executives do every day."

and the improvements coming from the business units. Other functions, including HR, legal, finance, and sales, now operate in much the same way.

2. Speeding a corporate transition. In 2015 General Electric rebranded itself as a "digital industrial company," with a focus on digitally enabled products. Part of the transformation involved creating GE Digital, an organizational unit that includes all 20,000-plus of the company's software-related employees. Brad Surak, who began his career as a software engineer and is now GE Digital's COO, was intimately familiar with agile. He piloted scrum with the leadership team responsible for developing industrial internet applications and then, more recently, began applying it to the new unit's management processes, such as operating reviews. Surak is the initiative owner, and an engineering executive is the scrum master. Together they have prioritized backlog items for the executive team to address, including simplifying the administrative process that teams follow to acquire hardware and solving knotty pricing issues for products requiring input from multiple GE businesses.

The scrum team members run two-week sprints and conduct stand-up meetings three times a week. They chart their progress on a board in an open conference room where any employee can see it. Surak says, "It takes the mystery out of what executives do every day. Our people want to know if we are in tune with what they care about as employees." The team collects employee happiness surveys, conducts root cause analysis on the impediments to working more effectively, and reports back to people throughout the organization, saying (in effect), "We heard you. Here is how we will improve things." Surak believes that this shows the organization that "executives work in the same ways as engineers," increasing employee motivation and commitment to agile practices.

3. Aligning departments and functions on a common vision. Erik Martella, the vice president and general manager of Mission Bell Winery, a production facility of Constellation Brands, introduced agile and helped it spread throughout the organization. Leaders of each department served as initiative owners on the various agile teams within their departments. Those individual teams achieved impressive results, but Martella worried that their time was being spread too thin and that department and enterprise priorities weren't always aligned. He decided to pull department leaders into an executive agile team focused on the enterprise initiatives that held the greatest value and the greatest opportunity for cross-functional collaboration, such as increasing process flows through the warehouse.

The team is responsible for building and continually refining the backlog of enterprise priorities, ensuring that agile teams are working on the right problems and have sufficient resources. Team members also protect the organization from pet projects that don't deserve high priority. For instance, shortly after Martella started implementing agile, he received an e-mail from a superior in Constellation's corporate office suggesting that the winery explore a personal passion of the sender. Previously, Martella might have responded, "OK, we'll jump right on it." Instead, he replied that the winery was following agile principles: The idea would be added to the list of potential opportunities and prioritized. As it happened, the executive liked the approach—and when he was informed that his suggestion had been assigned a low priority, he readily accepted the decision.

Working on agile teams can also help prepare functional managers—who rarely break out of their silos in today's overspecialized organizations—for general management roles. It exposes them to people in other disciplines, teaches collaborative practices, and underscores the importance of working closely with customers—all essential for future leaders.

6 Destroy the Barriers to Agile Behaviors

Research by Scrum Alliance, an independent nonprofit with 400,000-plus members, has found that more than 70% of agile practitioners report tension

between their teams and the rest of the organization. Little wonder: They are following different road maps and moving at different speeds.

Here's a telling example: A large financial services company we examined launched a pilot to build its next mobile app using agile methodologies. Of course, the first step was to assemble a team. That required a budget request to authorize and fund the project. The request went into the batch of submissions vying for approval in the next annual planning process. After months of reviews, the company finally approved funding. The pilot produced an effective app that customers praised, and the team was proud of its work. But before the app was released, it had to pass vulnerability testing in a traditional "waterfall" process (a protracted sequence in which the computer code is tested for documentation, functionality, efficiency, and standardization), and the queue for the process was long. Then the app had to be integrated into core IT systems—which involved another waterfall process with a six-to-nine-month logjam. In the end, the total time to release improved very little.

Here are some techniques for destroying such barriers to agile:

Get everyone on the same page. Individual teams focusing on small parts of large, complex problems need to see, and work from, the same list of enterprise priorities—even if not all the teams responsible for those priorities are using agile processes. If a new mobile app is the top priority for software development, it must also be the top priority for budgeting, vulnerability testing, and software integration. Otherwise, agile innovations will struggle in implementation. This is a key responsibility of an executive team that itself practices agile.

Don't change structures right away; change roles instead. Many executives assume that creating more cross-functional teams will necessitate major changes in organizational structure. That is rarely true. Highly empowered cross-functional teams do, by definition, need some form of matrix management, but that requires primarily that different disciplines learn how to work together simultaneously rather than separately and sequentially.

Name only one boss for each decision. People can have multiple bosses, but decisions cannot. In an agile operating model it must be crystal clear who is responsible for commissioning a cross-functional team, selecting and replacing team members, appointing the team leader, and approving the team's decisions. An agile leadership team often authorizes a senior executive to identify the critical issues, design processes for addressing them, and appoint a single owner for each innovation initiative. Other senior leaders must avoid second-guessing or overturning the owner's decisions. It's fine to provide guidance and assistance, but if you don't like the results, change the initiative owner—don't incapacitate him or her.

Focus on teams, not individuals. Studies by the MIT Center for Collective Intelligence and others show that although the intelligence of individuals affects team performance, the team's collective intelligence is even more important. It's also far easier to change. Agile teams use process facilitators to continually improve their collective intelligence—for example, by clarifying roles, teaching conflict resolution techniques, and ensuring that team members contribute equally. Shifting metrics from output and utilization rates (how busy people are) to business outcomes and team happiness (how valuable and engaged people are) also helps, as do recognition and reward systems that weight team results higher than individual efforts.

Lead with questions, not orders. General George S. Patton Jr. famously advised leaders never to tell people *how* to do things: "Tell them *what* to do, and they will surprise you with their ingenuity." Rather than give orders, leaders in agile organizations learn to guide with questions, such as "What do you recommend?" and "How could we test that?" This management style helps functional experts grow into general managers, and it helps enterprise strategists and organizations evolve from silos battling for power and resources into collaborative cross-functional teams.

AGILE INNOVATION has revolutionized the software industry, which has arguably undergone more rapid and profound change than any other area of business over the past 30 years. Now it is poised to transform nearly every other function in every industry. At this point, the greatest impediment is not the need for better methodologies, empirical evidence of significant benefits, or proof that agile can work outside IT. It is the behavior of executives. Those who learn to lead agile's extension into a broader range of business activities will accelerate profitable growth.

HBR Reprint R1605B

Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmermann**
University of Applied Sciences
of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

Making Sense of Agile Methods

Bertrand Meyer

From the Editors

Bertrand Meyer runs agile methods and practices through his personal friend-or-foe test. Find out more about his experiences and opinions about the hype, ugly, good, and even brilliant aspects of agile! —*Cesare Pautasso and Olaf Zimmermann*

SOME 10 YEARS ago, I realized I had been missing something big in software engineering. I had heard about Extreme Programming (XP) early, thanks to a talk by Pete McBreen at a summer school in 1999 and another by Kent Beck himself at TOOLS USA in 2000. But I hadn't paid much attention to Scrum. When I took a look, I noticed two striking discrepancies in the state of agile methods.

The first discrepancy was between university software engineering courses, which back then (things have changed) often didn't cover agility, and the industry buzz, which was only about agility.

As I started going through the agile literature, the second discrepancy emerged: an amazing combination of the best and the worst ideas, plus much in between. In many cases, when faced with a new methodological approach, a person can quickly deploy what in avionics is called Identification Friend or Foe (IFF): will it help or hurt? With agile, IFF fails. It didn't help that the tone of most published discussions on agile methods (with a few exceptions, notably *Balancing Agility and Discipline: A Guide for the Perplexed*[1]) was adulatory. Sharing your passion for a novel approach is commendable, but not a reason to throw away your analytical skills. A precedent comes to mind: people (including me) who championed object-oriented programming a decade or so earlier might at times have let their enthusiasm show, but we didn't fail to discuss cons along with pros.

Beyond the Boasts and Good Intentions

The natural reaction was to apply a rule that often helps: when curious, teach a class; when bewildered, write a book. Thus was *Agile! The Good, the Hype and the Ugly* born.[2] (Also see the edX online course Agile Software Development; www.edx.org/course/agile-software-development-ethx-asd-1x-0.) My first goal was to provide a concise tutorial on agile methods, addressing a frequently heard request for a comprehensive, no-frills, news-not-editorial description of agile concepts. The second goal was analytical: offering an assessment of agile boasts.

These boasts are impressive. The title of a recent book by one of the creators of Scrum promises "Twice the Work in Half the Time."[3] Wow! I'll take a productivity improvement of four any time. Another book by both of the method's creators informs us, "You have been ill served by the software industry for 40 years—not purposely, but inextricably. We want to restore the partnership."[4] No less! (Was any software used in producing that statement?)

The agile literature has a certain adolescent quality ("No one else understands!"), but ideas aren't born in a vacuum. I quickly realized that the agile movement was best understood as evolution rather than revolution. Although you wouldn't guess it from some of the agile proclamations, the

0740-7459/18/$33.00 © 2018 IEEE

software engineering community didn't wait until the Agile Manifesto (agilemanifesto.org) to recognize the importance of change; every textbook emphasizes the role of the "soft" in "software." For example, my 1995 book *Object Success*,[5] a presentation of object technology for managers, advocated designing for change and stressed the preeminence of code over diagrams and documents. For these issues as well as for some of the other agile ideas—the importance of tests, the necessity of an iterative process—the agile contribution was not to invent concepts but to convince industry to adopt them.

In looking at matters such as support for requirements change, I encountered yet another discrepancy: between grand intentions and timorous advice. It's great for the Agile Manifesto to "welcome change," but producing changeable software is a technical issue, not a moral one. It is hard to reconcile such lofty goals with the deprecation of software techniques that actually support change: information hiding, deemed ineffective in *Leading Lean Software Development*,[6] and design for extension and reuse, pooh-poohed in *Extreme Programming Installed*.[7] (Full citations appear in *Agile! The Good, the Hype and the Ugly*, sections 4.4.4 and 4.4.5.)

There are more eyebrow-raising agile pronouncements, but we shouldn't let them obscure the agile school's major contributions. After all, marketing buzz goes only so far; developers and managers, driven mostly by pragmatic considerations, haven't embraced agile ideas—more accurately, *some* agile ideas—without good reasons. (For my part, I wouldn't have spent the better part of four years reading more or less the entire agile literature, practicing agile methods, and qualifying as a proud Certified Scrum Master, if I thought the approach was worthless.) But you need a constantly alert IFF to sort out the best and worst agile ideas.

Sorting Out the Agile Ideas

The final chapter of *Agile! The Good, the Hype and the Ugly* summarizes the book's analysis by listing items in each of the three categories, complementing the Good with the subcategory of the truly Brilliant. Here are a few significant examples in each category.

The Hype

Let's start with the Hype, which can also be called the Indifferent: ideas that have been oversold even though their impact is modest. An example is pair programming: the practice of developing software in groups of two people, one at the keyboard and one standing by, speaking out their thought processes to each other. XP prescribes pair programming as the standard practice. One use of pair programming is as a source of research papers: you can measure and compare the outcome (development time and number of bugs) of two groups working on the same topic, one with pair programming and the other using traditional techniques. Such studies are relatively easy to conduct using student projects in a university software engineering course.

These studies, of which there are many, show that pair programming has no significant advantage or disadvantage compared to other techniques such as code inspection (for example, see "Two Controlled Experiments Concerning the Comparison of Pair Programming to Peer Review"[8]). The truth is that pair programming is an interesting practice, to be used on occasion, typically for a tricky part of the development requiring competence from two different areas, but there's no reason to impose it as the sole mode of development. It's also easy to misuse by confusing it with mentoring, an entirely different idea.

Other examples of the Indifferent include the role of open spaces (although office layout deserves attention, some of the best software was developed in garages), self-organizing teams (different projects and different contexts will require different styles of project management), and the charming invention of planning poker.

The Ugly

More worrying are the agile recommendations that fall into the Ugly category. Perhaps the most damaging is the widespread rejection of "Big Up-Front Everything": up-front requirements, up-front design. For a typical example, see "Agile Design: Intentional yet Emergent."[9] The rationale is understandable: some projects spend too much time in general preliminary discussions, a phenomenon sometimes called *analysis paralysis*, and we should strive instead to start some actual coding early.

This healthy reaction doesn't justify swinging the pendulum to the other extreme. No serious engineering process can skip an initial step of careful planning. It is good to put limits on it but irresponsible to remove it. The project failures I tend to see nowadays in my role as a consultant (or project rescuer) are often due to an application of this agile rule: "We don't need no stinkin' requirements phase; we're agile. Let's just produce user stories and

implement them as we go." A sure way to disaster.

User stories themselves also fall into the Ugly. They are a good way to validate requirements by ensuring that the requirements handle commonsense user interaction scenarios, but an insufficient basis for specifying requirements. A user story describes one case; piling up case over case doesn't give you a specification. What it gives you is a system that handles the user stories—the exact planned scenarios—and possibly nothing else.

In *Agile! The Good, the Hype and the Ugly*, I cite at length the work of Pamela Zave from AT&T, who has for decades studied feature-based design of telecommunication systems. (A collection of Zave's research papers on requirements and other topics is at www.research.att.com/people/Zave_Pamela/custom/indexCustom.html.) The problem with individual features is that they interact with each other. Such interactions, often subtle, doom any method that tells you to build a system by just implementing feature after feature. Everything will look fine until you suddenly discover that the next user story conflicts with previously implemented ones, and you have to go back and rethink everything. Although no universally accepted answer exists to the question of how best to write requirements, object-oriented analysis, which looks past individual scenarios to uncover the underlying data abstractions, is a good start (see chapter 27 of *Object-Oriented Software Construction*[10]).

Toward the Good

Rejections of up-front tasks and reliance on user stories for requirements are examples of the harmful advice you'll find proffered—sometimes in the same breath as completely reasonable ideas—in agile texts. As I come to the Good and the Brilliant, it's useful to include an example of a technique that, depending on how you use it, qualifies as part of either the worst or the best.

I mentioned that, at the beginning of the last decade, agile methods had found their way into industry but not academia. Students learn their trade not just from courses but also from summer internships in industry. In one of my first classes as a newly minted professor in 2002, I gave a lecture on software design. A third-year student came to me afterward and asked why I was still teaching such nonsense. Everyone knows, he said, that nowadays no one does design; we just produce "the simplest thing that can possibly work" and then refactor. I was stunned (not having realized how far XP ideas had percolated).

He was wrong: no magic process can, through refactoring, turn bad design into good. Refactoring junk yields junk. Understood this way, refactoring would fall into the Ugly. Yet refactoring also belongs to the Good by teaching us that we should never be content with a first software version simply because it works. Instead, we should apply the systematic habit of questioning our designs and looking into what could be done better. In other words, whatever the student thought, the right approach is to work hard, up front, on producing a good design—and later on to make it even better through refactoring.

Some other positive contributions of agile methods are for everyone to see, because agile ideas have already exerted a major influence on software development. Most visibly, no project in its right mind would go into the scheme (which looks crazy, but not so long ago was the norm) of splitting the task into large chunks leading to separate subprojects, and trying to reconcile them months down the road. The splitting is easy; it's the reconciliation that can be a nightmare. Divergent assumptions, often implicit, preside over the design of the various components and propagate into the depths of each of them, rendering them incompatible. The only remedy is to catch such divergence right away.

Although iterative development isn't an agile invention, agile made it the default and particularly promoted the use of short iterations. ("Short" has evolved to mean increasingly shorter. Just a few years ago, promoting six-week sprints sounded audacious. Today we're hearing about one-week or sometimes one-day sprints. This isn't even taking into account the spread of DevOps processes with their rapid fine-grain interleaving of development, testing, and deployment.) Continuous integration and continuous testing are natural complements to this core idea. These and a number of other agile precepts are truly Good.

Finally, the Brilliant

I mentioned that some of the Good deserves an upgrade to Brilliant. Here are just two examples. Both are ideas that figure in the agile literature, although with less emphasis than others that, to me at least, are less significant.

The first deserves substantial discussion, but I will just state it: no branching. Repeat after me: *branching is evil*. The second appears in Scrum texts, but without a name; I call it the Closed-Window Rule. It states that the list of tasks for an iteration (a sprint, which, as noted, is

ABOUT THE AUTHOR

BERTRAND MEYER is a professor of software engineering at Politecnico di Milano and Innopolis University. Contact him at bertrand.meyer@inf.ethz.ch.

short) can't grow. No matter who requests an addition—queen, hero, or laborer—everyone will be told "no." The proposed functionality will have to wait until the next sprint.

There's an escape mechanism (an *exception* in programming-language terms): if the addition is truly essential, you can cancel the sprint and start afresh. This possibility will address truly urgent cases but is so extreme as to be used only rarely. The beauty of the Closed-Window Rule is that it brings stability to software projects, preventing the constant influx of supposedly good ideas that disrupt the development. Some of those ideas might not look so good when you wake up sober the next morning; the Closed-Window Rule fosters a process of attrition and selection in which only the fittest ideas survive.

These survivors won't have that long to wait. The rule would be unworkable with the long steps of old-style project development, but with a typical one-month sprint, the average delay will be two weeks, during which the ideas will get the opportunity to mature. Few suggestions of added functionality are so critical that they cannot wait two weeks.

Benefitting from agile methods is a matter of spotting and rejecting the Ugly, ignoring the Hype, and taking advantage of the Good and Brilliant. Industry, which usually has its feet solidly on the ground, understood this situation early. Despite some agile proponents' absolutist claims (adopt every single one of my precepts, or else …), every project I've seen embraces a subset of the chosen method's ideas, rejecting those that don't fit its culture or needs.

Agile methods are no panacea. Like most human endeavors, they have their dark side, but that hasn't prevented them from improving the practice of software development in concrete ways. In any case, they don't invalidate the knowledge of software engineering accumulated over the preceding decades. Some of their beneficial insights contradict specific elements of this traditional wisdom, but for the most part they complement and expand it.

Agile is not a negation of what came before. It is one more brick in the patient construction of the modern software engineering edifice.

References

1. B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2003.
2. B. Meyer, *Agile! The Good, the Hype and the Ugly*, Springer, 2014.
3. J. Sutherland, *Scrum: The Art of Doing Twice the Work in Half the Time*, Random House, 2015.
4. K. Schwaber and J. Sutherland, *Software in 30 Days: How Agile Managers Beat the Odds, Delight Their Customers, and Leave Competitors in the Dust*, John Wiley & Sons, 2012.
5. B. Meyer, *Object Success: A Manager's Guide to Object Orientation, Its Impact on the Corporation and Its Use for Reengineering the Software Process*, Prentice Hall, 1995.
6. M. Poppendieck and T. Poppendieck, *Leading Lean Software Development*, Addison-Wesley, 2010.
7. R. Jeffries, A. Anderson, and C. Hendrickson, *Extreme Programming Installed*, Addison-Wesley, 2001.
8. M. Müller, "Two Controlled Experiments Concerning the Comparison of Pair Programming to Peer Review," *J. Systems and Software*, vol. 78, no. 2, 2005, pp. 166–179.
9. M. Cohn, "Agile Design: Intentional yet Emergent," blog, 4 Dec. 2009; www.mountaingoatsoftware.com/blog/agile-design-intentional-yet-emergent.
10. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.

Requirements: The Key to Sustainability

Christoph Becker, University of Toronto

Stefanie Betz, Karlsruhe Institute of Technology

Ruzanna Chitchyan, University of Leicester

Leticia Duboc, State University of Rio de Janeiro

Steve M. Easterbrook, University of Toronto

Birgit Penzenstadler, California State University, Long Beach

Norbert Seyff, University of Applied Sciences and Arts Northwestern Switzerland

Colin C. Venters, University of Huddersfield

// Software's critical role in society demands a paradigm shift in the software engineering mind-set. This shift is driven by requirements engineering. //

SOFTWARE SYSTEMS are a major driver of social and economic activity. Software engineering (SE) tends to focus on the technical elements—artificial systems with clear boundaries and identifiable parts and connections, modules and dependencies. But software systems are embedded in other technical systems and in socioeconomic and natural systems. This embedding is obvious when the interaction is explicit, such as environmental monitoring or flight control software.

However, software-intensive systems have become so essential to societies that the resulting sociotechnical systems' boundaries and interactions are often hard to identify. For example, communication, travel booking, and procurement systems influence the socioeconomic and natural environment through far-reaching effects on how we form relationships, how we travel, and what we buy. The engineering process rarely makes these effects explicit. Their lack of visibility makes assessing a software system's long-term and cumulative impacts difficult.

Designing for sustainability is a major challenge that can profoundly change SE's role in society. But what does it mean to establish sustainability as a major concern in SE? As software engineers, we're responsible for our software's long-term consequences, irrespective of the primary purpose of the system we're designing. Requirements are the key leverage point for practitioners who want to develop sustainable software-intensive systems. Here, we present two examples that illustrate the changes needed in SE and show how considering sustainability explicitly will affect requirements activities.

Sustainability in Software Engineering

Sustainability is the capacity to endure, so a system's sustainability describes how well it will continue to exist and function, even as circumstances change. Sustainability has often been equated with environmental issues, but it's increasingly clear that it requires simultaneous consideration of environmental resources, societal and individual well-being, economic prosperity, and the long-term viability of technical infrastructure.

A technical system's sustainability differs considerably from that of a socioeconomic system. Software

0740-7459/16/$33.00 © 2016 IEEE

engineers tend to focus on sustainability's technical dimension, in which it's simply a measure of the software system's longevity.[1] However, to understand broader sustainability issues, we must ask which system to sustain, for whom, over which time frame, and at what cost.[2] This involves five interrelated dimensions:[3]

- *The individual dimension* covers individual freedom and agency (the ability to act in an environment), human dignity, and fulfillment. It includes individuals' ability to thrive, exercise their rights, and develop freely.
- *The social dimension* covers relationships between individuals and groups. For example, it covers the structures of mutual trust and communication in a social system and the balance between conflicting interests.
- *The economic dimension* covers financial aspects and business value. It includes capital growth and liquidity, investment questions, and financial operations.
- *The technical dimension* covers the ability to maintain and evolve artificial systems (such as software) over time. It refers to maintenance and evolution, resilience, and the ease of system transitions.
- *The environmental dimension* covers the use and stewardship of natural resources. It includes questions ranging from immediate waste production and energy consumption to the balance of local ecosystems and climate change concerns.

Complex software-intensive systems can affect sustainability in any of these dimensions. Changes in one system, in one dimension, often have impacts in other dimensions and other systems. For example, consider a hard-to-maintain software system (technical sustainability). Excessive maintenance costs affect the owning company's financial liquidity (social and economic sustainability). This might limit its growth and even threaten its survival (economic sustainability).

Similar tradeoffs occur across other dimensions. For example, carbon offsets incentivize environmentally sustainable behavior through tradeoffs with the economic dimension. The triple-bottom-line perspective requires a business to account for social and environmental as well as financial outcomes.[4] The corresponding business practices have led to a surge in the number of social enterprises, which achieve survival rates above average for new businesses.[5]

> We need to consider systems' immediate features and effects and their longer-running aggregate and cumulative impact.

Increasingly, software engineers need to understand the effects by which software system design decisions can enable or undermine the sustainability of socioeconomic and natural systems over time (see the sidebar, "Classifying the Systemic Effects of Software"). Because sustainability is inherently multidisciplinary, any effort to define it involves concepts, principles, and methods from a range of disciplines and makes an integrated view crucial for effective system design. The notion of sustainability design brings these concerns together using systems-thinking principles (see the sidebar, "Sustainability Principles for Software Engineering").

A Tale of Two Projects

A software system's impact on its environment is often determined by how the software engineers understand its requirements. This impact's foundation is set in the decisions on which system to build (if any at all), the choices of whom to ask and whom to involve, and the specification of what constitutes success.

The following examples describe two projects to develop a procurement system that supports purchasing products and contracting services in a private company in the energy sector. Products, services, and suppliers must pass the company's approval process and be registered in the system before a purchase. This approval considers the supplier's reliability, capacity to deliver, and, in some cases, adherence to international standards of environmental management, health, and safety.

The examples are inspired by a real-world project.[6] The first example reflects typical software projects, which don't use sustainability design. The second shows what could happen if a project applied sustainability design. Terms in italics indicate aspects that are common to both projects, for easy comparison.

Development without Sustainability Design

The *project's purpose* is to maximize the organization's procurement efficiency, increase the financial return, and ensure suppliers' compliance

CLASSIFYING THE SYSTEMIC EFFECTS OF SOFTWARE

Many critical effects in sociotechnical systems play out over time. So, we need to consider not just our systems' immediate features and effects but their longer-running aggregate and cumulative impact. We distinguish three orders of effects.[1]

Immediate effects are the direct effects of the production, use, and disposal of software systems. This includes the immediate benefit of system features and the full life-cycle impacts, such as a life-cycle assessment (LCA) would include. An LCA evaluates the environmental impact of a product's life from the extraction of raw materials to its disposal or recycling.

Enabling effects arise from a system's application over time. This includes not only opportunities to consume more (or fewer) resources but also other changes induced by system use.

Structural effects represent "persistent changes observable at the macro level. Structures emerge from the entirety of actions at the micro level and, in turn, influence these actions."[1] Ongoing use of a new software system can lead to shifts in capital accumulation; drive changes in social norms, policies, and laws; and alter our relationship with the natural world.

Consider Airbnb.com. Its immediate effects include resources consumed and jobs created during its development, energy consumed during its deployment, and the room renting and booking services it offers. Its enabling effects include changes in how its users make travel arrangements as alternatives to hotel bookings and in how property owners rent out space.

These enabling effects (the "sharing economy") have been both praised and criticized for their far-reaching structural impacts. For example, Airbnb represents a substantial share of the buy-to-let market in major cities. The continuing price surges in these cities' hot spots have been linked to the density of buy-to-let properties. Many of these exist only because of the arbitrage that services such as Airbnb.com provide. The system enables transactions that provide a higher return on investment than long-term rentals. This has caused major concerns in several large cities.

Reference

1. L.M. Hilty and B. Aebischer, "ICT for Sustainability: An Emerging Research Field," *ICT Innovations for Sustainability*, Springer, 2015, pp. 3–36.

with certain rules. The criteria for selecting products and services focus on price, delivery time, and payment conditions.

Using a *stakeholder* influence matrix, the project leader focuses on those stakeholders who can "stop the show." A few influential stakeholders determine the project *scope* early on so that the project can focus on a minimal design scope to maximize project speed. The project team moves swiftly to determine the *boundaries* of the software to be; the only scoping questions revolve around the software's interfaces with neighboring systems.

The project's *success criteria* are to develop and deliver the system within the given budget and time. The question of feasibility centers on the software project investment's expected amortization period. *Risk analysis* focuses on economic risks that could inhibit project completion.

Requirements elicitation requests stakeholders' input through structured forms to identify what they want the system to do. Additionally, the team analyzes previous systems and consults business process documents. Requirements prioritization is determined by functional requirements and economic constraints and is completed quickly because the core stakeholder group has a strong consensus.

The *requirements specification* is documented following the software requirements specifications template from IEEE Standard 830. System *measurement and monitoring* employ performance and availability indicators. The system is completed on time and within budget and shows a reasonably low rate of faults, so the project is considered a success at completion.

Development with Sustainability Design

Consider conducting the same project while treating sustainability as a first-class concern in line with sustainability design principles (see the sidebar, "Sustainability Principles for Software Engineering").

While discussing the *project's purpose*, the initial project team discusses the company's values and responsibilities and identifies opportunities to support the

company's sustainable development. For example, the system can support sustainability in the supply chain by making transparent the carbon footprint of purchases and facilitating the selection of providers who apply sustainable practices. This doesn't change the overall project objectives, but it influences subsequent steps.

The *scope* of analysis starts with an inclusive, integrated view of the procurement processes, material flows into the company, and the local community's social and political environment. When defining possible system *boundaries*, the team experiments with multiple perspectives and works jointly with the procurement department and others.

The team expands the set of *stakeholders* and draws on knowledge beyond the team by using a stakeholder impact analysis. This analysis considers enabling and structural effects to identify those most affected by the project, including those external to the company. Stakeholders include local supplier representatives, service delivery organizations, process analysts, the chief technology officer, and the strategic-planning and foresight group.

To keep the number of stakeholders manageable, a sustainability expert acts as a surrogate stakeholder for others in the community and the further environment that the system might affect. A team member is assigned to each of the five sustainability dimensions so that responsibility for identifying possible effects is clear and effective communication with additional stakeholders can take place. These team members consult relevant experts in areas such as supply chain sustainability, carbon accounting, and socially responsible procurement. They also consult anthropologists analyzing and interpreting current technological developments and their impact on society.

The team agrees that the project's *success criteria* are not restricted to whether it's delivered on time and within budget, but will be *measured and monitored* over the 36 months after project completion. In this period, the team will measure a set of indicators covering the five sustainability dimensions. It will try to measure

SUSTAINABILITY PRINCIPLES FOR SOFTWARE ENGINEERING

The following principles are based on "Sustainability Design and Software: The Karlskrona Manifesto."[1]

- Sustainability is systemic; a system can never be treated in isolation from its environment.
- Sustainability is multidimensional; the five key dimensions are economic, social, environmental, technical, and individual.
- Sustainability is interdisciplinary; sustainability design in software engineering requires an appreciation of concepts from other disciplines and must work across disciplines.
- Sustainability transcends the software's purpose; any software can impact the sustainability of its socioeconomic, sociotechnical, cultural, and natural environments.
- Sustainability is multilevel; it requires us to consider at least two spheres during system design: the system under design and its sustainability, and the wider system of which it will be part.
- Sustainability is multi-opportunity; it requires us to seek interventions that have the most leverage on a system[2] and to consider the opportunity costs.
- Sustainability involves multiple timescales; it requires long-term thinking to address the timescales on which sustainability effects occur.
- Sustainability isn't zero-sum; changing a system's design to consider the long-term effects doesn't automatically imply making sacrifices now.
- System visibility is a necessary precondition and enabler for sustainability design. This is because only a transparent status of the system and its context, made visible at different abstraction levels and perspectives, can enable system designers to make informed responsible choices.

For more on this, see www.sustainabilitydesign.org.

References

1. C. Becker et al., "Sustainability Design and Software: The Karlskrona Manifesto," *Proc. 37th IEEE Int'l Conf. Software Eng.* (ICSE 15), 2015, pp. 467–476.
2. D.H. Meadows, *Leverage Points: Places to Intervene in a System*, Sustainability Inst., 1999.

- technical debt,
- social reputation and improved relations with the local community,
- individual aspects such as privacy compliance and the satisfaction of those involved in the procurement process,
- environmental aspects such as the total carbon footprint of the products and services acquired, and
- amortization of the project costs and improved cost–benefit relations in procurement.

During *risk analysis*, the team considers internal and external risks related to systemic effects in all five dimensions. For example, considering the evolving regulations on environmental accountability as a risk, the team develops a set of transparency requirements for the system. It also identifies uncertainties about future shifts in procurement as sustainable products become more competitive. So, it includes a feature to monitor these uncertainties.

During *requirements elicitation*, the team employs participatory techniques. The inclusive perspective lets the project leverage contributions from a broader set of stakeholders, including local service providers. In a series of workshops, the team uses a sustainability reference goal model to derive specific sustainability goals for the project and align them with other system goals, while deriving extended usage scenarios with the local community representatives.

The resulting *requirements specification* is based on a template that includes checklists for sustainability criteria and standards compliance in all five dimensions. The document is circulated among all the stakeholders and is shared with regulatory agencies to demonstrate that the project meets relevant sustainability rules. So, it's also used more actively in subsequent stages.

Sustainability Debt

The system resulting from this procurement project is different when development takes into account sustainability principles and therefore long-term consequences.

Focusing on sustainability design, software engineers must adopt a mind-set quite different from the puzzle-solving attitude often found in engineering and business. Now, the objective is to identify and understand "wicked problems": problems that are deeply embedded in a complex system with no definitive formulation and no clear stopping rule. In such cases, every solution changes the nature of the problem, so little opportunity exists for trial-and-error learning.[7,8] Instead, we need an adaptive, responsive, and iterative approach emphasizing shared understanding.

Figure 1 highlights selected immediate, enabling, and structural effects of the procurement system in the five sustainability dimensions. Consider a system feature that tracks individual products' carbon footprint, letting users choose products with lower footprints. The compound structural effect in the economic dimension can benefit local suppliers with environmentally sustainable production and can lead to a reduced carbon footprint.

The diagram in Figure 1 supports interactive collaboration among stakeholders to discover, document, and validate the system's potential effects. Not all effects will be positive. For example, automating product selection rules to minimize the carbon footprint takes away the manager's freedom to make decisions in the procurement process.[9] This can reduce mutual trust between the organization's members.

The diagram also facilitates a conversation about *sustainability debt*: decisions made for the present situation have invisible effects that accumulate over time in each of the five dimensions.[10] When we increase energy consumption, reduce individual privacy, impose technical barriers, or incur additional financial costs, we incur debts in these dimensions to different stakeholders. Making these effects visible is the first step to understanding and considering them in system design decisions.

Requirements Are the Key

In those two projects, a series of decision points occurred during system design. Many of them were requirements-engineering activities that occurred repeatedly in all iterations throughout the projects. Each decision influenced the decision space of subsequent choices and profoundly affected the system and its effects. Table 1 highlights how key activities change when we consider sustainability design principles.

Requirements' leverage becomes clear when we consider their relationships with engineering techniques. We develop techniques to quantify, construct, and test artifacts and to control whether the results fall in an acceptable range. However, for design concerns such as usability, performance, maintainability, or sustainability, such techniques are only applied once a need has been identified. Without such a need, the engineering techniques will remain unused and hence have no effect on the project.

For example, techniques for increasing technical sustainability

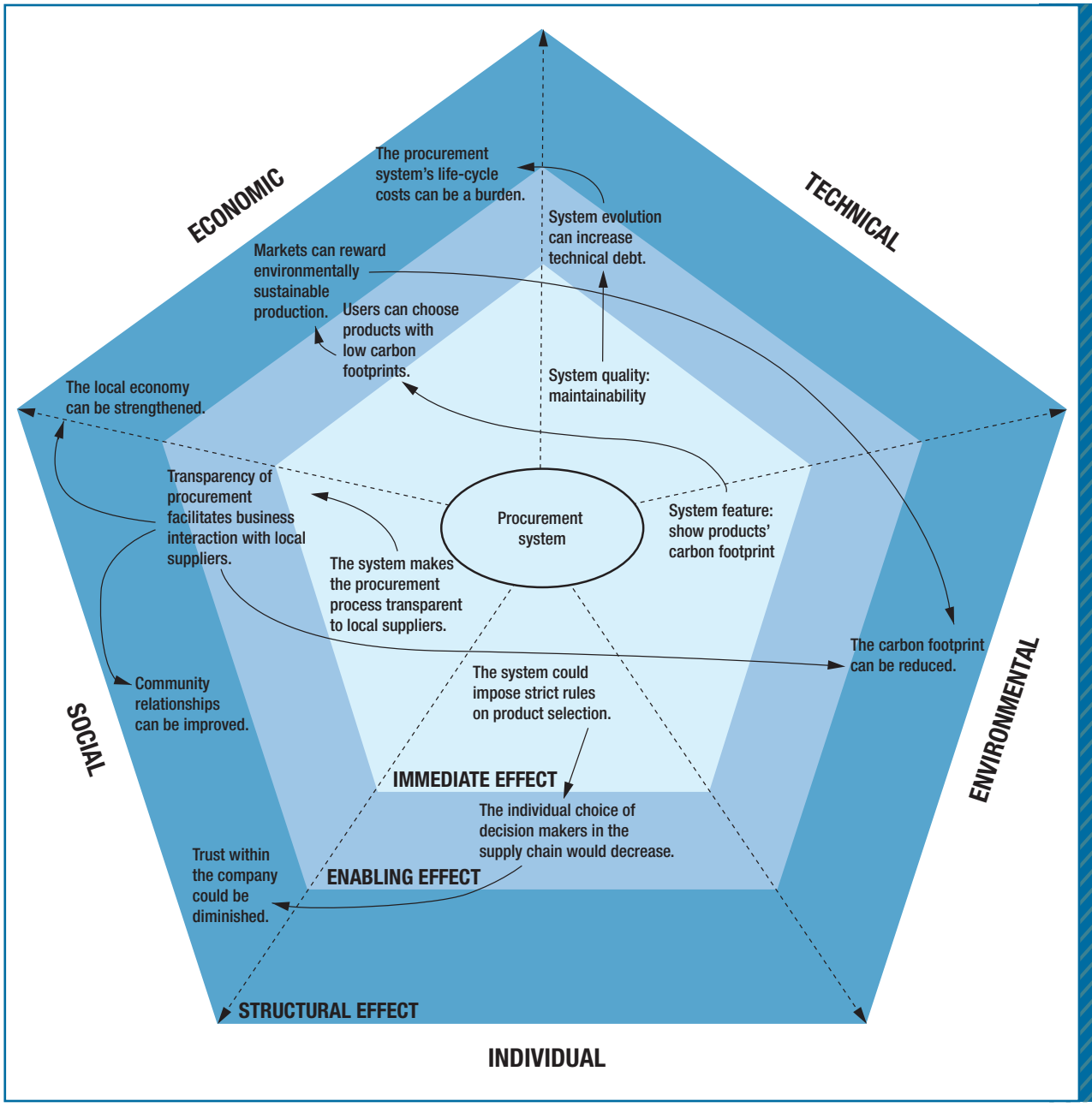

FIGURE 1. Selected immediate, enabling, and structural effects of the procurement system in the five sustainability dimensions. The diagram supports interactive collaboration among stakeholders to discover, document, and validate the system's potential effects.

abound, ranging from architectural design patterns to documentation guidelines. Yet, because applying these techniques often involves an up-front investment of effort, it occurs only when a longer life expectancy of a system is recognized and expressed. On the other hand, a stated requirement for which no technique yet exists will lead to an identified gap in technological ability. This means that in practice, systemic changes to the activities in Table 1 will dominate the effects of whatever techniques we develop to support these activities.

So, requirements engineers play a key role in sustainability. As "sustainability engineers," they go beyond a narrow system perspective and follow an interdisciplinary,

TABLE 1

Table 1. Software engineering practices for sustainability.*

Task	Standard current practice	Focus of future practice
Mind-setting	The world is a puzzle, and we should solve the problem.	The world is complex, and we should first understand the dilemmas.
Determination of the project objective and the system purpose, boundary, and scope	Focus on the immediate business need and key system features. Don't question the project's or system's purpose.	Emphasize how the project can affect sustainability in all dimensions. Strive to advance sustainability in multiple dimensions simultaneously. Experiment with different system boundaries to understand the alternative impacts.
External constraint identification	See constraints as imposed by the direct environment of the system and its technical interfaces. Minimize the constraints considered, but include legal, safety, security, technical, and business resources.	See constraints in each dimension as opportunities. Look for constraints from additional sources, starting with company corporate-social-responsibility policies, legislation, and sustainability standards.
Stakeholder identification	Minimize the number of stakeholders involved, and focus on those who have influence. Focus on internal stakeholders, and exclude unreachable stakeholders.	Maximize stakeholder involvement in an inclusive perspective integrating external stakeholders, and involve those who are affected. Assign a dedicated role to be responsible for sustainability, and introduce surrogate stakeholders to represent outside interests.
Success criteria definition	Focus on the financial bottom line at project completion. Measure the business outcome and financial return on investment.	Focus on advancing multiple dimensions simultaneously, including financial aspects, and take into account that most effects occur after project completion.
Requirements elicitation	Focus on the features and immediate effects the stakeholders want.	Help the stakeholders understand the system's enabling effects. Use creativity techniques and long-term scenarios to forecast the potential structural impact.
Risk identification	Identify risks that threaten timely project completion within the budget.	Include the effects on the system's wider environment. Include enabling and structural effects and risks that can develop over time.
Tradeoff analysis	View tradeoff analysis as a prioritization and selection problem, and let the key stakeholders decide.	Strive to transform sustainability tradeoffs into mutually beneficial situations. Ensure that a wider range of stakeholders (or their surrogates) discuss sustainability tradeoffs.
Go/no-go decision	Base the decision on feasibility, financial costs and benefits, and risk exposure to project participants—that is, internal stakeholders.	This continues to be an internal business decision but is documented to show to external audiences that it took into account sustainability indicators and enabling effects. The decision is based on a consideration of positive and negative effects in all five dimensions.
Requirements validation	Let key stakeholders verify that their interests are captured.	Ensure broad community involvement focused on understanding effects.
Project completion	Verify whether success criteria are met on the completion date. After that, focus on maintenance and evolution.	Evaluate the effects in all five dimensions over a certain time frame after completion, aligned with the expected timescale of effects.
Requirements documentation	Current templates ignore long-term effects and sustainability considerations.	Templates require information about sustainability as a design concern and support analysts with checklists.

** For a description of the dimensions mentioned in the table, see the section "Sustainability in Software Engineering."*

systems-oriented, stakeholder-focused approach, supported by higher management and executives. Their task is to understand the nature of software-intensive systems and the impact those can have on their social, technical, economic, and natural environments and the individuals in those environments.

This responsibility is reflected in the new UK Standard for Professional Engineering Competence, which specifies that engineers are to "act in accordance with the principles of sustainability, and prevent avoidable adverse impact on the environment and society."[11] It's up to SE curricula developers to equip

future software engineers with the competences required to simultaneously advance goals in all five dimensions, beyond the technical and economic.

For a long time, concerns about such effects have taken a backseat in SE, but this is changing as standards are being adjusted. For example, the working group WG42 on ISO/IEC 42030 (Architecture Evaluation) is discussing energy efficiency and environmental concerns at the software architecture level. In addition, the IEEE P1680.1 Standard for Environmental Assessment of Personal Computer Products is being revised.

Although these steps are important, a full consideration of all five sustainability dimensions is needed on the level of quality models, system documentation templates, and the analysis of systemic effects throughout system life-cycle stages. Requirements engineers will often be responsible for introducing relevant standards in each of the five dimensions into the elicitation and specification process. To support this, revisions of the ISO 25000 series should incorporate sustainability considerations related to software systems' quality attributes. In addition, ISO 29148 should acknowledge the importance of system characteristics beyond interaction with human users and encourage consideration of the systemic effects of software systems in RE.

Software's critical role in society demands a paradigm shift in the SE mind-set. Sustainability design emphasizes an appreciation of wicked problems over a focus on puzzles and pieces, systems thinking over computational problem solving, and an integrated understanding of systems over a divide-and-conquer approach to systems analysis.

Although these challenging shifts won't come easy, taking such perspectives provides an opportunity to stand out, an invitation to innovate, and an occasion for software engineers and companies to distinguish themselves with a unique selling point in a competitive market. We also have the opportunity to help shape broader sustainability policy. A shift to a sustainable society requires large-scale change both in government policy and in engineering and business practice; neither on its own will suffice. But regulatory change is much easier if it builds on established best practices, so software practitioners must take the lead.

If you agree that we, as software engineers, have a responsibility for the long-term impact of the systems we design, the sustainability design principles provide an opportunity to get started. We can and should start now, and practitioners can lead the way. We need to collect experiences in applying sustainability principles in SE and learn from the process. An important way to make this vision of software as a force for sustainability a reality is by cooperation between industry and academia.

Successful collaborations to integrate sustainability concerns into established practices can significantly and positively influence the long-term effects of the systems we design. To facilitate this, we must do three things.

First, we must identify and tackle causes of unsustainable software design. For this, industry can invite academics to research, analyze, and reengineer their current development processes and practices for improved sustainability.

Second, we must develop exemplar case studies that demonstrate the benefits of sustainability design in SE. For this, early adopter industrial collaborators can partner with academics to apply research findings such as those summarized in Table 1 and report on longer-term results.

> Software's critical role in society demands a paradigm shift in the software engineering mind-set.

Finally, we must build competences in the theory and practice of sustainable design into the training of all software engineers. Industry can make the demand for software practitioners trained in sustainability principles explicit by requiring specific competences from potential employees. Researchers and educators should develop improved curricula that incorporate sustainability principles and ensure that future software professionals possess the competences needed to advance sustainability goals through SE.

Let's get started.

Acknowledgments

This research is supported by the Deutsche Forschungsgemeinschaft project EnviroSiSE (PE2044/1-1); FAPERJ (210.551/2015); CNPQ (14/2014); NSERC (RGPIN-2014-06638); the European Social Fund; the Ministry for Science, Research, and the Arts Baden-Württemberg;

ABOUT THE AUTHORS

CHRISTOPH BECKER is an assistant professor at the University of Toronto, where he leads the Digital Curation Institute, and a senior scientist at the Vienna University of Technology. His research focuses on sustainability in software engineering and information systems design, digital curation and digital preservation, and digital libraries. Becker received a PhD in computer science from the Vienna University of Technology. Contact him at christoph.becker@utoronto.ca.

STEFANIE BETZ is a senior research scientist in the Karlsruhe Institute of Technology's Department of Applied Informatics and Formal Description Methods. Her research centers on sustainable software and systems engineering, particularly from the perspective of requirements engineering and business process management. Betz received a PhD in applied informatics from the Karlsruhe Institute of Technology. Contact her at stefanie.betz@kit.edu.

RUZANNA CHITCHYAN is a lecturer in the University of Leicester's Department of Computer Science and a member of the Centre for Landscape and Climate Research. Her research centers on requirements engineering and architecture design for software-intensive sociotechnical systems and sustainability. Chitchyan received a PhD in software engineering from Lancaster University. Contact her at rc256@leicester.ac.uk.

LETICIA DUBOC is a lecturer in the State University of Rio de Janeiro's Department of Computer Science and an honorary research fellow at the University of Birmingham. Her research focuses on software system sustainability and scalability, particularly from the perspective of requirements engineering and early analysis of software qualities. Duboc received a PhD in computer science from University College London. Contact her at leticia@ime.uerj.br.

STEVE M. EASTERBROOK is a professor in the University of Toronto's Department of Computer Science and a member of the School of the Environment and the Centre for Global Change Science. His research focuses on climate informatics—specifically, applying computer science and software engineering to the challenge posed by global climate change. Easterbrook received his PhD in computing from Imperial College London. Contact him at sme@cs.toronto.edu.

BIRGIT PENZENSTADLER is an assistant professor of software engineering at California State University, Long Beach. Her research focuses on software engineering for sustainability and resilience; she leads the university's Resilience Lab. Penzenstadler received a habilitation in environmental sustainability in software engineering from the Technical University of Munich. Contact her at birgit.penzenstadler@csulb.edu.

NORBERT SEYFF is a professor in the School of Engineering and the Institute of 4D Technologies at the University of Applied Sciences and Arts Northwestern Switzerland and a senior research associate in the University of Zurich's Department of Informatics. His research focuses on requirements engineering and software modeling, particularly on empowering and supporting end-user participation in system development. Seyff received a PhD in computer science from Johannes Kepler University Linz. Contact him at norbert.seyff@fhnw.ch.

COLIN C. VENTERS is a senior lecturer in software systems engineering at the University of Huddersfield. His research focuses on sustainable software systems engineering from a software architecture perspective for presystem understanding and postsystem maintenance and evolution. Venters received a PhD in computer science from the University of Manchester. Contact him at c.venters@hud.ac.uk.

and the Vienna Science and Technology Fund (WWTF) through project BenchmarkDP (ICT2012-46). Special thanks to our friend and colleague Sedef Akinli Kocak, a PhD researcher at Ryerson University, for her contributions to this article.

References

1. H. Koziolek, "Sustainability Evaluation of Software Architectures: A Systematic Review," *Proc. Joint ACM SIGSOFT Conf.—QoSA and ACM SIGSOFT Symp.—ISARCS on Quality of Software Architectures—QoSA and Architecting Critical Systems—ISARCS* (QoSA-ISARCS 11), 2011, pp. 3–12.
2. J.A. Tainter, "Social Complexity and Sustainability," *Ecological Complexity*, vol. 3, no. 2, 2006, pp. 91–103.
3. B. Penzenstadler et al., "Safety, Security, Now Sustainability: The Nonfunctional Requirement for the 21st Century," *IEEE Software*, vol. 31, no. 3, 2014, pp. 40–47.
4. J. Elkington, "Enter the Triple Bottom Line," *The Triple Bottom Line: Does It All Add Up? Assessing the Sustainability of Business and CSR*, A. Henriques and J. Richardson, eds., Earthscan, 2004, pp. 1–16.
5. "Who Lives the Longest? Busting the Social Venture Survival Myth," E3M, 2014; http://socialbusinessint.com/wp-content/uploads/Who-lives-the-longest_-FINAL-version2.pdf.
6. C. Bomfim et al., "Modelling Sustainability in a Procurement System: An Experience Report," *Proc. IEEE 22nd Int'l Conf. Requirements Eng.* (RE 14), 2014, pp. 402–411.
7. H.W. Rittel and M.M. Webber, "Dilemmas in a General Theory of Planning," *Policy Sciences*, vol. 4, no. 2, 1973, pp. 155–169.
8. S. Easterbrook, "From Computational Thinking to Systems Thinking: A Conceptual Toolkit for Sustainability Computing," *Proc. 2nd Int'l Conf. Information and Communication Technologies for Sustainability*, Atlantis Press, 2014; doi:10.2991/ict4s-14.2014.28.
9. J.A. Klein, "A Reexamination of Autonomy in Light of New Manufacturing Practices," *Human Relations*, vol. 44, no. 1, 1991, pp. 21–38.
10. S. Betz et al., "Sustainability Debt: A Metaphor to Support Sustainability Design Decisions," *Proc. 4th Int'l Workshop Requirements Eng. for Sustainable Systems* (RE4SuSy 15), 2015; http://ceur-ws.org/Vol-1416/Session2Paper4.pdf.
11. *UK Standard for Professional Engineering Competence (UK-SPEC)*, Engineering Council, 2014.

Embedding Reflection and Learning into Agile Software Development

Jeffry Babb, West Texas A&M University
Rashina Hoda, University of Auckland
Jacob Nørbjerg, Aalborg University

// The theoretical underpinnings of agile methods emphasize regular reflection as a means to sustainable development pace and continuous learning, but in practice, high iteration pressure can diminish reflection opportunities. The Reflective Agile Learning Model (REALM) combines insights and results from studies of agile development practices in India, New Zealand, and the US with Schön's theory of reflective practice to embed reflection in everyday agile practices. //

IMPLICIT IN THE agile tenet—"at regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly"[1]—is the fact that agile teams should regularly assess their process and its outcomes. Individuals and teams are expected to engage in regular reflection to maintain and improve their software processes.[1–5] Using reflective practice, agile developers can determine the degree to which the process, or aspects of it, can be expanded, adapted, altered, or abandoned.[3] Thus, agile software development projects are expected to consistently yield results in both areas of productivity (regular delivery of working software) and the reflection required to maintain an effective software process. However, in practice, this balancing act between productivity and continuous process improvement and learning is nontrivial to achieve and sustain.[3] Individuals and teams often perform under sustained pressure to deliver customer value (iteration pressure) and might abandon or under-engage in core agile practices related to reflection and learning, leading to diminished opportunities for knowledge sharing, learning, and reflection.[3,6] Without an ongoing reflective dialog about their practices, software teams may experience process erosion.[7]

In this article, we present the Reflective Agile Learning Model (REALM), showing where and how to integrate reflective practice in agile software development. The model combines insights and results from industrial studies of agile development practices in India, New Zealand, and the US with Schön's theory of reflective practice.[8–10] Although a theoretical synthesis of agile methods and Donald Schön's reflective practice isn't a novel suggestion,[2,5,11,12] our approach underscores the effectiveness of reflective practice to sustain and improve an agile development team. REALM provides concrete guidelines for embedding reflective practice into an iterative and agile software engineering

0740-7459/14/$31.00 © 2014 IEEE

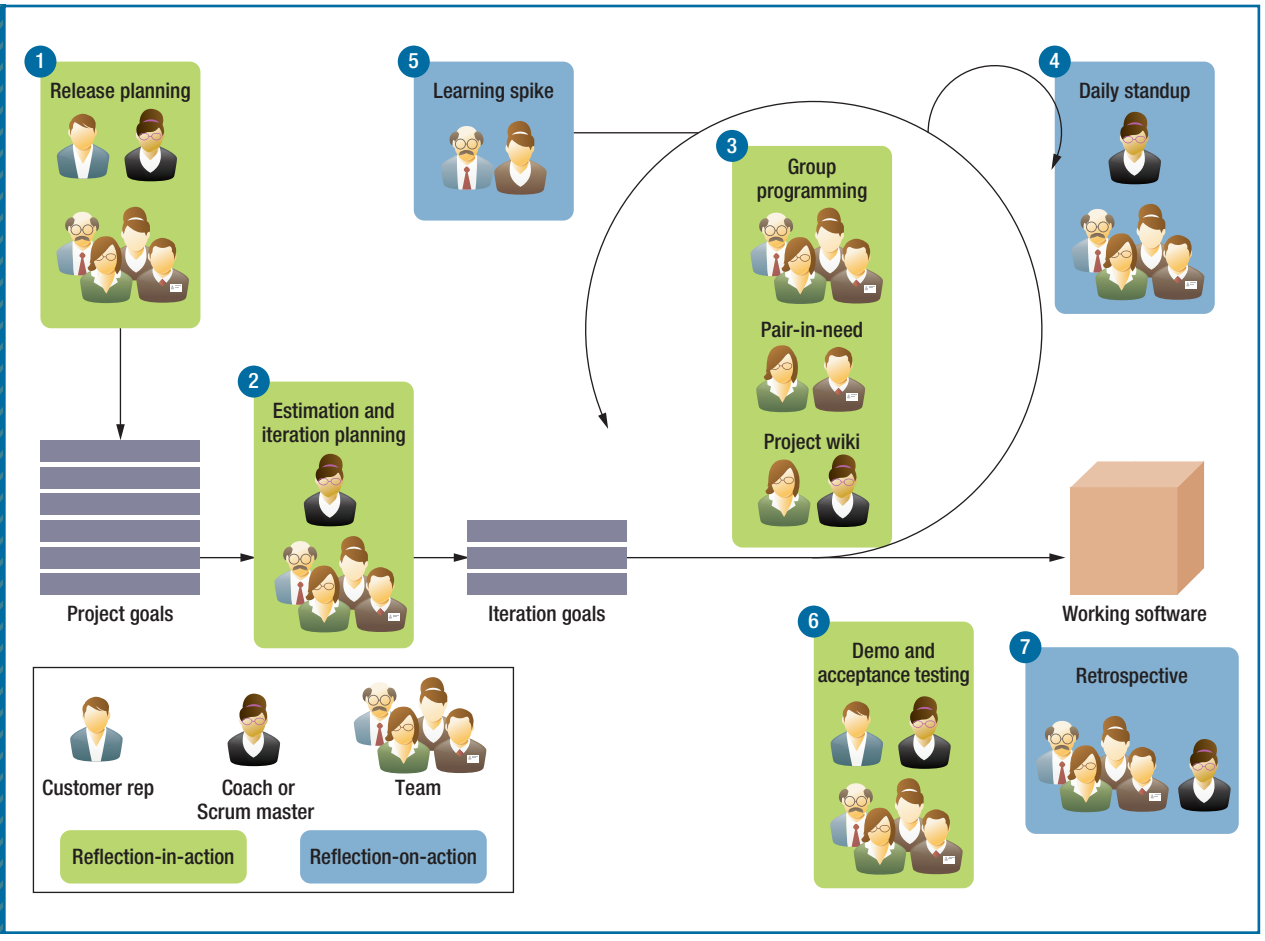

FIGURE 1. The Reflective Agile Learning Model (REALM). Standard and adapted agile practices with implicit reflection-in-action opportunities are depicted in green; learning-focused agile practices with explicit reflection-on-action opportunities are depicted in blue.

development cycle. This model is an evolution of previous models but has been refined to present a reflective process maintenance perspective.[3,6]

Drawing upon Evidence

We created REALM by drawing upon evidence from two industrial-studies of agile practices: a longitudinal Action Research study of agile software development in a small software company in the US, and a Grounded Theory study of software teams and practices involving 58 software practitioners from 23 different companies in New Zealand and India, most of whom were using agile methods such as Scrum, extreme programming (XP), or a combination of the two. Both studies involved observations of workplaces and practices plus direct face-to-face interviews of team members as a means to collect qualitative data.

A prominent finding emerging from both studies independently was the teams' adaptations of standard agile practices to suit particular organizational or project contexts. In particular, we found that the practices most likely to be lost, sacrificed, or compromised over time were those most closely related to learning and reflection.[3,6]

A model for the infusion of reflective practice into XP was a salient outcome of the US study. Developers engaged in daily learning and reflection habits that reinforced and complemented activities already endemic to agile methods. We refined and augmented REALM via the learning-focused practices of the Scrum and XP teams in the Grounded Theory study. The findings from both studies complement one another and combine to inform a new model of reflection in agile practice.

TABLE 1

Learning and reflection opportunities in the Reflective Agile Learning Model (REALM).

Agile practices	Reflection-in-action
Release planning	Learning about the project, domain, and product
Collective estimation and iteration planning	Learning about each other's skills and areas of expertise, and the team's collective capacity to delivery (velocity)
Group programming	Learning new skills and expertise from each other
Pair-in-need	Novices learning about the team and project as they pair with experts; pairs learning a new technology or skill as they work on a challenging task
Project wiki	Capturing project-based learning about the domain and individual reflections (blogs)
Demo and acceptance testing	Learning from acceptance test failures that fuel refinements and guide product vision
Agile practices	**Reflection-on-action**
Daily standup	Reflection on action performed the previous day, planning for the day's actions, and highlighting impediments
Learning spike	Adapted practice that embodies reflection on action as some members devote exclusive time to new learning
Retrospective	Standard agile practice devoted exclusively to reflection on actions performed in previous iteration

A New REALM of Application

As part of a learning team, a reflective practitioner is one who engages in two separate but intertwined reflective processes: *reflection-in-action* concerning the individual responses to shaping the problem at hand and its possible solutions, and *reflection-on-action* concerning a post hoc evaluation process for extending and validating individual and team repertoire. Schön defines repertoire as an accumulation of ideas, examples, situations, and actions gleaned from the whole of a practitioner's experience such as it is accessible for understanding and action.[9]

At first, this might seem either too trivial or too obtuse to systematically guide a software process. However, without ongoing and explicit attention to reflection, even an agile development team can lose its ability to learn and improve. Thus, the purpose of REALM is to routinize awareness of ongoing reflection in and on action in agile development projects. The model's development is strongly grounded in the reflection-in and -on action practices identified across the two studies. Figure 1 illustrates how REALM highlights opportunities for reflection-in-action, which is implicit in standard and adapted agile practices (green); reflection-on-action includes explicit learning-focused agile practices (blue). Table 1 summarizes the model's steps.

Reflection-in-Action

A practitioner who's more conscious of her reflection-in-action will be more apt to take note of her reactions and solutions to new problems in a manner that can inform her repertoire. This is useful both to the practitioner and to the team she works with. In REALM, these reflections are most evident in release planning, estimation and iteration planning, group programming, pair-in-need, project wikis, and product demo and acceptance testing. During these agile activities, developers are most engaged with the materials of design and development.

Release planning. In release planning, overall domain context, project goals, and product features are discussed, which drives the tempo and direction of team practices throughout the project. Any clues that a project might reveal about past contexts and experiences will become evident to the developer at this stage. The reflective practitioner, with heightened awareness, could be more apt to draw from repertoire at this point.

Collective estimation and iteration planning. The team collectively selects and estimates tasks for the current iteration, and in so doing, team members acquire and confirm

knowledge of each other's skills and areas of expertise. Learning about the team's collective capacity to deliver (velocity) starts here. Including the whole team—not just developers or leads, but stakeholders, too—ensures a shared vision and understanding of the project based on inputs from multiple perspectives and collective agreements. Teams can use this as an excellent opportunity to discuss customer expectations with regard to team productivity, which is a key element in managing learning throughout the project. The reflective practitioner will look for metaphors and patterns to allow past experience to guide and clarify task formulation and prioritization.

> Individuals and teams need to constantly add to and draw from their repertoire.

Group programming. Software development acutely draws on tacit knowledge and repertoire. The working rhythm and velocity of the team will be established, and any corrections and adjustments, particularly those made "on the spot," will most benefit from reflection-in-action. Overtly and tacitly, habits, skills, and knowledge pass between team members in a revelatory manner in terms of project trajectory and risk mitigation.

Pair-in-need. Agile teams might not pair-program all the time in practice. However, they do pair up on an as needed basis such as when a newcomer to the team pairs with an experienced member to learn "the tricks of the trade." Other instances include team members pairing up to learn and apply a new technology or skill to solve a challenging task.

Project wikis. Teams often capture project-based terminology and context-specific conventions in a wiki format shared between the team and the customer. Individuals might also engage in writing blogs to capture and share learning gained from project experiences.

Product demo and acceptance testing. Here the team can again draw on its collective repertoire, as it must adjust to the customer's interaction with and feedback regarding the software's emerging functionalities. Although agile methods call on teams to constantly involve the customer, the customer can rarely immerse into the same "zone" as the developers at the height of the development process, making for a constant translation process. Reflection-in-action at the moment of iteration demonstration and acceptance will have an appreciable impact on the next iteration and the nature of the product.

Reflection-on-Action

Reflection-on-action is the "after-action report" on how repertoire has changed. It's during reflection-on-action that teams attempt to answer questions regarding what to keep, discard, and modify in repertoire. Opportunities for reflection-on-action in REALM are the retrospective, the daily standup, and learning spikes.

Retrospective. While most agile methods call for a retrospective, in practice, it's an activity that can easily be skipped or skimped. Retrospectives provide a means of examining the thoughts, experiences, and notes on the various consultations with and examinations of both individual and team repertoire. This is a process of making the tacit explicit by asking of our actions, "What did/does this mean?" Therefore, reflection-on-action in general, and in particular during the retrospective, is both vital and challenging.

Daily standup. Repertoire is mostly called on during the "action" stages of a project: a daily standup meeting allows the practitioner and the team to recall recent reflection-in-action while it's still fresh. This daily routine constitutes a critically important component for engaging in reflective practice to benefit process maintenance. Throughout daily practice, practitioners on the team will ideally accumulate observations and reflections. Thus, the daily standup is an opportunity to aggregate reflections to reinforce or challenge team and individual repertoire. During the daily standup, the team can discuss project velocity and story burndown, as well as develop a platform for observations on repertoire use and understanding.

Learning spikes. Another test of repertoire is the development of proof-of-concept "spike" solutions to determine individual and team ability to branch out into new techniques, tools, or knowledge. Spike solutions

enhance individual and team repertoire via experiential learning.

Practical Examples

During the US study, the software team worked over a nine-month period to make reflection-in-action a habit and coerce reflection into an explicit activity. Among other items, they adopted the habit of on-the-spot reflection-in-action by "microblogging" in a team wiki to accumulate their thoughts and observations. At the end of the day, the developers also contributed to a team blog for reflection later that night or the next morning:

> *"I put it in the wiki right then, I knew I wanted to come back to it later. There have been a lot of times I knew I wanted to come back to something that I looked at six months ago, but I couldn't remember where I found it, so I don't want to lose this knowledge." – small shop owner and lead developer, Virginia, US*

This explicit process of reflective practice was slowly ingrained into the team by habits engaged in throughout the day. During development, particularly when pair-programming, developers would consciously remember to record tidbits of reflective writing in their team wiki. This habit of microblogging was used to "feed" the daily standup with a synopsis of interesting observations and other helpful items. This activity became a team expectation to be engaged in equally alongside more traditional activities that gauge project velocity, coordinate task completion, and so on. These daily habits further supported the project retrospectives (reflection-on-action) that formalized the development of individual and team repertoire. Because iteration pressure to deliver maximum customer value as quickly and efficiently as possible was quite high, reflection-in-action helped team members annotate their experiences in the hope that their body of knowledge could be tested, verified/validated, and documented:

> *"I think that the blogging, the wiki, and the daily standup are very important ways to keep an active project on track. We have so many projects going on that if we do not follow up, either through the daily standup or through the wiki, the projects will get off track." – developer, Virginia, US*

The blogs and wikis used by the US team certainly aren't novel tools, but the way in which they facilitated the habits necessary to understand the tacit-to-explicit process of reflective practice was. Daily and regular engagement of these tools provided the team with a means of raising awareness and consciousness of how individual practitioners framed, shaped, and set problems in their daily work.

In the Grounded Theory study, mature teams tended to use standard practices such as retrospectives effectively to drive action based on reflections on previous iteration:

> *"With every retrospective, we certainly came up with ideas to improve our process, and I think with all those retrospective sessions under our belt, with all the experience sizing, planning, everything combined, it really made us evolve as a team." – business analyst, NZ*

Reflections during retrospectives led to teams introducing adapted practices such as the learning spike to concentrate on exclusive learning. For example, a team in New Zealand, as a result of a retrospective session, discovered that manual testing was placing high demands on team productivity. As a result of team discussions and with management support, they agreed to conduct a learning spike where team members took exclusive time off project work to create an automated test suite:

> *"We've just basically reduced our velocity and taken the time to do those things because we knew they were important. We made a call that we were not going to wimp out, and go back to the manual testing...we've taken automation a lot further... doing 100 percent automation." – agile coach, NZ*

While the spike led to a reduction in project velocity in the short-term, the team clearly benefitted from automating the testing in the long-term across all projects. Lessons from this experience were captured and shared by the agile coach as presentations to the local agile community.

Pair-in-need was a common practice used across teams in India and New Zealand to boost learning while benefitting from investing two minds on a particularly challenging problem:

> *"The way we do it is that if things are unpredictable, we always take up user stories as a pair … if something [task] requires—this is complex, this is design-intensive—we sit together and pair it." – agile coach, India*

While newer teams (with less than a year of experience) in the Grounded Theory study struggled to embed learning in everyday practice

ABOUT THE AUTHORS

JEFFRY BABB is an assistant professor of computer information systems at West Texas A&M University. His research interests include small-team software development, agile software development, and mobile application development. Babb received a PhD in information systems from Virginia Commonwealth University. Contact him at jbabb@wtamu.edu.

RASHINA HODA is the director of the Software Engineering Processes, Tools and Applications research group in the University of Auckland's Department of Electrical and Computer Engineering. Her research interests include self-organizing teams, agile software development, and human-computer interaction. Hoda received a PhD in computer science from Victoria University of Wellington. Contact her at r.hoda@auckland.ac.nz.

JACOB NØRBJERG is an associate professor in Aalborg University's Department of Computer Science. His research interests are in systems development organization and management and software process improvement. Nørbjerg received a PhD in information systems development from the University of Copenhagen. Contact him at jacobnor@cs.aau.dk.

to varying degrees, mature teams mastered reflection-in and -on action. Using standard and adapted reflective practices, these teams were able to discover better ways of working and also worked better together. They became highly self-organizing in nature.

From Learning Teams to Learning Organizations

In the software process context, a learning organization[8] is an environment most able to engage in reflective practice.[2] A learning organization recognizes the importance of individual and collective knowledge within the organization and the need to manage it as an asset.[2]. A learning organization values each member's professional development, realizing that effective people will make an effective product. Whereas reflective practice focuses on developing individual repertoire, a learning organization promotes the accumulation of individual repertoire.[9] In this sense, a learning organization provides a working environment that values learning as a daily activity.[8]

In essence, the introduction of reflective practice to a software development organization facilitates a learning system in which no particular software development method is as important as the ability to adapt and develop knowledge appropriate in the team. This leaves little doubt as to why agile methods have been so compelling in software engineering for the past decade: agile practically demands a learning organization. Moreover, the habits required to engage in reflective practice would more likely be fostered and nurtured in a learning organization.

REALM illustrates the relevance of reflective practice to software engineering for organizational learning as it helps accumulate individual and team repertoire. As individuals build their repertoire, and in turn establish a team repertoire, both the team and the individual are reinforced. Team reflections-on-action can reinforce individual reflection-in-action to cement learning in a manner that strengthens individuals, who in turn strengthen team acumen, and ultimately the organization itself.

Recommendations for the Reflective Agile Practitioner

A reflective agile practitioner is any software engineering professional—developer, tester, manager, coach—empowered by supportive management,[3] who engages in reflective practice as a means to enhance his or her personal and team productivity and fuel long-term process and team improvements. Based on our experiences with REALM, we can provide some recommendations for such reflective practitioners.

Harness Standard Agile Practices to Implicitly Capture Reflection-in-Action

It's easy to "go through the motions" when performing various agile practices. By being conscious of the learning opportunities present in everyday agile practices as suggested in REALM (such as release planning), an agile practitioner can gain significant knowledge about the product domain, usage context, required technologies (including learning new skills), and individual and team capabilities.

Make Effective Use of Retrospectives to Achieve Continuous Improvement

The retrospective is a standard agile practice explicitly designed to capture reflection-on-action. However, retrospectives are often easily abandoned or diminished to lip-service as teams get more used to the process and begin to focus exclusively on delivery. Reflective practitioners should hold on to the practice of retrospectives and use them effectively to reflect on their processes, thereby achieving continuous improvement.

Introduce Adapted Agile Practices to Explicitly Capture Reflection-on-Action

It's common for practitioners to devise new or adapted practices to overcome particular challenges and needs, such as learning spike, pair-in-need, and project wikis. A reflective practitioner should be open to inventing his or her own strategies or endeavor to be in sync with the latest in the practitioner community to help with learning and applying strategies devised by other practitioners.

Share Lessons within the Organization and Wider Software Community

Presenting experience reports at local and international conferences or workshops is a great way to distribute the knowledge and learning gained from personal experiences. Most agile conferences or workshops host practitioner tracks that welcome experience reports and presentations. Blogging is another effective practice to achieve similar outcomes with limited investments. Popular social networking communities also provide ample avenues to share with and learn from the wider practitioner community. Such efforts are typically well-received and even commended by peers and management and enable the practitioner to emerge as a thought-leader in his or her professional sphere. An even less demanding mechanism to share lessons with the wider community is to participate in research efforts that serve to capture the experiences of several practitioners, which in turn can help inform the international software community.

Although reflection as a means of learning is a highly espoused benefit and expected outcome of agile software methods, its practice doesn't explicitly provide guidance on how to create an environment that will support ongoing learning and reflection. Theoretically, however, the focus of agile principles on continuous learning presents a unique opportunity to benefit from the application of the reflective practice paradigm developed by Schön as a means to achieve process maintenance (and evolution).

REALM uses reflective practice to develop micro habits for uncovering tacit knowledge among a software development team. Whether an individual must adopt a new beneficial habit or cease an existing detrimental one, habit-forming is a behavior modification concern that's "easier said than done." As it has been suggested that practitioners capable of success with agile methods should generally be high-functioning, the agile tenet for self-organization is highly correlated with the habits required for successful reflective practice.

Using REALM, software teams can consider ways to implement reflective practice to achieve and sustain a much-emphasized element of agile methods: the ability to adapt and evolve methods for a continuously improving software process at the individual, team, and organizational levels.

References

1. A. Cockburn and J. Highsmith, "Agile Software Development, the People Factor," *Computer*, vol. 34, no. 11, 2001, pp. 131–133.
2. O. Hazzan and J. Tomayko, "The Reflective Practitioner Perspective in eXtreme Programming," *Extreme Programming and Agile Methods: XP/Agile Universe 2003*, F. Maurer and D. Wells, eds., Springer Berlin Heidelberg, 2003, pp. 51–61.
3. R. Hoda, J. Babb, and, J. Nørbjerg, "Toward Learning Teams," *IEEE Software*, vol. 30, no. 4, 2013, pp. 95–98.
4. N.B. Moe, T. Dingsøyr, and T. Dybå, "A Teamwork Model for Understanding an Agile Team: A Case Study of a Scrum Project," *Information and Software Technology*, vol. 52, 2010, pp. 480–491.
5. S. Nerur and V. Balijepally, "Theoretical Reflections on Agile Development Methodologies: The Traditional Goal of Optimization and Control is Making Way for Learning and Innovation," *Comm. ACM*, vol. 50, no. 3, 2007, pp. 79–83.
6. J. Babb, R. Hoda, and J. Nørbjerg, "Barriers to Learning in Agile Software Development Projects," *Agile Processes in Software Engineering and Extreme Programming*, H. Baumeister and B. Weber, eds., Springer, 2013, pp. 1–15.
7. G. Coleman and R. O'Connor, "Investigating Software Process in Practice: A Grounded Theory Perspective," *J. Systems and Software*, vol. 81, no. 5, 2008, pp. 772–784.
8. C. Argyris and D. Schön, *Organizational Learning: A Theory of Action Perspective*, Addison Wesley, 1978.
9. D. Schön, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, 1983.
10. D. Schön, *Educating the Reflective Practitioner: Toward a New Design for Teaching and Learning in the Professions*, Jossey-Bass, 1987.
11. J.A. Highsmith, *Adaptive Software Development: A Collaborative Approach*, Dorset House, 2000.
12. J. McAvoy and T. Butler, "A Failure to Learn by Software Developers: Inhibiting the Adoption of an Agile Software Development Methodology," *J. Information Technology Case and Application Research*, vol. 11, no. 1, 2009, pp. 23–45.

Daily Stand-Up Meetings

Start Breaking the Rules!

Viktoria Stray, University of Oslo and SINTEF

Nils Brede Moe, SINTEF

Dag I.K. Sjøberg, University of Oslo and SINTEF

// Daily stand-up meetings are commonly used for software teams to collaborate and exchange information, but conducting them in a way that benefits the whole team can be challenging. We describe factors that can affect meetings and propose recommendations for improving them. //

IF YOU WORK on an agile project, you are probably familiar with some format of the daily stand-up meeting because it is a popular practice.[1,2] Some staff may see the meeting as a necessary venue for communication with the team while others consider it a waste of time. We have researched daily stand-up meetings for almost a decade and helped software engineering companies find ways to improve them. In this article, we share our findings and provide recommendations on how to conduct effective daily stand-up meetings.

Agile business methods introduced the practice of daily stand-ups to improve communication in software development teams. In these meetings, team members share information and thus become aware of what other members are doing, with the hope that team members will then align their actions to fit the actions of the others, i.e., mutual adjustment.[3] Furthermore, meetings that improve access to information foster employee empowerment.[4]

Despite clear guidelines for how the 15-min meeting should be conducted,[5] through our visits to more than 40 companies and several hundred teams, we have found that implementing daily stand-ups in a way that benefits the whole team is quite challenging. So, how can we adjust the practice to boost team performance? Understanding how to conduct these meetings for self-managing software development teams requires more than just examining the group's inner workings; we must also understand the organizational context surrounding them. Therefore, we have conducted a multiple-case study in four software companies to identify positive and negative aspects of daily stand-up meetings and provide recommendations on how to make them more valuable.

Research Background

The four companies we studied belong to different application domains. ITConsult is a Norwegian IT consulting company, TelSoft is an international telecommunications software company, GlobEng is a global software company that provides services for the engineering industry, and NorBank is a Nordic bank and insurance company. Data collection and analysis took place from 2010 to 2018.

To gather different perspectives on the meeting, we interviewed 60 project members with roles at all levels in 15 different teams (see Table 1). The overall discussion topic was teamwork and meetings, with a particular

Digital Object Identifier 10.1109/MS.2018.2875988
Date of current version: 16 April 2020

focus on daily stand-up meetings. More details on the interview guide and research context are available.[6,7] We observed and documented 102 daily stand-up meetings. Eight of the meetings from TelSoft and NorBank were audiotaped, transcribed, and coded based on a validated coding scheme for team meeting processes.[8] Figure 1 shows the proportion of words stated in each of the six categories we identified. The number of words can be considered to be a proxy for the time spent.

Daily Stand-Ups: What, How, When, and for Whom?

The interviewees expressed that discussions other than answering the three Scrum questions were most valuable. Furthermore, the value of the stand-ups was affected by meeting facilitation behavior, the dependencies among tasks, and the time of day the meetings were held. In addition, we found that junior team members were more satisfied with the meetings than senior team members. See Table 2 for a summary of the main benefits of and difficulties for conducting daily stand-ups.

What Is Discussed in the Meetings?

Participants in daily stand-ups are traditionally supposed to answer a variant of only the following three questions: "What did I do yesterday?" (Q1), "What will I do today?" (Q2), and "Do I see any impediments?" (Q3). We found that the teams addressed far more than the three questions within the 15 min. Analysis of the transcribed meetings showed that, on average, only 34% of the meeting was spent on answering the three questions (see Figure 1). The teams spent almost as much time (31%) on elaborating on problem issues, discussing possible solutions, and making decisions, which is also known as *problem-focused communication*.[8]

According to Scrum, the daily stand-up should not be used to discuss the solutions to obstacles raised. However, in the interviews, problem-focused communication was the most frequently mentioned positive aspect of these meetings. The teams highly valued having an arena for quick problem solving, even if the problem-solving sessions were as short as 1 min.

Because the daily stand-up is characterized by limited time, uncertainty, and incomplete information, it is reasonable to question whether spending this time on making decisions is worthwhile. A thorough definition and analysis of the problem characterizes a successful problem-solving process, which may be time-consuming.

To understand decision making in this context, we use the theory of naturalistic decision making,[9] which postulates that experts can make correct decisions under challenging conditions, such as time pressure, uncertainty, and vague goals, without having to perform extensive analyses and compare options. They can use their experience to recognize problems that they have encountered previously. They form mental simulations of the current problem, which they use to quickly suggest appropriate solutions. Because problem-focused communication is essential for problem solving,[8] and the stand-up is the only daily coordination arena for the whole team, we argue that the meeting is necessary for effective decision making in agile teams.

Both meeting transcripts and observations showed that the problem-focused communication usually happened after the team members had

Table 1. The data sources.

Company	ITConsult	TelSoft	GlobEng	NorBank	Total/average
Locations visited	Poland, Norway	Malaysia, Norway	China, Poland, Norway, United Kingdom	Norway	Nine sites
Number of teams studied	2	3	7	3	15
Average team size (minimum/maximum)	9.5 (9/10)	9.3 (9/10)	8.3 (5/13)	14 (13/14)	10.3 (5/14)
Number of interviews	15	20	13	12	60
Number of stand-up meetings observed	19	39	13	31	102
Average stand-up meeting duration in minutes (minimum/maximum)	10.8 (4/18)	15.5 (7/24)	11.6 (4/18)	8.9 (4/13)	11.7 (4/24)

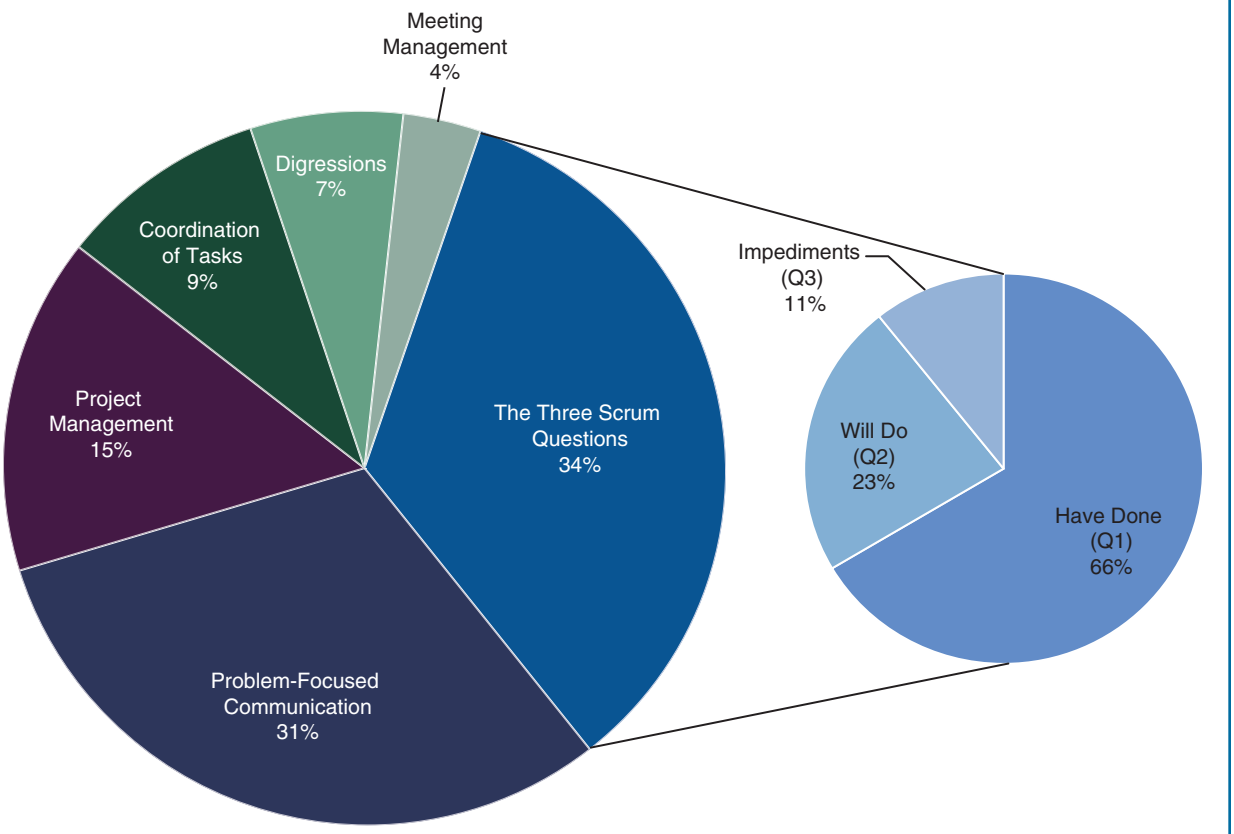

FIGURE 1. The proportion of topics discussed in daily stand-up meetings.

Table 2. The benefits and problems of daily stand-up meetings.

Main benefits
• Problems are identified, discussed, and resolved quickly.
• The team's cohesion and shared commitment improve.
• There is a greater awareness of what other team members are doing.
• Interactions are better coordinated through mutual adjustment.
• The decision-making process becomes more effective.
Main problems
• The information shared is not perceived to be relevant, particularly due to the diversity in roles, tasks, and seniority.
• Managers or Scrum masters use the meeting primarily to receive status information.
• Productivity decreases because the day is broken into slots.

described what they were going to do (Q2) or had mentioned an obstacle (Q3). Just relaying what they had been doing (Q1) rarely triggered problem-focused communication. Therefore, Q2 and Q3 are clearly most important for making quick decisions. While the interviewees expressed they were least happy to spend time on Q1, the teams actually spent the most time on that question, mainly because talking about what had been done often ended up as cumbersome reporting with superfluous details.

The challenge of knowing what others need to know tended to result in team members trying to include

everything they had worked on, independent of its relevance to others. In one team that did not spend much time on Q1, the Scrum master explained, "We don't want to know the details of what you did yesterday, but something that is useful for the whole team. We had to change the mind-set and find out what to say in the meeting. We practiced a lot and improved."

How Are the Meetings Facilitated?

If the behavior of the meeting facilitator made team members address the facilitator, we observed that the stand-up often became a status-reporting meeting. If the facilitator managed to make team members talk to each other, it tended to become a discussion meeting. To illustrate the two types of meetings, we show conversation charts from two observations in TelSoft (Figures 2 and 3). The lines represent an interaction between two participants.

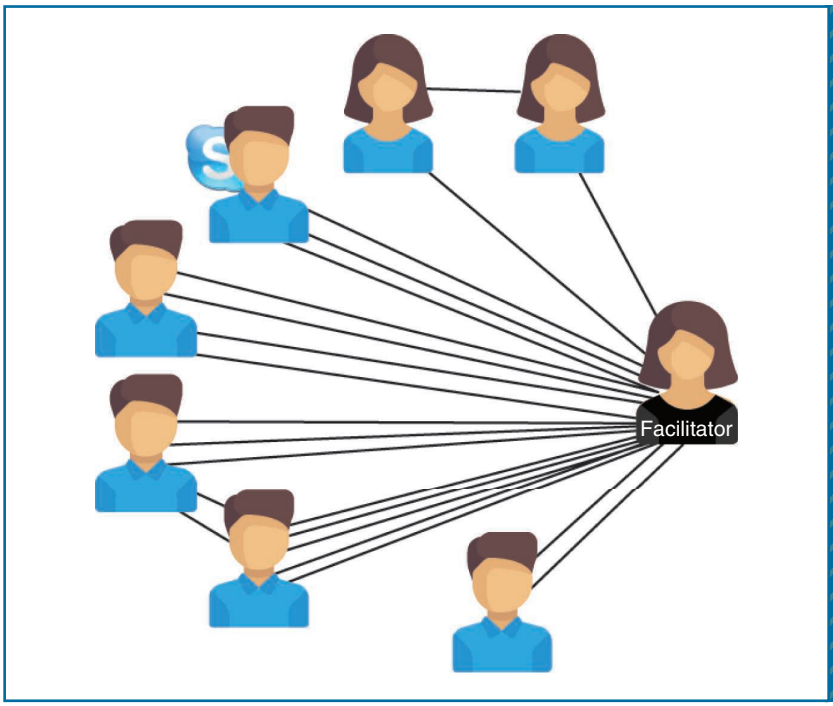

FIGURE 2. A conversation chart for a reporting meeting.

Figure 2 shows a reporting meeting in which the facilitator controlled the conversation by allocating turns for speaking. In this way, team members naturally directed their responses to her and had fewer interactions with each other. Figure 3 shows a discussion meeting in which team members engaged in what the other members were saying. The facilitators who managed to get people to talk to each other were usually involved in the day-to-day work and solved team tasks, so therefore had less need for status information.

Team members who practiced daily stand-ups as reporting meetings expressed negative attitudes. One member said it felt like having oral exams every day and another said, "No one in the team really wants to be at the status meeting." When the regular facilitator of such meetings was absent, the team usually either dropped the meeting that day or they conducted a discussion meeting instead.

One developer explained, "Today the Scrum master was not there, and we probably had our best daily meeting. Because usually, the meetings are focused on information useful to the Scrum master, but not for us. He is also more interested in what we have done than what we are going to do." Other research has found that poor meetings may have a negative effect beyond the meetings themselves, such as reduced job satisfaction, coworker trust, and well-being.[4,11]

When the facilitator had a personal interest in the status of specific tasks, they often gave team members working on these tasks more attention, leading to an unbalanced contribution. One team member said, "The Scrum master chooses the next person to talk based on the tasks that are important to her." We observed that people who spoke early were given the time they wanted, while the Scrum master cut short those who spoke last, because the meeting was approaching its time limit. It is negative for a team if, for example, developers are given more time to speak than testers. To make the team productive, everyone in the team should speak roughly the same amount of time.[10]

Facilitating the meeting can be even more challenging when the team members are distributed beyond the physical location of the meeting, which was the case in many of the teams we studied. The teams with facilitators who used video instead of only audio had more discussion meetings. One developer explained, "Use of video changed our meetings because if you see

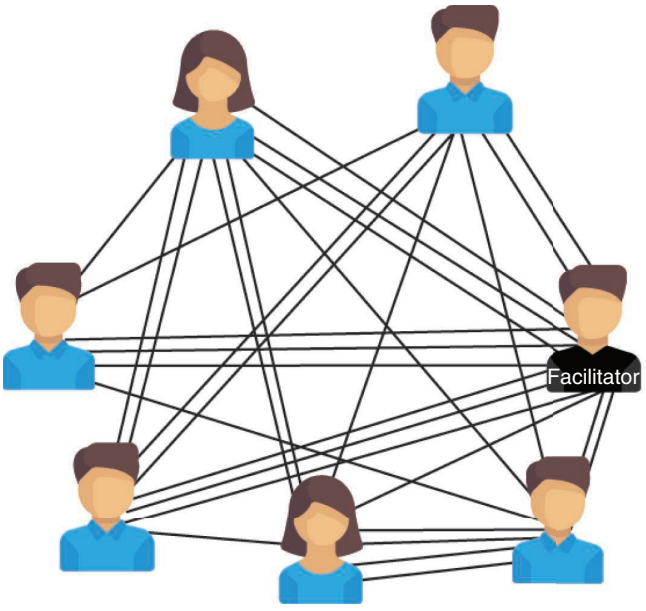

FIGURE 3. A conversation chart for a discussion meeting.

people's reactions and their gestures, you immediately know whether they are listening or are bored, and whether they don't understand what you said." Additionally, using a large video screen at the right height for standing helped team members to acquire the feeling that everyone was a part of the same standing circle, which further increased team spirit and improved the communication.

When Are the Meetings Held?

Most agile teams conduct their daily stand-ups in the morning. A few team members told us that it is good to start the day by gathering the team, to hear what people will be working on and whether it will face any obstacles. Nevertheless, in many cases, we found that an early start time reduced the perceived value of the meeting and the effectiveness of the team. We observed that early meetings were often delayed by a few minutes because some participants arrived late for work, which made the time spent in the meeting more than 15 min for those who arrived on time.

More importantly, many team members considered the meeting to be a disruptive interruption, particularly during programming activities. One developer said, "I find the meeting useful, but the interruption it causes is disturbing. I have to detach myself from what I am working on, and when I get back it takes some time to start again where I left off." Therefore, even when arriving early, some team members waited until after the meeting to start working on challenging issues, and instead spent their time before the meeting on tasks that did not require concentration. This behavior is natural because other research has found that most developers need more than 15 min to resume work after an interruption.[12] Starting early in the morning (e.g., 8:30 a.m.) or late in the afternoon (e.g., 3:30 p.m.) was also seen as undermining the much-valued flexibility regarding work hours in the companies.

Participants said that starting later in the day (e.g., 10:00 a.m.) was also problematic because there was not enough time between the daily meeting and lunch to do anything useful. Many staff members felt that starting right before lunch was a better option because then there was no added interruption and the team could go and eat lunch together.

Who Benefits Most From the Meetings?

The way work was assigned and coordinated affected the dynamics of the stand-ups. In some teams, there were few interdependencies between different members' tasks, mainly because they had specialized roles and low knowledge redundancy; that is, team members had expertise in different technical areas and worked on separate modules. Therefore, what one team member worked on was often uninteresting or irrelevant for members other than the Scrum master or technical lead, so staff usually did not pay attention when others were talking.

When there is little interdependence among team members, there is less need for mutual adjustment and problem-focused communication and, as a result, most of the time is instead spent on reporting. Particularly, there is less interdependence in larger teams because everyone's work depends on only a fraction of others' work. Accordingly, we observed that larger teams had more reporting meetings. NorBank had

the largest teams, partly as a consequence of having BizDevOps teams, which include people from business, development, and operations. Discussing problems among such a variety of roles would have required too much time and would not have been relevant for all attendees.

When more people attend a meeting, there is less time available for each person to be active, so team members are generally less satisfied with large meetings. Consistent with what we found in a recent survey of stand-ups, there was a significant decrease in meeting satisfaction when the team comprised more than 12 members.[2] Furthermore, we found that experience played an important role with respect to who benefitted from the meeting. Junior team members were generally more positive than senior ones. These senior team members often perceived the daily stand-ups as having little personal value because they already knew what was going on and did not receive any new information in the meeting.

Moreover, the positive attitude toward problem-solving discussions is more relevant to junior staff because the problems they encounter are easier to solve in a daily stand-up format. Senior team members often work on more complex tasks in which the problems require more expertise and time to resolve than a daily stand-up allows. Also, when an experienced team member works on a complex task, there might be little progress for some days, and team members do not see repeating what is said in the previous meeting as valuable. One senior developer said, "The intervals are too short to report on; often you feel that you did not have much progress from the day before."

We have also seen that senior team members attend more meetings, both internal to the team and external, in addition to the daily stand-ups. The daily meeting is then considered to be an additional daily interruption, which may reduce the well-being of senior team members because a higher meeting load negatively affects fatigue levels and subjective workload.[11]

Recommendations

In our longitudinal study, we made a set of initial recommendations on the basis of our early interviews and observations. Several teams then tried these guidelines. While there is no clear recipe for conducting a successful daily stand-up that fits all companies, we make the following recommendations based on all our data collection and analysis.

Omit Question 1

To avoid the daily stand-up becoming a status-reporting meeting that does not realize its potential benefits, we suggest that team members do not report what has been accomplished since the last meeting (Q1). Eliminating this question will reduce status-reporting and self-justification, in which participants explain in detail what they have done and why they have not achieved as much as expected. More time will then be available to discuss and solve problems.

While it is valuable for team members to know the status and progress of the other team members' tasks, such information can be shared more efficiently by other means. For example, some teams display their status on a visual board, and some use a chatbot in Slack (an electronic communication tool) to collect status information from everyone, which is later posted to the whole team.

Share Facilitation Responsibility

Facilitators need to consciously allot equal time to all participants. It is easy (often unconsciously) to allow some people to talk more than others because the information from these people may be of particular interest to the facilitators. Furthermore, facilitators must be aware of how the allocation of who speaks during the meeting affects the conversation flow. Try both an approach in which the tasks on the board are discussed and a round-robin approach in which the team members know, without interruptions, who will speak next.

An alternative to having one appointed facilitator is to circulate the facilitator role among a set of team members, which is what some of the observed teams did. Leadership that is broadly distributed instead of centralized in the hands of a single individual is known as *shared leadership*. This form of leadership can empower a team and foster both the task-related and social dimensions of a group's function, such as trust, cohesion, and commitment. Because new facilitators may influence how the meeting is carried out, sharing the facilitator role will give the team opportunities to reflect upon different ways to conduct the daily stand-ups. If the facilitator is not the same person as the team leader, it is less likely that the daily stand-up becomes a status-reporting meeting.

Meet Right Before Lunch

If a team decides to conduct daily stand-ups, the meetings must fit the rhythm of the team members' days and weeks. It is important to find the least disruptive time to reduce the potential for fragmented work. In many cases, conducting the meeting early in the morning has some drawbacks. Right

Table 3. Recommendations for daily stand-up meetings.

Stop asking, "What did you do yesterday?"
• Reduce the time spent on status reporting and self-justification. • Focus on future work, particularly considering dependencies and obstacles. • Spend time discussing and solving problems as well as making quick decisions.
Optimize the communication pattern.
• Share the leadership to increase joint responsibility, such as by rotating the facilitator role. • Team members should communicate with each other and not report to the facilitator.
Find the least disruptive time.
• Consider scheduling the meeting right before lunch to decrease the number of interruptions.
Find the frequency that offers the most value.
• In a team that communicates well, stop meeting daily if three or four times a week is sufficient. • In a large team, some members do not need to meet as frequently as others, depending on the interdependencies among tasks and members.

before lunch may be a better time, as it will merge two interruptions. Many teams tried this, and one respondent commented on not having the meeting in the morning: "There are many advantages with the new time; one is that people have different preferences when it comes to what time to arrive at work. When having the stand-up late, all of us are able to have moments of flow before the meeting. It is also much easier to remember what you are working on. I know the trend has spread to other teams."

Furthermore, hungry team members are more prone to end the meeting on time and, in addition, they are more likely to have lunch together, which will stimulate team cohesion. They can also continue discussions on their way to lunch and during lunch, if needed.

Adapt the Frequency

The daily stand-up meeting is, as the name indicates, intended to be conducted daily. However, we found that not all teams benefit from meeting five times a week. For example, small, collocated teams that have a high degree of informal communication may not need a daily meeting. Furthermore, in a large team there will be less need for mutual adjustment among all of the team members. Therefore, it might be better to split the team into smaller groups and hold separate meetings for staff that works together on a daily basis, with less frequent meetings for the whole team. Alternatively, team members may post answers to the three Scrum questions electronically on a daily basis, and then have a less frequent physical meeting in which they can concentrate on problem-focused communication.

We have worked with many teams in different companies who have experimented with our recommendations (Table 3). They have stopped asking Q1 ("What did you do yesterday?") and instead started sharing the facilitation responsibility to optimize communication patterns and changed the frequency and time of meetings. In particular, setting the correct meeting time (right before lunch) has been widely adopted and appreciated. Furthermore, we have yet to meet any teams who have reinstated Q1 after a trial period without it. Members appreciate focusing on the work that is ahead of them and being allowed to discuss problems and solutions. One recommendation that has been adopted less frequently is sharing the facilitator role, probably because Scrum masters and team leaders feel that it is their responsibility to facilitate the meetings.

When we help companies, we always advise them to collect information about the daily meeting, to gain insights regarding their team and how to enhance its productivity. Having a person from another team draw a conversation chart (see Figure 2) is a technique that may provide insight as well as a valuable basis for discussions about daily stand-ups. Relevant questions to discuss are as follows.

- "How can all team members benefit from the meeting?"
- "Are people talking to each other or to the facilitator?"
- "Is it a reporting or discussion meeting?"
- "Who should facilitate the meeting?"
- "Is every team member talking approximately the same amount of time?"
- "Should the meeting focus on people or tasks?"

We advise all agile teams to try to improve their meetings by experimenting with breaking the rules that they follow. Taking responsibility for improving the meetings will make the

team more self-managed. An agile software team should continuously inspect and adapt the daily stand-up meeting to fulfil the team's needs. Some of the adjustments might be about challenging the old mindset, such as conducting the meetings daily and in the morning, and discussing only the three Scrum questions. Teams that make appropriate adjustments become more productive.

References

VersionOne, "VersionOne 13th Annual State of Agile Report," May 2019. [Online]. Available: https://explore.versionone.com/state-of-agile/13th-annual-state-of-agile-report

1. V. Stray, N. B. Moe, and G. R. Bergersen, "Are daily stand-up meetings valuable? A survey of developers in software teams," in *Int. Conf. Agile Processes in Software Engineering and Extreme Programming*, 2017, pp. 274–281.
2. H. Mintzberg, *Mintzberg on Management: Inside our Strange World of Organizations*. New York: Free Press, 1989.
3. J. A. Allen, N. Lehmann-Willenbrock, and S. J. Sands, "Meetings as a positive boost? How and when meeting satisfaction impacts employee empowerment," *J. Bus. Res.*, vol. 69, no. 10, pp. 4340–4347, Oct. 2016.
4. K. Schwaber and J. Sutherland, "The Scrum guide," Nov. 2017, pp. 1–19. [Online]. Available: http://scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf
5. V. Stray, D. I. K. Sjøberg, and T. Dybå, "The daily stand-up meeting: A grounded theory study," *J. Syst. Softw.*, vol. 114, pp. 101–124, Apr. 2016.
6. H. Nyrud and V. Stray, "Inter-team coordination mechanisms in large-scale agile," in *Proc. ACM XP 2017 Scientific Workshops*, pp. 1–6. doi: 10.1145/3120459.3120476.
7. S. Kauffeld and N. Lehmann-Willenbrock, "Meetings matter: Effects of team meetings on team and organisational success," *Small Group Res.*, vol. 43, no. 2, pp. 130–158, Mar. 2012.
8. G. Klein, "Naturalistic decision making," *Hum. Factors*, vol. 50, no. 3, pp. 456–460, June 2008.
9. C. Duhigg, "What Google learned from its quest to build the perfect team," *New York Times*, Feb. 25, 2016. [Online]. Available: https://www.nytimes.com/2016/02/28/magazine/what-google-learned-from-its-quest-to-build-the-perfect-team.html
10. A. Luong and S. G. Rogelberg, "Meetings and more meetings: The relationship between meeting load and the daily well-being of employees," *Group Dyn.*, vol. 9, no. 1, pp. 58–67, 2005.
11. C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," *Software Quality J.*, vol. 19, no. 1, pp. 5–34, Aug. 2011.

ABOUT THE AUTHORS

VIKTORIA STRAY is an associate professor in the Department of Informatics, University of Oslo, Sweden, and is a researcher at SINTEF. Her research interests include agile methods, global software engineering, teamwork, coordination, and large-scale development. Stray received a Ph.D. in software engineering from the University of Oslo. She is a Member of the IEEE and ACM. Contact her at stray@ifi.uio.no.

NILS BREDE MOE is a senior scientist at SINTEF and an adjunct professor at the Blekinge Institute of Technology. His research interests relate to organisational, sociotechnical, and global/distributed aspects. Moe received a Ph.D. in computer science from the Norwegian University of Science and Technology, Trondheim. Contact him at nilsm@sintef.no.

DAG I.K. SJØBERG is a professor at the University of Oslo. His research interests are the software lifecycle, including agile and lean development processes, and empirical research methods in software engineering. Sjøberg received a Ph.D. in computing science from the University of Glasgow. He is a Member of the IEEE and ACM. Contact him at dagsj@ifi.uio.no.

What Do We Know about Defect Detection Methods?

Per Runeson, Carina Andersson, and Thomas Thelin, *Lund University*
Anneliese Andrews, *University of Denver*
Tomas Berling, *Ericsson Microwave Systems*

A survey of defect detection studies comparing inspection and testing techniques yields practical recommendations: use inspections for requirements and design defects, and use testing for code.

Detecting defects in software product development requires serious effort, so it's important to use the most efficient and effective methods. *Evidence-based software engineering* can help software practitioners decide which methods to use and for what purpose.[1] EBSE involves defining relevant questions, surveying and appraising available empirical evidence, and integrating and evaluating new practices in the target environment.

This article helps define questions regarding defect detection techniques and presents a survey of empirical studies on testing and inspection techniques. We then interpret the findings in terms of practical use.

The term *defect* always relates to one or more underlying faults in an artifact such as code. In the context of this article, defects map to single faults. Thus, we use the terms *defect* and *fault* interchangeably, as have many of the authors whose work we refer to.

What influences the choice of method?

The choice of defect detection method depends on factors such as the artifacts, the types of defects they contain, who's doing the detection, how it's done, for what purpose, and in which activities. Factors also include which criteria govern the evaluation.

These factors show that many variations must be taken into account. When you search the evidence for the the pros and cons of using some defect detection method, you must choose specific levels of these factors to guide the appraisal of empirical evidence.

Artifact

Which artifact are you assessing? Requirements? Design? Code? Testing requires an executable representation—that is, code—while inspection can apply to any artifact. Most experiments, by necessity, use small, artificial artifacts. Using industrial artifacts improves the study's generalizability but also leads to more confounding factors.

Types of defects

What types of defects do the artifacts contain? There's a big difference between gram-

0740-7459/06/$20.00 © 2006 IEEE

matical errors in code and missing requirements in a requirements specification. Testing and inspection methods might be better or worse for different types of defects. In this article, we first classify defects on the basis of their origin: *requirements*, *design*, or *code*. Second, many empirical studies categorize defects along two dimensions, as Victor Basili and Richard Selby proposed.[2] The first dimension classifies defects as either an *omission* (something is missing) or a *commission* (something is incorrect), while the second dimension defines defect classes according to their technical content.[2,3] Other classifications focus on the defect's severity in terms of its impact for the user: *unimportant*, *important*, or *crucial*.[4]

Actor

Who's the reviewer or tester? A freshman student in an experimental setting? An experienced software engineer in industry? What's the incentive for doing a good job in an empirical study, which isn't part of a real development project?[5] To confound matters further, experienced students have been observed to outperform recently graduated engineers.[6]

Technique

Which techniques are you using for inspection and testing respectively? Because there are many techniques for each, we refer to inspection and testing as *families* of verification techniques. For testing, we distinguish between structural (white box) and functional (black box) testing.

Purpose

What's the purpose of the inspection and testing activity? The activity might contribute to *validation*—that is, assuring that the correct system is developed—or to *verification*—that is, assuring that the system meets its specifications—or to both. The primary goal of both inspection and testing is to find defects, but more specifically, is it to *detect* a defect's presence, for later isolation by someone else, or is it to *isolate* the underlying fault? Testing reveals a defect's presence through its manifestation as a failure during execution. Inspections point directly at the underlying fault.

Furthermore, there are secondary purposes for inspection and testing. Inspections might contribute to knowledge transfer, for example, whereas testing in a test-driven design setting produces test specifications that also might constitute a detailed design specification.[7]

Table 1

Defect origins and detection

	Defect detection activity					
Phase of defect origin	**Requirements inspection**	**Design inspection**	**Code inspection**	**Unit test**	**Function test**	**System test**
Requirements	1	2	2	2	2	1
Design	N/A	1	2	2	1	2
Coding	N/A	N/A	1	1	2	2

Defect detection activities

A defect might originate in one development stage and be detected in the same or a later stage. For instance, a missing interface in a design specification could propagate to the coding stage, resulting in missing functionality in the code. You might detect this design defect during a design inspection, code inspection, unit test, function test, or system test. Because defect detection focuses on abstraction levels, we consider that *primary* defect detection activities are at the same level of abstraction and *secondary* defect detection activities are at a different level. For example, for design defects, design inspection and functional testing are primary activities, and code inspection and unit testing are *secondary* defect detection activities.

Table 1 illustrates primary and secondary defect detection activities for defects originating from requirements, design, and coding (listed in column 1). Cells numbered 1 represent primary defect detection activities, and cells numbered 2 classify secondary activities.

Evaluation criteria

What are the criteria for selecting techniques? Should you choose the most effective or the most efficient method? *Efficiency* in this context means the number of defects found per time unit spent on verification, and *effectiveness* means the share of the existing defects found.

Survey of empirical studies

Available sources of empirical evidence are experiments and case studies. Experiments provide good internal validity.[8] They can manipulate the investigation's context and control many parameters. However, achieving high ex-

Table 2

Surveyed empirical studies on inspection versus testing

Study and year	Type	Technique	Artifact	Number of defects of certain types*	Actors	Purpose	Result†
Hetzel,[17] 1976	Experiment	Functional test vs. structural test and inspection	Code modules (PL/1) 3 programs, 64–170 statements each	–	39 students	Detection	Effectiveness: testing > inspection
Myers,[18] 1978	Experiment		Code (PL/1) 63 statements	15 defects	59 professionals	Detection	Effectiveness: inspection = testing; complementary, but different for some classes of defects
Basili and Selby,[2] 1987	Experiment		Code (Fortran, Simpl-T) 169, 145, 147, and 365 LOC)	4 programs, total 34 defects Om/com: 0/2 ini, 4/4 cmp, 2/5 cnt, 2/11 int, 2/1 d, 0/1 cos	32 professionals + 42 advanced students	Detection	Effectiveness and efficiency depend on software type
Kamsties and Lott,[19] 1995	Experiment (replication)		Code (C) 211, 248, and 230 LOC	3 programs, 6/9/7 defects 0/2/0 ini, 0/0/1 cmp, 3/2/3 cnt, 0/3/0 int, 2/1/2 d, 1/1/1 cos	27 and 15 students in replications 1 and 2, respectively	Detection and isolation	Effectiveness: no significant difference Efficiency: testing > inspection
Roper et al.[20] 1997	Experiment (replication)		Code[19]	3 programs, 8/9/8 defects	47 students	Detection	Effectiveness: no significant difference Efficiency: testing > inspection Combination better
Laitenberger,[11] 1998	Experiment	Inspection, then structural testing	Code (C) 262 LOC	13 defects, 2 ini, 4 cnt, 2 cmp, 2 cos, 3 d	20 students	Detection	Not complementary
So et al.,[12] 2002	Experiment	Voting, testing, self-checks, code reading, data-flow analysis, Fagan inspection	Code (Pascal) 8 programs, 1,201–2,414 LOC each	Experiment 1: 270 major faults Experiment 2: 179 major faults	26 and 15 students in experiments 1 and 2, respectively	Detection	Effectiveness: voting > testing > inspection; complementary Efficiency: testing < inspection
Runeson and Andrews,[13] 2003	Experiment	Inspection vs. structural testing	Code (C) 190 and 208 LOC each	9 in each version 0/1 ini, 5/0 cnt, 1/0 int, 0/4 cmp, 1/2 cos, 1/2 d	30 students	Detection and isolation	For detection: testing > inspection For isolation: inspection > testing
Juristo and Vegas,[14] 2003	Experiment (replication)	Functional test vs. structural testing and inspection	Code[19] + one new program	4 programs, 9 defects each Om/com: 1/2 ini, 2/2 cnt, 1/1 cos	196 students	Detection	Effectiveness: different for different fault types Testing > inspection
			Code[19]	3 programs, 7 defects each Om/com: 1/1 ini, 1/1 cnt, 0/1 cmp, 1/1 cos	46 students		
Andersson et al.,[15] 2003	Experiment	Inspection vs. functional testing	Design (text) 9 pages, 2,300 words	2 design document versions, 13/14 defects, 3/4 crucial, 5/6 important, 5/4 unimportant	51 students	Detection	Inspection > testing Different faults found (one version)
Conradi et al.,[16] 1999	Case study	Inspection, desk check, test	Design (code)	1,502 and 6,300 in two projects, respectively	Professionals		Inspection > testing
Berling and Thelin,[3] 2003	Case study	Inspection, unit test, subsystem and system test	Requirements, design, code	244 (45 likely to propagate to code)	Professionals	Isolation	Little overlap between testing and inspection

** om = omission, com = commission, ini = initialization, cmp = computation, cnt = control, int = interface, d = data, cos = cosmetic*
† "x > y" means x is better than y

Table 3

Average values of effectiveness and efficiency for defect detection

		Study	Inspection effectiveness (%)*,†	Inspection efficiency†,‡	Testing effectiveness*,§	Testing efficiency‡,§	Different faults found
Experiments	Code	Hetzel[17]	37.3	–	47.7; 46.7	–	–
		Myers[18]	38.0	0.8	30.0; 36.0	1.62; 2.07	Yes
		Basili and Selby[2]	54.1	Dependent on software type	54.6; 41.2	–	Yes
		Kamsties and Lott[19]	43.5 50.3	2.11 1.52	47.5; 47.4 60.7; 52.8	4.69; 2.92 3.07; 1.92	Partly (for some types)
		Roper et al.[20]	32.1	1.06	55.2; 57.5	2.47; 2.20	Yes
		Laitenberger[11]	38	–	9#	–	No
		So et al.[12]	17.9, 34.6	0.16; 0.26	43.0	0.034	Yes
		Runeson and Andrews[13]	27.5	1.49	37.5	1.8	Yes
		Juristo and Vegas[14]	20.0 –	– –	37.7; 35.5 75.8; 71.4	– –	Partly (for some types)
	Design	Andersson et al.[15]	53.5	5.05	41.8	2.78	Yes for one version, no for the other
Case studies		Conradi et al.[16]	–	0.82	–	0.013	–
		Berling and Thelin[3]	86.5 (estimated)	0.68 (0.13)	80	0.10	Yes

* *Percent of the artifact's defects that are detected.*
† *Single entries involve code reading; multiple entries in one cell are reported in this order: code reading, Fagan inspection.*
‡ *Detected defects per hour.*
§ *Single entries involve functional testing; multiple entries in one cell are reported in this order: functional test, structural test.*
# *Testing is conducted in sequence after the inspection.*

ternal validity in experiments is difficult. Case studies have a more realistic context but by nature are subject to confounding factors. A case study's generalizability depends on how similar it is to an actual situation. No single experiment or case study can provide a complete answer, but a collection of studies can contribute to understanding a phenomenon.

Many empirical studies have investigated defect detection techniques, inspections, and testing in isolation. Aybüke Aurum, Håkan Petersson, and Claes Wohlin summarize 25 years of empirical research on software inspections, including more than 30 studies that investigated different reading techniques, team sizes, meeting gains, and so on.[9] Similarly, Natalia Juristo, Ana Moreno, and Sira Vegas summarize 25 years of empirical research on software testing based on more than 20 studies.[10] They compare testing within and across so-called "families" of techniques. Despite the large number of studies, they conclude that our collective knowledge of testing techniques is limited.

We know even less about the relationship between inspection and testing and the effects of combining them. Oliver Laitenberger summarizes five experimental studies comparing inspection and testing methods as well as results from one experiment he conducted that combined inspection and testing.[11] Later, four experimental studies[12–15] and two case studies[3,16] added to the knowledge on inspections versus testing.

We used these 12 studies—nine experiments on code defects, one experiment on design defects, and two case studies on a comprehensive defect detection process—to search for evidence that would help developers choose between defect detection methods. We followed the procedures in the EBSE framework that Tore Dybå, Barbara Kitchenham, and Magne Jørgensen recently presented.[1] Table 2 summarizes empirical studies involving both inspection and testing, and table 3 presents data from the studies. We report the experiments' outcomes and relate them to the results of the case studies conducted. The issues covered include

- requirements defects,
- design defects,
- code defects,
- different defect types, and
- efficiency versus effectiveness.

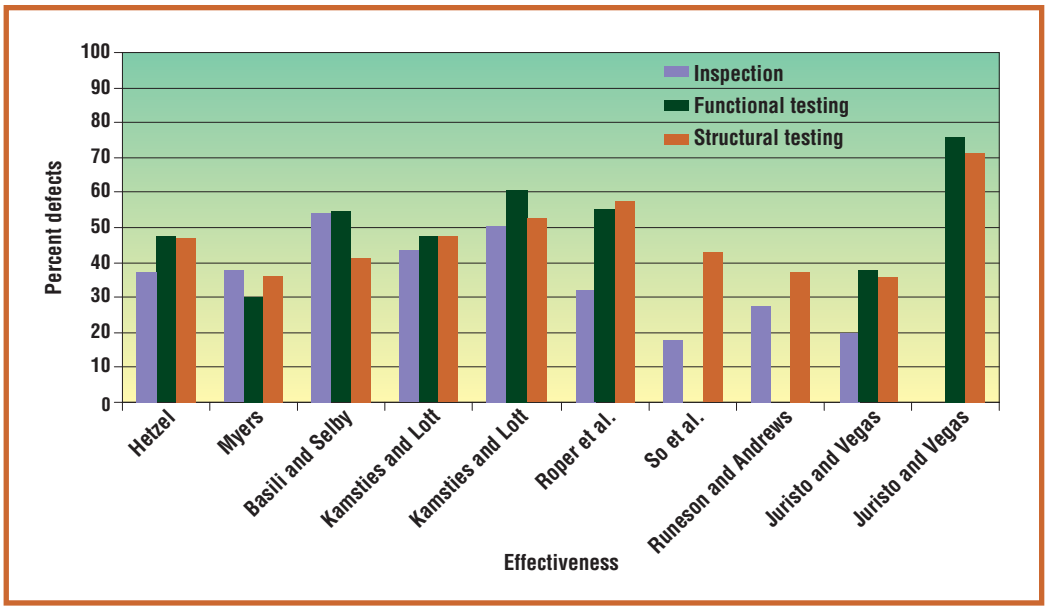

Figure 1. Average effectiveness of techniques for code defect detection.

Requirements defects

Several experiments compared different requirements inspection methods,[9] but none compared one to a testing method. The choice between requirements inspection and system testing is quite obvious and needs no experiments: spending effort up front to establish a good set of requirements is more efficient than developing a system on the basis of incorrect requirements and then reworking it. The case study by Tomas Berling and Thomas Thelin supports this assumption.[3]

Design defects

The key question regarding design defects is whether inspecting design documents or testing the implemented function is more efficient. Carina Andersson and her colleagues addressed this issue in one experiment.[15] They observed defect detection in a design specification and in a log from function test execution; they used experimental groups that varied greatly.

Inspections were significantly more effective and efficient than testing. The study's participants found more than half of the defects (53.5 percent) during inspection and fewer (41.8 percent) during testing (see table 3). Efficiency was five defects per hour for inspection and fewer than three per hour for testing.

The analysis didn't take into account rework costs. A defect detected during design inspection is much cheaper to correct than one detected in function testing, because the latter involves reworking the design and code. This implies even greater efficiency for design inspections compared to functional testing.

Berling and Thelin confirmed these results in their industrial case study, including five incremental project iterations.[3] Inspections detected on average 0.68 defects per hour, while testing detected 0.10 defects per hour (see table 3). For the fraction of faults that they estimated to propagate into code, the rate was 0.13 defects per hour. This case study also reports slightly higher effectiveness for inspections, but the differences are small. Reidar Conradi, Amarjit Singh Marjara, and Børge Skåtevik had similar results in their case study: they reported 0.82 defects per hour in design inspection and only 0.013 defects per hour in function testing.[16] So, the empirical data support design inspections as a more efficient means for detecting design defects.

Code defects

Most of the experiments investigated code defects. However, they revealed no clear answer as to whether code inspection or testing—functional or structural—is preferable.

James Miller[21] attempted to analyze the results from the first five studies (see table 2), but variation among the studies was too large to apply meta-analysis. So, we simply rank the techniques with respect to their effectiveness and efficiency. Figure 1 presents effectiveness—that is, the percentage of total defects that the different techniques found.

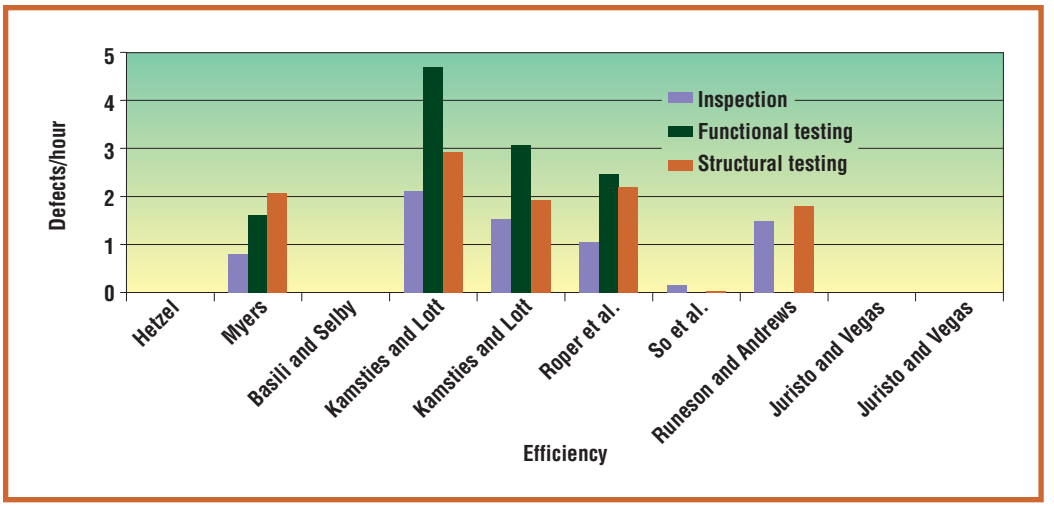

Figure 2. Average efficiency of techniques for code defect detection.

No technique emerges as a clear winner or loser. All three techniques are ranked most effective in at least one study *and* least effective in at least one study. However, except in one case, all researchers who compared the three techniques found the difference between the average effectiveness of the first and second techniques to be smaller than between the second and third. The data doesn't support a scientific conclusion as to which technique is superior, but from a practical perspective it seems that testing is more effective than code inspections.

Figure 2 presents the efficiency ranking—that is, the number of defects found per time unit—for those studies reporting efficiency data. (For the others, the space is left blank.) The efficiency rankings differ from the effectiveness rankings. Glenford Myers' study[18] ranked inspection most effective and least efficient, while a study by Sun Sup So and his colleagues[12] ranked testing most effective and least efficient. The study by Marc Roper, Murray Wood, and James Miller[20] ranked functional testing most effective and structural testing most efficient, but the differences are small.

The studies by Erik Kamsties and Christopher Lott[19] and by Per Runeson and Anneliese Andrews[13] distinguish between *detection* and *isolation* of defects. In the former study, this moved inspections to the second rank. In the latter study, this distinction didn't change the rank between the two studied techniques. The study by Roper, Wood, and Miller[20] also distinguishes between detection and isolation, which improved inspection's performance, but it still ranked third.

We caution that these studies were conducted in isolation and thus didn't take into account secondary issues such as the information-spreading effects of inspections, the value of unit-test automation, cost, and intrinsic values of a test suite or test-driven design.

Different defect types

The studies vary widely in the types of defects the artifacts contain. If that's indeed an important factor, then the absence or presence of different types of defects might affect efficiency and effectiveness. Table 2, column 5 shows the types of defects in each study, if they were known.

Comparing the studies is difficult. First, only five of the studies report defects by the same type scheme and can be used to investigate whether the differences depend on the fault type, not only on the technique. Second, the frequencies of the different defect types vary widely among the remaining studies. Third, only a fraction of defects are found; so, while we might know each artifact's defect type frequencies, we might not know the defect type frequencies of the defects that were found. Given the low proportion of defects found, this makes cross-study comparison difficult, if not impossible. Fourth, classification schemes involve subjective judgment that can confuse classification results.

Few studies investigate the relationship between defect types and inspection versus testing. One study found a statistically significant difference between defect types discovered by inspection and by testing.[13] This implies that the techniques' performance is sensitive to the

The studies indicate pros and cons for the two families of defect detection techniques, but no clear winner.

defect type. The same study indicates that subjects tend to prefer testing over inspection; this is a piece of qualitative information that should be taken into account.

Kamsties and Lott found two significant differences in performance between defect types; inspections gave worse results in finding omissions type faults and control class faults.[19] Juristo and Vegas replicated this experiment twice.[7] In the first replication, cosmetic faults were most difficult to find, irrespective of technique. However, defect type didn't affect code inspection, contradicting an earlier study by Basili and Selby.[2] Functional testing performed better than structural testing for faults of omission, and testing techniques performed better than inspection. Basili and Selby couldn't detect a clear pattern as to which technique detects which defect types more easily.

In summary, the jury is still out as to the effect of defect types on the performance of inspection versus testing.

Effectiveness and efficiency

Absolute levels of effectiveness of defect detection techniques are remarkably low. In all but one experimental study, the subjects found only 25 to 50 percent of the defects on average during inspection, and slightly more during testing (30 to 60 percent). This means that on average, more than half the defects remain! The Berling and Thelin case study reported 86.5 percent effectiveness for inspections and 80 percent effectiveness for testing.[3] However, these are based on an estimated number of defects that the technique could possibly find, not on the total number of defects in the documents.

The experimental studies found 1 to 2.5 defects per hour. The size of the artifacts was at most a few hundred lines of code. This is small from an industrial perspective, where professionals deal with more complex artifacts, struggle with more communication overhead, and so on. Consequently, the efficiency in the industrial case studies is lower: 0.01 to 0.82 defects per hour. The variation is also much larger, which might be due to different company measures of efficiency.[3]

The practical implication of the primary defect detection methods' low effectiveness and efficiency values is that secondary detection methods might play a larger role than the surveyed empirical studies concluded.

Validity of the empirical results

The studies we summarized indicate pros and cons for the two families of defect detection techniques, but no clear winner. But how valid are the results?

We can analyze threats to the validity of empirical studies along four dimensions: *internal*, *conclusion*, *construct*, and *external* validity.[8] From a practitioner point of view, external validity is the most important.[22] From a researcher point of view, internal validity is traditionally considered the key to successful research. However, it's important to balance all dimensions of validity to achieve trustworthy empirical studies.

The surveyed experimental studies' internal validity seems quite high. Established researchers conducted the studies, mostly in student environments with experienced students. On the other hand, the inconclusive results indicate the presence of factors that weren't under experimental control. By using established analysis methods, the studies also seem to limit conclusion validity threats.

For case studies, on the other hand, high internal and conclusion validity is harder to achieve. Confounding factors might interact with the factor under study and threaten internal validity. Because data collection is often set up for purposes other than empirical study, conclusion validity might be threatened.

Both experiments and case studies potentially threaten construct validity, because they use different instantiations of defect detection methods to represent test and inspection techniques. The studies listed in table 2 present large variations within as well as between the families of techniques, both in experiments and case studies.

The major threat for experiments is external validity, because they're conducted on small artifacts, mostly with students as subjects. The case studies suffer less from external threats, although there's a risk that the conditions specific to a particular case greatly influence the outcome.

What's the answer?

Our analysis of existing empirical studies showed no clear-cut answer to the question of which defect detection method to choose. However, from a practical viewpoint, these findings tend to be true:

- For *requirements defects*, no empirical evidence exists at all, but the fact that costs for requirements inspections are low compared to implementing incorrect requirements indicates that reviewers should look for requirements defects through inspection.
- For *design specification defects*, the case studies and one experiment indicate that inspections are both more efficient and more effective than functional testing.
- For *code*, functional or structural testing is ranked more effective or efficient than inspection in most studies. Some studies conclude that testing and inspection find different kinds of defects, so they're complementary. Results differ when studying fault isolation and not just defect detection.
- *Verification*'s effectiveness is low; reviewers find only 25 to 50 percent of an artifact's defects using inspection, and testers find 30 to 60 percent using testing. This makes secondary defect detection important. The efficiency is in the magnitude of 1 to 2.5 defects per hour spent on inspection or testing.

Several studies call for replication and further work on this topic. Factors seem to be at work that aren't measured or controlled but that nonetheless influence defect detection methods' performance. It might be useful to investigate "softer" factors such as motivation and satisfaction[5] and in particular to apply methods in practice, monitor them, and follow up.

Practitioners can use our empirical results in two ways: either as a guideline for a high-level strategy of defect detection or in a full EBSE fashion,[1] finding an answer to a specific question.

In strategy definitions, the summary of the empirical studies can act as a general guide for which defect detection methods to use for different purposes and at different stages of development. Our findings aren't novel or strictly empirically based. However, defining such a strategy would benefit many organizations and projects. Making a defect detection strategy explicit helps communicate values internally, making project members aware of how their work fits into the complete picture. The strategy could also be a starting point for EBSE-based investigations of more specific trade-offs between different methods.

Table 4

Example definition of factor levels

Factor	Example
Artifact	Code modules
Types of defects	Omitted, crucial, and important interfaces
Actor	Novice testers and programmers
Technique	Any feasible technique
Purpose	Design verification
Defect detection activity	Code inspection and unit testing
Evaluation criteria	Effectiveness

To precisely specify which defect detection technique to use, you can apply a full EBSE cycle,[1] along with the variation factors we outlined earlier. Table 4 shows examples of variation factors that are important in searching for a feasible defect detection technique.

These factors help frame the general question into a more precise one: "Which defect detection technique should we use to detect as many critical and important omission defects in interfaces as possible, in code inspection and unit testing, conducted by novice testers and programmers?" Then you can survey the studies in table 2 in detail as a basis for your decision. You might choose a combination of structural and functional testing, for example. Then to anchor the decision, you'd take into account the organization's previous experience. As a final step in the cycle, you must monitor and evaluate the selected technique's performance.

Furthermore, if we feed our industrial evaluations back to the research community, we can increase the body of knowledge about these methods' usefulness.

References

1. T. Dybå, B.A. Kitchenham, and M. Jørgensen, "Evidence-Based Software Engineering for Practitioners," *IEEE Software*, vol. 22, no. 1, 2005, pp. 58–65.
2. V.R. Basili and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.*, Dec. 1987, pp. 1278–1296.
3. T. Berling and T. Thelin, "An Industrial Case Study of the Verification and Validation Activities," *Proc. 9th Int'l Software Metrics Symp.*, IEEE CS Press, 2003, pp. 226–238.

About the Authors

Per Runeson is a professor of software engineering at Lund University and leader of the Software Engineering Research Group. He also holds a senior researcher position funded by the Swedish Research Council. His research interests include software development methods and processes, particularly for verification and validation. He received his PhD in software engineering from Lund University. He serves on the editorial boards of *Empirical Software Engineering Journal* and the *Journal of the Association of Software Testing*. Contact him at Dept. of Communication Systems, Lund Univ., Box 118, SE-22100 Lund, Sweden; per.runeson@telecom.lth.se.

Anneliese Andrews is a professor in and chair of the University of Denver's Department of Computer Science. Her research interests include software design, testing, and maintenance as well as quantitative approaches to software engineering data analysis. She received her PhD in computer science from Duke University. She serves on the editorial board of the *Empirical Software Engineering Journal*. Contact her at the Dept. of Computer Science, Univ. of Denver, 2360 S. Gaylord St., John Greene Hall, Rm. 100, Denver, CO 80208; andrews@cs.du.edu.

Carina Andersson is a licentiate in software engineering and a doctoral candidate at Lund University. Her research interests include software development verification and validation processes and software quality metrics and models. She received her MSc in engineering physics with industrial management from Lund University. Contact her at the Dept. of Communication Systems, Lund Univ., Box 118, SE-22100 Lund, Sweden; carina.andersson@telecom.lth.se.

Tomas Berling is a specialist at Ericsson Microwave Systems and a researcher at the IT University of Göteborg. His research and application interests include system verification and validation of complex software systems. He received his PhD in software engineering from Lund University. Contact him at Ericsson Microwave Systems, SE-431 84 Mölndal, Sweden; tomas.berling@ericsson.com.

Thomas Thelin is an associate professor of software engineering at Lund University. His research interests include empirical methods in software engineering; software quality; and verification and validation with emphasis on testing, inspections, and estimation methods. He received his PhD in software engineering from Lund University. Contact him at the Dept. of Communication Systems, Lund Univ., Box 118, SE-22100 Lund, Sweden; thomas.thelin@telecom.lth.se.

4.T. Thelin, P. Runeson, and C. Wohlin, "Prioritized Use Cases as a Vehicle for Software Inspections," *IEEE Software*, vol. 20, no. 4, 2003, pp. 30–33.

5. M. Höst, C. Wohlin, and T. Thelin, "Experimental Context Classification: Incentives and Experience of Subjects," *Proc. 27th Int'l Conf. Software Eng.*, ACM Press, 2005, pp. 470–478.
6. M. Höst, B. Regnell, and C. Wohlin, "Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment," *Empirical Software Eng.*, vol. 5, no. 3, 2000, pp. 201–214.
7. K. Beck, *Test Driven Development: By Example*, Addison-Wesley, 2002.
8. C. Wohlin et al., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.
9. A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-Art: Software Inspections after 25 Years," *Software Testing, Verification, and Reliability*, vol. 12, no. 3, 2002, pp. 133–154.
10. N. Juristo, A.M. Moreno, and S. Vegas, "Reviewing 25 Years of Testing Technique Experiments," *Empirical Software Eng.*, vol. 9, nos. 1–2, 2004, pp. 7–44.
11. O. Laitenberger, "Studying the Effects of Code Inspection and Structural Testing on Software Quality," *Proc. 9th Int'l Symp. Software Reliability Eng.*, IEEE CS Press, 1998, pp. 237–246.
12. S.S. So et al., "An Empirical Evaluation of Six Methods to Detect Faults in Software," *Software Testing, Verification, and Reliability*, vol. 12, no. 3, 2002, pp. 155–171.
13. P. Runeson and A. Andrews, "Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection," *Proc. 14th Int'l Symp. Software Reliability Eng.*, IEEE CS Press, 2003, pp. 3–13.
14. N. Juristo and S. Vegas, "Functional Testing, Structural Testing, and Code Reading: What Fault Type Do They Each Detect?" *Empirical Methods and Studies in Software Engineering*, R. Conradi and A.I. Wang, eds., Springer, 2003, pp. 208–232.
15. C. Andersson et al., "An Experimental Evaluation of Inspection and Testing for Detection of Design Faults," *Proc. IEEE/ACM Int'l Symp. Empirical Software Eng.*, IEEE CS Press, 2003, pp. 174–184.
16. R. Conradi, A.S. Marjara, and B. Skåtevik, "An Empirical Study of Inspection and Testing Data at Ericsson, Norway," *Proc. 24th NASA Software Eng. Workshop*, NASA, 1999; http://sel.gsfc.nasa.gov/website/sew/1999/topics/marjara_SEW99paper.pdf.
17. W.C. Hetzel, "An Experimental Analysis of Program Verification Methods," doctoral dissertation, Dept. of Computer Science, Univ. of North Carolina, Chapel Hill, 1976.
18. G.J. Myers, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections," *Comm. ACM*, Sept. 1978, pp. 760–768.
19. E. Kamsties and C.M. Lott, "An Empirical Evaluation of Three Defect-Detection Techniques," *Proc. 5th European Software Eng. Conf.*, LNCS 989, Springer, 1995, pp. 362–383.
20. M. Roper, M. Wood, and J. Miller, "An Empirical Evaluation of Defect Detection Techniques," *Information and Software Technology*, vol. 39, no. 11, 1997, pp. 763–775.
21. J. Miller, "Applying Meta-Analytical Procedures to Software Engineering Experiments," *J. Systems and Software*, vol. 54, no. 1, 2000, pp. 29–39.
22. A. Rainer, T. Hall, and N. Baddoo, "Persuading Developers to 'Buy into' Software Process Improvement: Local Opinion and Empirical Evidence," *Proc. Int'l Symp. Empirical Software Eng.*, IEEE CS Press, 2003, pp. 326–355.

Editor: **Eoin Woods**
Endava
eoin.woods@endava.com

Agility, Risk, and Uncertainty, Part 1: Designing an Agile Architecture

Michael Waterman

SOFTWARE ARCHITECTS IN agile environments face the dilemma of determining how much effort goes into architecting up front, before development starts. This is an issue that agile methodologies and frameworks don't address and that's becoming more critical as agile development gets used for a wider range of problems. This article is the first of two that discuss findings of my recent research based on the experiences of 44 agile practitioners,[1] to help shed light on the problem.

If architects (whether individual architects or the development team as a whole) spend too much time architecting up front, they'll struggle to be agile. This is partly because they're making architectural decisions too early—decisions that could later prove to be wrong or subject to change. It's also because this excessive effort will delay development and the opportunity for early customer feedback.

On the other hand, too little architecting results in ad hoc decisions that might not support the system's *architecturally significant requirements*—the requirements that impact the architecture. If the team is spending all its time fixing architectural problems that should have been thought through up front, it's not delivering customer value. And, just as if the team spent too much time architecting, it will struggle to be agile.

Somewhere in between is the optimal level of architecting that maximizes the team's architectural agility. How can we determine this optimal level? To answer that, we must realize that the optimal level depends highly on the team's context. This context includes whether the requirements are stable, the amount of technical risk, how early the customer wants to start using the system (perhaps earning revenue from it), the team's agility, how agile-friendly the team's environment is (such as business and administration or project stakeholders), and the team's architectural and technical experience.

The Agile Architecture

The context determines the team's ability to design an agile architecture—one that supports the team's agility and hence its ability to respond to change. I define agile architecture using two dimensions:

- It has been designed using an agile process.
- It's modifiable and tolerant of change.

The former means that the architecture isn't a static set of decisions—it evolves as the system evolves and as the team revisits architectural decisions. The latter means that the architecture can adapt more easily as the system changes.

Neither dimension by itself is necessarily sufficient for the architecture to be agile. An architecture can be designed using an agile process but not be particularly modifiable and tolerant of change. As Simon Brown said, "In my experience … teams are more focused on delivering functionality rather than looking after their architecture."[2] An architecture can be designed to be modifiable and tolerant of change, but if it's not designed using an agile process,

0740-7459/18/$33.00 © 2018 IEEE

Table 1. Tactics for designing agile architecture and their impacts.

Tactic	Impact on responsiveness to change	Reduces up-front design effort?
Keep designs simple	Increases modifiability	Yes
Prove the architecture with code iteratively	Increases modifiability	Yes
Use good design practices	Increases modifiability	No
Delay decision making	Increases tolerance to change	Yes
Plan for options	Increases tolerance to change	No

it can't fully welcome change (the second principle of the Agile Manifesto; agilemanifesto.org) as an intrinsic characteristic.

My research found that teams design agile architectures using five tactics:

- Keep designs simple.
- Prove the architecture with code iteratively.
- Use good design practices.
- Delay decision making.
- Plan for options.

Each tactic differently impacts the required responsiveness to change and the amount of up-front design. Table 1 summarizes these impacts.

Keeping designs simple is all about designing only for what's immediately required: no overengineering or gold-plating, no designing for what might be required (called YAGNI—You Ain't Gonna Need It—in Extreme Programming[3]). Simplicity reduces the detail in the design and the up-front effort. It increases the ease with which the design can be modified because there's less design to change. Of course, simplicity guarantees that the architecture will need to be updated later as the system grows and the requirements evolve. However, that's desirable because the decisions are delayed until the team has the best understanding about how to implement them.

Proving the architecture with code iteratively means simultaneous design and development, which is particularly useful if uncertainty exists about whether the architecture will meet the requirements. Simultaneous design and coding lets the team prove the architecture with real code, rather than through analysis, and refine the design if necessary. This lets the team come up with the simplest solution that works. So, like the simplicity tactic, this increases modifiability and reduces the up-front effort.

Good design practices to improve modifiability include separation of concerns (for example, service-oriented architecture, microservices, and encapsulated modules) and using quality management tools to ensure good design at the class level. Although good design practices should be an architect's goal no matter what development process he or she uses, they're especially important in agile development because they reduce the effort required to change the design. Good design practices don't reduce the up-front effort; indeed, they can increase it because they require extra discipline. However, the benefits they bring are worth that extra effort[2] because changes can be isolated and limited to small parts of the system.

Delaying decision making is related to simplicity: this tactic aims to reduce the impact of trying to predict requirements that aren't fully understood yet. Decisions are left as late as possible without delaying development, which allows the team as much time as possible to understand the requirements. Delaying decisions makes the architectural decisions more tolerant of change: fewer decisions must be revisited as requirements change, and the design requires less change.

Planning for options means the team makes decisions that retain flexibility and don't close off future options. The architecture is designed to be easily extended. Although planning for options is related to good design practices, it's less about general modularity and encapsulation and more about understanding what might need changing later and ensuring that the design isn't optimized so much that it makes those changes difficult. This tactic doesn't reduce up-front effort (and might increase it if, for example, the team uses extra levels of indirection to make changes to a particular technology layer easier).

The participants in my research used some or all of these five tactics to produce smaller, simpler, and more agile architecture designs that grow and evolve as the participants' understanding of the systems develops. Hence, these tactics support their teams' ability to respond to change.

In tension with minimizing design effort and deferring decisions as long as possible is the architect's need to reduce technical risk. Risk has a big impact on architects' ability to design an agile architecture and thus how much up-front design they do. I'll discuss this impact in Part 2.

ABOUT THE AUTHOR

MICHAEL WATERMAN is a solution architect at Specialised Architecture Services. Contact him at mike@specarc.co.nz.

References

1. M. Waterman, J. Noble, and G. Allen, "How Much Up-Front? A Grounded Theory of Agile Architecture," *Proc. 2015 Int'l Conf. Software Eng.* (ICSE 15), 2015, pp. 347–357.
2. S. Brown, *Software Architecture for Developers*, Leanpub, 2013.
3. M. Fowler, "Is Design Dead?," *Extreme Programming Examined*, G. Succi and M. Marchesi, eds., Addison-Wesley Longman, 2001, pp. 3–17.