Editors: Nancy Eickelmann ■ Motorola ■ nancy.eickelmann@motorola.com
Jane Huffman Hayes ■ University of Kentucky ■ hayes@cs.uky.edu

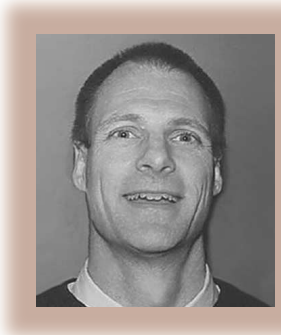
Trends in Software Process: The PSP and Agile Methods

John A. Stark and Ron Crocker

Water cooler discussions often gravitate to trends—the "latest" software process, programming language, or Web development paradigm. Some of them justly deserve the title "trend" or "fad;" others do not. In this article, two practitioners share their experience with current, yet diverse, software process approaches that are starting to greatly affect the industry. John Stark describes Science Applications International Corporation's (SAIC) experience with the Personal Software Process. Ron Crocker shares Motorola's experience with agile methods, a lighter-weight process approach. It appears that despite their diversity, both approaches have resulted in improved productivity and in similar or even higher-quality levels of the resulting products (from previous releases).—Jane Hayes

The Personal Software Process

The Software Engineering Institute at Carnegie Mellon University developed the Personal Software Process and its cousin, the Team Software Process, to give individual engineers and project teams guidance on effective day-to-day work habits.



The PSP and TSP have proven themselves in many organizations worldwide because they

- Greatly increase a team's ability to work to a plan and deliver on schedule
- Teach engineers how to measure and analyze their performance, enabling more efficient learning and improvement
- Provide sophisticated quality management, letting engineers create much higher-quality code than they are used to

The PSP and TSP have also increased productivity and reduced cycle time.

SAIC has created a process for small software teams based on PSP principles. You're meant to customize the out-of-the-box PSP process to meet individual and organizational needs. Our process uses all the PSP elements, such as personal design and code reviews, design standards, inspections, process improvement proposals, detailed task planning, earned value, quantitative defect and process management, and frequent postmortem analyses. We also added several teamwork elements and a methodical risk-management approach.

We've completed two highly successful software development and maintenance projects with this process. Mars, the first real-life use of

0740-7459/03/$17.00 © 2003 IEEE

For Further Reading

K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
A. Cockburn, *Agile Software Development*, Addison-Wesley, 2001.
R.T. Crocker, *Large-Scale Agile Software Development*, Addison-Wesley, to be published in 2003.
C. Larman, "The Agile Unified Process: Why it Reduces the Risk of Failure," Valtech, 2001.
K. Schwaber and M. Beedle, *Agile Software Development with Scrum*, Prentice Hall, 2001.

PSP training at SAIC, was a new development effort. The application was a PC-based data manipulation and formatting tool with a graphical user interface, written in C++ with the Unified Modeling Language. A team of three developers worked on it for 38 weeks using an iterative life cycle. They recorded and analyzed detailed task and defect data weekly.

The second project, Pluto, involved maintaining an existing application for nine weeks. The application and development environment resembled that of Mars; in fact, two of Pluto's three developers came from the Mars project. The developers used lessons learned from Mars postmortems to fine-tune the process for Pluto. Because Pluto was a maintenance project, the work involved applying enhancements—rather than creating whole C++ classes from scratch—each requiring a few lines of code in several places (for example, a new GUI button, computational code to perform the function, plus report generation code).

After completing both projects, we had quantitative, unbiased data describing their performance. The same process clearly did not work equally well for both projects. The same life cycle and phase sequence applied well, but the effort required for each phase differed significantly between the two projects. Furthermore, we discovered that using the same phase definitions for both project types didn't work well. We must complete more projects of each type to confidently describe the magnitude of the differences.

Mars and Pluto are the most effective projects in our organization's historical database. Of the 16 projects in the five-year database,

- Mars and Pluto had the two best effort estimates (Pluto was within 5 percent of actual).
- Mars had the best schedule performance (within 15 percent of plan). Pluto was over by 29 percent (two weeks) but included more functionality than originally planned.
- Pluto had the lowest—and Mars the third lowest—post-delivery defect density after at least six months of operation. This led to a noticeable lack of maintenance headaches and, therefore, to happy customers.

Probably the most rewarding result of using the PSP-based process is that the process elements and management activity did not slow down the project pace, as you might expect. Instead, Mars and Pluto's total productivities were, respectively, 37 percent and 213 percent above our historical average.

The same process clearly did not work equally well for both projects.

Agile methods

With agile methods—such as Extreme Programming, Scrum, Crystal, Grizzly, and even the Unified Process—we fundamentally expect requirements to change. (The "Agile Manifesto," http://agilemanifesto.org, provides a more complete definition of the term agile.) Instead of trying to predict the future, these methods encourage incremental design and implementation driven by real feedback from working software. More formal design activities are not anathema to agile methods. Rather, they encourage sufficient amounts of these activities to meet the project's needs. (See Alistair Cockburn's *Agile Software Development* [Addison-Wesley, 2001] for the definition of "sufficient" in this context.)

At Motorola, we ran our projects—implementations of new cellular radio technologies—as agile projects. We chose an agile approach because our principal difficulty wasn't the technology per se; instead, the technology was not "fully cooked." We expected that as we implemented the technology, aspects of it would emerge and change the requirements and implementation. With this expectation, we attacked the problem from several directions concurrently; we worked on the system's highest-risk portion first, but we also worked on the less risky portions to prepare for integrating the entire system.

To convert from chaos to order, we used an agile practice we call *strata*—relatively independent end-to-end layers cutting across the system at a given semantic layer. One of us, Ron Crocker, defines this practice further in *Large-Scale Agile Software Development* (Addison-Wesley, to be published in 2003). (In a previous article, we term the same concept *cluster*.[1] We changed the name because cluster evokes a mental image of clumps or groups, while strata brings a correct picture of layers.) Our project had five strata: radio, call-processing control (signaling), data bearer service (bearer), data collection, and the communication link to the base station (ATM) (see Figure 1). The subsystems (numbered at the bottom of the

columns) represent physical elements in the system—for example, Subsystem 2 might represent a subscriber unit (in this case, a mobile phone). The bars crossing the columns are the strata; they represent all functionality associated with a specific aspect of the system. The names of the subsystems and the strata are mostly uninteresting; what is important is that the strata cross subsystem boundaries and necessarily involve members of all the involved teams.

The strata gave us the main advantage of early end-to-end feedback on the real implementation of a stratum's focus area. Consider the signaling and bearer strata; because these layers include both the mobile and base station parts, early verification generally requires a functional radio link before we can validate anything, and the bearer stratum depends on the signaling stratum to create the path. To solve this problem, we encouraged the building of "scaffolding" to support the strata.

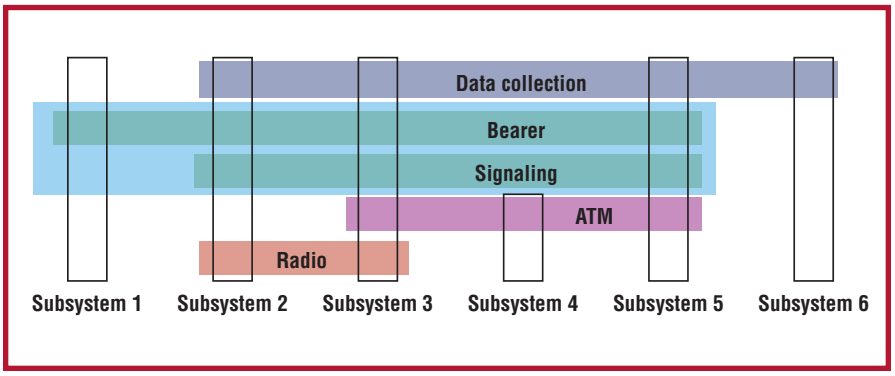


Figure 1. The project's strata and numbered subsystems.

For the signaling stratum, we created the *channel sim* to look like the real radio link and to validate the signaling end-to-end. Likewise, we could do end-to-end validation of the bearer stratum using the same channel sim.

As a secondary benefit, the strata's end-to-end nature let us build and verify complete cross-subsystem implementations of the system's various functions. This functionality included the necessary integration activities among the corresponding team members. Because this work occurred early in the project, when the pressure was low and the time to shipping seemed far away, it gave the integration teams time to form into high-performing teams. At the end, when the pressure was high and time was short, they were highly effective at diagnosing and fixing problems. Likewise, the released system requires integration of the strata; the light blue box in Figure 1 represents one such integration. As strata are integrated, the integration points are within subsystems and correspondingly within teams.

Our results were promising. Not only did we release two systems of substantial size (> 500 KLOC each, mostly C++) in about a year, each with less than five significant defects, we did so while maintaining a high level of change accommodation. The strata approach proved valuable because the early integration increased our confidence in the system's functionality while building high-performing cross-subsystem teams. Both the time required and the costs (total costs and staffing levels) were less than anticipated using our organization's standard approach, and the product's quality was somewhat better than anticipated. What we gave up in formality of process and artifacts was replaced with the communication and feedback the strata approach provided. We had trouble reporting project status to our senior management so that they could decide if the project was proceeding successfully. As the strata approach became more familiar to everyone, this difficulty disappeared.

Reference

1. R. Battin et al., "Leveraging Resources in Global Software Development," *IEEE Software*, vol. 18, no. 2, Mar./Apr. 2001, pp. 70–77.

John A. Stark is the chief software engineer for Science Applications International Corporation's Predictive Technologies Division. Contact him at starkj@saic.com.

Ron Crocker is a fellow of the technical staff in Motorola's Network Advanced Technology group. He's currently the principal investigator of performance implications of architectural decisions. Contact him at ron.crocker@motorola.com.