Exploring XP for Scientific Research

William A. Wood and William L. Kleb, *NASA Langley Research Center*

Extreme Programming, as an agile programming methodology, focuses on delivering business value. In the realm of exploratory, long-term, small-scale research projects, prioritizing near-term tasks relative to their business or scientific value can be difficult. Assigning even a qualitative monetary value can be particularly challenging for government research in enabling fields for which business markets have not yet developed. The conflict between near-term value and long-term research objectives leads to a culture clash when applying basic XP practices.

We decided to explore this culture clash when the Langley Creativity and Innovation Office solicited bids for exploring nontraditional methodologies for aerospace engineering research. C&I was looking for a way to produce extraordinary gains in productivity or enable entirely new applications. We submitted a bid and received one-year funding to perform a short prototyping assessment of XP at the NASA Langley Research Center. We conducted the project using a GNU/Linux operating system, the Emacs integrated development environment, and the Ruby programming language.[1,2] We had prior experience programming related algorithms for the advection-diffusion equation using Fortran but no experience in team software development, object-oriented design, unit testing, or programming with Ruby.

As the project began, we realized that we first had to deal with several cultural conflicts before implementing the 12 XP practices.

Eight of Extreme Programming's 12 practices are seemingly incompatible with the existing research culture. However, despite initial awkwardness, the authors successfully implemented an XP prototype-assessment project at the NASA Langley Research Center.

Cultural conflicts

Kent Beck lists nine environments that he says don't work well with XP.[3] Six of these nine are counter to the existing culture at Langley. However, Beck prefaces his assertions with the caveat that the list is based on his personal experiences: "I haven't ever built missile

0740-7459/03/$17.00 © 2003 IEEE

nosecone software, so I don't know what it is like. … If you write missile nosecone software, you can decide for yourself whether XP might or might not work." The software we were developing was intended for aerothermal predictions on nosecones of hypervelocity vehicles (see the "Our Software Product" sidebar), so we set out to decide for ourselves if XP could work in such a research-oriented environment. Here, we detail the six counter environments as they apply to our project.

First, according to Beck, the "biggest barrier to the success of an XP project" arises from an insistence on complete up-front design rather than just steering the project along the way. In February 2002, NASA announced a $23.3 million award to Carnegie Mellon University "to improve NASA's capability to create dependable software." Two-week training courses in the Software Engineering Institute's Team and Personal Software Processes have already begun, complete with 750 pages of introductory textbooks. The TSP assigns two-thirds of the project time to requirements gathering, documenting, and design. It does not allow coding, with the possibility for steering, until the final third of the project. Furthermore, significant steering can trigger a "relaunch," in which you start the requirements and design processes all over again. We used XP exclusively, avoiding the institutional bias toward the PSP and TSP.

The second cultural practice at odds with XP is big specifications. Langley's ISO 9001 implementation includes a 45-page flowchart for software quality assurance (LMS-CP-4754) and a 17-page flowchart for software planning and development (LMS-CP-5528), in which only one of the 48 boxes, located 75 percent of the way through, contains "Code and Test." At the risk of being ISO noncompliant, we ignored the approved software process, deferring the issue to when, or if, an ISO audit uncovers the discrepancy.

Third, Beck observes that "Really smart programmers sometimes have a hard time with XP" because they tend to "have the hardest time trading the 'Guess Right' game for close communication." Not only do researchers typically have doctoral degrees, but the reward structure under which they operate is based on peer review of the person's stature in the field. The Research Grade Evaluation Guide[4] emphasizes individual stature over team membership. We had to suppress our desire to be recognized for solo achievement and believe that two people doing XP would be more productive than the sum of our individual efforts.

Fourth, although adopting XP for large teams has been a frequent subject of debate, we faced the opposite problem—a small team of only two people. Maintaining the distinct roles of programmer, customer, recorder, and coach was challenging. With very small teams, the literature was unclear as to which tasks we could safely perform solo and which would rapidly degenerate into cowboy coding. In addition, with only two developers, we couldn't have the cross-fertilization benefit of rotating partners. Plus, interpersonal conflicts were a potential problem—if communication turned to confrontation, there were no other team members to play the role of mediator. Addressing these concerns required diligence in delineating roles and a conscious decision to stay focused on productive work. To rein in cowboy coding, we used test-driven pair programming exclusively when implementing features. We also preferred pair programming during refactoring but allowed solo refactoring when scheduling conflicts precluded pairing. However, we agreed not to break any existing tests or add any functionality.

Fifth, according to Beck, "Another technology barrier to XP is an environment where a long time is needed to gain feedback." One role of a government research center is to pursue long-term, revolutionary projects. Development cycles can last a decade, and the feedback loop on whether the project is headed in

Our Software Product

The research value we aimed to deliver was a software test bed for evaluating the performance of a numerical scheme to solve a model advection-diffusion problem. The model employs a multistage Runge-Kutta strategy for temporal evolution with multigrid sequencing. The particular algorithmic research feature is a strategy for the pointwise optimization of the Runge-Kutta coefficients to achieve particular damping characteristics as a tool for convergence acceleration.[1] For more information on the actual software product,[2] contact the authors.

1. W.L. Kleb, W.A. Wood, and B. van Leer, *Efficient Multi-Stage Time Marching for Viscous Flows via Local Preconditioning*, Paper 99–3267, Am. Inst. Aeronautics and Astronautics, 1999.
2. W.A. Wood and W.L. Kleb, "Runge-Kutta Circular Advection Problem Solver," *NASA Tech Briefs*, vol. 26, no. 12, 2002, p. 38.

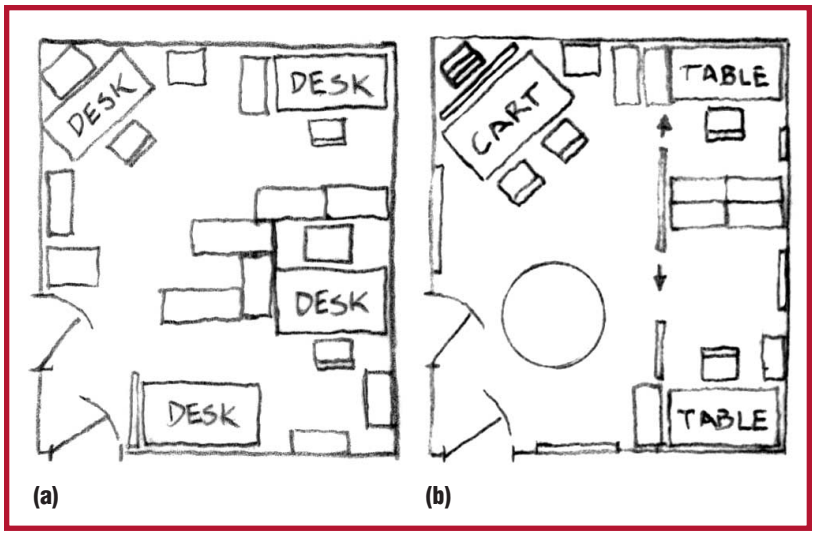

Figure 1. (a) The original 15′ × 17′ office layout: large, isolated work spaces with desks separated by towering bookcases and joined by a narrow aisle. (b) The refactored layout: small, isolated work spaces with tables and a large common area consisting of a Beowulf cluster, a pair programming station, a conference table, and whiteboards. We can move the partition at the upper right to further isolate a private work area, and all three areas can now accommodate pair programming.

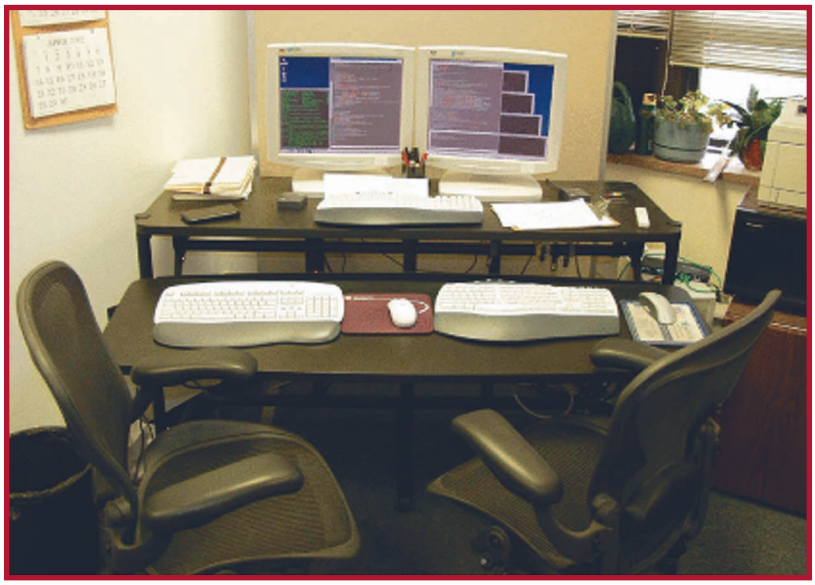

Figure 2. The pair programming station has two task chairs, a 60-inch-wide Anthro AdjustaCart, wireless keyboards and mice, and two LCD displays supporting a merged 2560 × 1024-pixel desktop. You can see the sustenance items—refrigerator, microwave, fresh-air supply, and plants—on the right.

a fruitful direction is often measured in years. XP prefers steering inputs on a days-to-weeks time frame. We didn't know if we could recast long-term research goals into small, tangible increments suitable to XP's two-to-three-week iteration cycles. Fortunately, in practice, although the research feedback timescale remains large, we successfully decomposed the technical features into small iteration chunks by following XP's simple design practice.

Finally, Beck cautions against "senior people with corner offices" because of the barriers to communication. At research centers, senior engineers typically have individual offices or cubicles, and colleagues are spread over multiple buildings at the local campus. Projects can also involve collaboration with an off-site coworker, such as a university professor. For our experiment, we refactored our office cubical layout into a commons and alcoves[5] development room (see Figure 1), with copious marker board space and a pair programming station. The pair programming station had simultaneous dual keyboard and mouse inputs connected to a 16-processor Beowulf cluster (see Figure 2).

Implementing XP practices

We made a serious effort to apply the 12 XP practices by the book (see Table 1 for a summary of our experience). Due to the six cultural conflicts just listed, eight of the 12 XP practices presented implementation challenges.

On-site customer

The biggest challenge was the on-site customer. XP places a premium on the customer–developer relationship, requiring an on-site customer. Both the customer and developer have clearly defined roles with distinct responsibilities, interacting on a daily basis. The customer focuses the developer on the business value, while the developer educates the customer on the feasibility and cost of feature requests. In the context of long-term research, the technologies being explored might be immature or uncertain, years removed from commercial potential. In this situation, the funder is too far removed from the research to serve as a suitable customer. So, the researcher is essentially the "customer" for his or her own development effort—at least for several years. What happens to the balance of power between customer and developer when they are the same person?

This was a significant concern for us, because we were primarily writing software for our own use. With two team members, we decided that, during the planning game, the indi-

Table 1

Our experience with XP practices

XP practice	Degree of adoption	Comments
Planning game	Full	We followed it by the book.[1]
Small releases	Full	Two-week iterations worked well for a project of this scope.
Metaphor	Full	We used a naive metaphor because both players spoke the same jargon.
Simple design	Full	We accepted this with skepticism, but it made future optimization easier than expected.
Test-driven development	Full	Comprehensive test coverage is the key to agility.
Refactoring	Full	This was integral to test-driven development; we followed *Refactoring: Improving the Design of Existing Code*,[6] often verbatim.
Pair programming	Full	We added new functionality only when working in pairs, improving source code readability.
Collective ownership	Full	We used CVS code control with no access restrictions.
Continuous integration	Full	We implemented four levels of automated tests with feedback ranging from seconds to hours.
Sustainable pace	Partial	Concurrent duties and nonoverlapping schedules made the pace difficult to sustain.
On-site customer	Partial	Our most difficult practice was very beneficial when implemented diligently. It can lead to loss of focus if not followed.
Coding standards	Full	Code had to be mutually understandable to both members of the pair. Clarity, over consistency, was the guideline.

vidual with the most to gain from using the software would serve as the customer while the other individual would serve as the developer. During coding, we both served as developers until questions arose, at which point one of us answered the question in the customer role. Switching roles proved challenging for the individual performing dual jobs.

During the planning game, thinking of stories was difficult without simultaneously estimating their cost. Communication was essential, and we had to make a conscious effort to think in a goal-oriented way. Furthermore, the customer had to remain focused on end results rather than think of the developer's work. However, forcing a user-oriented viewpoint helped focus the research effort, and although difficult and uncomfortable, the explicit role of customer during the planning game improved the research project's value. Even outside the context of XP, we recommend a planning game with a customer role for other research projects as an effective focusing tool.

Simple design

Another potential showstopper was the requirement for simple designs. Performance is always an issue for numerical analysis, and past experience with procedurally implemented and speed-optimized algorithms has verified the exponentially increasing cost of changing the fundamental design of elaborate codes. The lure of premature optimization for the developer is strong, particularly in the absence of a business-value-oriented customer.

We accepted this practice with skepticism, because poorly conceived numerical analysis algorithms can be prohibitively time-consuming to run. Our approach was to include performance measures in the acceptance tests to flag excessive execution times and then to forge ahead with the simplest design until the performance limits were exceeded. Once we encountered a performance issue, we used a profiler to target refactorings that would speed the algorithms enough to pass the performance criteria. The speed bottlenecks were not always intuitive, and it became evident that premature optimization would have wasted effort on areas that were not the choke points while still missing the eventual culprits.[7]

Pair programming

Pair programming also appeared to be a poor fit at first—it seemed a small team would suffer from not being able to rotate pairs. However, we actually achieved productivity gains through pairing. The pair pressure effect led to intense sessions that discouraged cutting corners, and the constant code review produced cleaner, more readable code that was much easier to modify and extend. Also, even though we each had over 20 years worth of programming experience, there was still some

Table 2

Work effort for two-week iterations over two release cycles

	Iteration						Total
	1.1	1.2	1.3	2.1	2.2	2.3	
Estimated hours	19	14	15	8	17	29	102
Actual hours	22	8	8	8	30	18	94
Velocity	1	2	2	1	0.5	1.5	1

cross-fertilization of tricks and tips that accelerated individual coding rates.

Collective ownership

The collective code ownership practice conflicted with our established practices in the field and conflicted with the promotion criteria. We didn't experience any problems, but we don't yet know the long-term impact of not having sole code ownership with regard to promotion potential. However, we anticipate that the more prolific research output enabled by XP will more than compensate for the loss of single code ownership in terms of professional prestige.

Sustainable pace

XP's 40-hour week was problematic, though perhaps for an inverse reason as encountered in programming shops. We had only about 10 hours per week mutually available for joint programming, with the rest of the time absorbed by responsibilities for other tasks or unavailable owing to conflicting schedules. We thus negotiated pair programming time during daily stand-up meetings, and we always added new functionality during joint sessions in a test-driven format. With the pair-created tests serving as a safety net, we allowed solo refactoring to increase the rate of progress. Also, we occasionally conducted disposable spikes and framework support on our own.

Test-driven development

Testing, perhaps ironically for a scientific research community, was not commonly done at the unit level prior to this project, and it also wasn't clear what would be the appropriate granularity for tests. However, we implemented four levels of fully automated testing. We wrote unit tests using an xUnit framework for each class. We ran the collection of all unit tests along with an instantiation of the algorithms devoid of the user interface as the integration test, running in a matter of seconds. Smoke tests, running in under a minute, exercised complete paths through the software, including the user interface. Full stress tests, which took hours, included acceptance tests, performance monitoring, distributed processing, and numerical proofs of the algorithms for properties such as positivity and order of accuracy. We could initiate all levels of testing at any time and automatically executed all forms nightly.

Metaphor

We conducted a search for a system metaphor and eventually selected the naive metaphor, as no other analogy seemed suitable. The naive metaphor worked well, since both the customer and developer (often the same person) spoke the same jargon.

Continuous integration

Continuous integration conflicted with the traditional approach of implementing algorithms in large chunks. We addressed this by assembling a dedicated integration machine and by crafting scripts to automate development and testing tasks. The planning game and simple design helped pare implementations down to small chunks suitable for frequent integration.

Results

The pilot project consisted of two release cycles, each subdivided into three two-week iterations, for a total project length of 12 weeks. Table 2 lists the estimated and actual pair time spent working on stories and tasks for each iteration. The times reported do not include time spent on the planning game (usually two hours at the start of each iteration). The overall average velocity for the project was approximately one, and the average time per week spent on development was approximately eight hours.

The measured velocities show variations in the work pace from twice as fast as expected to half speed. For the first iteration, the actual time to code the selected features took about as long as we expected. For the next two iterations, our proficiency improved such that we completed tasks in about half the time expected. By iteration 2.1 (first iteration of second release cycle), we incorporated the im-

proved work rate into our predictions of task times, halving the times that would have been predicted for the first cycle, so that although the "velocity" dropped to one, the actual work pace was remaining constant.

During iteration 2.2, the velocity dropped dramatically to one-half. We had become overly confident and sloppy. The tasks were not well defined during the planning game and the task times were not well estimated. The process of breaking requirements down into specific tasks during the planning game is a key part of system designing in XP, and we struggled with the poor design. Although we could sense during the iteration that the rate of progress had slowed, the velocity measurement clearly made the productivity drop visible, and the short iteration cycle kept the problem from dragging on. We took the planning game more seriously for the final iteration and productivity improved.

We produced 2,545 lines of Ruby code for an average of 27 lines per hour (productivity of a pair, not an individual). A breakdown of the types of code written shows that the average pair output was the implementation of one method with an associated test containing six asserts every 45 minutes. This productivity includes design and is for fully integrated, refactored, tested, and debugged code. Our prior performance on similarly scoped projects not developed using XP has shown an average productivity of 12 lines per hour, or 24 lines per hour for two workers. However, this historical productivity is for designed and integrated, but not tested, code. Furthermore, a subjective opinion of code clarity shows a strong preference toward the pair-developed code.

Of the total software written, 912 lines were for production code, 1,135 lines were for test code, and 498 lines were for testing scripts and development utilities. The production code contains 120 method definitions, exclusive of attribute accessor methods. The automated test code, both unit and acceptance, contain 128 specific tests implementing 580 assertions. A prior, non-XP project we performed implementing comparable functionality required 2,144 lines of code, approximately twice as large as the current production code. We attribute the reduction in lines of code per functionality to merciless refactoring, the concisely expressive nature of the dynamically typed Ruby language, and the continuous code review inherent in pair programming. Table 3 summarizes our productivity results.

Table 3

Productivity measures

		Total	Per pair-hour [a]
Production code	Lines	912	10
	Functions	120	1.3
Test code	Lines	1135	12
	Tests	128	1.4
	Assertions	580	6
Utility scripts	Lines	498	5

[a] *Total pair output per hour is the sum of this column, and includes design, keyboarding, and debugging time.*

Our results indicate that the XP approach to software development is approximately twice as productive as similar historical projects we've undertaken. This study implemented functionality at the historical rate but also supplied an equal amount of supporting tests, which are critical to the research effort's scientific validity and were not included in the historical productivity rates. Furthermore, the functional code base is about half as many lines of code as expected, and the code's readability was much improved. Continual refactoring, emergent design, and constant code review, as provided by XP, are largely responsible for the improved code aesthetics.

Our pilot project's evaluation endorses using XP for mission-critical software development at NASA's Langley Research Center. Currently, XP practices are being adopted by the High-Energy Flow Solver Synthesis project, which has been in progress for three years and employs 10 to 15 people (see http://hefss.larc.nasa.gov). The HEFSS legacy code contains over 500,000 lines of Fortran90, a procedural language that is more challenging to apply agile development methods to than is an object-oriented language such as Ruby. Full XP techniques are being used with Ruby for a variety of maintenance, support, and testing tasks: Fortran templating, automated generation of complex arithmetic code versions, generalized Fortran unit testing framework development, continuous regression suite automation, and release, installation, and configuration scripting.

About the Authors

William A. Wood is a researcher in the Aerothermodynamics Branch of the NASA Langley Research Center. His research interests include reentry aerothermodynamics. He received his PhD in aerospace engineering from Virginia Tech. He is a member of the American Inst. of Aeronautics and Astronautics. Contact him at william.a.wood@nasa.gov.

William L. Kleb is a researcher in the Aerothermodynamics Branch of the NASA Langley Research Center. His research interests include reentry aerothermodynamics. He is also a PhD candidate in aerospace engineering at the University of Michigan. He is a member of the American Inst. of Aeronautics and Astronautics. Contact him at william.l.kleb@nasa.gov.

Although the HEFSS team's transition to the XP development model is not fully complete, response from the participants has been favorable. Adhering to XP practices enhanced the development pace, bug trapping, and maintenance tasks. The prior software development process was not well defined and did not track progress metrics, precluding a quantifiable statement about the benefits of XP adoption for HEFSS.

References

1. D. Thomas and A. Hunt, *Programming Ruby: The Pragmatic Programmer's Guide*, Addison-Wesley, 2001.
2. Y. Matsumoto, *Ruby in a Nutshell: A Desktop Quick Reference*, O'Reilly & Associates, 2002.
3. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.
4. *Workforce Compensation and Performance Service: Research Grade Evaluation Guide*, Transmittal Sheet TS-23, Office of Personnel Management, Washington, D.C., 1976; www.opm.gov/fedclass/gsresch.pdf.
5. C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, Center for Environmental Structure, Oxford Univ. Press, 1977.
6. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
7. E.M. Goldratt and J. Cox, *The Goal: A Process of Ongoing Improvement*, 2nd ed., North River Press, 1992.