Extreme Programming in a Research Environment

William A. Wood and William L. Kleb

NASA Langley Research Center, Hampton VA 23681, USA
{W.A.Wood, W.L.Kleb}@LaRC.NASA.Gov

Abstract. This article explores the applicability of Extreme Programming in a scientific research context. The cultural environment at a government research center differs from the customer-centric business view. The chief theoretical difficulty lies in defining the customer to developer relationship. Specifically, can Extreme Programming be utilized when the developer and customer are the same person? Eight of Extreme Programming's 12 practices are perceived to be incompatible with the existing research culture. Further, six of the nine "environments that I know don't do well with XP" [Beck, 2000] apply. A pilot project explores the use of Extreme Programming in scientific research. The applicability issues are addressed and it is concluded that Extreme Programming can function successfully in situations for which it appears to be ill-suited. A strong discipline for mentally separating the customer and developer roles is found to be key for applying Extreme Programming in a field that lacks a clear distinction between the customer and the developer.

Key words: XP, extreme programming, customer, scientific application, testing, research, software development process

1 Introduction

Extreme Programming (XP), as an agile programming methodology, is focused on delivering business value. In the realm of exploratory, long-term, small-scale research projects it can be difficult to prioritize near-term tasks relative to their monetary value. The assignment of even qualitative value can be particularly challenging for government research in enabling fields for which business markets have not yet developed. This fundamental conflict between near-term business value and long-term research objectives is manifested as a culture clash when the basic practices of XP are applied. A brief introduction to these problematic practices follows.

XP places a premium on the customer/developer relationship, requiring an on-site customer as one of its twelve practices. Both the customer and developer have clearly defined roles with distinct responsibilities. Both interact on a daily basis, keeping each other honest and in sync. 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 may be immature or uncertain, years removed from commercial potential. In this situation the researcher can become the only customer, at least for the first several years, of their own development effort. What happens to the balance of power between customer and developer when they are the same person? Can a person serve two masters?

The government research lab environment conflicts with the pair programming and collective code ownership practices of XP because the compensation system, based on the Research Grade Evaluation Guide[2], emphasizes individual stature. Another practice, the 40-hour week, is problematic, though perhaps for an inverse reason than encountered in programming shops. The experience of the present team is that only about 10 hours per week are mutually available for joint programming, with the rest of the time absorbed by responsibilities for other tasks or unavailable due to conflicting schedules.

Another practice that is a potential show-stopper is 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 to change the fundamental design of elaborate codes. The lure of premature optimization for the developer is very strong, particularly in the absence of a business-value oriented customer.

Three more of the core practices were perceived to be a poor fit with the research environment because it was not clear how to implement them for a science application as opposed to a business application. Continuous integration conflicts with the traditional approach of implementing algorithms in large chunks at a time. Testing, perhaps ironically for a scientific research community, was not commonly done at the unit level, and in fact the appropriate granularity for testing was not evident. Finally, only the naive metaphor seemed to present itself.

The following section discusses the existing culture at a research laboratory, detailing the inherent conflicts with the XP values. The next section provides the background for a pilot project to evaluate the applicability of XP for scientific computing. The project was conducted under the auspices of a plan to explore nontraditional but potentially high payoff strategies for the design and assessment of aerospace vehicles. Specific observations concerning the implementation of XP practices in a research programming environment are enumerated. The results of the pilot project are then presented with conclusions drawn as to the effectiveness of XP in the context of research-oriented programming.

2 Culture

Beck[1] presents a list of nine "environments that I know don't do well with XP." Six of these nine are counter to the existing culture at this research center. Beck prefaces his assertions of inapplicability with the caveat that the list is based upon his personal experiences and that, "I haven't ever built missile nosecone software, so I don't know what it is like." The software developed for the re-

search situation considered here is in fact intended for aerothermal predictions on nosecones of hypervelocity vehicles, and so the present study accepts Beck's challenge that, "If you write missile nosecone software, you can decide for yourself whether XP might or might not work." The counter-indicators to using XP as they apply to the present research situation are detailed in this section along with strategies for coping with them.

Addressing the issues in the order presented by Beck, the "biggest barrier to the success of an XP project" arises from an insistence on complete up-front design at the expense of "steering." In February 2002, NASA announced a $23.3M award to Carnegie Mellon "to improve NASA's capability to create dependable software." Two-week training courses in the Personal Software Process (PSP) developed by Carnegie Mellon have already begun, complete with a 400-page introductory textbook. The PSP assigns two-thirds of the project time to requirements gathering, documenting, and design. Coding, with the possibility for steering, is not allowed until the final third of the project. Further, significant steering can trigger a 're-launch', where the requirements and design process is started all over again. The present project blended PSP and XP in a 0:100% ratio, and so far has not encountered any administrative consequences.

Another cultural practice at odds with XP is "big specifications." The ISO 9001 implementation at the Center 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 contains "Code and Test", located 75% of the way through. Despite threats of being ISO non-compliant, the present project simply ignored the approved software process, deferring the issue to when, or if, an ISO audit uncovers the discrepancy.

Beck observed, "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." The members of the research teams typically have doctoral degrees, though not in computer science. The reward structure under which the researchers operate is based upon peer review of one's stature in the field, leading to individual success or project management being highly valued, whereas team membership is not as valued. Adopting XP for the first time required a lot of trust, suppressing some long-held programming styles in the belief that two people doing XP would be more productive than the sum of their individual efforts.

While the adoption of XP for large teams has been a frequent subject of debate, the present study faces the opposite problem, a small team of only two people. Maintaining the distinct roles of programmer, customer, recorder, and coach were perceived to be challenges to the adoption of XP. With very small teams the literature was unclear as to which tasks could safely be performed solo and which others would rapidly degenerate into cowboy coding. Also, with only two developers there would not be the cross-fertilization benefit of rotating partners. Another potential problem for the small team is inter-personal conflicts. When communication turns to confrontation, there are no other team members to play the role of mediator. Addressing these concerns required diligence in de-

lineating roles and a conscious decision to keep the team focused on productive work. To reign in the cowboy coding, test-driven pair programming was used exclusively when implementing features. Pair programming was also preferred during refactoring, but solo refactoring was permitted when scheduling conflicts precluded pairing and no tests were broken or added.

"Another technology barrier to XP is an environment where a long time is needed to gain feedback." A role of a government research center is to pursue long-term, revolutionary projects. Development cycles can be over a decade in length. The feedback loop on whether or not the project is headed in a fruitful direction can be measured in years. XP prefers steering inputs on a days-to-weeks time frame. It remained to be seen if long-term research goals could be recast in small, tangible increments suitable to XP's 2–3 week iteration cycles. In practice, the research feedback time scale remains large, but for development purposes the technology features were able to be decomposed into small iteration chunks, following the simple design XP practice.

Beck cautions against "senior people with corner offices," because of the barriers to communication. At research centers the senior engineers typically have individual offices.[1] Further, colleagues are spread over multiple buildings at the local campus. Projects could also involve a collaboration with an off-site co-worker, such as a university professor. A trip to the junk furniture warehouse and borrowed time in a wood shop allowed for the one-to-a-cubical office layout to be refactored into a commons-and-alcoves[3], Figure 1.

3 Background

Despite the counter-indicators to the use of XP for scientific programming needs, the present project successfully competed for one-year funding to perform a spike[2] evaluation of XP. The funding source had specifically solicited bids exploring nontraditional methodologies for the field of aerospace engineering research that might produce extraordinary gains in productivity or enable entirely new applications. This evaluation of XP for a research environment was conducted by two researchers in three phases: learning, preparing, and implementation.

Neither investigator had prior experience with XP. Learning was achieved through a combination of personal reading[3] and sponsorship of the Modern Programming Practices Lecture Series[4] through the co-located Institute for Computer Applications in Science and Engineering. The investigators also transitioned from procedural programming to object-oriented technologies, believing that switch was a necessary, though not sufficient, prerequisite for flattening the cost-of-change curve for software development and maintenance.

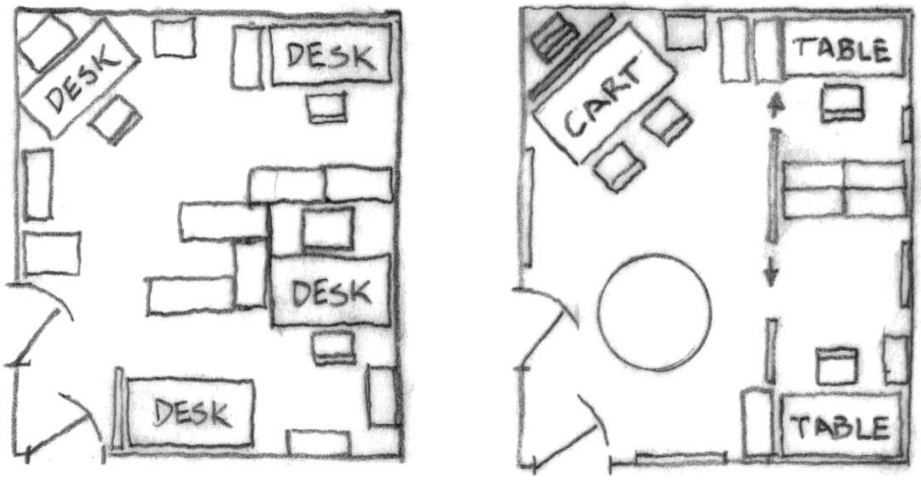
---

[1] Or cubicles.

[2] A short-term prototyping assessment.

[3] Bibliographic information is provided for books [1, 4–23], articles [24], and websites [25–27] that were found to be helpful.

[4] For a list of speakers and supporting material, see `http://www.icase.edu/series/MPP`.



(a) Original (b) Refactored

Fig. 1. The $15' \times 17'$ office layout transitioned from large, isolated work spaces with desks separated by towering bookcases and joined by a narrow aisle to small isolated work spaces employing tables and a large common area consisting of a Beowulf cluster, a pair programming station, a conference table, and white boards. Note: the partition at the upper right of (b) can be moved to further isolate one or the other private work areas and all three areas can now accommodate pair programming

In preparation for the XP experiment, environmental barriers were addressed. The office was refactored into an open development room with copious marker board space and a pair programming station was constructed with simultaneous dual keyboard/mouse inputs as shown in Figure 2, connected to a 16-processor Beowulf cluster. The development environment is: GNU/Linux operating system, Emacs IDE, and the Ruby programming language [28, 29].

The research value to be delivered by the spike project was a software testbed for evaluating the performance of an optimally adaptive Runge-Kutta coefficient strategy for the evolution of an advection-diffusion problem,

$$u_t + \underbrace{\vec{\lambda} \cdot \nabla u}_{\text{advection}} = \underbrace{\mu \nabla^2 u}_{\text{diffusion}} , \tag{1}$$

in a multigrid context [30, 31]. Integration of Eq. (1) with application of Gauss' Divergence theorem leads to

$$\frac{\partial}{\partial t} \int_{\Omega} u \, d\Omega = \oint_{\Gamma} \left(-u\vec{\lambda} + \mu \vec{\nabla} u \right) \cdot \hat{n} \, d\Gamma \ . \tag{2}$$

The desired Runge-Kutta strategy would optimize the damping of the high-frequency errors in the discrete representation of the temporal evolution in

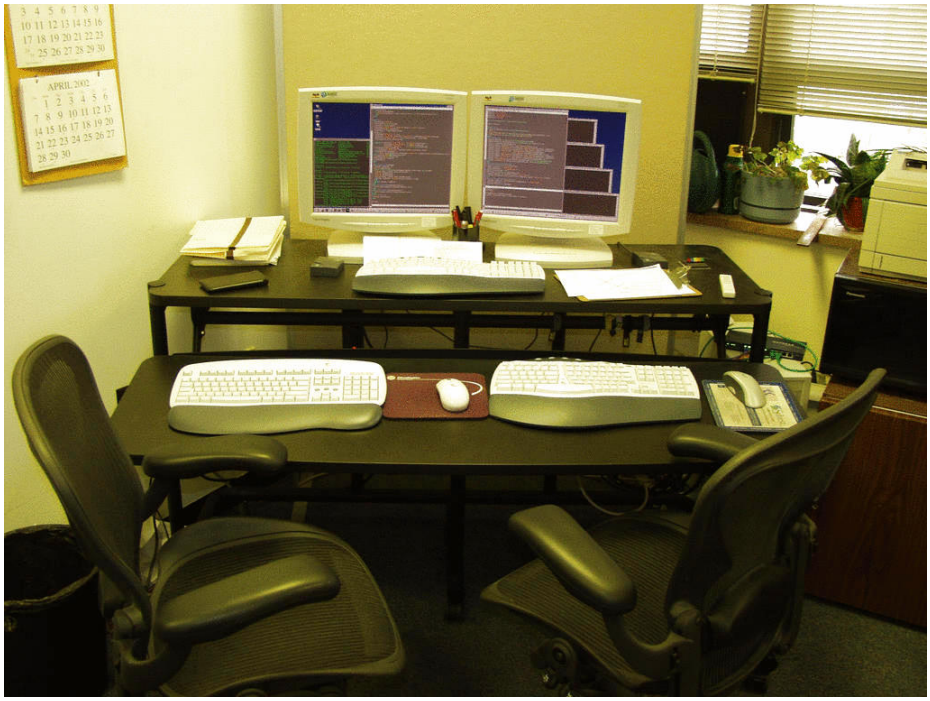


Fig. 2. Pair programming station consisting of two Herman Miller Aeron task chairs, a 60"-wide Anthro AdjustaCart, Logitech wireless keyboards and mice, Belkin keyboard and mouse switches, and two Viewsonic 18" LCD displays supporting a merged $2,560 \times 1,024$-pixel desktop. The sustenance items, refrigerator, microwave, fresh-air supply, and plants, can be seen at the right

Eq. (2), while the multigrid scheme applied to the spatial discretization serves to alias discrete low-frequency errors into higher harmonics, which are efficiently damped by the temporal operator.

Both investigators had independent prior experience programming related algorithms for the advection-diffusion equation using FORTRAN. Neither investigator had experience in team software development, object-oriented design, unit testing, or programming with the Ruby language.

4 Methodology

A serious effort was made to apply the 12 XP practices by the book. As described in Sect. 1, eight of the practices presented challenges for implementation. These challenges were caused by perceived environmental, historical, or cultural barriers.

The biggest challenge was to have an on-site customer, specifically when the customer and developer are the same person. In the present case, the developers

were writing software for their own use. With two team members it was decided that the individual with the most to gain from using the software would serve as the customer while the other individual would serve as the developer during the planning game. During coding, both individuals served as developers until questions arose, at which point one individual would have to answer in the customer role. This switching of roles proved to be challenging for the individual performing dual jobs. During the planning game it was a challenge to think of stories without simultaneously estimating their cost. The game required a lot of communication and conscious efforts to think goal-oriented and remain focused on end results when playing the customer, rather than thinking of the work about to be performed as the developer. It was found that forcing a user-oriented viewpoint helped to focus the research effort, and it is believed that, while difficult and uncomfortable, the explicit role of customer during the planning game improved the value of the research project. Even outside the context of a programming assignment, a planning game with a customer role is recommended for other research projects as a highly effective focusing tool.

The simple design practice was accepted with skepticism, as poorly conceived numerical analysis algorithms can be prohibitively time consuming to run. Past experience with procedural algorithms suggests that performance issues need to be planned up-front. The approach of the present team 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 a performance issue was encountered, a profiler was used to target refactorings that would speed the algorithms just 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 (cf. [32]). Also, the adherence to simple designs made the identification and rectification of the bottlenecks easier at the appropriate time.

As discussed in Sect. 2, pair programming appeared to be a poor fit, and the small team would suffer from not being able to rotate pairs. However, productivity gains were in fact achieved through pairing. The pair pressure effect led to intensive sessions that discouraged corner cutting, and the constant code review produced much cleaner, more readable code that was much easier to modify and extend. Also, even though each developer had over 23 years programming experience, there was still some cross-fertilization of tricks and tips that accelerated individual coding rates.

The collective code ownership practice was counter to the established practice at this research center and conflicted with the promotion criteria. The present team agreed to collective code ownership and did not experience any problems. The long term impact of not having sole code ownership with regards to promotion potential is not yet known.

The research environment is different from a programming shop, in that other activities occupy most of a person's time, and the present effort found only about 10 hours per week for pair programming, instead of the recommended

40 hour practice. New functionality was always added during joint sessions in a test driven format. With the pair-created tests serving as a safety net, solo refactoring was permitted to increase the rate of progress. Also, disposable spikes and framework support were occasionally conducted solo.

Unit testing was not commonly done prior to the present effort, and it was not clear what to test, in particular the appropriate granularity for tests. Four levels of fully-automated testing were implemented. Unit tests using an xUnit framework were written for each class, and the collection of all unit tests along with an instantiation of the algorithms devoid of the user interface was run 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, taking hours, included acceptance tests, performance monitoring, distributed processing, and numerical proofs of the algorithms for properties such as positivity and order of accuracy. All levels of testing could be initiated at any time, and all forms are automatically executed nightly.

A search for a system metaphor was conducted for a while and eventually the naive metaphor was selected as no other analogy seemed suitable. The naive metaphor worked well, as both the customer and developer spoke the same jargon, being the same people.

Continuous integration was addressed 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 to frequent integration.

5 Results

The pilot project consisted of two release cycles, each subdivided into three two-week iterations, for a total project length of 12 weeks. The estimated and actual time spent working on stories and tasks for each iteration is listed in Table 1. The times reported do not include the time spent on the planning game. Typical

Table 1. Work effort for two-week iterations, over two release cycles

Iteration	*1.1*	*1.2*	*1.3*	*2.1*	*2.2*	*2.3*	*Total*
Estimated hours	19	14	15	8	17	29	102
Actual hours	22	8	8	8	30	18	94
Velocity	1	2	2	1	$\frac{1}{2}$	$1\frac{1}{2}$	1

lengths for the planning game at the start of each iteration were two hours. The overall average velocity for the project was about one, and the average time per week spent on development was about eight hours.

The team produced 2,545 lines of Ruby code, for an average of 27 lines per hour (productivity of a pair, not 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. Prior performance by the team 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 integrated, but neither tested nor debugged, code. Further, 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, 1135 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, contains 128 specific tests implementing 580 assertions. A prior, non-XP project by the present team implementing comparable functionality required 2,144 lines of code, approximately twice as large as the current production code. The reduction in lines of code per functionality is attributable primarily to merciless refactoring and secondarily to the continuous code review inherent to pair programming.

6 Conclusions

Despite counter-indicators of XP being at odds with the existing research software development culture and the initial awkwardness of several of the practices, an XP spike project was successfully implemented. Attention to the XP rules, blind trust in the XP values, and the diligent role playing of the customer and developer parts were key to this success. The conscious and deliberate separation of the customer role from the developer, even when embodied by the same individual, was found to provide a benefit to the research project in general, beyond the scope of the software development. This benefit was manifest as a focusing of the research effort to tangible, targeted goals.

The team consisted of two people, and undoubtedly missed the benefits XP brings through pair rotation. While not preferred, it was found that some refactoring could be safely performed solo when supported by sufficient automated testing. This compromise was necessitated by the realities of conflicting schedules with a part-time work commitment. Predominantly, the initial cultural counter-indicators to using XP were found in fact to not preclude the use of XP in the research context, although the long-term impact on promotions and prestige due to lack of clear code ownership is not known. It is anticipated that the more prolific research output enabled by XP will more than compensate for the loss of single code ownership upon prestige in the field.

The results of the present study indicate that the XP approach to software development is approximately twice as productive as similar historical projects undertaken by members of the team. This study implemented functionality at the historical rate, but also supplied an equal amount of supporting tests, which are critical to the scientific validity of the research effort, and which were not included in the historical productivity rates. Further, the functional code base is

about half the lines of code as would be expected from past experience, and the readability of the code is considered to be much improved. Continual refactoring, emergent design, and constant code review as provided by XP are largely responsible for the improved code aesthetics.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000)
2. Workforce Compensation and Performance Service: Research grade evaluation guide. Transmittal Sheet TS-23, Office of Personel Management, Washington, DC (1976) Also available as `http://www.opm.gov/fedclass/gsresch.pdf`.
3. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns · Buildings · Construction. Center for Environmental Structure. Oxford University Press (1977)
4. Beck, K., Fowler, M.: Planning Extreme Programming. XP. Addison-Wesley (2001)
5. Jeffries, R., Anderson, A., Hendrickson, C.: Extreme Programming Installed. XP. Addison-Wesley (2001)
6. Succi, G., Marchesi, M., eds.: Extreme Programming Examined. XP. Addison-Wesley (2001)
7. Newkirk, J., Martin, R.C.: Extreme Programming in Practice. XP. Addison-Wesley (2001)
8. Wake, W.C.: Extreme Programming Explored. XP. Addison-Wesley (2002)
9. Auer, K., Miller, R.: Extreme Programming Applied: Playing to Win. XP. Addison-Wesley (2002)
10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
11. Yourdon, E.: Death March: Managing "Mission Impossible" Projects. Prentice-Hall (1997)
12. Brooks, Jr., F.P.: The Mythical Man-Month: Essays on Software Engineering. Anniversary edn. Addison-Wesley (1995)
13. Kernighan, B.W., Pike, R.: The Practice of Programming. Addison-Wesley (1999)
14. Cockburn, A.: Surviving Object-Oriented Projects. Addison-Wesley (1998)
15. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Object Technology. Addison-Wesley (2000)
16. Booch, G.: Object Solutions: Managing the Object-Oriented Project. Object-Oriented Software Engineering. Addison-Wesley (1996)
17. Booch, G.: Object Oriented Design with Applications. Ada and Software Engineering. Benjamin/Cummings (1991)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Resuable Object-Oriented Software. Professional Computing. Addison-Wesley (1994)
19. Meyer, B.: Object-Oriented Software Construction. 2nd edn. Prentice-Hall (1997)
20. Hunt, A., Thomas, D.: Pragmatic Programmer: From Journeyman to Master. Addison-Wesley (1999)
21. DeMarco, T., Lister, T.R.: Peopleware: Productive Projects and Teams. 2nd edn. Dorset House (1999)
22. Highsmith, III, J.A.: Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House (2000)
23. Machiavelli, N.: The Prince. Bantam classic edn. Bantam Books (1513)

24. Gabriel, R.P., Goldman, R.: Mob software: The erotic life of code. http://oopsla.acm.org/oopsla2k/postconf/Gabriel.pdf (2000) ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).
25. http://www.c2.com/cgi/wiki?ExtremeProgramming (2000)
26. http://www.xprogramming.com/ (2000)
27. http://www.extremeprogramming.org/ (2000)
28. Matsumoto, Y.: Ruby in a Nutshell: A Desktop Quick Reference. O'Reilly & Associates (2002)
29. Thomas, D., Hunt, A.: Programming Ruby: The Pragmatic Programmer's Guide. Addison-Wesley (2001)
30. Kleb, W.L., Wood, W.A., van Leer, B.: Efficient multi-stage time marching for viscous flows via local preconditioning. AIAA Paper 99–3267 (1999)
31. Kleb, W.L.: Optimizing Runge-Kutta Schemes for Viscous Flow. PhD thesis, University of Michigan (2003) In preparation.
32. Goldratt, E.M., Cox, J.: The Goal: A Process of Ongoing Improvement. 2nd edn. North River Press (1992)

7 Biographies

Bill Wood: PhD Virginia Tech, Aerospace Engineering. 1987–Present: NASA Langley Research Center. Currently in the Aerothermodynamics Branch.

Bil Kleb: PhD Candidate University of Michigan, Aerospace Engineering. 1988–Present: NASA Langley Research Center. Currently in the Aerothermodynamics Branch.