The XP Programmer:
The Few-Minutes Programmer

Laurie Williams, *North Carolina State University*

Written over two decades ago, *The One-Minute Manager* is still a top-selling book.[1] In just over 100 pages, it exudes advice to managers on short (one-minute) techniques for achieving a positive result with their employees, improving their productivity and job satisfaction. The authors explain how communication and consistency can improve results with minimal time and effort.

In its own way, the Extreme Programming methodology exudes this same advice: Be communicative with everyone on the team—including customers, end users, and business folks. Consistently and doggedly strive to understand and deliver what your customers want with the highest possible quality. Frequently offer feedback—a one-minute manager would never hold back until a formal appraisal cycle to give feedback to employees. Similarly, XP developers never spend extended periods of time on any one development practice, as in a waterfallish process. Instead, every few minutes (every one minute is probably a stretch), a developer iterates from one practice to another (such as test, design, code, test, design, code, and so on). The purpose of this "every few minutes" cycling is to get feedback early and often on decisions that have been made—from requirements decisions to design decisions and the like.

0740-7459/03/$17.00 © 2003 IEEE

Few-minutes project management

Some software development projects begin with lengthy cycles of determining a contractual set of requirements and developing an extensive project management plan for project completion. Relatively, XP teams spend just a few minutes on each of these activities—many times each day.

XP programmers gather requirements as short, natural-language statements that are essentially the customer's words written on small index cards. Called *user stories*, these cards are commitments for further conversation between the customer and developers and are not intended to completely specify requirements. Generally, they represent desired working code that the developers can produce for the customer to try, as opposed to documents, designs, or database schemas customers can only review.

The customers prioritize their user stories, and the developers estimate how long implementing the requirement will take. If an estimate exceeds the length of one iteration (which is typically one to three weeks), the customer and developer work together to break the user story into multiple stories. Ultimately, each user story's developer estimate will be no longer than one iteration.

When user story cards are complete with requirements statements, customer priorities, and developer estimates, the planning game begins. To play, the customer and some developers lay the user story cards out on a table. They have a joint understanding of how much time is available for the release (a small number of iterations) and how many people will be involved in development. They slide the cards around on the table until the customer chooses his or her highest-priority user stories to keep the team busy for the release on the basis of the developers' time estimates.

An important premise in XP is that the developers work at a sustainable pace (that is, they should stop when they are tired). XP incorporates this practice into project planning because tired programmers are more error-prone, and the defects injected during their overtime could take longer to find and fix than the overtime itself. Additionally, requiring developers to dedicate too much time to their jobs, as is common in the industry, can eventually affect their personal life and cost the project—for example, by resulting in high turnover.

Once the customer chooses the user stories, software developers select which ones they'd like to "own" and implement. Because a user story is intentionally an incomplete description of the requirements, an important XP practice is to converse with representative customers. Before beginning a story, the story's owner spends a few minutes with the customer to better understand what he or she wants. As implementation proceeds, the customer and developers will often spend few-minute periods clarifying requirements and demonstrating work in progress. On the basis of these short inspections by and conversations with the customer, the developers can adapt their work to best suit the customer's desires.

Each day, the development team spends a few minutes in a stand-up meeting. The team intentionally stands up in a circle during the meeting to motivate members to keep the meeting as short as possible. One by one, each developer talks for a few moments about

- The prior day's accomplishments
- Any obstacles, difficulties, or stumbling blocks faced
- What he or she plans to accomplish during the current day on the basis of the selected stories and tasks

These short meetings make team members publicly accountable for their progress and plans and provide an opportunity to get help from teammates in overcoming problems.

After each iteration, the customer views the working software and provides feedback. The developers can calculate the iteration's *velocity* to best plan how much work they can accomplish in future iterations. Velocity is basically the number of days' worth of work accomplished during the iteration period. The customer and developer then replay the planning game so that the customer can change or reprioritize the requirements in the pipeline and developers can update their estimates. As such, detailed project management is only done for the immediate future—the next one- to three-week iteration.

Few-minutes design

Some software development projects can proceed with lengthy cycles of defining a product architecture and developing high- and

Consistently and doggedly strive to understand and deliver what your customers want with the highest possible quality.

XP teams value face-to-face communication and spend essentially the whole day, every day, communicating.

low-level system design. Again, XP teams spend just a few minutes on each of these activities many times each day.

Before starting development, the team spends a few minutes coming up with a system metaphor or a "story that everyone—customers, programmers, and managers—can tell about how the system works."[2] The metaphor guides a system of names for objects and their relationships and a common vocabulary. As opposed to a system architecture, the metaphor shapes the system, and this shape evolves over time. When a pair of programmers starts working on a user story, they spend a few minutes brainstorming how they would implement the functionality. This could take the form of sketching a class diagram on a white board (which would be erased after implementing the code), a quick CRC (Class Responsibility Collaborator) card session,[3] or possibly some dabbling in code right away.

When coding, XP programmers iteratively spend a few minutes writing automated unit test cases (which will fail when run, because the code hasn't been implemented yet), followed by a few minutes of implementation code development to pass the tests they just wrote. This practice, called *test-driven development* (TDD),[4] keeps the system's concrete design as simple as possible. The programmers write only the code that must pass the test cases; they don't develop an impressive superstructure to handle some anticipated but not yet requested or prioritized requirement. They often write these automated test cases using an xUnit testing framework (see http://xprogramming.com/software.htm), and the tests grow and reside with the implementation code (although in a different code hierarchy). The developers generally don't spend even a few minutes loading up their code with comments. Instead, they use descriptive variable names to ensure their code is simple and reveals their intentions. This practice ameliorates the general tendency of out-of-date comments and provides "executable" documentation, which is obviously kept current.

Programmers do not own the code they write. Instead, the code belongs to the team—so anyone can change or enhance anyone else's code to complete their job without waiting even a few minutes for the original programmer's permission. The programmers can change each other's code because of the extensive TDD test cases, which provide an alert if a change breaks some previously implemented functionality. Ideally, throughout the day, programmers spend a few minutes refactoring[5] their code or improving the code's concrete design without changing the functionality.

Few-minutes feedback

Frequent feedback is essential. XP teams, including the on-site customer, spend a few minutes every day providing this feedback.

One form of feedback is acceptance test cases. The customer writes at least one acceptance test case for each user story. Initially, this test case provides early feedback to the developer on whether he or she understands the user story properly. Ron Jeffries calls this the Card, Conversation, Confirmation cycle (www.xprogramming.com/xpmag/EXPCardConversationConfirmation.htm), which involves writing a user story card, having the customer and developer converse about the card's requirement, and using the acceptance test case to confirm the requirement's completion. At the end of each iteration, the customer can see the completed, working software that implements the prioritized user stories. He or she can then run the test cases to ensure the system works as expected. If it doesn't, the development team has spent a recoverable amount of time going down the wrong path. The customer can revise his or her user stories and acceptance test cases and prioritize fixes to the running software so that the developer can better meet his or her needs in the next planning game.

Several development practices provide feedback every few minutes to the programmers on the implementation's correctness. With TDD, the developers continuously run their code against a suite of automated test cases. By doing this, they constantly get feedback that they are writing "clean code that works."[4] Similarly, pair programming[6] provides feedback on a minute-by-minute basis. With pair programming, two developers work together at one computer, collaborating on the same design, algorithm, code, or test. The two are perpetual brainstorming partners. This practice provides the driver (the programmer at the keyboard) with feedback on his or her ideas and code. Together, the TDD and pair programming focus on continual quality assurance.

When a pair completes some functionality, they ensure that their new automated test cases run and that the automated test cases in

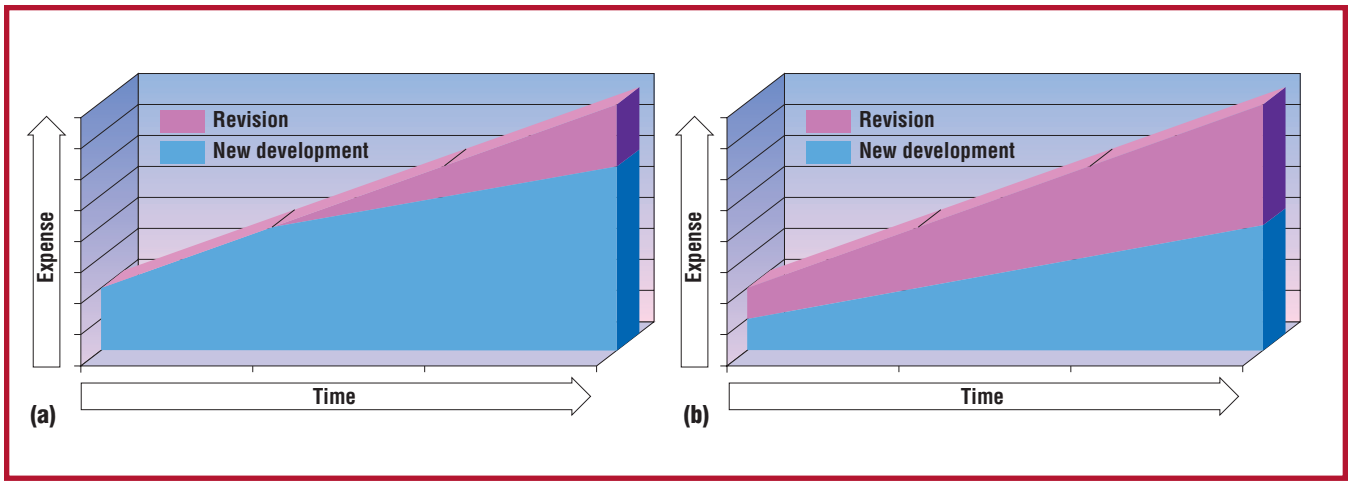

Figure 1. (a) Desired cumulative expense in traditional software development; (b) observed cumulative expense in agile development.

the code base run. They then put their new code into the code base. Pairs integrate their code into the code base at least once per day, receiving immediate feedback if a problem exists with the build and their new code.

Few-minutes team building

On some software development projects, the team members do not know each other that well. Consider the stereotypical programmer who comes in each morning with headphones on, listing to music, codes while emailing questions to teammates all day, and leaves after many hours of work (only then removing the headphones).

XP teams value face-to-face communication and spend essentially the whole day, every day, communicating. They value colocation so that everyone on the team (which includes the developers, testers, customers, analysts, coach, and manager) can talk face-to-face whenever anything needs clarification or organization. Instead of listening to music and remaining isolated, the programmer talks to his or her partner all day as they both code. This communication breaks down barriers, creating camaraderie and communities of practice within the team. Through the daily stand-up meetings, team members learn what others are working on and struggling with and how they can help each other so that the whole team succeeds.

XP betting

Software engineers have a long-held belief that they must "get out defects" as early as possible. They must do the requirements right the first time, or there will be detrimental effects throughout development. They must develop a solid architecture before they start coding, or the project will be in trouble. The XP contention that it is better to spend a relative few minutes on these kinds of activities can seem bewildering.

Consider a team that plans to have two members spend three months developing a solid, well-specified, and inspected requirements document. The team will certainly spend six person-months on this document. Instead, an XP team might have two people spend a week documenting the best set of user stories the customer can create at that time. In this case, the team spends two weeks developing this first version of the requirements document—thus banking five and a half months. They can later spend this time on revising requirements or the product on the basis of current knowledge, including the customer's perception of the working software that has been developed thus far. The XP bet is that they will not have to spend more than the saved five and a half months to revise the requirements or the product and that the customer's current view of their desires is better than their initial view.

The two theoretical graphs in Figure 1 represent this betting process and depict the cumulative expense incurred in software development over time. Figure 1a represents the traditional method and philosophy of development. Most of the expense is spent on new development, whether for a document, a design, or code. Inevitably, some revisions will occur during the development cycle, although the aim is to minimize revision expense.

Conversely, Figure 1b represents an XP

Joint Theme with *Computer*

Extreme Programming is just one methodology in the emerging class of agile software development methodologies.[1–3] The June issue of *Computer* features broad perspectives on agile software development. One article in the issue places agile software development in the context of the history of iterative and incremental development. Two others focus on balancing agile software practices with the need for more stringent, plan-driven approaches. Additionally, two experience reports share perspectives on introducing agile processes in an organization. Finally, Barry Boehm and Kent Beck debate how to achieve agility through discipline.

References

1. K. Beck et al., "The Agile Manifesto," 2001, http://agilemanifesto.org.
2. A. Cockburn, *Agile Software Development*, Addison-Wesley, 2001.
3. J. Highsmith, *Agile Software Development Ecosystems*, Addison-Wesley, 2002.

project's cumulative expense. In this case, the expense incurred for new development and for revision has a much different profile, demonstrating a large increase in revision and a correspondingly large decrease in new development. However, strong anecdotal evidence suggests that the additional revision does not exceed the expense that would have been incurred had extensive up-front requirements engineering, planning, and designing occurred. So, both graphs indicate the same level of expense incurred over similar time periods. (Researchers are investigating the validity of these anecdotal claims.)

In a traditional project, development proceeds with a "do it right the first time" philosophy with the objective of "satisfying the original contract." This is an appropriate approach for projects without a significant degree of requirements variability. In an XP project, development proceeds with a "do it right the last time" philosophy with the objective of "delighting the customer." The XP, few-minutes programming approach is appropriate for projects with a significant degree of requirements variability.

The XP articles in this issue share experiences of few-minutes programmers in three different domains and environments. Jonathan Rasmusson discusses using XP in a bleeding-edge technology project for a North American energy company. William Wood and William Kleb share their experiences with XP in a NASA research environment. Finally, Orlando Murru, Roberto Deias, and Giampiero Mugheddu relate their experiences using XP in an Internet startup in Italy. These teams have succeeded by being communicative and consistent. We hope you can learn from their experiences.

Acknowledgments

Many thanks to Kent Beck, Ron Jeffries, and my graduate students for their helpful suggestions on this Guest Editor's Introduction. Also, I couldn't have published this special issue without the astute reviews, suggestions, and shepherding of the program chairs (Michele Marchesi, Giancarlo Succi, and Don Wells) and program committees of the XP/Agile Universe and XP 2002 conferences. I also thank the authors who cheerfully withstood my infatuation with Microsoft Word's "editing" feature as they improved and extended their conference papers. (Now they know what it's like to be one of my graduate students!)

References

1. K.H. Blanchard and S. Johnson, *The One-Minute Manager*, Berkley Publishing Group, 1983.
2. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.
3. D. Bellin and S.S. Simone, *The CRC Card Book*, Addison-Wesley, 1997.
4. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.
5. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
6. L. Williams and R. Kessler, *Pair Programming Illuminated*, Addison-Wesley, 2003.

About the Author

Laurie Williams is an assistant professor of computer science at North Carolina State University. Her research interests include agile software development methodologies and practices, software reliability, software testing, and e-commerce. She received a PhD in computer science from the University of Utah. She is the coauthor of *Pair Programming Illuminated* (Addison-Wesley, 2003) and *Extreme Programming Perspectives* (Addison-Wesley, 2003). She is a member of the IEEE and ACM. Contact her at williams@csc.ncsu.edu.