Tomoo Matsubara ■ Matsubara Consulting ■ tmatsu@xa.so-net.ne.jp

eXtreme Programming Development through Dialog

Robert C. Martin

To those who are unfamiliar with the method, *eXtreme Programming* might conjure up visions of programmers madly hacking away without concern for analysis, architecture, design, or consequences. Indeed, people have frequently made such accusations. Yet such claims are not only baseless, they are diametrically opposed to what XP is all about.

XP views people, rather than paper, as a project's most potent element.

XP is the brainchild of Kent Beck, and he describes it in his landmark book *eXtreme Programming Explained* (Addison-Wesley, 2000). (A review of this book appears on page 113 in this issue.) It is a software development method that views people, rather than paper, as a project's most potent element. Its primary motive is to start a dialog between the people involved in a project. It facilitates this by identifying the parties in the conversation and arming them with the clear knowledge of what they are responsible for communicating to others.

The Dialog

The two primary roles in XP are *customer* and *programmer*. The customer is the person or group who represents the users. The customer is responsible for identifying the features (known as stories) that the programmers must implement, providing detailed acceptance tests for those stories and assigning priority to them. Thus, the programmers should implement the stories the customer wants, in the order the customer wants, and pass any tests that the customer specifies. Programmers may not implement anything that the customer does not explicitly request. On the other hand, programmers are responsible for estimating how long it will take to implement the stories. Customers may not question or challenge those estimates or make commitments to third parties that do not reflect those estimates.

With this division of responsibility in place, it is clear that a project plan represents a dialog between the customer and programmers. The protocol of that dialog is also clear. The customer tells the programmers what stories he or she wants in the next iteration. The programmers add up the estimates for those stories and tell the customer whether they are possible. The customer can remove or swap stories but can't get more in the iteration than the programmers estimate is possible. The programmers can tell the customer how long something will take, but they can't select the stories to be implemented.

How could this possibly work? After all, no customer is going to simply accept the programmers' estimates; instead, the customer will push back on those estimates, demand better performance, and wheedle and cajole the programmers into making their estimates more acceptable. And no programmer is simply going to implement only the features that the customer wants and in the order he or she wants. Rather, the programmers are going to build lots of infrastructure first and work on

0740-7459/00/$10.00 © 2000 IEEE

only the technologically "cool" things—or the things that look good on their resumes. Or so goes the common assumption.

This is part of the essential distrust that exists between customers and programmers. XP breaches this distrust with a single concept: feedback.

The Two-Week Plan

The concept is simple. Maybe you can't trust me for a year; but you can probably trust me for two weeks. If I prove to be untrustworthy after two weeks, you've lost very little. But if I prove trustworthy, you will probably be willing to trust me for another two weeks.

In XP, the time from when the customer directs the programmers to work on a story until when the story is actually working in the system is typically two weeks. For that short time period, the customers and programmers can suspend their distrust of each other.

Thus, in XP, the planning horizon is only two weeks. No detailed plans are created past an iteration's horizon. Participants create all detailed plans when each iteration begins, and the plans are meaningful only during the iteration—indeed, very few artifacts of any kind are created that outlast one. This includes project plans, analysis models, design models, and so forth. Rather than make long plans and complex models, we do two weeks worth of work, assess where we are, and then plan the next two weeks.

This rapid succession of planning cycles is like a protracted conversation. The participants enter into a kind of rhythm in which they make very short-term plans, execute those plans, and then evaluate the outcome and make the next set of plans.

How can this work? Don't we need to have all the requirements up front? Don't we need to build extensive models of the problem and solution domains? Don't we need a project plan that describes how the whole project will work? Without these things, won't the project descend into chaos?

This works simply because the rapid feedback removes errors. Every two weeks, the customer looks at the system being built and assesses whether the last two weeks have resulted in an improvement. If not, the customer directs the next two weeks of work in a way that he or she thinks will improve the product. Moreover, after the first few iterations, the customer gets a feel for the development team's velocity and can determine if it is fast enough for his or her purposes. Finally, any time the customer feels that there is nothing further to be gained by continuing the project, he or she can stop it and take away a working system.

Quality Control

But won't constantly reworking the code reduce it to an unmaintainable mess? XP is acutely sensitive to poor code quality and addresses it in several ways. First, XP demands simplicity from the programmers—they must leave the code in the simplest possible state that passes all the acceptance tests. Thus, when code is reworked from iteration to iteration, it is also continually reduced to the simplest state the programmers can find.

Second, programmers are not allowed to work on the code alone. The programmers team up in pairs, and each pair sits at a workstation and programs. The result is that every line of code is the result of a conversation between at least two programmers.

Finally, before adding code to the system, the programmers must write a failing unit test that the new code must make successful. This ensures that as the program grows, a copious suite of tests grows with it. These tests keep the quality of the software high and give the programmers the courage they need to continually rework the code into its simplest form—an operation known as *refactoring*.

Thus, there are several layers of dialog in an XP project. The customers have an ongoing dialog with the programmers, the programmers have an ongoing dialog with each other, and the test cases have an ongoing dialog with the program. But are these dialogs sufficient for software development? Or do we need heavier and more formal processes?

The answer is that, for many projects, development through dialog works very well. The customers and programmers learn to trust one another. They have conversations about new stories the customers want added to the system, without feeling the need to commit them to a large requirements document. The customers can see the pace of the project from iteration to iteration, and they eventually grow comfortable with the programmers' estimates without requiring huge detailed plans. The code grows as customers add new stories iteration after iteration, but the code does not rot. It is continually refactored into a clean and simple state, all without huge analysis and design models.

Over the past few decades, we have sought to solve the software industry's problems by creating ever-larger processes that produce ever-more complex and arcane artifacts prior to software production. Perhaps all we really needed was to learn how to talk to each other.

Robert C. Martin is president of Object Mentor, Inc., a firm of highly accomplished software experts that offers high-level object-oriented software design consulting, training, and development services to major corporations around the world. In 1995 he authored the best-selling book, *Designing Object Oriented C++ Applications Using the Booch Method* (Prentice Hall). From 1996 to 1999 he was the editor-in-chief of *C++ Report*. Contact him at rmartin@objectmentor.com.