

Using Extreme Programming in a Maintenance Environment

Charles Poole and Jan Willem Huisman, *Iona Technologies*

During the 1990s, Iona Technologies' flagship product was a Corba-based middleware product called Orbix. We have since created a newer version of Orbix, but here we discuss how we developed and, more importantly, now maintain the older version, which represents the Corba specification's early evolution. We also review the inherent problems of code entropy due to specification instability and time-to-market pressures. Over the years, because of these pressures and the rapid growth

in Iona's engineering team, we often ignored good engineering processes and practices. As a result, the code base's overall health degenerated and was salvaged only through two successful reengineering efforts and a series of infrastructure projects designed to improve overall engineering practices. It wasn't until later that we realized how closely our efforts were tied to Extreme Programming.

The problems

When Iona moved from a start-up to a market-leading provider of middleware products in 1999, it faced four major problem areas: processes and practices, testing, code entropy, and team morale. An initial reengineering project that started in 1997 was concluding, and although it focused a tremendous amount of resources on reengineering, improving testing, and making nec-

essary changes to the code base to comply with the latest version of the Corba specification, it did not sufficiently address the four problem areas; it merely skimmed the surface of what needed to be done.

Processes and practices

In early 1999, you could have asked two different engineers from the same Orbix team how they did their jobs, and each would have replied differently. This reflected the team's lack of process documentation and visibility and its failures to make process a part of each engineer's personal software practices. Also, there was no focus on process improvement. The junior engineers assigned to maintenance and enhancement had no experience with good engineering practices and had only a rudimentary understanding of process. To compound the problems, the maintenance and

The authors review how efforts to introduce industry-level best practices led to Extreme Programming, which improved the team's ability to deliver quality support and product enhancements.

enhancement team used disparate source control elements across two globally distributed development sites and lacked a well-defined and tested interface between configuration units. Dependency management was a nightmare.

Testing

In general, quality was never one of Orbix's strong points. We didn't document test coverage very well, so good metrics were not available. Our test suites were cumbersome, difficult to run, and impossible to accurately report. Also, the test suite used to provide interoperability testing with other Iona products and product components—or against other Corba middleware products—was not automated across the platform set to which we delivered our product. Thus, system testing our product releases was difficult. Furthermore, instead of having the development and maintenance teams monitor quality, a team that was detached from the engineering effort monitored it through checklists and forms.

Code entropy

By the end of 1997, we had already patched and repatched the Orbix code hundreds of times to address numerous customer issues as well as the changing Corba specification. These patches often used band-aid approaches to resolve problems or add functionality and were a major factor in the code's rapid entropy. Many of these unreviewed changes were made to a code base that was never designed to withstand the punishment meted out by many of Iona's larger customer deployment environments. Customers demanded fixes and faster resolve times.

The code's degradation further accelerated when Iona removed most of the senior development staff from the maintenance team so that they could develop new products. Finally, there was limited acceptance of the well-documented style guides that Iona's chief architect specified for all code development, making it difficult to read the code. Poor structuring of the source code's directory space also made it difficult to quickly become familiar with the overall code structure.

Team morale

People indicated in reviews that they didn't feel cohesiveness in the team. Many reported

poor visibility of the projects on which people worked. In general, we felt underappreciated and overworked.

Initial history of change

In 1999, prior to the release of Kent Beck's *Extreme Programming Explained*,¹ Iona undertook several projects to address its problem areas. It is in the context of these projects that it introduced Extreme Programming, realizing that many of the things it was implementing were elements of XP.

Reengineering

The second reengineering effort initially focused on resolving problems—which many customers reported—related to poorly implemented modules. Bug clusters highlighted in our defect-tracking system clearly identified these problem areas. Today, this reengineering effort continues as a part of our XP effort, which helps engineers resolve problems. An additional outcome of the initial effort was reducing the code's complexity by stripping out unused code and implementing several patterns that have made it easier to maintain, test, and understand the code. We've reduced code size by over 40 percent.

Improving engineering practices

Jim Watson, one of Iona's distinguished engineers, led the reengineering project. In addition to providing strong technical insight into coding problems, he was also instrumental in promoting stronger engineering practices. He initiated several practices to encourage growth in the engineering team, including weekly presentations that gave everyone an opportunity to present technical topics of relevance or interest such as merge strategies, patterns, and estimation. He emphasized code reviews, adherence to source-code management guidelines, and ownership of and responsibility for code style. He made engineers consider how they could constantly improve the quality of their work in areas such as test coverage, and he established a more proactive approach to problem solving. His practices still strongly influence the team.

Automating everything

We established a separate project to clearly understand the build and test dependencies across the older product set's various

We were suddenly on the road to a consistent and automated build, test, and release infrastructure for nightly product builds and tests.

elements and to fully automate the build and test process. This is an ongoing project, but so far we've been able to build and unit test the complete product set on a nightly basis. Efforts continue to better automate the interoperability and system tests, which still require some manual intervention.

Team consolidation

Before starting the reengineering projects, over 70 engineers maintained and enhanced the Orbix products. By the time we completed the major portion of those projects, we had reduced the team to around 40 but were servicing three times the number of customers. Today, the team size is down to 25. In addition to the personnel consolidation, the team managed a single mainline of code using a well-defined set of common rules that govern how to merge fixes and enhancements into the consolidated mainline.

Extreme maintenance

The second reengineering effort resulted in a remarkable transformation—we were suddenly on the road to a consistent and automated build, test, and release infrastructure for nightly product builds and tests. The reengineering and refactoring efforts eliminated much of the code complexity and stagnation, resulting in a clean, well-structured code base that conformed to Iona code standards. So, if we saw so much improvement, why consider XP?

Despite our progress, we had yet to resolve issues of testing, visibility, morale, and personal work practices. From a management standpoint, we wanted to do more with less: higher productivity, coupled with increased quality, decreased team size, and improved customer satisfaction. The engineers wanted more time to do a good job instead of always feeling pressured to deliver fix after fix. So we started looking at XP and learned that many of its elements come naturally to teams working the maintenance game. According to Kent Beck, "Maintenance is really the normal state of an XP project."¹

In our earlier projects, we created a set of maintenance processes that describe how a bug is created, prioritized, analyzed, fixed, and delivered to the customer. The Generation 3 team also had a set of processes that explain how to request, describe, prioritize, and implement incremental enhancements

and deliver them to customers. Clearly, these processes already incorporated important XP elements—*extreme maintenance* means following the XP model while a product is in the mainstream of its product life cycle.²

The synergy was so great that we started to use the term XP to describe the set of processes and practices the Generation 3 element and product teams used. We then started the XP experiment in earnest in an attempt to resolve our outstanding issues. Table 1 presents all of the practices presented in *Extreme Programming Explained*¹ and describes how we approached them.

Metaphor

When customers report a bug, they describe it in the form of a short story that explains what they are trying to do and how the problem manifests itself. We then note this story in our defect-tracking system and make it available to the customer and everyone working with that customer.

When we receive an enhancement request or functional requirement, we present it as a specification on our requirements Web page. Because of the need for global visibility, and because customer service representatives and engineers must be able to update the system, unlike the bug reports, these requirements are not presented as index cards and don't read as stories (yet). However, they do contain essentially the same information as traditional requirements. For the purpose of our XP experiment, we left the enhancement request until a later improvement phase and instead focused on tracking the bug-fixing stories in a well-structured Web-based queuing system, visible to the internal customer (or Customer Service), engineers, and engineering managers.

We liked the idea of using index cards to track tasks and decided to implement a storyboard for each of the element teams comprising our consolidated Generation 3 team. Our goal was to improve the poor visibility of each engineer's work effort. We wrote each task, regardless of whether it was a customer issue or internal project task, on a color-coded card (see Figure 1). Nirmalya Sengupta, our Generation 3 operations lead, then sat down with our customer (or Customer Service representative), so he could prioritize his list of bug and internal tasks. The customer could also view internal tasks and ask to have his tasks escalated above internal tasks. Cur-

Table 1**Extreme Programming practice adoption**

Extreme Programming	Adoption status	The Orbix Generation 3 experience
The planning game	Partial adoption	We haven't found it necessary to follow a strict iteration planning process. However, to understand and manage capacity, it is still critical to monitor estimates of ideal time and actual time to completion. Most bugs in our product take about two weeks to fix, so we fell into two-week patch release iterations. Our customer service and sales organizations act as our customer proxies, providing a coordinated twice-weekly prioritization of bug issues for each patch. They also run the test cases that are our acceptance tests for each customer story against each patch delivered to a customer. We still get interruptions and midcycle reprioritization, but it's unavoidable.
Small releases	Followed religiously	Short release cycle requirements drive our support efforts, so our customer patch cycle is two weeks. Our point release cycle has moved from months down to weeks. Without automated nightly regression and unit testing, we wouldn't have come this far. Without a single button integration test capability, engineers wouldn't have been able to ensure that changes had not broken code. Without optimizing our test suites around test time, we wouldn't have been able to provide the necessary timely feedback.
Metaphor	Adopted from start	Our metaphor is the customer story as detailed in our bug-tracking system. Our customer supplies our acceptance test for each story.
Simple design	Partial adoption	There hasn't been as much focus on simple design, because the product is complex, and we continue to use high to midlevel design documents to help people quickly understand the over all system and some system components. In our maintenance environment, there is a clear story on which to focus. We tend not to generate design documents for fixes but instead focus on implementing the story and passing the acceptance and regression tests.
Testing	Adopted from start	Test first is naturally a part of the maintenance process in the Generation 3 team. We don't work on bugs unless a test case is developed and all test cases are run nightly. We also use code reviews to back up the lack of great test coverage.
Refactoring	Adopted from start	We took this on as one of the cornerstones of our efforts in improving code maintainability and stability. More often than not, the call is to focus on refactoring as a part of an engineer's personal practices. Sometimes in refactoring, we have found ourselves doing wholesale reengineering, but this doesn't happen often.
Pair programming	Sparingly adopted	Although we've tried it, it is not something the team widely practices. We've used code reviews to try and address code standards, design of fixes, and so forth.
Collective ownership	Adopted from start	We have people changing code everywhere. Some have a stronger knowledge of certain areas, but we have fewer people maintaining the code and we can't afford not to have people working on all parts of the code. For the most part, we've used pairing to gain an edge. We also have a strong set of code standards we enforce not only at code reviews but also as a part of the check-in process in our source-control system.
Continuous integration	Adopted from start	It took us 18 months to build an automated testing system that was comprehensive but also easy to use and fast. Also, we don't use integration stations. Instead we use a mutex file (requiring check in and check out) in our source control system to ensure that no one else is integrating while we are merging into the mainline. Our merge process is strong and followed quite well.
40-hour week	Not adopted	We haven't felt courageous enough to tackle this.
On-site customer	Adopted from start	We use our customer service team and sales force to act in the role as a customer. They set priorities and generate acceptance tests.
Coding standards	Adopted from start	We had already started this one when we began using XP.

rently, only customer issues are rigidly prioritized, but we intend to integrate internal tasks into the prioritization process as well. (Kent Beck suggests having a Web-accessible digital camera positioned so that people can regularly look at and zoom in on the storyboards—but that's for a future project.)

An engineer estimates the completion date after 24 hours of initial analysis. When the

task is completed, the engineer records the actual date of completion, and if there were delays, she notes reasons for the delays, observations, and lessons learned on the back of the task card. We remove all closed tasks from the board, extract the data, record it in a spreadsheet, and store the task card in a task log.

In general, bug-fixing tasks should be completed as two-week pieces of effort. If, after

Pair programming in XP is critical to improving collaboration and greatly facilitates mentoring and improvements in engineering practice.

analyzing an issue, an engineer estimates more than two weeks' worth of work, she splits the story into several tasks. Anything more is a refactoring project and should be structured more along the lines of an enhancement with clear incremental delivery of functionality based on tasks extracted from the original customer story.

We also incorporated the daily stand-up meeting into the team (see Figure 2)—an XP element. Each team took 15 to 30 minutes to review their progress, and each person took a couple of minutes to talk about what he or she was working on that day and the previous day. The focus was not only on improving visibility but also on encouraging communication between team members. The qualitative results were immediate. Some engineers found it difficult to explain why they spent so much time on issues or why they looked at things that weren't high priorities on the team's list of tasks.

Testing

Automation is essential, and one of the first initiatives we undertook in 1999 was to automate our test suites and speed them up. We completed an extension of that effort in 2000, which lets developers build and test the entire product set on any platform combination we support by clicking a button from a Web-based interface or using the workstation command line. These initiatives let us test the entire product set against the Corba specification, the Orbix API, and customer-supplied test cases on a nightly basis or at the engineer's discretion on the 17 platforms we support. Both the stories describing the functionality that the software supports and those describing the bugs that customers encountered have tests associated with them. Engineering requires each customer to provide a test case for each reported bug prior to doing any work on the issue. This test case is automatically included in the test suite's nightly runs.

Any engineer must be able to build the entire product or system and run the test suite to get immediate feedback on changes made. Nightly builds are a reasonable intermediate step to this test-on-demand requirement, and again the processes we decided to follow map well to the testing idea at the XP model's foundation—test before you implement, and test everything often.

Pairing

Pair programming in XP is critical to improving collaboration and greatly facilitates mentoring and improvements in engineering practice. However, convincing engineers that it is useful and should be incorporated into their work practices is extremely difficult.

Our pair programming experiment came about accidentally in late 2000. The Generation 3 team worked on a particularly difficult set of customer issues. We had a customer engineer working onsite with us. (Having customers onsite is another XP principle, although normally we treat our onsite product managers and customer service representatives as our customers.) At different times, the customer teamed up with various members of the Generation 3 team to do some pair programming. Likewise, members of the team gravitated toward a pair-programming approach as they progressed with the issues on which they were working for this particular customer.

In the end, the qualitative feedback from the team was positive. They felt that they worked more effectively and with a higher level of communication as opposed to working independently on individual issues. Also, the senior staff enjoyed some of the mentoring that naturally occurred as they interacted with the junior members of the group. The overall productivity during this period was also high and morale improved. Engineers got a lot of positive reinforcement from each other, and engineering practices and quality improved as we actively noted coaching interactions between engineers. We also got no patch rejections for the work these engineers did.

We are currently trying to formally implement pair programming. Until we do, we have asked each engineer to take on one or two tasks each quarter in which they pair with another team member. We'll note the tasks in which they worked in pairs and will show them the data at the end of the quarter to

Task:		Details:	
Area:			
Customer:			
Engineer:			
Date queue	Date active	Expect finish	Date closed

Figure 1. The color-coded task card.

(hopefully) reinforce the results we got with our initial pair-programming experiment.

Small releases

In addition to describing functionality, XP focuses on increments of functionality that engineers can quickly merge into the mainline. They merge in bug fixes quite rapidly, and with the improved nightly build and test and the one-button test, they can make systems available as a temporary or permanent patch within one day. Any initial effort should focus on releasing a patch every iteration for a large, complex system—and automation and improved testing enable that end. For projects approaching XP at midlife, we encourage focusing on automation at any costs. Rapid turnaround time can only improve customer satisfaction. Unfortunately, because we just recently started collecting metrics on mainline breakage, we don't have statistics to show what is currently a qualitative analysis of Iona's improvement.

In addition, because the code base we were working with was a legacy code base, we did not have much to say about how it was put together. However, we needed to make it maintainable and add functionality if required. As mentioned earlier, one of the improvement efforts we undertook was a major reengineering of elements of the Orbix product. This effort incorporated stories that typically took three to four weeks to implement. By keeping these increments relatively short, we could more effectively manage the delivery schedule of our last minor point release by modifying the project's scope but not its delivery date, because we were almost always at a point where we could deliver a functional product.

Refactoring

With regards to refactoring, analyzing bug statistics is important in identifying areas of code that might need refactoring or reengineering. This form of customer feedback is an important part of targeting improvements and stopping code entropy. Refactoring should always be on an engineer's mind when analyzing a problem or enhancement request: "How can I make this area of the code work better?" as opposed to "How can I quickly fix this problem (using the band-aid approach)?"

We addressed refactoring in our effort to improve personal practices. Recent code re-

views have revealed that refactoring has become part of the engineers toolkit when approaching a problem. Along with analyzing a problem, they now identify whether the area of code they are working on is a candidate for refactoring, and they follow up after delivery with a refactoring task on the storyboard. Sometimes this extends into a reengineering effort if the scope of improvement is increased, and for some customer issues, we can't afford the time to do the refactoring effort due to commitments to restore times. However, for the most part, we've seen fewer problems in our new releases, a significant decrease in code entropy (as measured by comments from code reviewers), no patch rejections over the last few months, and reduced code complexity (again, as measured by comments from code reviewers). Unfortunately, we have not been able to collect complexity metrics to verify this.

Facilities strategy

A maintenance team environment should combine personal and team space to enhance interactions and collaborations that should occur in the team. Although not an explicit XP principle, our facilities strategy continues to be key to XP's success. The pair programming could not have happened without changing the engineers' workspace, and the collective ownership, testing, and continuous integration would have suffered.

Prior to initiating our XP project, Iona's engineering workspace consisted of the typical floor plan—it had multiple bays, each con-

Figure 2. A stand-up meeting at the storyboard.



Figure 3. Orbix Generation 3 cubicles (a) prior to initiating the Extreme Programming project and (b) after.



taining six shoulder-height cubicles (see Figure 3a). The engineers and engineering managers initiated a project to restructure the space in which they worked to enhance team communication. They created a large central area by eliminating 20 percent of the individual cubicles and establishing nondelineated group workspaces. The group workspaces consist of either a round or rectangular meeting table or workstation area. In addition, white boards and flipcharts on tripods were made available in each work area. They also purchased a couch, and a catering service now delivers snacks on a weekly basis (see Figure 3b).

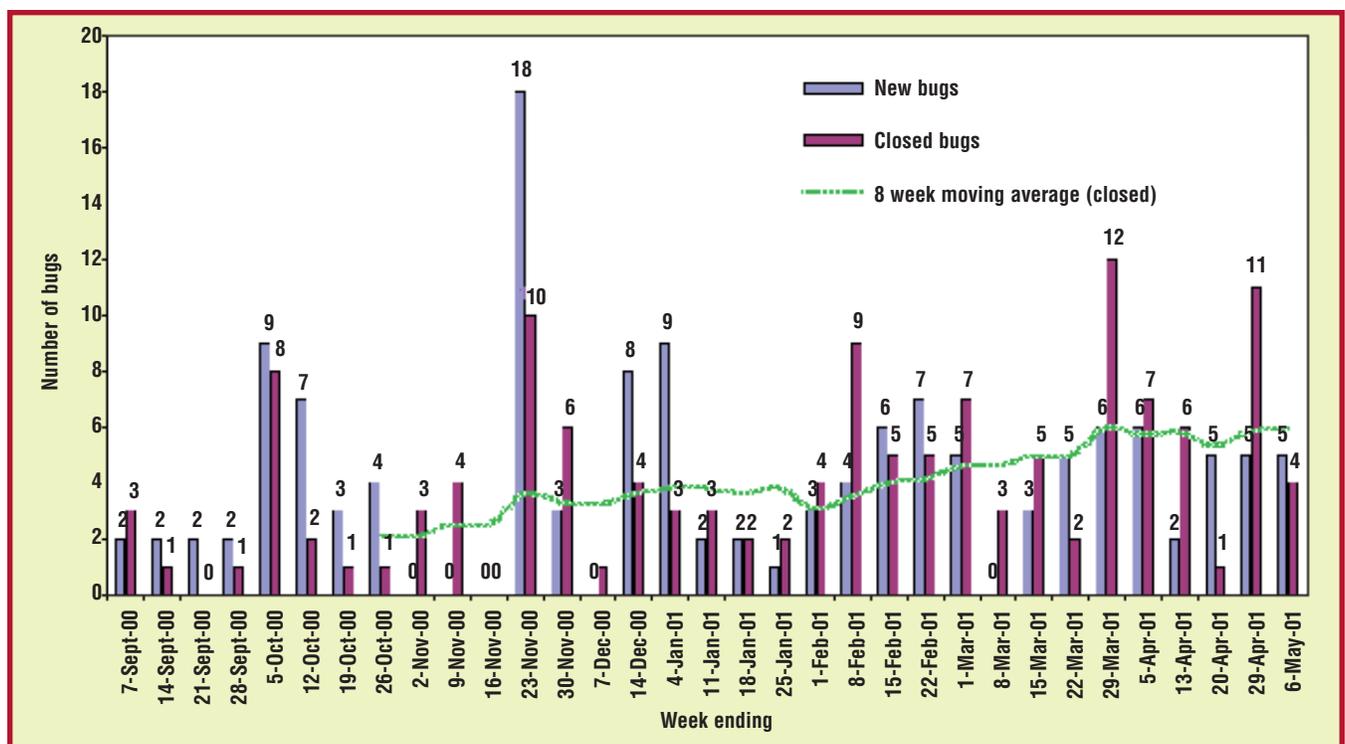
The results are noticeable. Code reviews happen in the group area; pair programming happens at the workstations. People discuss their ideas on the whiteboards and large flipcharts. There is a measurable increase in the visibility of what everyone is doing on the team and the number of conversations and

collaborations that occur. These changes to the environment were instrumental to our success in improving overall team morale. We also have had a visible increase in the number of team interactions—not only on code reviews but also regarding ongoing tasks.

Qualitative improvements

Our metrics (although not particularly extensive) seem to show some quantitative improvements in productivity during the period in which we used pair programming to address specific deliverables or bug fixes for a major customer. The productivity is based on a constant work force with no change in overall work habits in terms of hours spent fixing bugs. This productivity increase has continued beyond the initial experiment and is perhaps attributable to more people using pair programming and open collaborative spaces created during that period.

Figure 4. Team bug-fixing productivity.



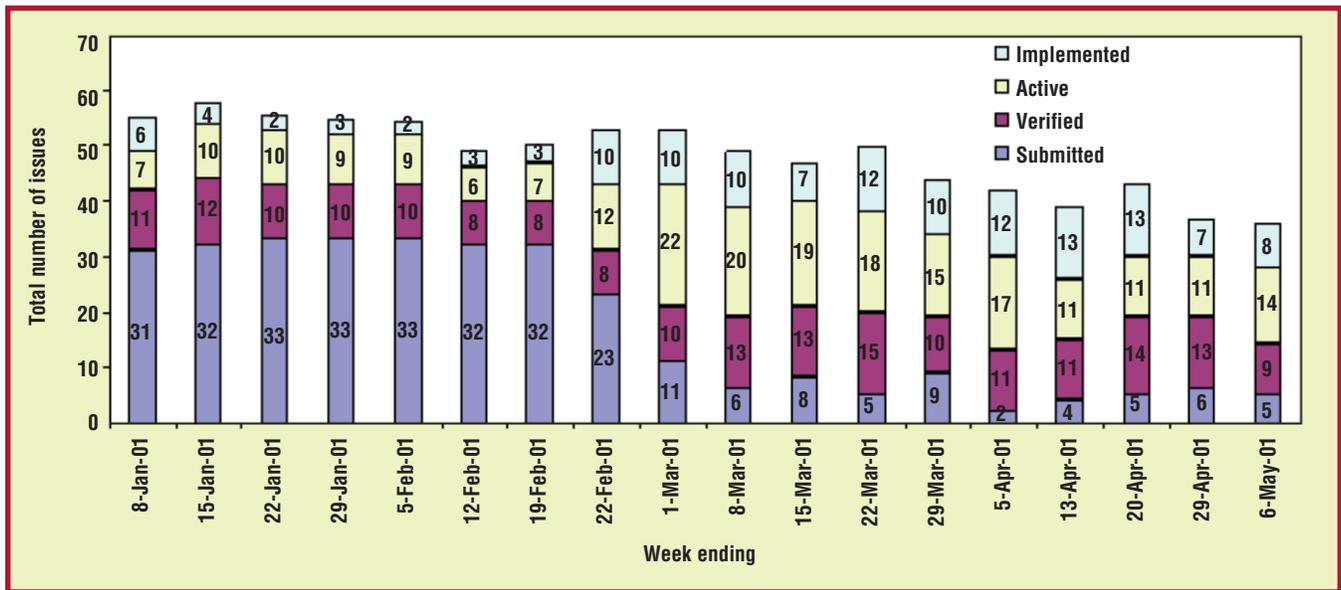


Figure 5. Workflow queues.

As Figure 4 shows, the month of November (the period of this effort) saw a measurable peak in productivity over and above the previous several months. This was an improvement of 67 percent, based on the next most productive five-week period, even though the total number of people-weeks missed due to holidays during the two most productive periods is relatively equivalent (Christmas in the later half of December and the Indian holiday Diwali at the end of October and early November). Looking at the trend line, we see continued improvement, even though the overall team size was reduced in March 2001 from 36 to 25.

Figure 4 shows several things. First, the new bugs represent issues that the team must address. An eight-week running average trend line would show that this averages out to four issues a week. Second, the number of closed issues represents the number of issues that are resolved in a patch to the customer. This patching cycle represents our XP iteration and is on the order of two to three weeks (note the closure peaks).

One of the greatest success stories is improvements in visibility. This is the greatest benefit to the team. Having a storyboard on which to daily prioritize tasks and discuss progress encourages best practices, lets people see what others are doing, and lets management gauge progress. Figure 5 shows the dramatic improvements in our workflow queues as a result of using the storyboard. When we installed the board in February 2001, it focused people's attention on issues that went unverified for significant amounts of time. We also saw a dramatic increase in the number of issues that people started to

actively work on. Visibility alone was a strong motivational factor in this turnaround. In the future, we hope to scale this practice across multiple development sites.

How can XP help us further improve? First, improving the pair programming initiative can improve our lack of cross-training among the code base's many modules. It is not a practice for which this is intended but is a useful side benefit to a real problem as the team grows smaller. Engineers have accepted the benefits, but we are still in the process of structuring a more general approach to ensuring that the practice of pairing becomes part of the Iona culture. Earlier this year, we proposed having an individual sign up for each new bug, enhancement request, or refactoring effort, making that individual responsible for grabbing a partner and working through the issue. Our technical leads felt comfortable with this approach and they are now using it as a part of the standard process.

Second, metrics are critical to the replanning game, but getting engineers to contribute is difficult. To plan how long it will take to complete a story, you must know how long it took to complete a similar piece of work based on the estimates of the engineer to whom the work is assigned. "As the basis for your planning, assume you'll do as much this week as you did last week."³ Kent Beck and Martin Fowler call this rule Yesterday's Weather, and we think it is an appropriate analogy.

We are currently running an internal project to improve our ability to enter metrics into our defect tracking system and report on them

ADVERTISER INDEX NOVEMBER / DECEMBER 2001

Advertising Personnel

James A. Vick
IEEE Staff Director, Advertising
Businesses
Phone: +1 212 419 7767
Fax: +1 212 419 7589
Email: jv.ieeemedia@ieee.org

Marion Delaney
IEEE Media, Advertising Director
Phone: +1 212 419 7766
Fax: +1 212 419 7589
Email: md.ieeemedia@ieee.org

Marian Anderson
Advertising Coordinator
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: manderson@computer.org

Sandy Brown
IEEE Computer Society,
Business Development Manager
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: sb.ieeemedia@ieee.org

Debbie Sims
Assistant Advertising Coordinator
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: dsims@computer.org

Atlanta, GA
C. William Bentz III
Email: bb.ieeemedia@ieee.org
Gregory Maddock
Email: gm.ieeemedia@ieee.org
Sarah K. Huey
Email: sh.ieeemedia@ieee.org
Phone: +1 404 256 3800
Fax: +1 404 255 7942

San Francisco, CA
Matt Lane
Email: ml.ieeemedia@ieee.org
Telina Martinez-Barrientos
Email: Telina@husonusa.com
Phone: +1 408 879 6666
Fax: +1 408 879 6669

Chicago, IL (product)
David Kovacs
Email: dk.ieeemedia@ieee.org
Phone: +1 847 705 6867
Fax: +1 847 705 6878

Chicago, IL (recruitment)
Tom Wilcoxon
Email: tw.ieeemedia@ieee.org
Phone: +1 847 498 4520
Fax: +1 847 498 5911

New York, NY
Dawn Becker
Email: db.ieeemedia@ieee.org
Phone: +1 732 772 0160
Fax: +1 732 772 0161

Boston, MA
David Schissler
Email: ds.ieeemedia@ieee.org
Phone: +1 508 394 4026
Fax: +1 508 394 4926

Dallas, TX
Royce House
Email: rhouse@houseco.com
Phone: +1 713 668 1007
Fax: +1 713 668 1176

Japan
German Tajiri
Email: gt.ieeemedia@ieee.org
Phone: +81 42 501 9551
Fax: +81 42 501 9552

Europe
Glesni Evans
Email: ge.ieeemedia@ieee.org
Phone: +44 193 256 4999
Fax: +44 193 256 4998

Advertiser	Page Number
Boston University	15
California State University Northridge	6
Compaq	Cover 4
Eaton	12
John Wiley	Inside Front Cover
Monmouth University	16
ParaSoft	5
Requirements Engineering Conference 2002	1

IEEE Software

IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, California 90720-1314
Phone: +1 714 821 8380
Fax: +1 714 821 4010
<http://computer.org>
advertising@computer.org

automatically. The volume of issues makes it difficult to track using a simple spreadsheet, but we do it anyway. Our Operations Lead speaks to each developer and notes how they have progressed on the work they are doing. The limited metrics we have indicate that it takes an engineer on average two weeks to fix a bug.³ This matches the suggested length of time to implement a story in the XP model. By measuring the calendar time with the defect tracking system, we can measure the calendar time it took to resolve an issue and ultimately the team's velocity.³ This information is invaluable in planning what we will do in the next patch or release iteration. ☺

Acknowledgments

The authors thank Kent Beck for working with the Orbix Generation 3 team and the invaluable insights into the development process he provided that have motivated our teams to adopt XP. We also acknowledge the influence of ideas presented in *Refactoring: Improving the Design of Existing Code* (M. Fowler, Addison-Wesley, 1999) and *Extreme Programming Installed* (R. Jeffries, A. Anderson, and C. Hendrickson, Addison-Wesley, 2000). We presented an earlier version of this article at XP Universe in July 2001.

References

1. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Reading, Mass., 1999.
2. G.A. Moore, *Crossing the Chasm: Marketing and Selling High Tech Products to Mainstream Customers*, Harper Collins, New York, 1999, p. 47.
3. K. Beck and M. Fowler, *Planning Extreme Programming*, Addison-Wesley, Reading, Mass., 2000.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

About the Authors



Charles Poole is a senior engineering manager at Iona Technologies. His research interests include Extreme Programming in large distributed development environments and self-assembling software paradigms. He received his BSE in aerospace engineering from the University of Michigan and his MSE in astronautical engineering from the Air Force Institute of Technology. Contact him at charles.poole@iona.com.

Jan Willem Huisman

is currently working with Iona Technologies in Dublin as an engineering manager responsible for the maintenance team for the Orbix Generation 3 product. He has extensive experience in software quality control and software development. He has also been involved in the entire software development cycle in the areas of programming, technical design, testing, project management, and maintenance. Contact him at jan.willem.huisman@iona.com.

