Continuous Learning

Joshua Kerievsky
Founder, Programmer
Industrial Logic, Inc.
2583 Cedar Street
Berkeley, CA 94708
+1 510 540 8336
Joshua@industriallogic.com

ABSTRACT
Continuous learning is part of eXtreme Programming's spirit, implied in its values, and implemented, to a certain extent, in its practices. I've learned that to be really good at XP, teams can go even further with their practice of continuous learning. In this paper I describe specific continuous learning tools, including learning repositories, study groups and iteration retrospectives, which apply to programmers, coaches and entire XP teams.

Keywords
Learning, continuous learning, learning repository, study groups, retrospectives, iteration retrospectives.

1 INTRODUCTION

The spirit of continuous learning is at the heart of eXtreme Programming. Customers and developers learn continuously from iteration planning; developers learn continuously from pairing, swapping pairs, testing and refactoring; coaches learn continuously by communicating with everyone; and entire teams learn continuously from feedback generated by the XP process.

Thus, while continuous learning isn't a stated value or practice of XP, it is inherent to XP. In practice, this means that XP teams and individuals on those teams gradually learn and improve.

Experiencing the pace of these learnings led me to look for ways to shorten the learning curve. I discovered that by using a few powerful learning tools, a team could improve at a much faster rate.

These tools include using a learning repository and conducting regular technical study groups and iteration retrospectives.

2 TEAMS TAKE LEARNING FOR GRANTED

Since continuous learning isn't an articulated value or practice, customers and developers often take learning for granted. It's something that is just supposed to happen.

You might imagine that if a team truly appreciates the XP values of Feedback and Communication, then continuous learning will result. But in practice, this doesn't necessarily happen. Either teams don't associate Feedback and Communication with continuous learning, or they don't reflect enough to realize that they need to learn. This is understandable. Imagine if XP's 40-hour work week principle were not articulated but only implied. Do you think teams would strive to work just 40 hours?

I've noticed that XP teams often miss the chance to learn in ways that could significantly improve their performance. XP teams are very code-centric and focused on making functional software. When reflection and learning happen, it's often in a watered down, haphazard way.

Learning on an XP project today can be a bit like the practice of refactoring was before Kent Beck described it as an XP practice and turned up the knob on this practice to 10. Prior to XP, programmers would refactor their code when they felt like it, or maybe after code was shipped or released, but not *all of the time*. By articulating refactoring as a practice and defining the importance of doing it continuously ("mercilessly"), XP challenged programmers and teams to improve their process.

3 ECONOMIC INCENTIVE FOR LEARNING

Continuously refactoring, like all of the other XP practices, can be shown to have a direct effect on the economics of software projects. If a team refactors continuously, their code will be easier to understand, extend and maintain. As a result, the team will be more efficient, and that will allow them to get more done in less time. Bang, there's your economic incentive.

So is there also a direct economic incentive for practicing continuous learning? You better believe it.

Judy Rosenblum, who spent five years as Coca-Cola's chief learning officer and three years as Coopers & Lybrand's vice chairwomen for learning and education, says that learning *must* be connected directly to business. Organizations have to make learning a strategic choice. And to make that happen, organizations need leaders who see how important learning is to the continued health and success of their organizations. Such individuals must

effectively embed learning into their organization's processes and projects, as Rosenblum explains:

> Someone has to decide to make learning not just an individual experience but a collective experience. When that happens, learning isn't just something that occurs naturally – it is something that the company uses to drive the future of the business. [1]

You might think this is completely obvious – of course organizations need to keep learning! But is learning a main topic in executive meetings? Is it believed to be as important as marketing, sales and human resources? Most organizations would like to say, "Yes, we value learning," but in practice, they don't. They hope teams will learn, and they send people to training classes a few times a year, but they don't understand that continuous learning can have a huge impact on their bottom line. Instead, they overemphasize action.

Judy Rosenblum addresses this oversight:

> The sense of urgency creates a bias for action. And that, in turn, prevents organizations from taking the time to learn. You have this phenomenal asset – your organization's collective experience – but this bias for action keeps you from focusing on it. [1]

4 A BIAS FOR ACTION

XP teams, especially new or inexperienced ones, are often too action-centric. Customers want to keep producing stories and writing acceptance tests, while developers want to keep testing, coding and refactoring. But when does the process improve? Don't the customers want to get better at what they do: writing stories, planning, interacting with *their* customers, communicating with development, trimming fat from stories and tasks? And don't developers want to improve at refactoring, pair-programming, design, automated testing, patterns, integration, or the simple art of knowing when to ask for help?

Certainly they do. But do they make time to improve? You might think that they don't need to, that learning will just happen over time. But I've been frustrated by the slowness of this process, which is largely due to the lack of time devoted to group learning.

For example, I can learn three hugely valuable things in one day, but my team isn't going to know about these learnings because the process doesn't include time for sharing them.

You might argue that XP does include time for sharing, since XP advises that teams conduct daily stand-up meetings, in which participants physically stand up and give summaries of what they're working on and how they're doing. Isn't that a good time for sharing learnings? Absolutely not.

Stand-up meetings are meant to be quick events – they aren't appropriate for conducting learning sessions, in which reflection and dialogue are requisite. So when would be an appropriate time? More to the point: "What is the simplest, most cost-effective way to share learnings?"

5 A LEARNING REPOSITORY

My preference is to use a simple, security-free, browser-based *learning repository*, such as Ward Cunningham's Wiki [2]. I say security-free because I've seen tools that have too many security bells and whistles, and I've seen how no one enters content into these tools simply because they are too burdensome to use.

So your learning repository must be simple to use, but just installing it and asking folks to use it isn't enough, either. Teams need to establish usage conventions. For example, a team can decide that developers will quickly jot down learnings on index cards as they work, and when they integrate their code, they can integrate significant learnings as well. Doing this will rapidly produce a valuable learning repository. Here are just a few examples of what a team might record:

> *Database Layer XML Refactoring* (Jan 27, 2001) While working on the new XML framework, Eric and I discovered that the database layer had been given new responsibilities that really didn't belong there – the mixture of responsibilities complicated the original design. So we've refactored the XML code out of the database layer, and placed it into the new XML framework code. –Bob
>
> *Tapping Your Finger: A Pair-Programming Technique* (Jan 30, 2001) I discovered that instead of annoying my pair by telling her that she missed something, I can just tap with my finger on the offending spot in the code and give my pair the chance to figure out what was missing or incorrect. My pair, Mary, really liked this. It could be a good pair-programming technique for everyone on the team. –Sandra
>
> *Getting Stuck & Unstuck Thanks To The Customer* (Feb 1, 2001) Today Karen and I discovered that the task we had signed up for was actually way more complicated than we'd thought. We asked Rob (the coach) for help, and that triggered a 10-minute meeting with the customers, which resulted in a great idea from Jim (a customer) about a far simpler, better implementation. It sure helped us to ask a question rather than continuing to program. –Jerry

6 GROUP LEARNING

Once teams produce enough learning content they will need to reflect on and discuss it. There are two good places to do this: for technical matters, the best place is a programmer's study group; and for team, people or process matters, the best place is in an iteration retrospective. As you will see later, study groups and

retrospectives are powerful learning tools, both of which take time away from programming. Some may worry about this lost time. This is fear talking, saying "we have to act, we don't have time to learn or reflect."

Such fear is quite common on XP projects, particularly when it comes to refactoring. Under pressure, many developers will skip refactorings to go faster. They don't yet know that this will eventually slow down the entire team as the code becomes bulkier and more brittle.

It's not easy to understand that you have to slow down in order to go fast. Taking time to refactor seems as if it may slow you down, but it will actually make you go faster. Taking time to reflect and learn may also seem as if it's slowing you down, but it, too, will make you go significantly faster.

Nevertheless, a coach or team may still be uncomfortable taking the time to conduct group learning sessions because, unlike refactoring, this work doesn't have a directly visible effect on the code. But while the effect of group learnings on the code is indirect, it is nevertheless highly beneficial.

For example, I once worked with an XP team that had experienced a few bumpy iterations. They had not been refactoring enough, and after these bumpy iterations, it became harder to implement new code, given the heavy accumulation of code smells. One day I discovered a particularly potent design smell and pointed it out to my pair. He said that he had known about that problem for a few months. This alarmed me. I wondered, was he the only one who knew about this? Did other programmers or the coach know about it? What other potent smells were out there but unknown to the entire team?

It was clear to me that every programmer on the team needed to at least be aware of the system's potent smells. This would enable them all to pay attention to these smells and consider how to refactor them out of existence. So we began a process of documenting these potent smells on index cards, which we stacked on the group table. Doing this work enabled the group to learn, and those learnings eventually led to direct action.

7 LEARNING CAPABILITIES

But not everyone on a team will be able to spot particularly potent smells, or even know what to do with them once they are spotted. There is a capability issue here.

What if a system was originally designed to let Java directly output HTML, but it is clear to a few programmers that this approach is far from ideal? And what if no programmer on the team knows how XSLT could replace the Java/HTML code to radically simplify the system? Well, given the burdensome nature of this Java/HTML code, the team might try to refactor it a few times. But without coming up with an entirely new approach, the code will continue to be a burden.

Okay, what if one person on the team does happen to know about XSLT? Then the team has a chance to greatly simplify their system. But how did this one person know XSLT? Perhaps this person is a continuous learner, someone who regularly reads industry magazines to stay up on new technology developments. It's a good thing for the team that at least one person happens to be a continuous learner.

But this is certainly far from not optimal. I want teams to continuously learn, because doing so will help them produce simpler systems, faster than ever. Peter Senge, author of the profoundly important, best-selling book, "The Fifth Discipline: The Art & Practice of the Learning Organization," had this to say about team learning:

> Most of us at one time or another have been part of a great "team," a group of people who functioned together in an extraordinary way – who trusted one another, who complemented each others' strengths and compensated for each others' limitations, who had common goals that were larger than individual goals, and who produced extraordinary results. I have met many people who have experienced this sort of profound teamwork – in sports, or in the performing arts, or in business. Many say that they have spent much of their life looking for that experience again. What they experienced was a learning organization. The team that became great didn't start off great – it *learned* how to produce extraordinary results. [3]

The two most powerful learning tools that I suggest for use by XP teams to support the practice of continuous learning are study groups and retrospectives.

8 STUDY GROUPS

I'll begin with study groups. You may be amazed, as I often am, that there are programmers today who have never read Martin Fowler's book, "Refactoring: Improving The Design Of Existing Code" [4]. There are even programmers on XP projects who have never studied the refactoring catalog in this book, even though they are supposed to be refactoring all the time! Martin's book is one of those hard-cover classics that everyone is supposed to read, but don't because they perceive it to be too imposing or hard to understand, which is far from the truth.

So if programmers on XP projects don't know the refactoring catalog, how good do you suppose they'll be at refactoring? Of course, they can get better by pair-programming with developers who *do* know the refactoring catalog, but that can be a slow process, which may still fail to introduce them to important refactorings. In addition, if these programmers don't know anything about other areas of software development, like design patterns and good domain modeling practices, how good do you think they'll be at building a well-designed

system?

To continuously improve programmer's technical abilities, I recommend study groups. A programmer's study group will meet regularly in a comfortable place to delve into important technical topics. These topics can come from the group's learning repository, from books or articles, or even from a guest participant.

Ken Auer's XP company, Role Model Software, allocates time once a month for technical group learning sessions, which Ken calls "startegic focus time."

Attendance in a study group is optional, but having the meetings regularly, such as once per iteration, is vital. I recommend that groups meet for two hours if they want to delve deeply into a subject, though one-hour meetings are fine for covering topics quickly.

There are roles to be played in a study group and certain important safety rules and rituals to follow. Absolutely no one should play the role of lecturer or teacher in a study group. The group meets to conduct *group* learning. If someone is expert in a certain technical area, that individual ought to help others learn, not show off or talk down to participants.

Those who find it burdensome to study important books or articles on their own may be surprised to discover that group study can make learning easier and more insightful.

If you'd like to start a programmer's study group and learn how to run it successfully, I suggest that you study my pattern language on this subject, which is called Pools Of Insight: A Pattern Language for Study Groups [5]. You might even begin your first study group by studying the patterns in that language, as several groups have done.

9 RETROSPECTIVES

Study groups address programmer's needs for continuous technical learning, but when does the entire XP team -- customers, developers and coach -- come together to reflect and learn about how to improve? The whole team gathers together during Release and Iteration Planning meetings, but the primary purpose of those meetings is planning, not learning. So what happens when something in the current process isn't working well? Too often, there is simply no time to air the problem, discuss and resolve it.

What is commonly missing is the practice of holding retrospectives. Norm Kerth, author of the book "Project Retrospectives: A Handbook for Team Reviews" [6], describes a retrospective as an end-of-project review, involving everyone who participated on the project in examining the project to understand:

- What happened,
- What the community could learn,
- What the community could do differently next time.

The continuous-learning approach to retrospectives means they come not at the end of a project, but at the end of every iteration. Conducting iteration retrospectives will enable teams to quickly adjust and improve their performance, because they will be continuously revisiting these questions:

- What worked well?
- What did we learn?
- What should we do differently next time?
- What still puzzles us?

Norm gives very clear guidelines for successfully conducting retrospectives. I'll do my best to summarize them here, but you'll probably enjoy reading his book, which is destined to become a classic.

Norm recognizes that people have a fear of retrospectives – because they have a fear of being attacked, of being made to look foolish, of getting a poor performance review or of hurting someone's feelings. Yet no retrospective can succeed if people are afraid, or if there is an atmosphere of blame, criticism, sarcasm or even humor at other people's expense. Therefore, Norm lays down specific ground rules that help establish a safe environment for conducting a retrospective. Perhaps the most important of these ground rules is Kerth's Prime Directive of Retrospectives, which states:

> Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand. [6]

Once the group understands these safety ground rules, it's time to break out some butcher paper. This is paper that is usually 30 feet long and 6 feet high. Norm likes to hang this stuff from the walls, break it up into sections of a timeline (for example, 3 sections could signify each week of a 3-week iteration) and then have teams go off and identify key events or things that happened during each section. People then add their identified events and happenings to the various sections of the timeline, which is next mined for stories and team goals. Norm suggests that professional facilitators help lead this process. In fact, he believes it is vital that the facilitator be an outsider and not a member of the team involved in the retrospective.

The final part of a retrospective is perhaps the most important. This is when the participants take the lessons learned during the retrospective and turn them into concrete ideas for improving their development process. This is hard work. I would add that it is particularly hard on XP projects, since it is easy to think you've found a deficiency in the XP process, when all you've really found is a faux-deficiency. Chris Collins and Roy Miller describe how "process smells" can be identified during

retrospectives, and they advise people to be careful about how they choose to fix them:

> The key to retrospectives is to make sure you are solving the correct problem. Sometimes the tendency is going to be to add a practice to the process, where the real problem is in how you are implementing one of the twelve practices. [7]

So how much time should it take to run one of these iteration restrospectives? Clearly, if we spend too much time on them, we'll loose vital development time. I asked Norm about this, and his answer surprised me. He said that even for a 3-week iteration, he would begin with a retrospective that lasts 2.5 days. I thought that was excessive, but Norm explained that groups need to learn how to do retrospectives. When beginning to perform retrospectives, they need lots of time. As time goes on, they will get better and better at it, until it takes perhaps only a half-day. I marveled at the simple good sense of this advice: Take time early on to get good at doing retrospectives, and you won't need much time to do them in the future.

10 A CONTINUOUSLY LEARNING COACH

We've talked about ways for the programming team to continuously learn and ways for the team as a whole to continuously learn, but what about an XP team's coach?

It is critical that an XP Coach be a continuous learner, since the coach is the leader of the team. If the coach doesn't value learning, the team won't either.

A coach must lead by example. This means that the coach will seek out and obtain coaching and mentoring from the best sources available.

The coach must also ensure that continuous learning happens on a regular basis. And it can take quite a bit of courage to not cancel a programmer's study group meeting in the face of a looming release date.

Coaches must strive to learn about their customer's needs, team or personality conflicts, new technologies, and the latest wisdom about XP and other lightweight methods.

I recently learned of an excellent continuous learning technique for coaches from Rob Mee, who is an XP coach at Evant, a merchandise management company in San Francisco. Rob has learned that the best way for him to continue to learn about the system his team is building is to program. So Rob programs to learn, and says he now uses 50% of his time to do so.

11 CONCLUSION

Continuous learning isn't a new part of XP, but rather a core part of it, implied but not directly articulated. It is a practice that can help a good XP team rapidly become a great XP team. Try it and see what you learn.

ACKNOWLEDGEMENTS

I'm always indebted to my wife Tracy, who endures my writing stints and does a superb job of copy-editing. I'd also like to thank Norm Kerth for his interview, and Ward Cunningham and Eric Evans for reviewing this work.

REFERENCES

1. Webber, A. Will Companies Ever Learn? *Fast Company Magazine*, October 2000, 275-282
2. Cunningham, W. Web Site. On-line at http://c2.com/cgi/wiki?WikiWikiWeb
3. Senge, P. The Fifth Discipline: The Art & Practice of The Learning Organization. *Currency Doubleday*, 1990, 4.
4. Fowler, M., Refactoring: Improving the Design of Existing Code. *Addison-Wesley*, Reading, MA, 1999.
5. Kerievsky, J., Pools Of Insight: A Pattern Language for Study Groups. Web Site. On-line at http://www.industriallogic.com/papers/kh.html
6. Kerth, N. Project Retrospectives: A Handbook for Team Reviews. *Dorset House*, New York, NY, 2001.
7. Collins, C., Miller, R., Adaptation: XP Style. Submitted as a paper to XP 2001.