Adaptation: XP Style

Christopher T. Collins
Senior Software Developer
RoleModel Software, Inc.
342 Raleigh Street
Holly Springs, NC 27540 USA
+1 919 557 6352
ccollins@rolemodelsoft.com

Roy W. Miller
Software Developer
RoleModel Software, Inc
342 Raleigh Street
Holly Springs, NC 27540 USA
+1 919 557 6352
rmiller@rolemodelsoft.com

ABSTRACT
Others in the XP community (Beck, Fowler, et al.) have talked about the need to adapt XP to local conditions. They have addressed the "why" question in some depth, but not completely. They also have not addressed the "how" question beyond rather general statements. We will attempt to complete the "why" and to answer the "how."

Introspection is an internal habit that every member of the development team needs to have. They should be thinking about how their process could be better. When they come upon what we call a *process smell* (i.e. something that just does not feel right about their process), this is a signal to think about what the root cause might be. The practice of retrospectives makes those thoughts public. The combination of these two allows a team to adapt XP in a way that is consistent with its values, principles, and practices.

Keywords
Adaptation, introspection, proactive retrospective, process refactoring, process smells, process spike, reactive retrospective, XP.

1 WHY CHANGE YOUR PROCESS?

People and circumstances vary. Software projects should embrace this principle. Software projects are composed of people. The context (business priorities, politics, etc.) will vary. This means that no two software projects can be exactly alike. Heavyweight methods ignore this. They want to make people into replaceable parts and projects into scripted plays so that software development will be predictable. Unfortunately, they overlook the fact that people make software, and people are not predictable.

People developing software learn as they progress. Not incorporating new knowledge that improves results is rather silly. If a team notices that something in their development process is causing pain, they need to make adjustments.

Not changing a process that doesn't produce what you want is risky. Sooner or later, the problems will build to critical mass and slow you down. You need to remove the cruft from the process, just like you remove it from the code.

So, change is required. But how do you do it? The problem with much of the material written about XP to date is that it does not address this question well enough. We need to know the goal of change, and the path to get there.

2 XP MATURITY LEVELS

We need to define the way we look at XP. Beck has made a start by saying there are three levels of XP maturity:

1. XP (Out of the Box)
2. Adaptation
3. Transcendence

XP "out of the box" refers to doing XP in what Martin Fowler calls its "book form" [2]. Do it as written. After a few iterations, start to adapt it as necessary. At some point, you will stop caring about whether you are doing XP or not. The name becomes less important than the results, and the process becomes your own. XP "out of the box" is just the simplest process that could possibly work. Thus, it's the best place to start. As Fowler says, "Fundamentally, it's easier to change a small thing by adding bits, than to change a large thing by taking bits away" [2].

The goal is not to move through the levels as quickly as possible, but only when you need to. Process improvement for its own sake is an instance of YAGNI (You Aren't Going to Need It).

XP needs to offer some guidance about what mechanisms a team should use to move between the three levels of maturity.

3 THE CHANGE AGENT

Heavyweight methods suggest that process-focused groups outside the project should dictate proven change strategies to the project team. XP says that the team should adapt the process from within to produce the best possible results. Nothing drains a team more than having change dictated to it. For a team to be as effective as it can be, the people in it have to accept responsibility for improving their own process. How do they do that?

Fowler says, “The first part of self-adaptivity is regular reviews of the process. Usually you do these with every iteration.” He recommends further that more formal process reviews take place in 2-3 day offsite “retrospectives” run by a trained facilitator. He also says that “reviews are neither emphasized, nor part of the process, although there are suggestions that reviews should be made one of the XP practices” [2]. This doesn’t go far enough.

The way you should go about changing XP is to turn XP on itself, using the discipline’s values, principles, and practices to adapt it to your local conditions. If attempted changes to a process are inconsistent with the values of that process that are already in place, those changes will not work. What we need is essentially an “XP way” to change XP. We recommend that you start with three things:

1. a baseline for comparison
2. the value of introspection
3. the practice of retrospectives

4 THE BASELINE

You have to do XP first before you can know how to change it to fit your particular context. It is premature abstraction to change it before you do it. Michael Potts of Beech Aircraft said, “The Wright brothers’ design…allowed them to survive long enough to learn how to fly.” XP “by the book” will let you build great software while you are learning how to do it even better. Treat XP “by the book” as your training wheels. Once you feel you have it down, remove them and make the process your own.

There is a good rule of thumb to apply when writing code: make it run, make it right, make it fast. XP “by the book” is much like writing good code. Do XP as written first. That is making it run. Adapt the process for local conditions. That is making it right. Move past the book form to make the process wholly your own. That is making it fast.

Make it run first to get initial benchmarks that tell you whether or not your changes are effective. If you make a change that breaks something, revert to the baseline. Note that this does not mean you should do things by the book when they simply do not make sense. The point is that you should try to modify as little as possible before you know definitively what works and what doesn’t in your environment. For example, if automating an acceptance test for a particular function of your user interface isn’t worth the effort and the money, don’t feel the need to do it just because that’s what XP out of the box would require. Be smart about it.

The question is, when you *do* need to change XP, how do you do it in an XP way?

5 THE VALUE OF INTROSPECTION

Stephen Covey talked about “sharpening the saw” as one of the seven habits of highly effective people. That’s what introspection is. It is people thinking about their process, vocalizing and refining their thoughts about it, and encouraging others to do the same. People have to develop the habit and mindset of introspection, just like they have to develop the habits of thinking simply, communicating openly and honestly with others, tempering optimism with feedback, and being brave.

The values of XP are an admission of human nature. People implement XP, and we humans have certain foibles in common, generally speaking. We tend to be selfish. We tend to be dishonest with ourselves and with others. We tend to make things more complicated than they need to be. We tend to get scared by things we don’t understand. We fear failure. Stress exaggerates these traits. The brilliant thing about XP is that its values recognize this tendency for humans to retreat to our baser selves. The values expose that tendency, and provide a foundation for practices that help people overcome it. But there is something missing.

There is a part of human nature that others have addressed implicitly, but have not articulated. Humans tend not to think beyond the ends of their noses. We tend to get tunnel vision. We tend to take the path of least resistance by doing the first thing that “works”. This is a defense mechanism. Traditional approaches to developing software have conditioned us to be afraid of change, because they make change painful, both personally and professionally. Other writers have recognized the pain of change. That recognition is the primary reason for XP’s growing success (it makes change less painful). What other writers have not done is recognize that the habit of *introspection* is what enables change.

People have to develop the habit of looking for *process refactorings* just like they look for code refactorings. Introspection is something that good programmers should be doing anyway. As Andy Hunt and Dave Thomas said in *The Pragmatic Programmer*, being pragmatic is “thinking beyond the immediate problem, always trying to place it in its larger context, always trying to be aware of the bigger picture” [4]. What we are suggesting is that introspection should be a habit that applies to process as well. The other values also apply to process, and they support introspection. You should listen to your current process, share ideas for making it better, make the simplest process changes that could possibly work, and proceed with confidence.

6 PROCESS SMELLS

All of this introspecting is good, but when do you know that an apparent process problem is worth doing something about? You know it when it smells.

A *process smell* is an indication that something is wrong. Something doesn't feel right, so you think about it in order to articulate the source of the problem.

Examples of Smells

For example, suppose you suddenly feel the need to schedule bug fixes into an iteration. Or maybe you feel the need for a process to track bugs. These things are not inherently bad, but this is a process smell. You probably do not need to add anything to the process here. A likely source of the problem is that your acceptance tests are weak. This could be letting bugs slip through at the end of iterations, and allowing developers to call stories complete before they are. If you were doing it right, some critical percentage of acceptance tests for the new stories would be passing before you declared those stories "done."

As another example, suppose you notice that pairs seems to be staying together for several days at a time. XP suggests that pairs ought to rotate more often than that. Something could be wrong here. This could indicate that tasks are too long, which keeps people from switching as often as they could otherwise. They want a sense of completion on certain tasks, so they stick with tasks that take too long. This could indicate the need to shorten tasks in the next iteration.

As a third example, consider pair programming. Ward Cunningham talks about reflective articulation, or the art of verbalizing what you're doing and where you're going when you pair program. This keeps your pair fully engaged and able to help. Sometimes you'll notice that one member of a pair isn't fully engaged like that. His pair might be madly typing away, not talking very much, while he falls asleep. We sometimes refer to this as the "drooling smell." This requires some process fixing. Maybe in the next day's stand-up meeting you need to emphasize that pairing means that both members need to be fully engaged. Then again, perhaps the problem is you. When introspecting on how you are performing, one item that should be on your checklist is how well you are pairing. If your pairs consistently have trouble staying engaged when you drive, maybe you need to work on your pairing skills.

Subjective Smells

Not all smells are directly observable like this. Some smells are subjective. If people are bored with their jobs, loathe going to work in the morning, call in sick frequently, or feel like they have to work nights and weekends to stay on schedule, you have a process problem. If only one person on the team feels or behaves this way, perhaps that is an individual issue that you need to work through for the benefit of the team. If many team members feel or behave this way, you had better fix the process.

The Catch-All Smell

Not all smells are specific enough to identify discretely. Some can be difficult to track down. Fortunately, you have a "catch-all" smell to help you out. This is a remarkable decline in your velocity. A close second is a sharp increase in your acceptance test failure rate. This is really just a warning of future velocity problems.

Most process problems show up in your velocity eventually. They can get there in different ways. Perhaps your defect rate suddenly goes way up, which forces you to spend lots of time repairing code that worked in the past. Maybe somebody isn't able to be as productive as they have been, which drags down his pair.

7 THE PRACTICE OF RETROSPECTIVES

Once you have identified a *process smell*, and you have used introspection to articulate it, how do you tell the rest of the team about it? Retrospectives allow people to make introspection public. Retrospectives come in two forms:

- *proactive*, to keep things running smoothly
- *reactive*, to fix an immediate problem

The first category should be a natural part of the XP rhythm. Code a little, test a little, integrate a little, think a little. It's a tune-up, rather than an overhaul. XP isn't all code – that's just what you focus on most of the time. *Proactive retrospectives* aren't a formal affair. They should happen in stand-up meetings attended by a group of programmers with the habit of introspection.

Proactive retrospectives should look something like this. Someone on the team has a good idea for how to do something more efficiently. He talks to his pair about it to get his thoughts straight. In the next morning's stand-up meeting, he brings it up for the group to comment on based on their collective experience. If others have tried it already and it didn't work, they can say why and prevent people from wasting time exploring it. If the idea has promise, he and his pair can do what we call a *process spike*. Much like a coding spike, a *process spike* is exploring a process change to see if it will work, and how. When the spike is done, he and his pair can report back to the team in the next stand-up meeting. If it worked, the rest of the team can integrate it into the process.

The second category is necessary when something in the process starts to smell. Remember that velocity serves as the barometer for how well the process is running. If it rises or falls significantly, that signals the need for a *reactive retrospective*. If your iterations are short, you probably will not notice velocity changes during an iteration (unless they are extreme). It is a better idea to fold *reactive retrospectives* into iteration planning itself. That is the most frequent practical checkpoint where such an exercise makes sense, as Fowler suggested in his paper *The New Methodology* [3].

Reactive retrospectives should look something like this. When the team gathers in a conference room to plan the next iteration, they begin by reflecting on the past one. This includes a short brainstorming session to answer the four questions proposed by Norm Kerth for project post-mortems [5]:

1. What did we do well?
2. What have we learned?
3. What can we do better?
4. What puzzles us?

Then the team comes up with ideas for how to preserve the good and get rid of the bad. They develop a short list of *process spikes* to conduct before the next iteration.

Fowler says that organizations also should hold a 2-3 day offsite meeting after every release (this is in keeping with Kerth's project retrospectives) [2]. It would be hard to justify doing them more frequently. The important thing to note here is that those meetings are not a substitute for lightweight retrospectives.

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.

8 PUTTING IT ALL TOGETHER

One of the authors (Chris) was working on a project that used two-week iterations. One developer noticed a pattern of decreased intensity and focus in the middle of iterations. In a stand-up meeting, he suggested that the team should try one-week iterations to see if it helped. The team discussed it and decided to try it for an iteration or two. After a couple iterations, the team reevaluated the results and decided that the overhead of starting one-week iterations wasn't leaving enough time to get anything substantial done. Two-week iterations just felt right. Like a good XP team, they reverted to two-week iterations.

This is an example of how a member of the team applied the habit of introspection to identify a process smell. He used proactive retrospective to vocalize his thoughts in a public forum. The team performed a process spike to test the suggested adaptation, and learned from the spike.

9 BEYOND XP

We have talked about turning XP on itself in order to adapt it to local conditions. XP makes it convenient to include them as a natural part of development. But the practices we've discussed here apply outside the world of XP.

Agile methods such as XP lend themselves to inclusion of introspection and retrospectives, and cannot function very well without them. They depend on the professional discipline and dedication of the people using them as a replacement for formalism. But even heavyweight methods can benefit from having their participants actively think about how to improve their process.

You should actively reflect on your process to see if there are ways you can improve it, regardless of the process you are using. This includes developing the habit of introspection, and including retrospectives in your process so that you can learn from the past to improve the future. No process is foolproof enough, or complete enough, for users of that process to turn off their brains.

10 CONCLUSION

Kent Beck included "local adaptation" as a principle of XP. Fowler expanded this idea when he talked about taking XP beyond the boundaries. We do not believe either went far enough. We hope this paper helps to fill in the gaps.

Some of the things we advocate here are extensions of existing XP practices (such as *process refactoring*). We have suggested several new things, but we believe they are fundamental, consistent with XP in its current form, and light enough to be implemented without distracting teams from their primary goal of writing good code.

Good software development teams should develop the habit of introspection, practice retrospectives, identify process smells, and perform process spikes to test out adaptations.

We suggest that introspection could be the fifth value for XP, and the retrospectives could be the thirteenth practice. However, we are not advocating an explosion of new values and practices for XP. You can vary the practices of XP within existing parameters. Sometimes, though, that doesn't go far enough. It is not wrong to add something new. However, if you're going to add something to XP, that new thing must be aligned with the existing values and principles. In particular, it must not burden the process unnecessarily.

ACKNOWLEDGEMENTS

Many of the ideas in this paper have been adapted from material by Kent Beck, Martin Fowler, and Norm Kerth.

Thanks to Kent Beck, Martin Fowler, Joshua Kerievsky, Ken Auer, and the rest of the RoleModel Software team for reviewing and commenting on this paper.

REFERENCES

1. Beck, K. *Extreme Programming Explained: Embrace Change*, Addison-Wesley (1999).
2. Fowler, M. *Variations On a Theme of XP*. On-line at http://www.martinfowler.com.
3. Fowler, M. *The New Methodology*. On-line at http://www.martinfowler.com.
4. Hunt, A. and Thomas, D. *The Pragmatic Programmer*, Addison-Wesley (2000).
5. Kerth, N. On-line at http://www.retrospectives.com.