

Learning by Doing: Why XP Doesn't Sell

Kay Johansen

Verio Inc.
1203 North Research Way
Orem, UT 84097 USA
801-437-7534
kjohansen@verio.net

Ron Stauffer

e-automate Corporation
831 East 340 South
Suite 100
American Fork, UT 84003 USA
801-492-1705
rstauffer@e-automate.com

Dan Turner

Independent Consultant
801-426-5122
dan_turner@usa.net

ABSTRACT

If there isn't much value in Extreme Programming, why is it enthusiastically (some might say fanatically) embraced by some people? Yet if there is value, why do so many others dismiss it so readily? We think there is real value in XP, but the value isn't learned until you are deeply involved in actually doing XP. You have to "try it" before you "buy it." Unfortunately there's a very high barrier to trying XP due to prevailing cultural assumptions in the software industry.

In our presentation, we will:

- Discuss our experience introducing partial XP on a small team over a year and a half period of time.
- Report on our perceptions and those of other team members and management after the project was complete.
- Offer our conclusions as to why some members of the project thought XP was a blazing success while others walked away without buying it.

Keywords

XP, extreme programming, experience report, implementing extreme programming, selling extreme programming

1 PROJECT OVERVIEW

Our story begins at a small software development company historically supported through private funding and product revenue. The company had a shipping product with a code base that had been developed over the previous couple of years by a small team of programmers. Due to their small size, limited market position and somewhat precarious funding, the company had acquired a mindset of saying yes to every customer request. They made a lot of promises but had few developers. As the promised features became backlogged, the need to ship something quickly overrode the desire to keep good product design and to perform adequate testing before each release.

Eventually, cash flow issues forced the company to restructure itself and seek venture capital funding. The slate

was in some measure wiped clean in many areas of the company, including development. A strong VP of Engineering was hired who championed the need for change and created a protected environment where the development team had the opportunity and freedom to experiment with new methods.

Motivation

The first author, Kay Johansen, had been following the XP discussion on Ward Cunningham's Wiki [1] for about a year, and was beginning to feel XP might offer solutions to the problems that had continually frustrated her on previous projects:

- contributing programming effort to projects that came together late, if at all.
- enduring endless death marches.
- working on teams that had good ideas but never seemed able to come together to implement them.
- watching code bases of beloved products become mired and impossible to maintain due to years of patch and fix programming.

In mid-1999, Kay joined the company discussed in our paper.

The second author, Ron Stauffer, had his own frustrations. He had been at the company for over two years, and although the development team had been successful in shipping releases regularly, he was frustrated by the high levels of stress on the developers from:

- late, inadequately tested releases due to inaccurate scheduling and estimating.
- always knowing there was too much to do, because there was a constant backlog of features which had been promised to customers.
- working under impossible deadlines.
- doing rushed work to benefit individual customers rather than improving the product for the market.

In the following months, the two of us had many discussions about the different project problems we had

encountered and our desire to somehow improve the way we developed software.

Early stages

Pushed by the fact that Ron was the only remaining developer, and Kay needed to understand the code quickly, we adopted pair programming, which we'd heard about on the Wiki. We were both somewhat skeptical, but didn't see any other way to handle the situation.

Pair programming actually seemed to work.

We soon could tell that it was a superior way to get someone up to speed on the code. On previous projects Kay had had to struggle for weeks to get even partial understanding of the code. Now she was delighted to have full-time training from someone who understood the domain and the code. In mere minutes, she could get accurate explanations of how any given piece of code worked.

Ron welcomed the fact that he was learning better habits and more professional behaviors from an experienced developer. In Stephen Covey's words, he was "sharpening the saw." At the same time, Ron was learning that code is for developers and that it's better to write communicative code than "cool" code. Other developers would thank him. And he even learned to throw away code and start over, which required Kay to use a pretty big club.

Gradually, as Kay began to be able to contribute, we began to see other advantages of pair programming. We were able to come to solutions much more quickly together than we could individually. The code remained clean and consistent. We found that when bugs slipped through and were discovered by customers, the bugs were usually in code that we hadn't pair programmed. Slowly we learned that pair programming as peers was much safer and more productive than working individually. And we were learning more than just pair programming; we were becoming familiar with a new vocabulary that included such terms as "you aren't gonna need it," "merciless refactoring," and "code smells."

Based on our initial success with pair programming, we began to consider implementing other XP ideas.

We were uncomfortable about the lack of testing on the project. Until we read about unit testing in XP, we didn't know there were developers out there who actually did automated testing. We thought it was something too difficult and time consuming for use on normal projects. But when we saw other people were doing it, we thought perhaps we could too. Perhaps we could even get away from spending more time fixing bugs than writing code.

Kay decided to try writing some test cases with another programmer who had just been hired. They soon had several tests up and running on some new code. However, when they attempted writing exhaustive tests for existing

code, they got mired down in the details of testing every little accessor method. It would be a few months before we found out how to make test-first programming work, but our motivation to try it was growing.

Turning point

Our breakthrough insight into test-first programming came when Kay attended XP Immersion in December 1999.

Until this time we didn't have a very good understanding of XP. We'd read everything we could find on the subject, knew all the practices and were well versed in the literature, but still had a ways to go before the ideas really clicked.

This learning gap was essentially filled by the face-to-face interaction, participation and demonstrations Kay experienced at XP Immersion. She watched James Grenning and Michael Feathers demonstrate refactoring and unit testing on a small program, and participated for a week with a group of conference attendees on a mock XP project team, with an "onsite customer." Ron Jeffries coached the team on iteration planning and signing up for tasks.

Returning with this improved understanding (now based on experience), Kay was able to get VUnit working and train Ron on test-first programming. Ron's initial reaction was that the overhead of writing and maintaining the tests would significantly outweigh the benefits, but he felt that it was certainly worth a try. He trusted that if the overhead proved too high, Kay would agree to stop writing the test cases.

After the first week, we began to notice that having the test cases in place increased our confidence. Ron had always been concerned by the lack of testing the product had historically received, and had been at the company long enough to get bitten by the inevitable defects getting through to customers, necessitating long extra hours of cleanup work and damage control. He felt immensely more comfortable now that we were taking the time to think through problems at an incredibly high degree of granularity and support every line of code with tests.

As we worked longer with test-first programming, we began to see the value of having the immediate feedback of the red bar, helping us to focus on only one failure at a time, and minimizing the tendency to get lost with multiple distractions. We appreciated the security of knowing that if a test case ever failed in the future, the team would know immediately and could react in a timely manner, rather than find out days or weeks later that something was wrong, as had previously been the case.

Ron realized that his job was becoming more enjoyable. Having programmed "under the gun," in a mentality of scarcity for so long, he was learning first-hand the truth of Kent Beck's statement:

The mentality of sufficiency... creates its own efficiencies, just as the mentality of scarcity creates its own waste." [2]

It's true that the test cases took time to develop, but instead of "waiting until we have time," we just pretended we had enough time to write tests, and found to our surprise that we were moving at least as fast as before--only with more confidence.

The product's business objects layer fell naturally into unit testing. We were used to running manual testing against test databases on our local machines, so it was a natural extension to write some code that created a particular test database with predetermined test data. Each test case cleaned up after itself in its TearDown() method, leaving the test database in the same state as it was before the test ran.

This time when we wrote the tests, we didn't worry about "playing catch-up" and writing tests for existing code. We only wrote tests for new or changed code (a little tip Kay picked up at Immersion.) This gave us a sense of accomplishment without bogging us down with too many tests.

At first, the test suite was very small and ran in under a minute, but soon we began to run just the tests pertaining to the code we were working on, running the full test suite at regular intervals. Of course, we always made sure the full suite passed before we checked code back in. But at no time did the full test run take longer than five minutes, even when it had developed into a very extensive suite.

During this time, we also learned the value of simple design. As we designed a particular feature, we initially came up with a very elaborate scheme. After a couple of weeks trying to force the design to work, we stepped back, reflected for a moment, and decided that a much, much simpler solution would really be adequate. It took courage, because we were already close to the deadline and working overtime, but we threw all the code away and started over.

We were extremely grateful for this decision countless times over the next year. The piece of code turned out to be very pervasive, with the new feature creating a ripple effect that necessitated changes throughout the product, some of which we didn't catch until later. As it was, the changes were not too difficult to make, but had we released the original design to customers, we would have had to spend a lot of time later patching up the rest of the program so that it would work with the complex design.

By then, we had assimilated pair programming, test-first programming and simple design, but were still working as a "single pair plus one." Events led to the formation of a new project team to prepare the next product release, with Kay as the team lead, so we began working with other developers and had to answer questions of planning,

requirements gathering, scheduling and coordination. The team began with four programmers including the lead and grew to six programmers by the end of the project.

Implementation

Having had good success with the XP practices we'd tried so far, we looked to XP for a solution to the scheduling problems we'd encountered on so many projects in the past. We decided to implement the planning game.

Short increments appealed to Kay as a possible solution to the most common problem she'd observed on software projects—the "90 percent done" syndrome. She wanted to be able to answer Kent Beck's question, "When you're halfway through, are you halfway done?" which didn't seem possible if design and programming tasks were allowed to expand into long-term efforts with indefinite deadlines.

Having the customer speak with one voice appealed to Ron because of the problems he'd experienced when the company tried to respond to all their customers' different requests at once.

But in order to run the planning game, we needed an XP customer. We wanted to find someone who would:

- be physically close to the development team.
- elucidate requirements for the developers.
- keep them from attempting to implement too much and blow the schedule.

Our best option seemed to be to have this role filled by a product manager.

We began looking for a product manager who had the right mix of customer involvement, domain knowledge, knowledge of our product, and the ability to make tough prioritization decisions. We were fortunate to find such a person heading up the training department in our own company. Dan Turner, the third author, came to the company from one of our initial clients, had managed our product implementation projects for many of our other clients, and was the person in the company most in touch with the customers' needs and the problems they'd been encountering.

Dan enthusiastically agreed to take on the responsibility of customer/product manager. Although he didn't have product management experience, he had the more important background we wanted with the product itself, the market niche, and the customer needs.

XP was new to Dan but it didn't seem so strange, since he didn't have a lot of experience with other ways of developing software. As he began to work within the role of XP customer, he appreciated the structure the process provided and the ability it gave him to more effectively steer the project. He could see how XP encouraged open communication and information sharing among all

members of the team, especially between the product manager and the developers.

"Release planning" was partially done by the time Dan joined the team. Without the customer, we had to do release planning somewhat by guesswork. It wasn't too difficult to make an initial stab at a release schedule, since this release was primarily a complete redesign of the user interface. Kay quickly put together a rudimentary tracking system listing all the screens in the product. She assigned a ballpark estimate of "high", "medium" or "low" difficulty to each screen upgrade by eyeballing the current version of that screen and guessing.

After conversing with the other developers and getting their feel for the difficulty involved, we assigned estimates in "ideal days" to the high, medium and low categories.

Kay developed a projected release date by:

- completing one iteration to measure the team velocity.
- summing the number of ideal days for all the user interface tasks listed in the tracking system.
- dividing by the measured team velocity.

This was actually a projected "code complete" date--there was no automated acceptance testing planned for the product, so the actual release plan included time at the end for a traditional Quality Assurance acceptance test.

As soon as Dan joined the team, we were able to complete release planning by adding in estimates for additional feature requests he had uncovered in the process of gathering requirements. We presented the VP of Engineering with a "bare minimum" code complete date that represented all the user interface changes. We presented the other features with an estimate on each one of how many weeks it would add to the date. The executive team agreed to the minimum code complete date with the understanding that they could add features at any time before the last iteration by increasing the schedule by the estimated amount of time.

With the release planning done and the customer in place, the team was ready for some basic ground rules. With a mixed level of XP buy-in from the members of the team, but no one openly opposed to XP, we set the following policies as something everyone could agree to:

- All checked-in code will be either pair programmed or code reviewed.
- All code changes in the business objects layer will be supported by unit tests.
- A clean compile and clean run of the entire unit test suite must be obtained before checking in code.

- Attend iteration planning meetings (once every two weeks) and standup meetings (once every day.)
- At iteration planning, sign up for the same amount of work you completed last iteration.
- At iteration planning, the customer chooses which stories are available for sign up, amount of work not to exceed the combined amount the team completed last iteration.
- Only work on functionality required for stories you are currently signed up for.
- Should a story remain uncompleted at the end of the iteration, you may not sign up for the same story you had in the previous iteration (no second chance.)
- Team members will not be asked to work more than 40 hours per week.

This was a limited, "tolerant" version of XP, necessitated by acceptance testing resource constraints and the absence of buy-in from some new members of the team. Even so, most people in the engineering department perceived it as pretty radical.

Steady state

Having watched Ron Jeffries explain iteration planning at XP Immersion, Kay had enough understanding to set up iterations and iteration planning, and communicate their format and purpose to the developers and Dan. The concepts were simple enough that after we ran one planning meeting, everyone had an understanding of how the planning game worked.

Ron liked having the tasks be forced into iteration-size chunks. This usually meant they tended to be better defined and a sense of having several accomplishments in each iteration was helpful for morale.

He also learned, to his delight, that developers have no business estimating a task they are unfamiliar with; hence, having the customer write a card to spike a solution was a valid task in preparation for better estimating a follow-up task.

At the end of each iteration planning meeting, when it came time to sign up for tasks, Ron was consistently amazed how someone always seemed to want the tasks that he didn't want. And because he always got at least one or two tasks he wanted, signing up for a less desirable task was easy to do. After all, two out of three wasn't bad. There was always some good humored bartering taking place to make sure the tasks you had matched your velocity.

As the team experienced iterations and iteration planning meetings, the developers got to watch the customer deliberating, prioritizing, often even agonizing over features to include in the iteration. This was the first time in the company's history that scope was explicitly adjusted

according to the amount of work that was actually being performed by the team—the team's measured velocity. All the developers appreciated this.

We also appreciated from the 40-hour week. Aside from the obvious benefits of enjoying time away from the office, we noticed that there were times when, at the end of the day, we would be coding and hit a snag. In prior programming lives, we would have stayed at work coding and coding, often without making significant progress for the time spent. If we just went home before getting frustrated we seemed to have clearer minds and could solve problems while at home, allowing us to be immediately productive the next day rather than returning to a still unresolved problem left from the night before.

After code complete, we entered a bug fix cycle as Quality Assurance began to perform top-down tests on the product. At that point, we stopped having iterations because most bugs were small enough it wasn't worth the time it took to estimate them or combine them into estimable chunks. We quickly noticed how "different" it was to be without the iteration planning process. We wanted our iterations back.

Eventually the product shipped and we all went to a movie!

2 WHAT WE LEARNED

We believe the project was a success. We felt like we were finally meeting our dates, delivering better quality code and having a better work experience. Given the chance to do it over again, we would keep all the XP practices we did and add automated acceptance testing and incremental delivery. We experienced how the lack of these two practices prolonged the project and added unpredictability to the schedule by adding a period of bug fixes at the end of the cycle.

Five months after the product shipped, we conducted a project retrospective consisting of interviews of other members of the development team and the VP of Engineering. What were their perceptions of the project, the use of XP, the results? What practices would they keep, what ones would they not? The results of the project retrospective were so interesting to us that we added project introspections to the list of practices we'd start doing if we could "do it over again." We think introspectives during the project might improve buy-in from the team and management, and would probably alert us to needed course corrections.

What we learned from the project retrospective was that different members of the team had quite different perceptions of the benefits of XP. We (the team lead, senior developer and product manager) had the most commitment and belief in the value of the XP practices. Others had varying levels of support for the practices, especially testing and pair programming.

Testing

We did a large amount of unit testing and were much more convinced of its value by the end of the project than we were at the beginning. At first we thought it would be too much overhead, but once we decided to only write test cases for new or changed code, we felt less overwhelmed. The more experience we gained with test-first programming, the more clear it became that the extra time spent was made up for by less time spent in the debugger tracking down problems. We became more confident that we wouldn't be spending so much extra time down the road fixing customers' data problems caused by bugs in our product. And we saw how test cases allowed refactoring, so the design of the code could improve over time instead of being hacked beyond all recognition.

Other developers believed throughout the project that unit testing was too much work for too little benefit. One was eventually persuaded through logical argument that it was a good thing to do, but still preferred not to. Another remarked that he didn't see why unit testing was beneficial until he'd gotten into his next project, where he's now in Quality Assurance. Interestingly enough, the developer least convinced of the benefits of test cases came to our team from previous jobs in Quality Assurance.

Pair programming

Not every member of the team tried pair programming. Those who practiced it the most became the most comfortable with it. Whenever they paired, they felt that their code was consistently clearer and had fewer bugs. They felt like two programmers working together could come up with better solutions faster than either one working alone.

Kay noticed how knowledge of all areas of the code spread quickly among the team members practicing pairing. Another developer was quite new to programming and came on the project rather late. He felt like pair programming allowed him to start contributing much earlier than he would have otherwise. Other members of the team believed pairing was not an efficient use of time, opting instead to have their code reviewed after the fact. The VP of Engineering's view was that pair programming turned out to be a useful technique for learning the code in a mentor/student relationship.

We don't think that pair programming is "sellable." It seems that people are either receptive to the idea, or they are not. The problem is that pair programming has to be experienced to be appreciated; if someone is not receptive, they won't try it, and won't understand the advantages. Many of the benefits, such as the way pair programming provides peer support for high-discipline practices like testing and clear design, are not quantifiable.

Onsite customer/planning game

Dan was new to product management, and worried that he

didn't have experience defining requirements. He really appreciated the way the XP process provided a broad systematic approach to requirements gathering, followed by ruthless prioritization.

In order to create cards for each of the project tasks, he first had to identify all of the potential objectives: feature enhancements, bug fixes, and new functionality. Just gathering all this information was an improvement over the way requirements had been defined in the past. He collected feedback from end-user clients and from internal employees in all areas of the organization. This created a comprehensive list of potential product improvements, provided everyone with a broader, more holistic view of the project, and reduced the risk that critical objectives were being overlooked.

With a list of almost four hundred potential improvements, the team could easily have become overwhelmed, divisive, and disconnected. But doing prioritization in the planning game avoided this problem by focusing us on the objectives that received the highest ratings for a number of pre-determined criteria. Dan learned a lot about prioritization from the planning game: improvements that he initially felt were critical for the product ended up being "less important" when he had to prioritize them against all of the other improvements that could be made.

The developers also gained an appreciation of the value of prioritization. By sitting with the product manager while he worked on prioritization, they learned that he was working very hard to remove any less important functionality rather than expecting development to deliver the impossible.

Being able to sign up for tasks was a small, but greatly appreciated thing for all of the developers. They enjoyed "the right to accept your responsibilities instead of having them assigned to you." [3]

Short iterations had several benefits: Kay appreciated being able to gauge at regular, short intervals whether the project was on schedule. The developers appreciated the finite amount of work and the sense of accomplishment and closure at the end of an iteration. Two developers reported a dramatic reduction in their stress level when they moved into development from other areas of the company. They attributed this to no longer being demanded to produce more than they were capable of producing in a given amount of time.

Some team members reported reservations about the planning game. First, by only signing up for an estimated number of difficulty points, one developer felt that people were sometimes slacking off when all their cards were complete. The VP of Engineering also shared this perception. Second, the VP of Engineering would have preferred seeing a measure of each individual's velocity. We tracked individual velocity internally but only reported team velocity externally. Finally, two of the developers

viewed the cards as too "low-tech." They would have preferred an electronic solution.

40 hour week/Sustainable pace

The developers appreciated the 40 hour week, especially when it contrasted with their experience on prior projects. The VP of Engineering and the executive team, however, perceived the absence of overtime as lack of commitment on the part of team members. They viewed the team as not doing their part. This was a stated concern after the project was completed.

Simple design

Although simple design seems counter to what we have been taught about thinking ahead and building maximum flexibility into the code, we learned its value after having to seriously rethink and rebuild designs that were too difficult to implement. Over time we saw how the simpler design was easier to understand and gave us greater flexibility because we could modify the code without fear. We learned that simple design allowed us to get more simple things done rather than fewer complex things, in the same amount of time.

Summary

We were the three members of the team most sold on the benefits of the XP practices. We were the ones who followed the XP practices most closely. However, it was still a paradigm shift for us, especially for Ron and Kay, who were both skeptical at the beginning.

Making a paradigm shift isn't easy. For us it seemed to occur over time, as we experienced the new way of doing things and began to see where it solved some of the problems that had been so frustrating on past projects. The forces that cemented our change of thinking were an understanding of the XP practices (gained through doing them) combined with an understanding of the problems they were intended to solve (also gained through experience.)

As we practiced XP we developed a new, shared understanding. We learned through experience that:

- Testing is good
- Planning is good
- Prioritization is good
- Choosing your own responsibilities is good
- Continuously improving the code is good
- Two heads are better than one
- Simple design is good

Not everyone on the team had the same understanding or perceptions. Some felt there wasn't enough benefit in the practices to outweigh the costs, and so they were not inclined to spend a lot of time doing the practices, particularly testing and pair programming.

Interestingly, there was resistance just to the name. In the retrospective interview, we discovered that “extreme” was not a word the VP of Engineering liked to hear when connected with a serious software development effort. He felt that “moderation” is usually a better approach.

In summary, some of the reasons for not accepting or trying the XP practices were:

- The name
- 40 hour week looks lazy
- Pair programming is less productive
- Testing is too much work

Results wouldn’t sell XP in this case. What is an objective measure of the success level of a project? We felt we had excellent results; others felt we had average results. Some of the evidence we thought indicated success (such as the 40 hour week) were considered negative factors by others. This highlights the inevitable difference in perceptions that occurs whenever different people with different understandings interpret the same events.

Even if everyone agreed on how successful the project was, there were too many variables for anyone to be comfortable saying “results show, XP worked” or “results show, XP didn’t work.” Any number of factors combined to make the project turn out the way it did.

In an essentially subjective situation, we feel that actual participation gives weight to opinions. Through participation in a different paradigm, our understandings began to change, and consequently our opinions changed.

3 CONCLUSIONS

Learning by Doing

We found that understanding the value of the XP practices came only by actually doing the unfamiliar, new practices. Until we had first-hand experience with someone who had the understanding, we weren’t making much progress. It was when we experienced the practices at XP Immersion that we began developing new understandings.

If you’d like to try XP, we recommend attending XP Immersion, pairing with someone who has, or pairing with someone who has successfully implemented XP on another project.

XP Doesn’t Sell

What we believe after our experience on this project is that there is an underlying culture in the software profession that makes XP a very difficult sell. We in the industry are steeped in a particular culture that doesn’t even develop a vocabulary for talking about the problems that XP attempts to address.

Since we are embedded inside this culture, we’re usually not even aware of it. Changing our behavior, for example by adopting the XP practices, is a way of moving away from these shared assumptions long enough to recognize

them and perhaps challenge them. Merely discussing or observing different behavior is less likely to prompt that “leap” that allows us to see our own cultural assumptions.

Based on our experience of the software industry, seeing where individuals spend their time and what organizations reward, we think the following assumptions make it difficult for people to “buy” XP.

Inspecting Defects Out

We expect to inspect defects out, not build quality in. The traditional approach is to write all your code, then check it for errors. After all, if you were a good programmer you wouldn’t have put many errors in your code. We are constantly surprised at the amount of effort spent at the end of project cycles to find and fix bugs, but prefer to forget this when we move on to the next project. After all, based on our assumptions the logical conclusion is that we must have been “bad” programmers—a conclusion we don’t want to draw. Pair programming and test-first programming look like practices that require an inordinate amount of time to people who are unconsciously steeped in the culture of inspecting defects out.

Focusing on Effort

We tend to measure effort, not results. Results in software are very difficult to define, much less measure, whereas effort is quite easy to measure. The software culture has naturally grown around planning projects by man-hours and rewarding people for the number of hours spent working, rather than on value delivered. If this is the way things work at your organization, what good is a system that schedules based on delivering business value, and that encourages people to work smarter, not harder?

Assuming Predictability

We expect predictability. Can you blame us? Computers are very complex devices, and yet have a high degree of predictability, certainly in comparison with events in the world at large. So we are constantly surprised when schedules slip. Something must have gone “wrong” somehow! Those darn requirements “shouldn’t” have changed. We “should” have foreseen this or that when designing the system architecture. The programmers were “careless” and so we have a lot of bugs to remove. Again, we tend to bury these occasions, as they were clearly a result of some mistake on our part. The options-based approach embodied in XP’s planning game and assumed by XP’s simple design/refactoring makes no sense if projects are by and large predictable.

Underestimating People Factors

We expect to use technological or process solutions to solve people problems. We’re not sure why this is the case; could it be that software still shows its origins when it was largely a solo endeavor? Or is it just too alarming to trust project success to units as unreliable and unpredictable as people? We believe people are significant factors in project

success, but we concur with Alistair Cockburn that "remarkably few people in our field have devoted serious energy to understanding how these things called people affect software development." [4] If process and technology are the main determiners of project success, then the "soft" benefits of XP, namely increased communication, increased tendency to ask for and receive help, and increased support to keep people following good programming practices, are not compelling.

If you believe that quality comes from inspecting defects out, you don't need a tool for building quality in. If you believe that man-hours are the measure of success, you don't need a tool for delivering more value with fewer man-hours. If you believe projects are basically predictable, you don't need a tool for managing unpredictability. If you believe that social interaction is a small variable in project success, you don't need a tool for improving communication.

XP or any other solution offering these results isn't going to sell until and unless these perceptions change. Individuals change their understanding through experience, which they are not likely to gain if their assumptions prevent them from trying new things. Cultural change happens slowly.

We conclude that for the time being, in most organizations, XP won't sell. The best we can do is find people with which to build new shared understandings and work together as much as possible to validate whether these new understandings are in fact beneficial to the software development industry.

4 ACKNOWLEDGEMENTS

The authors wish to particularly thank Alistair Cockburn for his encouragement to study sociological factors in software development and to question assumptions.

Thanks to Zhon Johansen for his help in shaping our understanding of the XP practices.

Thanks to Object Mentor for XP Immersion.

Thanks to all the project team members and other employees for being patient with our crazy ideas.

5 REFERENCES

1. Web site online at <http://c2.com/cgi/wiki?FrontPage>
2. Beck, K. *Extreme Programming Explained: Embrace Change* (2000), xix.
3. Jeffries, R., Anderson, A., Hendrickson, C. *Extreme Programming Installed* (2001), 7.
4. Cockburn, A., Characterizing People as Non-linear First-order Components in Software Development (Oct.1999). On-line at <http://members.aol.com/humansandt/papers/nonlinear/nonlinear.htm>.