CHAPTER FIFTEEN

Building a Team with Collaboration and Learning

James Grenning

Around the turn of the millennium, I had a great experience. I was brought in to help one of our clients get a new project started. They were new to object-oriented (OO) design, and I was supposed to help them get an OO architecture defined and an initial plan in place. As it turned out, I spent almost half a year with them every other week. This was a time of great learning for them and for me. Not only did we have some fine accomplishments, but we also spent some time in turmoil.

The company, call it By the Book Systems (BBS), was very structured and prided itself on its quality and conformance to its processes. Like many big companies, BBS sometimes did things that hurt itself. In this tale, we'll see some of the brave and great things that were done at BBS and some outright blunders that can kill team morale and stifle progress.

Starting a new project is fun. I've had the pleasure of starting quite a few projects in my career, and it was good to be part of a team at BBS. As a consultant, I am too often an outsider, being on the team for only short periods of time. In this case, I was with BBS almost every other week. So I really became part of the team.

The team started small: initially Johnny (the systems engineer), Alan (the lead engineer), and me.

The first order of business was for Johnny to brief me on the requirements of the project. He had a 30-page requirements document written by the book with the shoulds and shalls just as prescribed.

The requirements document had plenty of good information in it, but in that form it was almost totally unactionable. Even with the requirements prioritized, it was not clear what the software had to do. Software does things; we needed an inventory describing what the software had to do. Thankfully, there is a popular solution to this problem. Use cases are used to catalog and define system behaviors. Use cases would really help us get a handle on what the software should, shall, and must do.

We had a small problem, though. Johnny had not used use cases before. If use cases were not part of the standard process, we might have to do some formal justification. We pulled out the process manual and discovered, much to our pleasure, that use cases were one of the acceptable forms for requirements articulation. That was good news.

Johnny and I locked ourselves in his office. As we read the requirements document, we discussed what the requirements meant and what the implied system behavior was. We named use case after use case, capturing them in a spreadsheet. After about a week's effort, we had extracted 105 use cases from Johnny's requirements document and we identified nine actors outside our system's boundaries. We understood the system boundaries and the system behaviors, at least at a talking and planning level. This activity is called *use case identification*. The idea is to name all the behaviors.

Around that time, Kent Beck published his book, *Extreme Programming Explained* [1]. It was all the buzz. Wars raged on the newly formed Yahoo! Groups and the established comp.object newsgroup. A great opportunity presented itself. Object Mentor, my employer at the time, had cornered the market on Extreme Programming (XP). Well, that's not really accurate, but it sounds cool. What really happened is that Bob Martin, Jim Newkirk, and Lowell Lindstrom (the Object Mentor management team) had formed an alliance with Kent Beck, Ron Jeffries, and Martin Fowler to train people in XP. Lowell invited me to attend the first XP Immersion. I took the week off from BBS to get immersed in XP. What an impact that unexpected week at the Deerfield, Illinois, Marriott had on me. It amazes me how small events can have such a profound effect.

I must admit, most of the world knew very little about XP and I was in the same boat. What I had heard concerned me. How could something called *Extreme* really be a good idea unless it had to do with skiing? Were helmets required, or elbow pads? Would I need a new life insurance policy? I think Kent was being provocative and I am glad he was. If it had been called something like *Sensible Programming*, everyone would have yawned and the waterfall process would have been kept as the ideal. As it happens, that simple, small book has made many people challenge the status quo.

After my week at Immersion, I returned to BBS with the enthusiasm of the newly converted. OK, I am exaggerating. I was never religious about XP, but thought many of the ideas could really break some logjams at BBS.

I started telling my colleagues about the interesting and revolutionary techniques in XP. They were interested, but reserved. "The quality people will never let us do something called Extreme," said Alan. Johnny was worried that more upfront design work and documentation were needed than XP prescribed. The big process culture made it difficult to imagine that a lightweight process like XP could be a good idea.

Source: "Extreme Programming and Embedded Software Development," by James Grenning [2].

James: Does the above line go with the "Practices of Exreme Programming" sidebar on the next couple of pages? --Prod

Selling Management

After a few more discussions about XP, Alan and Johnny were warming up to the idea. We briefed our manager, Fred, on XP and he was all for trying it, but we needed to go to the boss, Bud, our director. We knew there would be some contentious issues, specifically around keeping the process police at bay and providing the level of documentation that was needed. The documentation requirements at BBS were heavy. But it was my job as the team's mentor to challenge their current practices and help them grow.

I prepared a presentation on the practices of XP. We anticipated some of the issues that would come up and built a case for our recommendations. The presentation ended with a few open-ended questions to get a dialog going.

Bud: "This all sounds very good, but how will we handle the technical reviews required by our process?"

"We need to take the reviews off the critical path," I told Bud. I told him something he already knew: that the reviews temporarily block progress. Preparing for reviews was time-consuming and we also had the logistical problem of scheduling the senior engineers for our review meeting. We discussed how the current approach reviewed untried design ideas.

I explained: "Your most experienced people spend all their time in reviews rather than building the product. You need an experienced designer on each team." We talked about how design is a daily activity, not an event that ends when the sign-off happens.

Bud: "I think I understand. Are you saying we don't need reviews?"

"I'm not comfortable giving up reviews, but we will change what is reviewed and when," I responded. We knew that periodic reviews could not be avoided completely, but that we could find a more effective way to do them. The reviews stop the team's forward progress until the review meeting is held and the design is signed off. We told Bud that by moving the reviews off the critical path we could keep the team moving ahead doing continuous design, implementation, and testing.

"Currently, you use the most experienced people for reviewing the designs rather than creating the designs," I told him. We discussed how by having just one experienced

PRACTICES OF EXTREME PROGRAMMING

Extreme programming is a set of principles and practices for incrementally creating high-quality software. It's one of the original members of the agile development methodology family.

Customer team member

Teams have someone (or a group of people) representing the interests of the customer. The *customer* decides what is in the product and what is not. The *customer* is responsible for acceptance-testing the product. This means the customer team will likely have skilled QA team members.

User story

A user story represents a feature of the system. Stories are small; small enough to be completed within a single iteration by a single person. If the story is too big to complete in an iteration, split it into smaller stories. A story is like the name of a use case.

Planning game

In the planning game, the customer and the programmers determine the scope of the next release. Programmers estimate the effort required to complete each story. Customers select stories and package them into iterations that do not exceed the team's capacity. Stories are ordered by the customer according to their value and cost.

Small releases

The system is built in small releases. Each iteration lasts two weeks. A release is a group of iterations that provide valuable features to the users of the system.

Acceptance testing

Acceptance tests demonstrate that the story is complete. They provide the details behind the story and are defined before the story is started. Customers own the acceptance tests. The bulk of the tests must be automated so that they can be run at any time, usually multiple times per day.

Open workspace

To facilitate communications the team works in an open workspace with easy access to other team members, equipment, and project status information.

Test-driven design

Programmers write software in very small, verifiable steps. First, a small test is written, followed by just enough code to satisfy the test. Then another test is written, and so on.

Metaphor

The system metaphor provides an idea or a model for the system. It is part of the high-level design, facilitating communications. It provides a context for naming modules, classes, and functions in the software.

Refactoring

Refactoring is the process of keeping the design clean, incrementally. When systems evolve, the design will get messy. The idea of refactoring is to detect, identify, and fix design problems as they come up, while they are small and easy to fix.

Simple design

The design is kept as simple as possible for the current set of implemented stories. Frameworks and infrastructure evolve with the code through refactoring.

Pair programming
: Two programmers collaborate to solve one problem. Programming is not a spectator sport. Both programmers are engaged in the solution of the problem at hand.

Coding standards
: The code must have a common style to facilitate communication among programmers. The team owns the code; the team owns the coding style.

Continuous integration
: Programmers integrate and test the software many times a day. Big code branches and merges are avoided.

Collective ownership
: The team owns the code. Programmer pairs modify any piece of code they need to. Extensive unit tests help protect the team from coding mistakes.

Sustainable pace
: The team needs to stay fresh to effectively produce software. Too much overtime will result in reduced quality, burnout, and unpredictable outcomes. Everyone works hard but at a sustainable pace.

---

designer on the team we could, with daily design discussions and pair programming, keep the design quality high.

Bud asked: "OK, but what will be reviewed?"

"We will review what was built and tested rather than what we anticipated building," I told Bud. We discussed that at each iteration, we would capture the important architectural decisions and constructs. If design problems were discovered during the reviews, we could make changes with confidence. Our automated tests would lock in the behavior. Any changes required by the reviews could be fixed in the next iteration.

"Let's be clear on the purpose of the design document. Its purpose is to provide a high-level road map of the system," I added. "The details will be where they belong in the code and its tests."

Bud: "Wait a second, there is no detailed design documentation?"

"Not at all," I responded. "Using test-first programming we are going to have many automated unit tests. The test cases are the detailed documentation." We discussed how each test case is a focused code example with a specific precondition and desired outcome.

I added: "The tests are an executable specification." Once people learn to read the tests, the tests become a very effective form of detailed documentation. It's not like prose documents that you have to read and interpret. This spec can be executed and we can see if

there are any deviations, now and in the future. "It's the gift that keeps on giving," I added.

Being a little uncertain, I added, "Bud, as you know, we are new to this, but we think it makes sense. And anyway, with the one-month cycles, we can try this approach and revisit our decision each month."

Bud: "OK, we'll make continuous improvements." But then he added: "How do we know the tests are right?"

"We've been thinking about that one." I paused, and then continued: "We plan on pair programming helping us, but we think we should review the test cases. We want to make sure the module is doing the right thing and has a usable and well-engineered interface."

Bud: "OK, we review the tests to check that the interface is right, and also make sure the tests are testing the right stuff."

"Yes, that's the idea." We went on to further discuss and convince ourselves we were making a good decision. The key points we settled on are that the specific implementation details may not be as critical as having a good interface and the right behavior. Those things impact the larger architecture. I added: "Don't forget that if some implementation needs improvement, we have the tests backing us up to make sure that during refactoring the external behavior is preserved."

Bud: "OK, but our process requires code reviews, and they have helped us quite a bit. When do those happen? Does pair programming take their place?"

"We think so. That's one of these *Extreme* things that we are a little concerned about," I added. "Code reviews happen continuously with pair programming." I took a little time to describe how pair programming works. We take turns working with each other, maybe a few hours at a time. It's like a live code review, but could actually be better in some ways, and maybe worse in other ways. Two people are involved in making design decisions and naming the classes, functions, and variables in the code. The downside is if the pair gets shortsighted together.

Bud knew they were investing a lot in code reviews. There was plenty of preparation and meeting time. We discussed that pair programming can eliminate much of that overhead. He liked what he was hearing. We all thought that combining pair programming with reviewing test cases would result in better quality than today's document-driven process.

Bud: "We get the code review in real time to make sure our format standards are met. The test cases show what the code is supposed to do and confirm that it does it."

Bud was great. He knew the current process was slowing the teams down and believed that the common-sense XP practices could help accelerate our progress. He knew change was needed. He was excited about the potential for improved schedule predictability, improved quality, and improved fun at work. He said if he got just one of those and the other two did not decline, he would consider this XP project a success. Bud gave us the

green light to "go Extreme." We also got a "get out of jail free" card with the process police. "I'll sign any process waivers you need," Bud said.

In hindsight, Bud made me see one thing that enables a great team to form: a manager with vision, bravery, and a willingness to support his team. Bud supported us, believed in us, and made it safe for us to go outside the established norms.

Getting Started

Early on, we had the option to grow the team, but we decided to wait a month until we had an iteration under our belt. This way, we started the learning process, building our experience. In an iteration or two, we could coach the new people with a lot less guesswork.

Johnny, now our *customer*, Alan, and I wrote all our use cases on note cards. We wanted them to be easy to manipulate, just like Kent and Ron taught me at Immersion. At Immersion, I learned about *user stories*. To me, it seemed that a user story and the name of a use case were virtually equivalent. We considered changing our practice from use cases to user stories, but Johnny warned me: "If we decide to use user stories, we are going to have to write a justification to deviate from our standard process." Johnny also was concerned about the informality of user stories, and added, "Besides, I think that a user story is too light on detail. We would never get that approved."

We decided to continue using use cases; after all, it was an accepted BBS practice. We figured that we were going to push the envelope in many other areas, and uses cases seemed to be the right vehicle for us. We decided to pick our battles carefully and save our "get out of jail free" card for when we really needed it.

Just before we were to begin our first iteration, Johnny and I worked out the detailed steps for the core use cases. The core use cases had the highest value and the most architectural impact. We were ready to start iterating.

Looking into the future just a little: after the first iteration, we started considering the XP practice of automated *acceptance tests*. Like Kent and Ron suggested at Immersion, we built an application-specific testing language. We tried to make it readable and not too programmer-ish; the goal being that a domain expert could read and write the tests without being a programmer. After a few iterations, it became clear to us that the information in the use cases and the information in the acceptance tests was a form of duplication. Duplication is one of the wastes and liabilities that XP tries to prevent.

Eventually, I built a case for not detailing out the use cases at all. Instead, we would write the acceptance tests before the iteration, providing the details that were in the use cases. Acceptance tests would be an executable specification! And we would replace the use-case elaboration effort Johnny had to shoulder with writing acceptance tests.

As it turned out, this was too extreme for BBS. We kept doing both use cases and acceptance tests because that seemed easier than bucking the system yet another time. I guess

that's what Jerry Weinberg would call becoming a pickle. Put cucumbers in with the pickles, and what happens? "Cucumbers get more pickled than brine gets cucumbered." [3] XP got a little pickled, as did I.

Let's get back to getting our first iteration started. After estimating a few stories, I mean use cases, Johnny chose the first iteration. The initial use cases described core functionality, functionality that encompassed the most common operations the system would be expected to do thousands, if not millions, of times per day. Alan and I got to work. Ellen, a part-timer on the team, joined in and worked with us from time to time. She was a senior engineer winding down from her last project. She was ready to go Extreme with us. We agreed that all production code would be created using pair programming. It's good we had a third team member because with all the meetings that Alan got pulled into, it was good that I had Ellen to pair with when we could not keep Alan out of the meetings, and vice versa.

Our first iteration was a success. We delivered most of what we planned. Looking back, I see that this was quite an accomplishment. At Object Mentor, we came to know that first iteration as "iteration zero." Not just because we computer geeks like to start counting from zero, but because so many first iterations accomplish nothing. It was our first XP project. We did not know we were supposed to fail, so we delivered working core functionality instead.

This team did not fight each XP practice, as I later experienced as a coach. The team's desire to learn and try new things made the project a success. Sure, Ken Beck wrote the book, but it was our process.

We learned some of the ins and outs of test-first design and incremental development. Our manager, Fred, was thrilled with the output. "I used to get only a draft requirements spec after a month, but we've got working code!"

We had the base learning and initial code in place. It was time to ramp up the team.

Growing the Team

Now that Alan, Ellen, and I had a grasp on our technical practices, we added a couple of bright, new engineers, Paul and Alex. They were glad to be on the team learning OO design and XP, and contributing to a new product.

It was a good team; we worked well together. There was collaboration, cooperation, and excitement about our progress and learning. We pair-programmed, wrote CppUnit tests, refactored our design as we went, and created our own application-specific language to drive acceptance tests. We accomplished a lot.

Pressing the Envelope and the Process Police

The process police, lead by Marilee, were on our tails because of our lightweight documentation approach and deviation from the waterfall-style milestones process. Just hav-

ing Bud's "get out of jail free" card gave us some distance. But we did have to deal with breaking from the norms quite regularly.

At our first review meeting, we had completed a small handful of use cases. The design was simple, so writing the documentation was easy. It took only part of a day. Consequently, there wasn't much documentation. We had two reviewers, Jim and Art. Consider them the informants of the process police. They were not used to such thin documentation packages. Hey, come on! It covers only one month's work.

The fact that we had Bud's approval to push the status quo made all the difference. Our design decisions must have been good enough, because we spent most of the time discussing process and how we could get away with such thin documentation.

We gave a little more the next review, but they still wanted more. But no one could really say why. So we settled on continuing to evolve our thin architecture document, well-refactored code, and comprehensive unit and acceptance tests.

Learning

Paul and Alex were fairly new employees at BBS, with one or two years of experience, if I remember correctly. I got to spend the most time with them. The senior people kept getting pulled into meetings, so the three of us had plenty of time to work. Being the seasoned journeyman, it was like having two apprentices. We worked day to day, incrementally developing. Like I mentioned earlier, I was at BBS only every other week, on average. So on Fridays, we would plan the next week's work.

Paul and Alex were great. They always got everything done that we planned on doing. The code was well tested and it delivered the desired functionality. As their mentor, I reviewed their work when I returned the following Monday. They always got the code to work, but they usually put some code in the wrong place. What I was detecting was something that Martin Fowler calls "feature envy" [4].

Code problems now have names! Fowler and Beck did a fine job naming and describing a small catalog of "code smells." Feature envy was where one class was doing the job of another class. There were others: long method, large class, long parameter list, lazy class, and shotgun surgery, to name a few. Just having the name for some undesirable code quality helped us detect problems early and keep the code clean.

So Alex and Paul would leave these small code problems around, but it was no big deal; Monday became refactoring day. We had the tests, so it was easy and low-risk to move the envious code to its proper home. It was a learning experience. I would never have known to call it *feature envy* before XP. Paul and Alex could have been defensive about their work, but the learning spirit avoided hurt feelings. They also taught me plenty during our pairing sessions.

After a few cycles of this, it became a game. Paul and Alex would get the work done, and start hunting for code smells. On Monday mornings, I would come in, get a coffee, and

find them. Paul would say, "James, we got all the tests passing, but there are a few things you are not going to like." They were building skill in the first step to being successful refactorers! They had to find and identify code smells. They did not always know how to solve the smells, but they learned how to recognize the problems. Awareness is the first step toward change.

Pair programming really helped break down communication barriers. When you sit next to someone for a few hours you reveal your own strengths and weaknesses. It's a two-way street. With the right people and the right attitude, learning happens and trust grows. Our team atmosphere was such that it was safe to make mistakes and to learn.

Requirements Versus On-Site Customer

Johnny was an expert in the system. He was the designer and programmer on the original system years earlier. During my work on the project his role was systems engineer. From his vast knowledge of the system, he wrote the requirements spec. We worked together to extract the names of the use cases implied by the requirements. Johnny would elaborate the requirements just in time for us to design and implement them. Johnny was always under pressure to deliver the next batch of use cases. Sometimes it took him longer to figure out the use cases than it took for us to implement them. That always upset him. It was working quite well. I really came to appreciate having an on-site customer.

One day, Johnny, Paul, and I had finished reviewing the use cases and went to start the development work. We worked out the impact of the new use case on the existing design. We used a high-tech tool known as a whiteboard. The system design was in our heads, so we extracted enough of the design, putting it on the whiteboard, and talked through the changes. Once we were happy with our direction, Paul and I went to Paul's cube to get started.

Within minutes of starting, we had a problem. We read the detailed use case again. There was an area of interpretation. We had just discussed it with Johnny a half hour before, but Paul said, "Johnny said we should retry the transaction under these conditions." I said, "No, in this case we are supposed to log a message and continue. You're thinking about use case extension 4B; that's next week's work."

Of course, we discussed it (argued civilly about it) for 20 minutes. Then it dawned on me. We don't have to guess. We have an on-site customer. "Let's ask Johnny." I popped my head up over the cube walls. There was Johnny in his office with the door open 5 yards away. I'm surprised he did not hear us. Johnny settled the discussion in less than a minute. No emails were needed. No voice mails were left unanswered. No requirements review cycles were spun. We asked Johnny, he settled it, and we got back to work.

Some might say we needed a better spec; others would add that the use cases needed more detail. The documents would have been great to have, but I don't think that effort would have helped or have been cost-effective. The document would never be detailed

enough and usable enough where we would not need clarifications from our customer. The only form of requirement that has all the details is code.

There were other occasions where we needed clarification on the requirements. Many times Johnny asked for time to go off and study the existing code base. The legacy application that we were redesigning a part of was our de facto requirements document. It was the only document detailed enough to answer some of our nitty-gritty requirements questions. If all those nitty-gritty requirements had made it into a document, that document would have been as detailed as the code and probably not too helpful.

Trouble in River City

Things were going great until one ominous Monday morning. Wally, our newest engineer, showed up unannounced and with an attitude. He had some history with Ellen, and right away he started with a rude form of humor, cutting deep into her. After I told him to cut it out, he insisted he was kidding, but you could see it hurt her. The knifelike kidding did not stop.

Big companies have this bad habit of treating people like parts in a machine. The first sign is referring to people as *resources*. Someone in management thought that Wally was a plug-replaceable programming unit. Through some magic of "resource availability" and project priority, we got Wally this fateful morning.

Wally was bright, but he had no idea what our team culture was. Our team was collaborative, and we were excited about taking on the process and technical challenge the project offered. All the team members identified with the team, and wanted to be part of it and to live by the standards and work ethic we had adopted. Wally did not get or want to get why we were writing tests. He did not get why we were not doing a big design first. He did not get the short iterations and, surprise: he was no suitable pair partner. His *humor* disrupted standup meetings and pair programming sessions.

I recognized right away that Wally was not going to work out. Fred wanted us to try to integrate him into the team. We tried, but sand does not go well with contact lens cleaner. It's too abrasive. In the end, Fred saw it our way and got him off our team. In the short time he was with us, our morale and output dropped.

Teams Are Made of People, Not Resources

There is an important lesson here. People are not plug-replaceable. One bad attitude on a team can bring the whole team down. A bad apple can be especially poisonous in a collaborative team. I suppose someone like Wally could hide in a cube and be given individual work. He might even be able to be quite successful. But that was not our team or how we chose to work. In my career, the times I most enjoy are collaborative times like these. The least enjoyable times were made painful by a bad attitude.

When Wally finally did ship out, we recovered. Work was fun, and lots got done.

Companies Make Their Own Troubles

One of the XP practices is to adopt an open workspace. The practice involves giving the team its own space to work. In the center is a big table. Project information is plastered on the walls. Computers are set up on the table so that it's easy to collaborate.

We did not have an open workspace and we discovered that working in cubes was not great. The monitors were in the corners and that made working together tough. We started talking about taking down some cube walls so that we could have a good space for communication and collaboration.

I asked Johnny about moving some walls. He said, "We can't do that; we have union people who configure all the furniture. We have to go through them." I had envisioned how Peter Gibbons, the frustrated programmer from the movie *Office Space*, unfastened his wall and it toppled, freeing him from his cube. That was not going to happen. It was time to visit Bud.

I said, "Bud, the cube walls are hurting our ability to communicate. We should take down a few cubes and make a space for the team."

"Save your energy for something else; that one is a non-starter. We have a policy for furniture reconfiguration," Bud sighed. "Once a wall or office is changed in an office area," he said as he gestured to a lattice of 50 or more cubes, "the whole area must be brought up to the current office specs."

I replied: "Yeah, so what does that mean? Can we have an open workspace area?"

Bud: "I am not sure of that detail, but if we change one wall," he answered, pointing to the private offices along the outside wall, "all those senior engineers and managers will lose their private offices and doors. You can't make this one happen." Bud consoled us that if it was our process, he could do something about it. "But moving a wall gets the union involved, and there is no flexibility there."

This battle was over before it started. We could have pouted and whined. Instead, we got back to work and lived within what was possible. Pickled again.

Future Projects

After the success of our first project, Fred wanted to get a few more teams to go Extreme. This turned out to be more difficult than expected. First of all, Fred wanted to run each of the other teams the same way as our original team. "This is the way we do pilot projects," Fred told me. Of course, BBS has a process for trying new things, and we could not break that mold!

I warned him not to force each team into the exact same practices. I did not think it would work. Each team is unique and each team member has unique skills. There will be differences, and they won't own the process the way we did.

Well, we tried it anyway. The projects were successful, delivering two to three times more high-quality software per plug-replaceable-programmer days than the standard BBS development project. But as much as we tried, we could not get each team to follow the exact same practices. Of course, much of the core was the same, but there were many variations. Each team found its own way.

Some of the biggest obstacles I encountered were cultural. Their culture included pride in the processes they developed and followed. Those processes helped them do many great things. But they needed to get faster. Something had to change, but the culture resisted change. A year later, I spoke to a lead engineer in a nearby group and asked him whether his team was doing XP. He told me it was too much trouble. A team that wanted to go Extreme, as Bud called it, had to put together a 30-page process justification approved by the process police. Not many project teams volunteered to be the next XP project.

Collaboration Success Factors

We demonstrated that it is possible to improve on the status quo. We challenged the current practice and gained new understanding on how to get things done. It showed me again how important a good team is. A good team, working together, can get a lot more done than a group of individuals. How do we get a gelled and collaborative team?

In a blog by Alistair Cockburn [5], I found some interesting insight into team dynamics and collaboration. Alistair said, "People collaborate when they want to," and continued, "But what gets them to want to?" Through an informal study, he discovered that certain actions lead to an atmosphere of improved collaboration. Here is an abridged form of the top action categories:

- Lift others: Recognize others.
- Increase safety: Support others, challenge but adopt ideas.
- Make progress: Success breeds success.
- Add energy: Challenge, contribute.

In recalling this story, I can clearly see Alistair's action categories at work.

Bud's confidence in our team and willingness to give us the go-ahead really provided a huge lift to our team. Our learning and teaching styles lifted individuals. A counterexample was the assignment of Wally to the team. He did not lift people; he brought people down.

Bud's "get out of jail free" cards and approval of our proposed process changes gave us the safety to try new things. His realistic expectations added to this safety.

We had good people on the team. They were skilled but did not claim to know everything. The team's can-do attitude and continuous learning made work a joy. The early success gave confidence and provided a big boost in our journey. We found things that worked; we made our progress visible. Each day had new accomplishments that relied on the

team's collaboration. We challenged each other and the organization. It was a memorable team experience. Even though I could not convert the pickles to cucumbers, we changed the flavor of the brine!

References

1. Beck, Kent. 1999. *Extreme Programming Explained*. Reading MA: Addison-Wesley.
2. Grenning, James. 2002. "Extreme Programming and Embedded Software Development." Embedded Systems Conference (San Jose, CA).
3. Weinberg, Jerry. 1985. *The Secrets of Consulting*. New York, NY: Dorset House Publishing.
4. Fowler, Martin. 1999. *Refactoring*. Reading MA: Addison-Wesley.
5. Cockburn, Alistair. 2008. "Collaboration: the dance of contribution"; *http://alistair.cockburn.us/Collaboration%3a+the+dance+of+contribution*.