Launching Extreme Programming at a Process-Intensive Company

James Grenning, *Object Mentor*

A company that has traditional formal processes launched a project using many Extreme Programming practices. The author covers how XP was proposed to management, how the project seed began and grew, and some of the issues the team faced during its first six months.

This is a story about starting a project using an adaptation of XP in a company with a large formal software development process. Defined Process Systems is a big company (the name is fictitious). The division I worked with develops safety-critical systems and is building a new system to replace an existing legacy product. The project is an embedded-systems application running on Windows NT and is part of a network of machines that must collaborate to provide services. Through a systems engineering effort, DPS divided the system into subsystems, each of which was later designed by a development team. I was brought in to help one of the teams start its part of the project using iterative development, use cases, and object-oriented design techniques.

Shortly after starting this project, I spent a week in XP Immersion I, a one-week intensive training class in the techniques and philosophy of Extreme Programming.[1] Enthused by XP, I talked to the DPS team about applying some XP practices to our work. The company had already decided to try iterative development and OO techniques, departing from the division's standard process. They were game for something different, but how different? We decided to take the idea of applying XP practices to the director of engineering.

What we were facing

The standard practice in the division was created in response to past and current problems. Most of the developers have one to three years' experience. The more senior developers had the role of reviewing and approving the work of the less experienced developers. To their credit, the review team took the formal review process very seriously. They were good at it: they captured issues and defects on the Web, prepared reviewers, and held crisp review meetings.

In the world of Big Design Up Front and phase containment, these guys were good. I could end this article right now except for one problem: all this process added overhead to the development of software. To design something, they needed to figure out the design, document it in Rose, schedule a review meeting, and distribute materials to

0740-7459/00/$10.00 © 2001 IEEE

The need for documentation was ingrained in the culture, so we expected concern over XP's lack of formal documentation.

review. Then reviewers had to review the materials and enter issues on the Web, have a review meeting, document the meeting outcome, repair the defects and then close them on the Web, fix documents, and have the changes reviewed again.

All this up-front work was not keeping bugs out of the product. Unrealistic deadlines and surprises late in the project were taking their toll. Products were delivered late. Engineers were just getting their skills to a decent technical depth, but they were also burning out and heading for systems engineering or management.

The team struggled with how to begin the new project. Requirements were in prose format and fully understood only by the person who wrote them.

The list of problems was long: the existing process had a lot of overhead, deadlines were tight, engineers were running away, requirements were partially defined, and they had to get a project started. With all these issues, something had to change. Did they need something extreme like XP? I believed and hoped XP would help.

Choosing your battles

The DPS culture values up-front requirements documents, up-front designs, reviews, and approvals. Writing the product features on note cards, not doing any up-front design, and jumping into coding were not going to be popular ideas. To people unfamiliar with XP, this sounded a lot like hacking. How did we get by these objections?

Having been introduced to XP, the group understood what the main objections would be as we tried to sell XP to the management team. Like good lawyers, we prepared questions along with the anticipated answers for our presentation. We expected that the decision makers would consider some of the practices dangerous and unworkable at DPS. The need for documentation was ingrained in the culture, so we expected concern over XP's lack of formal documentation. Can the code *be* the design? Can we really build a product without up-front design? What if there is thrashing while refactoring? What about design reviews?

To paraphrase Kent Beck, one of XP's originators, "do all of XP before trying to customize it." I think that is great advice, but for this environment we would never have gotten the okay to mark up the first index card. We decided to choose our battles. We needed to get some of the beneficial practices into the project and not get hurt by leaving other practices behind. We did not omit practices that we didn't feel like doing; we tried to do as many as we could. We used the practices and their interactions as ways to sell around the objections.

We started by identifying the project's goals—to build a working product with reliable operation and timely delivery, with enough documentation to enable effective maintenance (no more, no less), and with understandable source code. This was as objectionable as motherhood and apple pie. The standard process would identify the same objectives.

We all agreed that a reliable working product was a critical output of the project. This was particularly important, as this was a safety-critical system. A tougher question was, what was enough documentation? This was where it got interesting. This application was not your typical XP target application—it was part of a larger system that multiple groups were developing at multiple sites. These other groups were using the standard, waterfall-style DPS process, not XP or short-iteration development. We had a potential impedance mismatch between the XP team and the rest of the project.

How much documentation?

Proposing no documentation would end the conversation. We decided to keep the conversation going and answer a question with a question. What did we want from our documentation? We needed

- enough documentation to define the product requirements, sustain technical reviews, and support the system's maintainers;
- clean and understandable source code; and
- some form of interface documentation, due to the impedance mismatch between groups.

These answers did not all align with XP out of the book, but they kept the conversation going. XP is not anti-documentation—it recognizes that documentation has a cost and that not creating the documentation

might be more cost effective. This, of course, violates conventional wisdom.

After acknowledging and addressing the project's objectives, I led the team through the cost-of-change pitch from Beck's *Extreme Programming Explained*.[1] The director, the manager, and some senior technologists agreed that XP addresses many of their current development problems. They also thought XP right out of the book would not work for them. What did we want to do differently?

Documentation and reviews were going to be the big roadblocks. We heard, "Requirements on note cards!?!" "I can't give a stack of note cards to the test team." "Bob in firmware needs the cards too." "Someone will loose the cards." I noticed that the company's "standard" process allowed use cases in the form described by Alistair Cockburn.[2] This is a text-based method, similar to user stories but with more details. Other DPS groups needed to look at the use cases, so we decided not to fight that battle—we had enough lined up already. We decided to use use cases.

Another objection was "We need documentation for the future generations of engineers that will maintain the product we are building. We need our senior people to look at the design to make sure your design will work." Our answer was that a good way to protect future software maintainers is to provide them with clean and simple source code, not binders full of out-of-date paper. Maintainers always go to the source code; it cannot lie, as documents can. XP relies on the source code being simple and expressive and uses refactoring to keep it that way.[3] The source code is the most important part of the design. At some point, the maintainers will need a high-level document to navigate the system.

A follow-on objection was that what one person thinks is readable source code is not to another. XP addresses this through pair programming. If a pair works hard at making the source readable, there is a really good chance that a third programmer who sees the code will find it readable, too. Plus, with collective code ownership, anyone can change the source code if need be.

Another expected objection was that code was not enough—we needed a documentation or transition strategy for whenever a project is put on the shelf or transferred to another team. Ideally, the documentation given to the maintainers describes the state of the software at the time it was shelved or transferred. This document can and should be written in a way that avoids needing frequent modification. As maintenance proceeds, the document will likely be neglected. Make the document high level enough so that the usual maintenance changes and bug fixes do not affect it. Let the documentation guide the future developers to the right part of the code—then they can use the high-quality, readable, simple source code to work out the details.

Following this strategy will not result in a huge document. Remember, you have some of the best detailed documentation available in the form of automated unit tests—working code examples of exactly how to use each object in the system. XP does not prohibit documentation; just realize it has a cost and make sure it is worth it. You can plan documentation tasks into any iteration. The advice here is to document what you have built, not what you anticipate building.

The next follow-on objection was that we'd never do the document at the end of the project. My reply: So, you would rather do a little bit at a time, and have to keep changing and rewriting it? Doesn't this sound like it will take a lot time? It does! So the management team must stick to its guns and do the high-level documentation task at the end of the project. Pay for it with less time wasted during development.

Documentation and reviews were going to be the big roadblocks.

Reviews

The director did not completely buy into this one and hence the final objection: "I still have my concerns. What if the design is no good? Do I have to wait until the end of the project to find out?" Pair programming was not reason enough for the management team to give up their review process. The big problem with DPS's development process was that it guaranteed that development would go slowly. The process went something like this: create a design, document it in Rose, schedule a review meeting, distribute the materials, have everyone review the materials, have the review meeting, collect the issues, fix the issues, maybe do another review, then finally write some code to see if the design works. This made sure the cost of change was high.

At the beginning of a project, you need to believe that the design can and will evolve.

Because of the safety-critical nature of the application, the management team was not willing to give up on reviews.

I proposed a more efficient way to do the reviews: the ask-for-forgiveness (rather than ask-for-permission) design process. We let the development team work for a month at a time on the system. At the end of the month, they assembled and reviewed a design-as-built review package. This took the review off the critical path, so the review process did not slow down the team. We agreed to document the significant designs within the iteration and review them with the review team. We gave the remaining issues that the reviewers found to the next iteration as stories. The idea here was to spend a small amount of time in the iteration documenting the design decisions that month. As it turned out, we really did not have to ask for forgiveness at all.

It's not about XP

It's about building better software predictably and faster. We compromised on a number of issues, but we did have agreement to use most of the XP practices: test-first programming, pair programming, short iterations, continuous integration, refactoring, planning, and team membership for the customer. Table 1 describes the XP practices we used and how we modified them to suit our needs. We added some process and formality: use cases, monthly design reviews, and some documentation. This adaptation of XP was a significant change in how DPS developed software, and we hoped to prove it was an improvement. The director thought XP offered a lot of promise for a better way to work that could lead to improved quality, faster development, better predictability, and more on-the-job satisfaction ("smiles per hour"). He said we were "making footprints in the sand." If we could improve quality, job satisfaction, productivity, or predictability, the director thought it might be worth doing XP. If we could improve any two of those factors, he thought there would be a big payoff.

The problems of poor quality, delivery delays, long delivery cycles, and burned-out engineers plague the software industry. XP struck a cord with our team's leaders. The techniques appeared to address some of the problems the team was facing, and the focus on testing and pair programming could help them build a quality product. XP's iterative nature could help the team determine how fast it could go and give the management team the feedback it needed. So, it's about getting the job done. XP is a set of techniques that seemed promising.

Projects can get stalled in the fuzzy front end.[4] This is a problem, especially in waterfall projects, where you must define all the requirements prior to starting the design process. In XP, as soon as you have defined a couple weeks' worth of user stories, development can start. Think of how hard it is to shave a month off the end of a project. Now think of how easy it would be to save that month just by starting development as soon as you have identified a month's worth of stories. Story development occurs concurrently with story implementation.

The first iteration

The team had three people, a customer and two developers, including me. We started by getting the unit test tool CppUnit set up and integrated with our development environment, VC++. This did not take long—the tools are pretty easy to use.

The project's customer (our systems engineer) gave us a requirements document. As we identified a functional requirement, we wrote it on a card. Each card named a use case. We did not bother to elaborate the use cases, just name them; in a few days, we had identified 125 use cases named. Picking the most important ones was relatively easy using this list.

In XP, the customer chooses the most valuable user stories and discusses them with the programmers. We were using use cases, a similar idea; our customer chose the most valuable use cases and elaborated them. For the early iterations, we decided to ignore use case extensions (which hold the special cases or variations) and keep the product definition simple.[2] We just assumed there were no special cases or error cases; because we were using XP, we believed we could ignore the details and not be penalized later. We also did not bother using use case diagrams, because they did not add any value to the development team. Our main goal in the first iteration was to build a little bit of the product, get some experience, and build some skill and confidence.

At the beginning of a project, you need to

Table 1

Summary of XP practices used

XP practice	Adoption status	Our experience
Planning game	Partially adopted	The team practiced scope limiting, task breakdown, and task signup techniques. We used use cases rather that user stories. We wrote the use cases from an existing requirements document and stored them in a database.
Small releases	Adopted	Iterations were one month long.
Metaphor	Not adopted	A metaphor had not yet evolved, and we didn't develop one. Instead, a high-level design evolved and was recorded in a small set of UML diagrams and explanatory text. It played the role of our metaphor.
Simple design	Adopted	The design did not have anticipatory elements. A monthly design-as-built review let the senior people monitoring the project see the team's design decisions.
Functional testing	Adopted	We developed functional tests in a custom scripting language. The tests demonstrated that the application logic met the customer's need. However, the team got behind on automated acceptance tests. This is not recommended.
Test-first design	Adopted	We wrote the first line of production code using test-first design. We wrote the program in C++ and used CppUnit 1.5 as our unit test tool.
Refactoring	Adopted	We refactored regularly, and the design evolved smoothly.
Pair programming	Adopted	We could develop tests, interfaces, and simulations on our own but used pair programming to create production code.
Collective ownership	Adopted	We collectively owned the code. On one occasion, new code that required special knowledge resulted in a module owned by one programmer. Development slowed on that part of the system.
Continuous integration	Adopted	During our first iteration, continuous integration was no problem. As soon as we added a second pair to the project, the team had integration problems. We quickly learned how to avoid collisions and do merges.
40-hour week (a sustainable pace)	Adopted	The team worked at a sustainable pace. A couple times, we put in overtime to meet the iteration goals.
On-site customer	Partially adopted	A systems engineer acted as the on-site customer. We derived our acceptance tests from the use cases. The customer was not directly responsible for these tests. The team scared the customer a couple times by delivering his use cases in less time than it took him to define them.
Coding standards	Adopted	The main coding standard was "make the code look like the code that is already there." The team used header and Cpp file source templates to provide the company-required comment blocks. The comment blocks were mainly noise that hid the code.
Open workspace	Not adopted	There was no open workspace. Workstations were in the corners, making pair programming awkward. The roadblocks to building a team workspace were political.

believe that the design can and will evolve. Otherwise, the desire to do up-front specification work will put the team into analysis paralysis. Knowing that the design can evolve sets the team free to start building the system as soon as some functionality is identified. Simplifying assumptions keeps complexity out of the code, at least temporarily. John Gall wrote, "A complex system that works is invariably found to have evolved from a simple systems that works."[5] Thus, it really helps to make these simplifying and scope-limiting decisions in *each* iteration. This lets the core features drive the design.

We had a planning meeting and discussed the features that were to be included in the first iteration. My partner and I had OOD experience but no real XP experience (except for the week I spent in class). We wanted a guide to our first iteration, so we spent about half a day with a whiteboard looking at design ideas. Because we were unsure of XP, we were not sure if we could really start coding without doing some design. We did a little bit of design, or as we called it a Little Design Up Front (LDUF). Ron Jeffries calls this a Quick design session.[6]

In our LDUF session, we found a group of collaborating objects we thought would meet the needs of our first stories. We worked from hand-drawn copies, not bothering with a diagramming or CASE tool. We knew things would change, and we did not want to spend our time making it look pretty. It gave us confidence and a vision of where we were going.

During the iteration planning meeting and our LDUF session, we identified some of

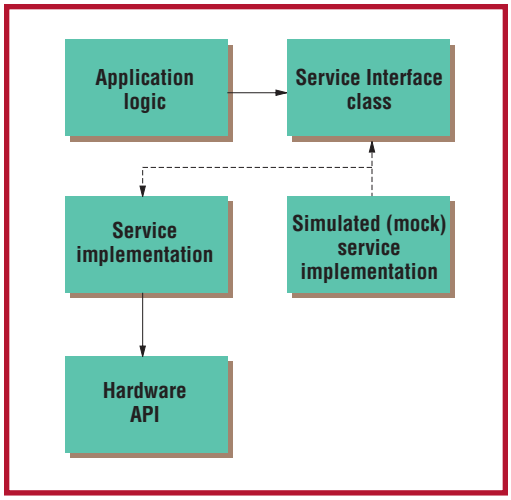


Figure 1. Simulating the interface by creating a Mock Object.

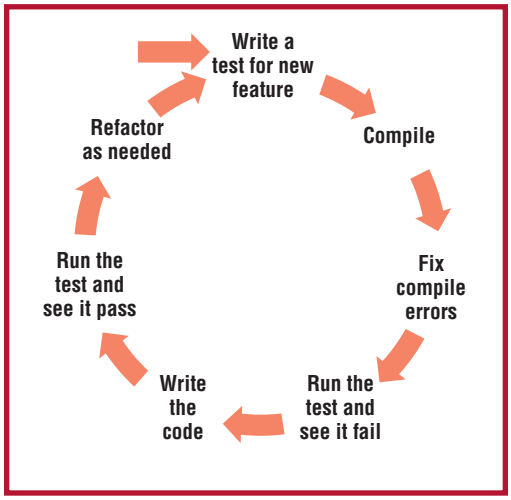


Figure 2. The test-first design process.

the interfaces we needed to support the iteration 1 features. These acted as placeholders for the real hardware. We then added a simulation of the interface by creating a Mock Object (see Figure 1).[7] This facilitated development and also kept volatile entities such as the database schema, GUI, hardware dependencies, and protocol from creeping into the application logic. We witnessed an example of designing for testability leading to reduced coupling in the application.

So we sat down to write our first line of code. We picked a candidate class from our hand-drawn LDUF diagram and followed the test-first design process. Figure 2 represents the test-first design process, XP's innermost feedback loop. Our first line of code was a test. The test did not compile. We fixed the compile. We ran the test. It failed. We fixed the test. We were finally in maintenance! Figure 2 presents a high-level look at XP's iterative process.

We also established pair-programming guidelines. We could develop test cases, simulators, and interface classes on our own but had to do all other production code in pairs. The first iteration had significant downtime, considering only one pair was working on the project. Meetings were a real productivity killer. Pair programming was fun and intense; we were able to stay on task. If one partner lost track of where we were going, the other partner quickly resynched their efforts. We taught each other about the tools and learned new skills.

The coding standard is a team agreement to make the code look the same—it is the team's code, so it should all look the same. It turned out that my partner and I had a compatible coding style. As more people joined the team, we established a self-documenting coding standard: "Make the new or modified code look like the code that is already there." Cool, a one-line coding standard! However, there was pressure to use the division's coding standard. From a "choosing your battles" point of view, we gave into using the standard comment blocks in front of each function.

Later iterations

We planted a seed of functionality at the center of this subsystem application and simulated its interactions with its environment. We brought in new stories that made the seed grow, incrementally adding new functionality and complexity. From its simple beginning of a half dozen classes and a few simulations, the clean, loosely coupled design evolved to about 100 classes.

Unit test volume grew. The unit tests saved us numerous times from unexpected side-effect defects. We could fix the defects immediately by adding the code that broke the tests.

We created our own acceptance test scripting language to drive transactions into the system and used text file comparisons of simulation output to confirm system operation. We were able to design simulations that stressed the system beyond the limits expected in the field. Unfortunately, the team got behind in acceptance testing. I do not recommend this.

Evolutionary design

Evolutionary design relieved a lot of pressure from the team. We didn't have to create the best design of all time for things we were not quite sure about—only the best design for what we knew about at that moment. We made good design decisions one at a time. Our automated tests and refactoring gave us the confidence that we could

continue to evolve the system.

Object-oriented design is an important supporting practice of XP. OO programming languages let you build software in small independent pieces, which test-first programming promotes.[8]

Project manager surprises

Not only were the programmers happy with their creation, but the project manager was as well. After the fourth iteration, he said, "I'd only have three documents by now! Instead I have a piece of the system that works!"

The manager discovered another benefit. Usually a project manager coordinating a team's work with other teams spends a lot of time juggling priorities and figuring out task dependencies. On the XP team, dependencies between features were almost nonexistent. We built features in the order of customer priority, not internal software framework order dictated by a BDUF (Big Design Up Front). The team was agile and able to adapt to the other subsystems' changing needs.

Building the team

We built the team slowly, while we were developing skills. We felt we could absorb one or two people per iteration. We did not let newcomers take tasks right away, but used them mainly as pair partners during their first iteration. Then, as they got to know the system and our practices, they started to take on tasks at the iteration planning meetings. We didn't assume that team velocity would increase when we added a new person to the team—we measure velocity, not predict it.

The DPS way of developing software made up for programmer inexperience by having senior engineers review the less experienced engineers' work. In XP projects, you must still address the spread of expertise; for instance, it is critical to have at least one senior engineer on the team. We don't give senior people a big title or special role, but we need them. They help spread the wealth of knowledge, and both they and their pair partners learn.

The end of the story

Unfortunately, I cannot present the story of how the project completed, because it was mothballed due to changing market needs. This is in the spirit of one of XP's mantras: work on the most important thing first. Nevertheless, the team and the managers were impressed with our results in terms of productivity and quality. Because of this project, we started two other pilot projects.

In my experience, when the engineers want XP, the management doesn't, and if management wants XP, the engineers don't. Where is the trust between management and engineering?

To managers: Try XP on a team with open-minded leaders. Make it okay to try new things. Encourage the XP practices. Provide good coaching. Challenge your team to go against the status quo. Recruit a team that wants to try XP rather than force a team to use XP. Make sure the organization sees that no one will be punished for trying something different. When handpicking XP practices, you might compromise the self-supporting nature of XP. Try as much of XP as you can. Iterations are short. Feedback comes often.

To engineers: Develop a sales pitch. Identify problems that you might solve. Identify the benefits, identify the risks. Do a pilot project. Iterations are short. Feedback comes often.

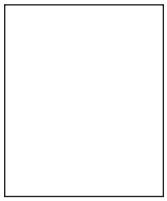
Acknowledgments

I thank the real client that provided the experience to write this article (who wished to remain anonymous). I also want to thank Chris Biegay and Jennifer Kohnke for a job well done in helping me prepare this article.

References

1. K. Beck, *Extreme Programming Explained*, Addison-Wesley, Reading, Mass., 1999.
2. A. Cockburn, *Writing Effective Use Cases*, The Crystal Collection for Software Professionals, Addison-Wesley, Reading, Mass., 2000.
3. M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison Wesley Longman, Boston, 1999.
4. S. McConnell, *Rapid Development*, Microsoft Press, 1996.
5. J. Gall, *Systemantics: How Systems Really Work and How They Fail*, 2nd ed., The General Systemantics Press, Ann Arbor, Mich., 1986.
6. R.E. Jeffries, *Extreme Programming Installed*, Addison-Wesley, Reading, Mass., 2001.
7. T. Mackinnon, S. Freeman, and P. Craig, *Endo-Testing: Unit Testing with Mock Objects, XP Examined*, Addison-Wesley, Reading, Mass., 2001.
8. R. Martin, "Design Principles and Design Patterns," www.objectmentor.com/publications/Principles%20and%20Patterns.PDF (current 12 Oct. 2001).

About the Author



James Grenning is the director of consulting at Object Mentor and is currently practicing and coaching Extreme Programming. Areas of interest are organizational change, software process improvement, and the business impact of agile software development. He helped create the Manifesto for Agile Software Development (http://AgileAlliance.org) and is a signatory. He is also a member of the ACM. He earned his BS in electrical engineering and computer science from the University of Illinois, Chicago. Contact him at Object Mentor, 565 Lakeview Parkway, Ste. 135, Vernon Hills, IL 60061; grenning@objectmentor.com.