Embracing Change with Extreme Programming

Extreme Programming turns the conventional software process sideways. Rather than planning, analyzing, and designing for the far-flung future, XP programmers do all of these activities—a little at a time—throughout development.

Kent Beck
First Class
Software

In the beginning was the waterfall (Figure 1a). We would get the users to tell us once and for all exactly what they wanted. We would design the system that would deliver those features. We would code it. We would test to make sure the features were delivered. All would be well.

All was not well. The users didn't tell us once and for all exactly what they wanted. They didn't know. They contradicted themselves. They changed their minds. And the users weren't the only problem. We programmers could think we were making great progress only to discover three-fourths of the way through that we were one-third of the way through.

If long development cycles were bad because they couldn't adapt to changes, perhaps what we needed was to make shorter development cycles. As Figure 1b shows, the waterfall begat iterations.

The waterfall model didn't just appear. It was a rational reaction to the shocking measurement that the cost of changing a piece of software rose dramatically over time. If that's true, then you want to make the biggest, most far-reaching decisions as early in the life cycle as possible to avoid paying big bucks for them.

The academic software engineering community took the high cost of changing software as a challenge, creating technologies like relational databases, modular programming, and information hiding. What if all that hard work paid off? What if we got good at reducing the costs of ongoing changes? What if we didn't have to settle for taking a cleaver to the waterfall? What if we could throw it in a blender?

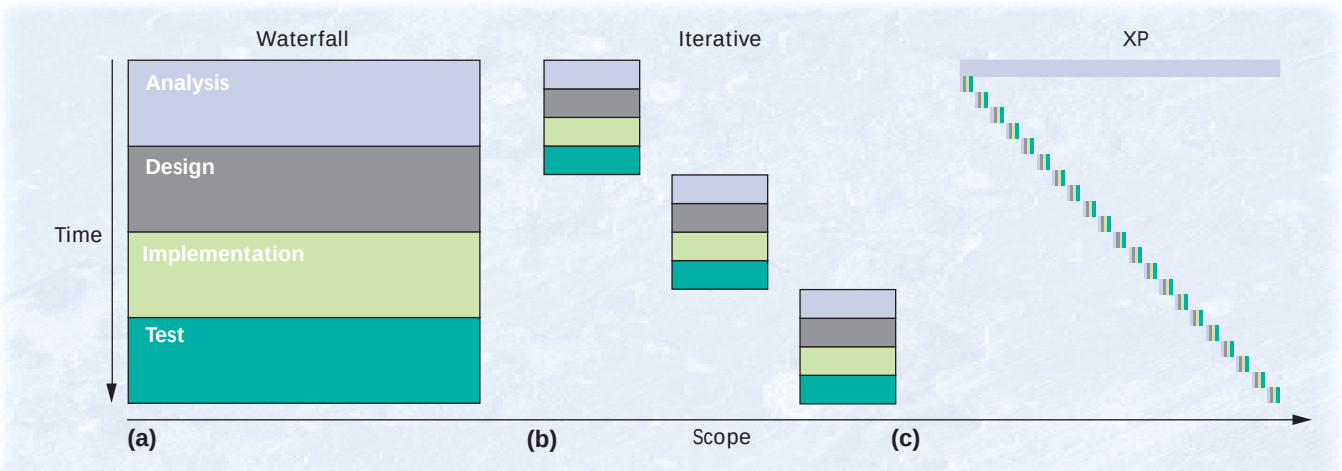

Figure 1. The evolution of the Waterfall Model (a) and its long development cycles (analysis, design, implementation, test) to the shorter, iterative development cycles within, for example, the Spiral Model (b) to Extreme Programming's (c) blending of all these activities, a little at a time, throughout the entire software development process.

0018-9162/99/$10.00 © 1999 IEEE

XP Practices

Here is a quick summary of each of the major practices in XP.

Planning game. Customers decide the scope and timing of releases based on estimates provided by programmers. Programmers implement only the functionality demanded by the stories in this iteration.

Small releases. The system is put into production in a few months, before solving the whole problem. New releases are made often—anywhere from daily to monthly.

Metaphor. The shape of the system is defined by a metaphor or set of metaphors shared between the customer and programmers.

Simple design. At every moment, the design runs all the tests, communicates everything the programmers want to communicate, contains no duplicate code, and has the fewest possible classes and methods. This rule can be summarized as, "Say everything once and only once."

Tests. Programmers write unit tests minute by minute. These tests are collected and they must all run correctly. Customers write functional tests for the stories in an iteration. These tests should also all run, although practically speaking, sometimes a business decision must be made comparing the cost of shipping a known defect and the cost of delay.

Refactoring. The design of the system is evolved through transformations of the existing design that keep all the tests running.

Pair programming. All production code is written by two people at one screen/keyboard/mouse.

Continuous integration. New code is integrated with the current system after no more than a few hours. When integrating, the system is built from scratch and all tests must pass or the changes are discarded.

Collective ownership. Every programmer improves any code anywhere in the system at any time if they see the opportunity.

On-site customer. A customer sits with the team full-time.

40-hour weeks. No one can work a second consecutive week of overtime. Even isolated overtime used too frequently is a sign of deeper problems that must be addressed.

Open workspace. The team works in a large room with small cubicles around the periphery. Pair programmers work on computers set up in the center.

Just rules. By being part of an Extreme team, you sign up to follow the rules. But they're just the rules. The team can change the rules at any time as long as they agree on how they will assess the effects of the change.

We might get a picture like the one shown in Figure 1c. It's called Extreme Programming.

ANATOMY OF XP

XP turns the conventional software process sideways. Rather than planning, analyzing, and designing for the far-flung future, XP exploits the reduction in the cost of changing software to do all of these activities a little at a time, throughout software development. (The "XP Practices" sidebar will give you a quick grasp of the practices and philosophy underlying XP. These practices are designed to work together, and trying to examine any one soon leads you to the rest. The "Roots of XP" sidebar on page 73 traces the historical antecedents of these practices.)

XP development cycle

Figure 2 shows XP at timescales ranging from years to days. The customer picks the next release by choosing the most valuable features (called *stories* in XP) from among all the possible stories, as informed by the costs of the stories and the measured speed of the team in implementing stories.

The customer picks the next iteration's stories by choosing the most valuable stories remaining in the release, again informed by the costs of the stories and the team's speed. The programmers turn the stories into smaller-grained tasks, which they individually accept responsibility for.

Then the programmer turns a task into a set of test cases that will demonstrate that the task is finished. Working with a partner, the programmer makes the test cases run, evolving the design in the meantime to maintain the simplest possible design for the system as a whole.

Stories

XP considers the period before a system first goes into production to be a dangerous anomaly in the life of the project and to be gotten over as quickly as possible. However, every project has to start somewhere.

The first decisions to make about the project are what it could do and what it should do first. These decisions are typically the province of analysis, hence the thin blue analysis rectangle at the top of Figure 1c. You can't program until you know what you're programming.

You put the overall analysis together in terms of stories, which you can think of as the amount of a use case that will fit on an index card. Each story must be business-oriented, testable, and estimable.

A month is a good long time to come up with the stories for a 10 person-year project. It's true that it isn't enough to explore all of the possible issues thoroughly. But forever isn't long enough to explore all of the issues thoroughly if you never implement.

Release

Notice in Figure 2 that we don't implement all of the stories at first. Instead, the customer chooses the smallest set of the most valuable stories that make sense together. First we implement those and put them into production. After that we'll implement all the rest.

Picking the scope for a release is a little like shopping for groceries. You go to the store with $100 in your pocket. You think about your priorities. You look at the prices on the items. You decide what to buy.

In the planning game (the XP planning process), the items are the stories. The prices are the estimates on the stories. The budget is calculated by measuring the team's output in terms of estimated stories delivered per unit time.

The customer can either load up a cart (pick a set of stories) and have the programmers calculate the finish date or pick a date and have the programmers calculate the budget, then choose stories until they add up.

Iteration

The goal of each iteration is to put into production some new stories that are tested and ready to go. The process starts with a plan that sets out the stories to be implemented and breaks out how the team will accomplish it. While the team is implementing, the customer is specifying functional tests. At the end of the iteration, the tests should run and the team should be ready for the next iteration.

Iteration planning starts by again asking the customer to pick the most valuable stories, this time out of the stories remaining to be implemented in this release. The team breaks the stories down into tasks, units of implementation that one person could implement in a few days. If there are technical tasks, like upgrading to a new version of a database, they get put on the list too.

Next, programmers sign up for the tasks they want to be responsible for implementing. After all the tasks are spoken for, the programmer responsible for a task estimates it, this time in ideal programming days. Everyone's task estimates are added up, and if some programmers are over and some are under, the undercommitted programmers take more tasks.

Over the course of the iteration, the programmers implement their tasks. As they complete each task, they integrate its code and tests with the current system. All tests must run or the code cannot be integrated.

As the customer delivers the functional tests during the iteration, they are added to the suite. At the end of the iteration, all unit tests and all functional tests run.

Task

To implement a task, the responsible programmer first finds a partner because all production code is written with two people at one machine. If there is any question about the scope or implementation approach, the partners will have a short (15-minute) meeting with the customer and/or with the programmers most knowledgeable about the code most likely to be touched during implementation.

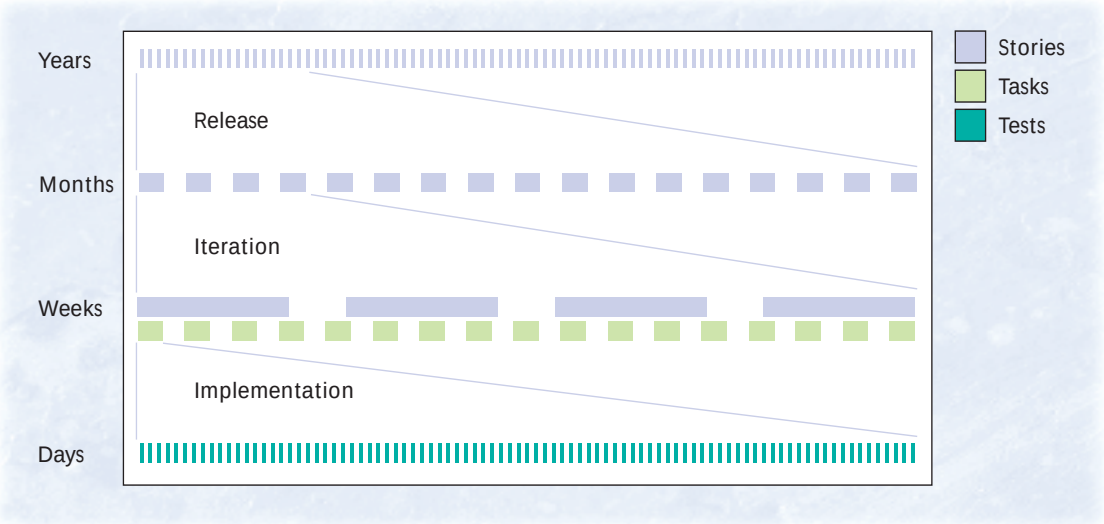

Figure 2. XP according to various timescales. At the scale of months and years, you have the stories in this release and then the stories in future releases. At the scale of weeks and months, you have stories in this iteration and then the stories remaining in this release. At the scale of days and weeks, you have the task you are working on now and then the rest of the tasks in the iteration. And at the scale of minutes and days, you have the test case you are working on now and then the rest of the test cases that you can imagine.

Roots of XP

The individual practices in XP are not by any means new. Many people have come to similar conclusions about the best way to deliver software in environments where requirements change violently.[1-3]

The strict split between business and technical decision making in XP comes from the work of the architect Christopher Alexander, in particular his work *The Timeless Way of Building*,[4] where he says that the people who occupy a structure should (in conjunction with a building professional) be the ones to make the high-impact decisions about it.

XP's rapid evolution of a plan in response to business or technical changes echoes the Scrum methodology[5] and Ward Cunningham's Episodes pattern language.[6]

The emphasis on specifying and scheduling projects from the perspective of features comes from Ivar Jacobson's work on use cases.[7]

Tom Gilb is the guru of evolutionary delivery. His recent writings on EVO[8] focus on getting the software into production in a matter of weeks, then growing it from there.

Barry Boehm's Spiral Model was the initial response to the waterfall.[9] Dave Thomas and his colleagues at Object Technology International have long been champions of exploiting powerful technology with their JIT method.[10]

XP's use of metaphors comes from George Lakoff and Mark Johnson's books, the latest of which is *Philosophy in the Flesh*.[11] It also comes from Richard Coyne, who links metaphor with software development from the perspective of postmodern philosophy.[12]

Finally, XP's attitude toward the effects of office space on programmers comes from Jim Coplien,[13] Tom DeMarco, and Tim Lister,[14] who talk about the importance of the physical environment on programmers.

References

1. J. Wood and D. Silver, *Joint Application Development*, John Wiley & Sons, New York, 1995.
2. J. Martin, *Rapid Application Development*, Prentice Hall,Upper Saddle River, N.J.,1992.
3. J. Stapleton, *Dynamic Systems Development Method*, Addison Wesley Longman, Reading, Mass., 1997.
4. C. Alexander, *The Timeless Way of Building*, Oxford University Press, New York, 1979.
5. H. Takeuchi and I. Nonaka, "The New Product Development Game," *Harvard Business Rev.*, Jan./Feb. 1986, pp. 137-146.
6. W. Cunningham, "Episodes: A Pattern Language of Competitive Development," *Pattern Languages of Program Design 2*, J. Vlissides, ed., Addison-Wesley, New York, 1996.
7. I. Jacobsen, *Object-Oriented Software Engineering*, Addison-Wesley, New York, 1994.
8. T. Gilb, *Principles of Software Engineering Management*, Addison-Wesley, Wokingham, UK, 1988.
9. B. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, pp. 61-72.
10. D. Thomas, "Web Time Software Development," *Software Development*, Oct. 1998, p. 80.
11. G. Lakoff and M. Johnson, *Philosophy in the Flesh*, Basic Books, New York, 1998.
12. R. Coyne, *Designing Information Technology in the Postmodern Age*, *MIT Press*, Cambridge, Mass., 1995.
13. J.O. Coplien, "A Generative Development Process Pattern Language," *The Patterns Handbook*, L. Rising, ed., Cambridge University Press, New York, 1998, pp. 243-300.
14. T. DeMarco and T. Lister, *Peopleware*, Dorset House, New York, 1999.

From this meeting, the partners condense the list of test cases that need to run before the task is done. They pick a test case from the list that they are confident they can implement and that will teach them something about the task. They code up the test case. If the test case already runs, they go on. Normally, though, there is work to be done.

When we have a test case and it doesn't run, either

- we can see a clean way to make it run, in which case we make it run; or
- we can see an ugly way to make it run, but we can imagine a new design in which it could be made to run cleanly, in which case we refactor the system to make it run cleanly; or
- we can see an ugly way to make it run, but we can't imagine any refactoring, in which case we make it run the ugly way.

After the test case runs, if we see how to refactor the system to make it even cleaner, we do so.

Perhaps during the implementation of this test case we imagine another test case that should also run. We note the new test case on our list and continue. Perhaps we spot a bigger refactoring that doesn't fit into the scope of our current test. We also note that and continue. The goal is to remain focused so we can do a good job and at the same time not lose the benefits of the insights that come during intense interaction with code.

Test

If there is a technique at the heart of XP, it is unit testing. As you saw above, unit testing is part of every programmer's daily business. In XP, however, two twists on conventional testing strategies make tests far more effective: Programmers write their own tests and they write these tests before they code. If programming is about learning, and learning is about getting lots of feedback as quickly as possible, then you

can learn much from tests written by someone else days or weeks after the code. XP primarily addresses the accepted wisdom that programmers can't possibly test their own code by having you write code in pairs.

Some methodologies, like Cleanroom,[1] prohibit programmers testing or in some cases even compiling their own programs. The usual process has a programmer write some code, compile it, make sure it works, then pass it on to a testing organization. The bench testing takes the form of single-stepping through the code and watching variables, or interpreting the results of print statements, or poking a few buttons to make sure the list item turns green.

The XP testing strategy doesn't ask any more work than the usual bench testing strategies. It just changes the form of the tests. Instead of activities that evaporate into the ether as soon as they are finished, you record the tests in a permanent form. These tests will run automatically today, and this afternoon after we all integrate, and tomorrow, and next week, and next year. The confidence they embody accumulates, so an XP team gains confidence in the behavior of its system over time.

As I mentioned earlier, tests also come from the customers. At the beginning of an iteration, the customers think about what would convince them that the stories for an iteration are completed. These thoughts are converted into systemwide tests, either directly by the customer using a textual or graphical scripting language or by the programmers using their own testing tools. These tests, too, accumulate confidence, but in this case they accumulate the customer's confidence in the correct operation of the system.

WHEN SOMETHING GOES WRONG

Talking about how a method works when it works perfectly is about like describing precisely how you will descend a monstrous patch of white water. What is interesting is precisely what you will do when the unexpected or undesired happens. Here are some common failures and possible Extreme reactions.

Acxiom: Working toward a Common Goal

Jim Hannula, Acxiom

On top of a data warehouse, Acxiom built a campaign management application using Forté's distributed OO development tool. The small development team—consisting of 10 developers—built the application by relying on sound OO principles and a strong team development approach.

During the final two years of the application's three years of development, the team—comprised of managers, business analysts, developers, testers, and technical writers—used Extreme Programming techniques, which proved to be instrumental in our success.

We know we have a good design if it's simple. Some of our past designs tried even to account for future iterations of our application. We discovered that we were not very good at that. If we use patterns and communicate well, we can develop a sound application that is flexible and can still be modified in the future.

Refactoring is a major part of our development effort. It was evident to us that if we were afraid to change some code because we did not know what it did, we were not very good developers. We were letting the code control us. If we don't know what the code does now, we break it and find out. It is better to implement a solid piece of code than it is to let a piece of code control the application.

Unit testing was a hard piece to implement because Forté did not have a ready-built testing framework. We developed our own testing framework and have been successful implementing it. Recently we started using Java as a development language and now use JUnit as a testing tool.

The key to XP is setting developer and team expectations. We have found all developers on the team must buy into Extreme or it doesn't work. We tell prospective developers if they do not want to follow our development style, this is not a good team for them. One person not buying in to the approach will bring down the whole team. XP focuses on the team working together to come up with new ideas to develop the system.

When we first started with XP, some of the developers did not want to follow it. They felt that it would hurt their development style and that they would not be as productive. What happened was that their pieces of the application were producing the most problem reports. Since

Team: managers, business analysts, developers, testers, and technical writers
Application: campaign management dbase
Time: three years

they were not developing in pairs, two people had not designed the subsystem and their skills were falling behind the other developers who were learning from each other. Two well-trained developers working together and with the rest of the team will always outperform one "intelligent" developer working alone.

A misconception about XP is that it stifles your creativity and individual growth. It's actually quite the contrary. XP stimulates growth and creativity and encourages team members to take chances. The key is to decide the direction of the corporation and stand behind the hard decisions.

XP is not extreme to our team. It's a method that uses a common-sense development approach. Everyone works together toward a common goal.

DaimlerChrysler: The Best Team in the World

Chet Hendrickson, DaimlerChrysler

Team: 10 programmers, 15 total
Application: large-scale payroll system
Time: four years

The C3 project began in January 1995 under a fixed-priced contract that called for a joint team of Chrysler and contract partner employees. Most of the development work had been completed by early 1996. Our contract partners had used a very GUI-centered development methodology, which had ignored automated testing. As a result, we had a payroll system that had a lot of very cool GUIs, calculated most employees' pay incorrectly, and would need about 100 days to generate the monthly payroll. Most of us knew the program we had written would never go into production.

We sought Kent Beck to help with performance tuning. He found what he had often found when brought in to do performance tuning: poorly factored code, no repeatable tests, and a management that had lost confidence in the project. He went to Chrysler Information Services management and told them what he had found, and that he knew how to fix it. Throw all the existing code away! The first full XP project was born.

We brought Kent in as head coach; he would spend about a week per month with us. Ron Jeffries was brought in as Kent's full-time eyes and ears. The fixed-price contract was cancelled, and about one-half of the Chrysler developers were reassigned. Martin Fowler, who had been advising the Chrysler side of the project all along and clashing with the fixed-price contractor, came in to help the customers develop user stories. From there, we followed Kent as he made up the rules of XP. A commitment schedule was developed, iterations were laid out, rules for testing were established, and paired programming was tried and accepted as the standard.

At the end of 33 weeks, we had a system that was ready to begin performance tuning and parallel testing. Ready to begin tuning because it was well factored and backed up by a full battery of unit tests. And, ready to begin parallel testing because a suite of functional tests had shown the customers that the required functionality was present.

That increment of C3 launched in May 1997, not as soon as we had hoped. We were slowed by two factors. First, we had decided to replace only the internals of the payroll system. We left all of the external interfaces intact. Matching up the output from our new system to the old payroll master ended up being a much larger task then we had originally estimated. Second, we decided not to launch during any pay period with special processing requirements, such as W-2 processing, profit sharing, or general merit pay increases. This effectively eliminates November through April.

Since the launch of the monthly system, we've added several new features, and we have enhanced the system to pay the biweekly paid population. We have been paying a pilot group since August 1998 and will roll out the rest before the Y2K code freeze in November 1999.

Looking back on this long development experience, I can say that when we have fallen short of keeping our promises to our management and our customers, it has been because we have strayed from the principles of XP. When we have driven our development with tests, when we have written code in pairs, when we have done the simplest thing that could possibly work, we have been the best software development team on the face of the earth.

Underestimation

From time to time you will commit to more than you can accomplish. You must reduce the occurrence of underestimation as much as possible by getting lots of practice estimating. If you are overcommited, you first try to solve the problem in the family. Have you slipped away from the practices? Are you testing, pairing, refactoring, and integrating as well as you can? Are you delivering more than the customer needs in places?

If you can't find any way to go faster, you have to ask the customer for relief. Staying committed to more work than you can confidently complete is a recipe for frustration, slipping quality, and burnout. Don't do that. Re-estimate based on what you've learned, then ask the customer to reschedule. We can only complete two out of three stories, so which two should we finish and which one goes in the next iteration or release? Is there a story that has more critical parts and less critical parts so we can split it and deliver the most important parts now and the less important parts later?

Uncooperative customers

What if you get a customer who just won't play the game? They won't specify tests, they won't decide on priorities, they won't write stories. First, by completing functionality iteration after iteration, and by giving the customer clear control over development, you are trying to build a trust relationship with the customer. If trust begins to break down, figure out if it's your fault. Can you do a better job of communicating?

If you can't solve the problem on your own, you have to ask the customer for help. Extreme programmers simply don't go ahead based on their own guesses. Explain or demonstrate the consequences to the customer. If they don't change, make your concerns more visible. If no one cares enough to solve the problem, perhaps the project isn't a high enough priority to go on.

Turnover

What if someone leaves? Won't you be stuck without documents and reviews? First, a certain amount of turnover is good for the team and for the people on the team. However, you'd like people to leave for positive reasons. If programmers go home at the end of every week seeing the concrete things they have accomplished for the customer, they are less likely to get frustrated and leave.

When someone leaves an XP project, it isn't like they can take away any secrets that only they know. Two people were watching every line go into the system. And whatever information does walk out the door, it can't hurt the team too much because they can run the tests to ensure that they haven't broken anything out of ignorance.

New people on an XP team spend the first couple of iterations just pairing with more experienced people, reading tests, and talking with the customer. When they feel ready, they can accept responsibility for tasks. Over the course of the next few iterations, their personal velocity will rise as they demonstrate that they can deliver their tasks on time. After a few months, they are indistinguishable from the old salts.

Programmers that don't work out with the team are a problem, too. XP is an intensely social activity, and not everyone can learn it. It also requires abandoning old habits, which can be difficult, especially for high-status programmers. In the end, though, the many forms of feedback in XP make it clear who is working out and who isn't. Someone who consistently doesn't complete tasks, whose integrations cause problems for other people, who doesn't refactor, pair, or test …. Everyone on the team knows the score. And the team is better off without that person, no matter how skilled.

Changing requirements

The bugaboo of most software development is just not a problem in XP. By designing for today, an XP system is equally prepared to go any direction tomorrow. Things that are like what you've already done will be easier, just by the nature of refactoring to satisfy "once and only once," but those are precisely the things that are most likely to happen. However, should a radically new requirement arise, you won't have to unwind (or live with) a lot of mechanism built on speculation.

I didn't initially realize the extent to which XP can adapt to changing requirements. The first version of XP assigned stories to all the iterations in a release, as part of release planning. The team discovered that they could get better results with less planning by only asking the customer to pick which stories should be in the present iteration. If a new story comes up, you

Ford Motor: A Unique Combination of Agility and Quality

Don Wells, Ford Motor

Finance Systems at Ford Motor has been developing the Vehicle Costing and Profit System (VCAPS), an analysis tool that produces reports on production revenues, expenses, net income, and profit. The input is a bill of materials, fixed costs and expenses, and variable costs such as labor hours. VCAPS assembles this data into detailed cost analysis reports to support corporate-level forecasting and decision making.

Ford started VCAPS in 1993 and built it with VisualWorks and GemStone Smalltalk. VCAPS is now being maintained with a small staff and is to be replaced with a newer system.

The VCAPS project challenged us two ways. First, the analysts wanted modifications and new functionality before each run. Constantly changing requirements kept us in reaction mode. We never caught up. Second, the system needed to be run in a limited span of time. But the system took a long time to process and required lengthy manual input before producing final output. A bug could waste precious time by requiring a rerun.

XP offered us a unique combination: agility to meet the volatile requirements on time and quality to avoid the dreaded rerun.

We began XP with the planning game. It was a failure. Customers and management were unaccustomed to negotiating schedules. The commitment schedule produced was perceived as lacking credibility and utility. We had to swap in Microsoft Project schedules, which could be modified without large meetings and could produce the kinds of artifacts management was used to seeing and taking action on.

We continued by adding a few unit tests. Automated unit testing was an enormous success. After a year, we had 40 percent test coverage and management had measured a 40 percent drop in bug reports. XP was being noticed.

Team: 12 programmers, 17 total
Application: cost analysis system
Time: six years

We solved problems by adding XP practices. Tests enabled continuous integration and small releases. These allowed us to roll in collective ownership and refactoring. We were working toward simple design. Building momentum, we tried pair programming. We had to work hard to get pair programming going. Our developers found it awkward; it took a while to become comfortable.

After a year and a half, the decrease in system failures had reduced the number of emergency releases to a point where customers and managers noticed far greater system stability. Overall, XP was very successful in our environment.

Tariff System: Tests You Can Read

Rob Mee, Independent consultant

Tariff System is a subsystem of a large Smalltalk/GemStone project at a major international container-shipping company. Using XP practices, Tariff System was taken from inception to production in three months by a team of three. The resulting system proved to be unusually stable and easy to maintain.

At the outset of the project, the team resolved to adhere to several core XP practices: always program in pairs, use the simplest design possible, refactor aggressively, and write extensive unit tests. All of these practices were very effective. One XP idea that initially seemed far-fetched was writing tests before writing the code that satisfied them. We were surprised to find that in fact this helped bring our designs into focus, enabling us to work more quickly.

Another practice we employed from the beginning was collecting requirements from users in the form of user stories. We had mixed results with this. As programmers focused on coding, we found the role of facilitating and negotiating with users difficult. More important was the fact that users needed lots of help writing stories that were both relevant and unambiguous. In the end, we felt that perhaps XP was missing a project role. We needed someone from the development team whose primary focus—and particular talent—was interacting with users.

In our efforts to refactor test cases and fixtures, we discovered that creating little languages for our major domain objects dramatically improved the readability and brevity of our test code. It also practically eliminated the time we spent thinking about how to create object instances when writing tests. We defined grammars for about ten of our domain classes. Here's a simple example used to construct a Service Offering:

newFromString: 'from Oakland to Tokyo shipping toys: 20ft containers $500; 40ft containers $1000'.

Team: three developers
Application: shipping tariff calculation system
Time: three months

The constructor uses a parser, automatically generated from a grammar, to produce the domain object. The code to instantiate this object using standard constructors would have taken many lines, would have been difficult to read, and would have distracted from the test case itself.

Eventually, we discovered that we could combine the individual domain languages into a larger description of the system as a whole, which proved to be a valuable tool in the expression of functional tests.

don't have to shuffle the remainder of the iterations, you just put it in the pile. One or two weeks later, if the story still seems urgent, the customer will pick it.

Planning one iteration at a time also introduces a pleasing self-similarity. At the scale of months and years, you have the stories in this release and then the stories in future releases. At the scale of weeks and months, you have stories in this iteration and then the stories remaining in this release. At the scale of days and weeks, you have the task you are working on now and then the rest of the tasks in the iteration. And at the scale of minutes and days, you have the test case you are working on now and then the rest of the test cases that you can imagine.

XP is by no means a finished, polished idea. The limits of its application are not clear. To try it today would require courage, flexibility, and a willingness to abandon the project should your use of XP be failing.

My strategy is first to try XP where it is clearly applicable: outsourced or in-house development of small- to medium-sized systems where requirements are vague and likely to change. When we begin to refine XP, we can begin to try to reduce the cost of change in more challenging environments.

If you want to try XP, for goodness sake don't try to swallow it all at once. Pick the worst problem in your current process and try solving it the XP way. When it isn't your worst problem any more, rinse and repeat. As you go along, if you find that any of your old practices aren't helping, stop doing them.

This adoption process gives you a chance to build your own development style—which you will have to do in any case—to mitigate the risks should XP not work for you and to continue delivering as you change. ❖

Reference

1. S. Prowell et al., *Cleanroom Software Engineering*, Addison Wesley Longman, Reading, Mass., 1999.

Kent Beck owns and operates First Class Software, your typical one-person consulting company masquerading behind a fancy name and an answering machine. In addition to two books and 50 articles, he is the author of the forthcoming Extreme Programming Explained: Embrace Change *(Addison Wesley Longman, Reading, Mass., 2000). Contact him at kentbeck@csi.com.*