1

Introduction to Extreme Programming

This chapter is a brief overview of the Extreme Programming (XP) methodology as it applies to developing enterprise-level software in Java. The tools described elsewhere in this book will help you realize many of the goals of XP, but you do not have to adopt the entire XP methodology to get value out of this book and chapter. Automated testing, for example, can help you refactor code regardless of whether you are doing pair programming. Continuous integration can help you detect and fix problems early in the lifecycle of the system regardless of whether your customer is on site. However, because this book discusses XP throughout, it is useful to have a short overview of the entire methodology. If you are already familiar with XP, you may want to turn directly to Chapter 2, "J2EE Deployment Concepts."

XP Overview

XP is a lightweight methodology that focuses on coding as the main task. With XP, the code-centric activities are in every stage of the software development lifecycle. Some practitioners of more traditional methodologies (most seem to be CASE tool vendors) have criticized XP, claiming that it involves reckless coding and is not a real process.

On the contrary, XP is an extremely disciplined methodology that centers on constant code review; frequent testing; customer involvement and rapid feedback; incessant refactoring and refining the architecture; continuous integration to discover problems early in the development process; ongoing design and redesign; and constant planning. The following sections of this chapter discuss the core values, principles, and practices of XP.

Four Values of XP

Kent Beck, originator of the XP methodology, defined four key values of XP:

- Communication
- Simplicity
- Feedback
- Courage

Communication is facilitated through pair programming, task estimation, iteration planning, and more. The goal of communication is to provide a place where people can freely discuss the project without fear of reprisal.

Simplicity may seem counterintuitive because it is common in our industry to overly complicate our development projects. The aim of a project is not for the alpha developer to show his technical prowess, but to deliver value to the customer. Don't over-design algorithms. Don't create an artificial intelligence system when all you need is a few if statements. Don't use EJB or XSLT just because doing so will look nice on your résumé. Choose the simplest design, technology, algorithms, and techniques that will satisfy the customer's needs for the current iteration of the project.

Feedback is crucial and is obtained through code testing, customers' stories, small iterations/frequent deliveries, pair programming/constant code reviews, and other methods. For example, if you are unit-level testing all the code you write several times a day, you are constantly getting feedback about the quality and production worthiness of the system. If something is wrong, you will know right away, instead of being unpleasantly surprised the night before product launch.

Courage is all about being brave enough to do what is right. Have you ever worked on a project where people were afraid to throw away code? I have. The code was horrible and convoluted, and it systematically broke every style, rule, and convention. Yet management and quite a few developers were afraid to throw away the code because they weren't sure how discarding it would affect the system. If you follow the tenets of XP, you will not have this problem. Unit regression testing builds an intense sense of courage. When you know the changes you make will not break the system in some unforeseen way, then you have the confidence to refactor and re-architect code. Testing is key to courage.

If, after several iteration of a project, you find a cleaner, less expensive, more performant way of developing a system, you will have the courage to implement it. I have, and it is a blast.

If the Code Smells, Refactor It

In his landmark book *Refactoring: Improving the Design of Existing Code*, Martin Fowler describes code that needs to be refactored as having a certain objectionable smell.

We have to have the courage to throw away code. Coding, like writing, gets better with revisions.

Five Principles of XP

Building on the four values of XP, Beck created five overriding principles for XP development:

- Provide rapid feedback
- Assume simplicity
- Make incremental changes
- Embrace change
- Do quality work

The idea behind rapid feedback is that the faster you get feedback, the more quickly you can respond to it and the more it guides the process of designing, coding, and delivering the system. This feedback does not simply focus on automated testing and continuous integration to spot problems early, but also encompasses short iterations that are given to the customer to ensure that the system they need is ultimately delivered. This constant steering and learning helps keep the cost of change low and enables the developer to assume simplicity.

Assuming simplicity means treating every problem as a simple problem until proven otherwise. This approach works well, because most problems are easy to solve. However, it is counterintuitive to common thought on reuse and software design. Assuming simplicity does not mean skipping the design step, nor does it mean simply slinging code at a problem. Assuming simplicity requires that you design *only* for the current iteration. Because you are getting rapid feedback from the customer as requirements change, you will not waste time redoing code that was designed to be thermonuclear-resistant when all that was needed was a little waterproofing. The fact is, the customer's requirements always change during the development process, so why not embrace the alterations? My feeling has always been that the true test of every design is when the rubber meets the road—that is, when you are coding.

Incremental change fits nicely with simplicity. Don't over-design a system. It is always better to do a little at a time. Let's say you have decided to redo a Web application that currently uses XSLT (a technology for transforming XML data) so that it instead uses JavaServer Pages (JSP; a technology that is commonly used to create dynamically generate Web pages), to improve performance of the system.

Instead of attacking the whole system, why not just incrementally change the pages that get the most traffic first, thus getting the biggest bang for the buck while minimizing risk? As more and more of the system is done, you may decide to leave some parts using XSLT; or, after partially finishing the JSP version, you may decide to do something else. By testing and deploying on a small scale, you can make decisions based on live feedback that you otherwise could not make. This is the rubber hitting the road. (By the way, I am making no design judgments on XSL or JSP. This is just an example for illustration.)

It has been said that a picture is worth a thousand words. Well, a working model deployed to production is worth a thousand pictures. This is the synergy among rapid feedback, incremental change, and keeping it simple. XP goes further than incrementally changing the system. XP relishes change; it embraces change.

Have you ever heard a developer or a project manager declare that once the requirements were set, the customer should not change them? I have. This requirement seemed logical, but wait—isn't the system being built for the customer?

Conversely, XP developers relish and embrace change. Change is expected, because you are delivering business value incrementally to the customer. The customer has plenty of time to give rapid feedback and request needed changes. This process improves the quality of the system by ensuring that the system being built is the one the customer really needs. Customer are happy because they can steer the next revision before the project gets too far off track from their current business needs.

One of the things that drove me to XP was the principle of quality work. I feel better about myself when I deliver quality work. Early in my career, my team was required to certify that every line of code was tested: 100 percent "go code" and 85 percent exception-handling code. At first I was shocked.

I soon realized that certifying the testing of every line of code caused us to write some extremely clean code. No one had the same three lines of code in three different places, because if they did, they would have to certify nine lines of code instead of three. This discipline led to less code to maintain and made it easier to make changes because we made the change in only one place.

From that point on, I loved testing. I liked knowing that the code I wrote worked and that I had a test to prove it. This attitude is extremely important, because we are constantly bombarded with new technology and things to learn. As I said, quality work drove me to XP. Previously I wrote my own tests using JPython, a scripting language for the JVM, which I still use for prototyping. Then, I heard about JUnit, and from JUnit I learned about XP.

Of course, the quality work principle applies to more than making you happy. You would much rather write quality code and deliver a quality solution to the customer than write sloppy code and deliver a shoddy solution that does not meet the customer’s need. Customers would much rather receive quality code than something that just does not work. It has been said that customers will sometimes forget that software was delivered late, but they will never forget poor quality.

When I was initially learning about XP, it seemed to be about 90 percent common sense, 8 percent "Aha!", and 2 percent revolution. So far, we have been covering the common sense and "Aha!" The next section covers these as well as the revolution.

Twelve Practices of XP

In his landmark book on XP, Beck iterated four basic practices: coding, testing, listening, and designing. These practices are expressed further in 12 major areas of practice, as follows:

- Planning game
- Small releases
- Simple design
- Testing
- Continuous integration
- Refactoring
- Pair programming
- Collective ownership

- 40-hour week
- On-site customer
- Metaphor
- Coding standard

Planning Game

The purpose of the planning game is to determine the scope of the current iteration. This step is centered on determining the tasks that are most important to the customer and accomplishing these tasks first. The planning game encompasses the customer's determining the scope of the project, priority of features, composition of the various releases, and delivery dates. The developers assist the customer with technical feedback by estimating task duration, considering consequences and risks, organizing the team, and performing technical risk management by working on the riskiest parts of the project first. The developers and the customers act as a team.

Time is recorded against stories to further refine your estimates of future stories, making project estimation more accurate as time goes on. Customer stories are recorded on index cards. These stories explain the features of the system. The developers work with the customer to decided which stories will be implemented for that iteration.

Small Releases

The philosophy behind the small releases practice is to provide the most business value with the least amount of coding effort. The features have to be somewhat atomic. A feature must implement enough functionality for it to have business value. This step may be counterintuitive, but the idea is to get the project into production as soon as possible. Small releases get feedback from the customer and reduce risk by making sure the software is what the customer wants. In essence, this step uses the Paredo rule: 80 percent of the business value can be completed with 20 percent of the effort. Small releases go hand in hand with the planning game to decide what features will give the biggest bang for the buck, and they also work with the practice of keeping designs simple.

Simple Design

The idea behind simple design is keep the code simple. The simplest design possible does not try to solve future problems by guessing future needs. The simplest design passes the customer's acceptance test for that release of the software.

The simplest design passes all the tests, has no duplicate logic code, and is not convoluted but expresses every developer's purpose. This step goes hand in hand with small releases. If your architecture is not expressed well and is built to anticipate future needs, you will not be able to deliver it as quickly. We are developers, not fortunetellers. We don't have a crystal ball, so the best way to anticipate customers' future needs is to give them a working system and get feedback from them. Most customers don't know exactly what they need until you deliver something tangible that they can react to. Remember, a picture is worth a thousand words, and a working model is worth a thousand pictures.

Testing

The practice of testing is key to XP. How will you know if a feature works if you do not test? How will you know if a feature still works after you refactor, unless you retest? If you admit that you don't know everything, and that the code will change and evolve, then you'll realize the need for a way to test the code when you change it.

The tests should be automated so you can have the confidence and courage to change the code and refactor it without breaking the system! This approach is the opposite of waterfall development.

Code is in a liquid state, so it can be re-architected, refactored, or thrown out and completely redone. Later, the tests can show that the system still works. Testing keeps the code fluid. Because tests typically check the public interface of classes and components, the implementation of the system can change drastically while the automated tests validate that the system still fulfills the contract of the interfaces. A feature does not exist unless a test validates that it functions. Everything that can potentially break must have a test. JUnit and friends will help you automate your testing.

Continuous Integration

Continuous integration is a crucial concept. Why wait until the end of a project to see if all the pieces of the system will work? Every few hours (at least once every day) the system should be fully built and tested, including all the latest changes. By doing this often, you will know what changes broke the system, and you can make adjustments accordingly instead of waiting until modifications pile up and you forget the details of the changes.

In order to facilitate continuous integration, you need to automate the build, distribution, and deploy processes. Doing so can be quite complicated in a J2EE environment. Ant can help you integrate with source control, Java compilation, creating deployment files, and automating testing. It can even send emails to the group letting them know what files broke the build or what tests did not pass.

Using Ant to perform continuous integration changes the whole development blueprint of your project. With a little discipline and setup time, continuous integration reduces problems linked with team development—particularly time spent fixing integration bugs. Integration bugs are the most difficult to uncover because often they lie unexposed until the system is integrated and two subsystems intermingle for the first time. Suddenly the system breaks in peculiar, unpredictable ways. The longer integration bugs survive, the harder they are to exterminate. Using Ant and JUnit to implement continuous integration often makes such a bug apparent within hours after it has been introduced into the system. Furthermore, the code responsible for this bug is fresh in the mind of the developer; thus, it is easier to eradicate. As Fowler and Foemmel state, continuous integration can "slash the amount of time spent in integration hell" (see their article at www.martinfowler.com/articles/continuousIntegration.html).

Ant can be used to automate the following tasks:

❑ Obtaining the source from configuration management system (CVS, Perforce, VSS, or StarTeam, and so on)

❑ Compiling the Java code

❑ Creating binary deployment files (JAR, WARs, ZIP)

❑ Automating testing (when used in conjunction with tools like JUnit)

Developers often talk about automated building and testing but seldom implement them. Ant makes automated building and testing possible and plausible. Ant and JUnit combine well to allow teams to build and test their software several times a day. Such an automated process is worth the investment of time and effort. Automated builds and tests need the following, as stated by Fowler and Foemmel:

- ❑ A single place for all the source code where developers can get the most current sources—typically, a configuration management system
- ❑ A single command to build the system from all the sources (in our case, an Ant buildfile)
- ❑ A single command to run an entire suite of tests for the system (in our case, an Ant buildfile that integrates with JUnit)
- ❑ A good set of binaries that ensures developers have the latest working components (such as JAR files that contain classes)

Refactoring

The act of refactoring enables developers to add features while keeping the code simple, thus keeping the code simple while still being able to run all the tests. The idea is to not duplicate code nor write ugly, smelly code. The act of refactoring centers on testing to validate that the code still functions. Testing and refactoring go hand in hand. Automated unit-level tests will give you the courage to refactor and keep the code simple and expressive.

Refactoring is not limited to when you need to add a feature—refactoring is different than adding a feature. However, the catalyst for refactoring is often the need to add features while keeping the code simple. XP says not to guess what future features your customer wants. You cannot design in a vacuum. As you receive feedback, you will need to add features that will cause you to bend the code in new directions. Many software project management books say that once a project matures, adding two new features will cause one existing feature to break. The books make this statement as if such an occurrence is normal and acceptable. However, this is not the nature of software development; this is the nature of using a methodology and development environment in which adding new features and refactoring are not coupled with testing. Testing makes refactoring possible.

You will know when you need to refactor, because you will hear a little voice in the back of your head nagging you: Don't take a shortcut, make it simple, make it expressive. If the code stinks, you will fix it. If you don't hear that little voice or you aren't listening, then your pair-programming partner is there to guide you in the right direction.

Pair Programming

Pair programming is probably the most revolutionary practice of XP—and it is usually the one managers have the hardest time swallowing. On the surface, their reaction is easy to understand: If our projects are behind, then how will having two programmers work on the same task help speed things up? Why have two developers with one keyboard and one monitor?

If you think it is expensive to have two developers work together, how expensive will it be when you have to replace the whole system because it is impossible to maintain and every change causes massive ripple effects? You have undoubtedly seen it happen, and it is not a pretty picture.

I know from experience that pair programming works. For one thing, it improves communication among team members. A large part of what we do depends on the work of other developers. The developer you team with one day is not necessarily the developer you will team with the next day. In fact, the developer you team with in the morning may not be the developer you will team with in the afternoon. This process breeds better communication and cross-pollination of ideas. How often does your team reinvent a piece of code that another developer worked on?

Also, if one person knows much about a particular technical subject matter (such as EJB or Oracle) or business domain (such as accounting, semiconductor equipment, or aerospace), what better way for other developers to learn than to pair-program with that person?

What about quality? Pair programming provides constant feedback and ensures that the person who is coding is refactoring, testing, and following the coding standard. As Solomon stated, "By iron, iron itself is sharpened. So one man sharpens the face of another."

And, while one developer is focusing on the coding, the other developer can be thinking about the bigger picture: how this code will fit in with the other system. Typically, the developer who is not coding at the time is taking notes so that nothing falls through the cracks.

A few careless programmers can ruin a project team. Just like one rotten apple can spoil the whole bunch, sloppy code breeds more sloppy code. Pretty soon the team is addicted to the quick fix. In addition, because more and more of the code reeks, no one wants to own it. Pair programming is in lockstep with the practice of collective ownership.

Collective Ownership

The XP practice of collective ownership states that anyone can make a change to the system. You don't have to wait. No one owns a class. Everyone contributes, and if you need to make a change or refactor something to get it to work with a new feature, you can. Besides, you have the tests to make sure your changes did not break the system, and the old version is still in source control if anyone needs to refer to it, which is rare.

In addition, because many classes have corresponding automated test code, anyone can see how the code and API of classes were suppose to work. So, collective programming goes hand in hand with automated testing. Testing is part of what makes collective ownership possible.

Some developers will know some sections better than other developers, but all take ownership and responsibility for a system. If no one knows a particular part, the unit tests exercise the API and check that you did not break the system with your changes. Thus, you do not have to wait for another team member (let's call him Jon) to fix something. If Jon is busy helping Doug, you will fix the system with Nick, your pair-programming partner. If Jon is sick, the knowledge of a particular subsystem is not lost. Because Jon was pair programming with Andy, Andy knows about the system—and, in fact, so does Paul.

In addition, if you did not practice collective ownership and you had some critical parts of the system, then everyone would need to wait until you were done. Or, you would have to work a very long week to accomplish the task so you wouldn't hold up Paul, Doug, Andy, and Nick. Your family would like you to spend more time with them, and you want to work a 40-hour week.

40-Hour Week

The 40-hour week practice states that if you cannot do your work in a 40-hour week, then something is wrong with the project. Let's face it, burned-out developers make lots of mistakes. No one is saying that you should never put in a 90-, 80-, 70-, or 60-hour week, but such a schedule should not be the norm. An unstated reality to this practice is to make your time count by working hard and getting the most out of your time. Long meetings should be rare, as should unnecessary red tape. Any activity that gets in the way of providing the most business value into the next release of your current projects should be avoided like the plague. Make the most of your time, and 40-hour weeks will be enough most of the time. In addition, keeping normal office hours will give you more opportunity to talk about features with your 9-to-5 customers.

On-Site Customer

The on-site customer practice states that if at all possible, the customer should be available when the developers need to ask questions. And, if at all possible, the customer should be physically located with the developers.

The customer must be available to clarify feature stories. The customer is also involved in the planning game, and this process is easier if the customer is not remote. The developers work with the customer to decided which stories are implemented for that iteration. The customer also helps write and refine the stories—which typically involves ad hoc and planned discussions with the developers. The stories, which are recorded on index cards, are written in a simple dialect that both the developers and customer understand, using a common metaphor.

Metaphor

A *metaphor* is a common language and set of terms used to envision the functionality of a project. These terms can equate to objects in the physical world, such as accounts or objects. Other times, metaphors can be more abstract, like windows or cells. The idea is for the customer and developers to have the same vision of the system and be able to speak about the system in a common dialect.

Coding Standard

XPers should follow the practice of using a coding standard. You must be able to understand one another's code. Luckily for us Java developers, we have the coding standard set by Sun; you should be able to make some light adjustments and make this coding standard your coding standard.

Your coding standard should state the proper use of threads, language constructs, exception use, no duplicate code logic, and so on. The coding standard should be more than just a guide on where to put your curly brace; it should denote style and common practices and conventions that your team will follow. Java developers have tons of reference material for coding style and standards. Like many things, developers will follow the standard if they voluntarily create it and buy into it.

Adopting XP?

As we've stated before, you do not have to adopt XP to get value out of this book. For example, automated testing can help you refactor code regardless of whether you are doing pair programming.

Here's another example: If you use Unified Modeling Language (UML) use cases with a case tool instead of stories written on index cards, continuous integration and small release cycles will still be beneficial for getting rapid feedback. The point is, you may decide to do things in addition to the process. Or, your corporate culture may have an adverse reaction to things like pair programming.

UML and CASE tools

Some XP advocates swear by never using CASE tools. They say that the only UML should be written with a pencil on an index card. I don't agree. As long as the CASE tool can continually keep the model in sync with the code, then the tool can be very beneficial. In addition, some CASE tools can speed development by generating the necessary boilerplate code for design patterns.

Beck notes that whether you draw diagrams that generate code or write out code, it is still code.

One of the first areas to focus on when adopting XP is automated testing. Begin by writing tests for code that you are about to add functionality to, refactor, or fix. The key is to add automated tests slowly to code written before you adopted XP, and always employ automated testing with newly developed code. Do not write tests for code you are not working on. Later, when you begin doing integration tests, you will want to automate the build, test, and integration cycle.

My company has adopted XP. We adhere to the 12 XP practices. However, I am not a purist. I believe that other software processes can benefit from automated testing, simple designs, continuous integration, incremental releases, and constant refactoring.

Beck states that XP's 12 practices will not fit every project. XP also will not fit every organization's culture. Regardless, J2EE development can benefit from automated testing, simple designs, continuous integration, incremental releases, and constant refactoring. This book focuses on tools to perform automated testing and continuous integration for J2EE to enable refactoring and incremental releases.

Summary

XP is a lightweight methodology that focuses on coding as the main task. XP is based on four values: communication, simplicity, feedback, and courage. Communication is facilitated through pair programming, task estimation, iteration planning, and more. Simplicity means avoiding making things overly complicated and insisting that the basics be addressed first and foremost. Feedback is given by way of testing, customer stories, small iterations/frequent deliveries, pair programming/constant code reviews, and so on. Courage means the courage to do what is right whether you have to refactor a working system, throw code away, cancel a project, or insist on quality.

XP is based on five principles: rapid feedback, assuming simplicity, making incremental changes, embracing change, and doing quality work. In his landmark book on XP, Beck iterated four basic practices: coding, testing, listening, and designing. These practices are expressed further in 12 major areas of practice: the planning game, small releases, simple design, (automated) testing, continuous integration, refactoring, pair programming, collective ownership, a 40-hour week, an on-site customer, metaphor, and adherence to a coding standard. This book focus on two practices of XP: automated testing and continuous integration.