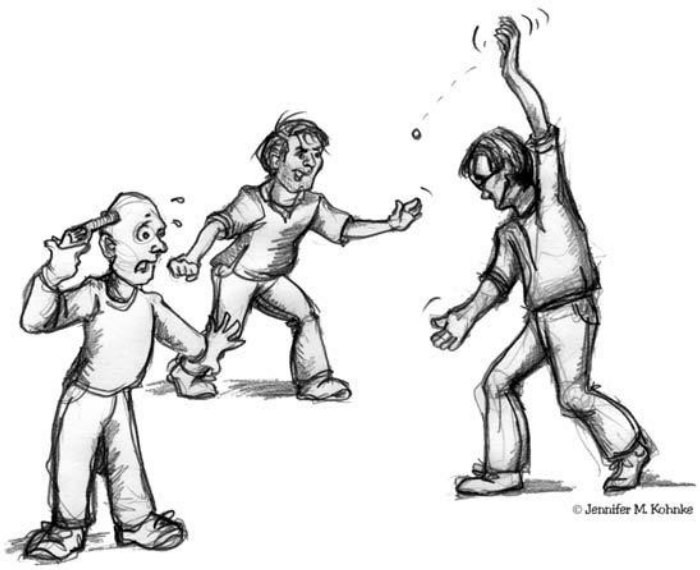
2

Overview of Extreme Programming



As developers we need to remember that XP is not the only game in town.

—Pete McBreen

The previous chapter gave us an outline of what agile software development is about. However, it didn't tell us exactly what to do. It gave us some platitudes and goals, but it gave us little in the way of real direction. This chapter corrects that.

The Practices of Extreme Programming

Extreme programming is the most famous of the agile methods. It is made up of a set of simple, yet interdependent practices. These practices work together to form a whole that is greater than its parts. We shall briefly consider that whole in this chapter, and examine some of the parts in chapters to come.

Customer Team Member

We want the customer and developers to work closely with each other so that they are both aware of each other's problems and are working together to solve those problems.

Who is the customer? The customer of an XP team is the person or group who defines and prioritizes features. Sometimes, the customer is a group of business analysts or marketing specialists working in the same company as the developers. Sometimes, the customer is a user representative commissioned by the body of users. Sometimes the customer is in fact the paying customer. But in an XP project, whoever the customers are, they are members of, and available to, the team.

The best case is for the customer to work in the same room as the developers. Next best is if the customer works in within 100 feet of the developers. The larger the distance, the harder it is for the customer to be a true team member. If the customer is in another building or another state, it is very difficult to integrate him or her into the team.

What do you do if the customer simply cannot be close by? My advice is to find someone who *can* be close by and who is willing and able to stand in for the true customer.

User Stories

In order to plan a project, we must know something about the requirements, but we don't need to know very much. For planning purposes, we only need to know enough about a requirement to estimate it. You may think that in order to estimate a requirement you need to know all its details, but that's not quite true. You have to know that there *are* details, and you have to know roughly the kinds of details there are, but you don't have to know the specifics.

The specific details of a requirement are likely to change with time, especially once the customer begins to see the system come together. There is nothing that focuses requirements better than seeing the nascent system come to life. Therefore, capturing the specific details about a requirement long before it is implemented is likely to result in wasted effort and premature focusing.

When using XP, we get the sense of the details of the requirements by talking them over with the customer, but we do not capture that detail. Rather, the customer writes *a few words* on an index card that we agree will remind us of the conversation. The developers write an estimate on the card at roughly the same time that the customer writes it. They base that estimate on the sense of detail they got during their conversations with the customer.

A user story is a mnemonic token of an ongoing conversation about a requirement. It is a planning tool that the customer uses to schedule the implementation of a requirement based upon its priority and estimated cost.

Short Cycles

An XP project delivers working software every two weeks. Each of these two-week iterations produces working software that addresses some of the needs of the stakeholders. At the end of each iteration, the system is demonstrated to the stakeholders in order to get their feedback.

The Iteration Plan. An iteration is usually two weeks in length. It represents a minor delivery that may or may not be put into production. It is a collection of user stories selected by the customer according to a budget established by the developers.

The developers set the budget for an iteration by measuring how much they got done in the previous iteration. The customer may select any number of stories for the iteration, so long as the total of their estimates does not exceed that budget.

Once an iteration has been started, the customer agrees not to change the definition or priority of the stories in that iteration. During this time, the developers are free to cut the stories up in to *tasks* and to develop the tasks in the order that makes the most technical and business sense.

The Release Plan. XP teams often create a release plan that maps out the next six iterations or so. That plan is known as a release plan. A release is usually three months worth of work. It represents a major delivery that can usually be put into production. A release plan consists of prioritized collections of user stories that have been selected by the customer according to a budget given by the developers.

The developers set the budget for the release by measuring how much they got done in the previous release. The customer may select any number of stories for the release so long as the total of the estimates does not exceed that budget. The customer also determines the order in which the stories will be implemented in the release. If the team so desires, they can map out the first few iterations of the release by showing which stories will be completed in which iterations.

Releases are not cast in stone. The customer can change the content at any time. He or she can cancel stories, write new stories, or change the priority of a story.

Acceptance Tests

The details about the user stories are captured in the form of acceptance tests specified by the customer. The acceptance tests for a story are written immediately preceding, or even concurrent with, the implementation of that story. They are written in some kind of scripting language that allows them to be run automatically and repeatedly. Together, they act to verify that the system is behaving as the customers have specified.

The language of the acceptance tests grows and evolves with the system. The customers may recruit the developers to create a simple scripting system, or they may have a separate quality assurance (QA) department that can develop it. Many customers enlist the help of QA in developing the acceptance-testing tool and with writing the acceptance tests themselves.

Once an acceptance test passes, it is added to the body of passing acceptance tests and is never allowed to fail again. This growing body of acceptance tests is run several times per day, every time the system is built. If an acceptance tests fails, the build is declared a failure. Thus, once a requirement is implemented, it is never broken. The system migrates from one working state to another and is never allowed to be inoperative for longer than a few hours.

Pair Programming

All *production* code is written by pairs of programmers working together at the same workstation. One member of each pair drives the keyboard and types the code. The other member of the pair watches the code being typed, looking for errors and improvements.[1] The two interact intensely. Both are completely engaged in the act of writing software.

The roles change frequently. The driver may get tired or stuck, and his pair partner will grab the keyboard and start to drive. The keyboard will move back and forth between them several times in an hour. The resultant code is designed and authored by both members. Neither can take more than half the credit.

Pair membership changes at least once per day so that every programmer works in two different pairs each day. Over the course of an iteration, every member of the team should have worked with every other member of the team, and they should have worked on just about everything that was going on in the iteration.

This dramatically increases the spread of knowledge through the team. While specialties remain and tasks that require certain specialties will usually belong to the appropriate specialists, those specialists will pair with nearly everyone else on the team. This will spread the specialty out through the team such that other team members can fill in for the specialists in a pinch.

Studies by Laurie Williams[2] and Nosek[3] have suggested that pairing does not reduce the efficiency of the programming staff, yet it significantly reduces the defect rate.

1. I have seen pairs in which one member controls the keyboard and the other controls the mouse.
2. [Williams2000], [Cockburn2001].
3. [Nosek].

Test-Driven Development

Chapter 4, which is on testing, discusses test-driven development in great detail. The following paragraphs provide a quick overview.

All production code is written in order to make failing unit tests pass. First we write a unit test that fails because the functionality for which it is testing doesn't exist. Then we write the code that makes that test pass.

This iteration between writing test cases and code is very rapid, on the order of a minute or so. The test cases and code evolve together, with the test cases leading the code by a very small fraction. (See "A Programming Episode" in Chapter 6 for an example.)

As a result, a very complete body of test cases grows along with the code. These tests allow the programmers to check whether the program works. If a pair makes a small change, they can run the tests to ensure that they haven't broken anything. This greatly facilitates *refactoring* (discussed later).

When you write code in order to make test cases pass, that code is, by definition, testable. In addition, there is a strong motivation to decouple modules from each other so that they can be independently tested. Thus, the design of code that is written in this fashion tends to be much less coupled. The principles of object-oriented design play a powerful role in helping you with this decoupling.[4]

Collective Ownership

A pair has the right to check out *any* module and improve it. No programmers are individually responsible for any one particular module or technology. Everybody works on the GUI.[5] Everybody works on the middleware. Everybody works on the database. Nobody has more authority over a module or a technology than anybody else.

This doesn't mean that XP denies specialties. If your specialty is the GUI, you are most likely to work on GUI tasks, but you will also be asked to pair on middleware and database tasks. If you decide to learn a second specialty, you can sign up for tasks and work with specialists who will teach it to you. You are not confined to your specialty.

Continuous Integration

The programmers check in their code and integrate several times per day. The rule is simple. The first one to check in wins, everybody else merges.

XP teams use nonblocking source control. This means that programmers are allowed to check any module out at any time, regardless of who else may have it checked out. When the programmer checks the module back in after modifying it, he must be prepared to merge it with any changes made by anyone who checked the module in ahead of him. To avoid long merge sessions, the members of the team check in their modules very frequently.

A *pair* will work for an hour or two on a task. They create test cases and production code. At some convenient breaking point, probably long before the task is complete, the pair decides to check the code back in. They first make sure that all the tests run. They integrate their new code into the existing code base. If there is a merge to do, they do it. If necessary, they consult with the programmers who beat them to the check in. Once their changes are integrated, they build the new system. They run every test in the system, including all currently running acceptance tests. If they broke anything that used to work, they fix it. Once all the tests run, they finish the check in.

Thus, XP teams will build the system many times each day. They build the *whole* system from end to end.[6] If the final result of a system is a CD, they cut the CD. If the final result of the system is an active Web site, they install that Web site, probably on a testing server.

4. See Section II.
5. I'm not advocating a three-tiered architecture here. I just chose three common partitions of software technology.
6. Ron Jeffries says, "End to end is farther than you think."

Sustainable Pace

A software project is not a sprint; it is a marathon. A team that leaps off the starting line and starts racing as fast as it can will burn out long before they are close to finishing. In order to finish quickly, the team must run at a sustainable pace; it must conserve its energy and alertness. It must intentionally run at a steady, moderate pace.

The XP rule is that a team is not *allowed* to work overtime. The only exception to that rule is the last week in a release. If the team is within striking distance of its release goal and can sprint to the finish, then overtime is permissible.

Open Workspace

The team works together in an open room. There are tables set up with workstations on them. Each table has two or three such workstations. There are two chairs in front of each workstation for pairs to sit in. The walls are covered with status charts, task breakdowns, UML diagrams, etc.

The sound in this room is a low buzz of conversation. Each pair is within earshot of every other. Each has the opportunity to hear when another is in trouble. Each knows the state of the other. The programmers are in a position to communicate intensely.

One might think that this would be a distracting environment. It would be easy to fear that you'd never get anything done because of the constant noise and distraction. In fact, this doesn't turn out to be the case. Moreover, instead of interfering with productivity, a University of Michigan study suggested that working in a "war room" environment may *increase* productivity by a factor of two.[7]

The Planning Game

The next chapter, "Planning," goes into great detail about the XP planning game. I'll describe it briefly here.

The essence of the planning game is the division of responsibility between business and development. The business people (a.k.a. the customers) decide how important a feature is, and the developers decide how much that feature will cost to implement.

At the beginning of each release and each iteration, the developers give the customers a budget, based on how much they were able to get done in the last iteration or in the last release. The customers choose stories whose costs total up to, but do not exceed that budget.

With these simple rules in place, and with short iterations and frequent releases, it won't be long before the customers and developers get used to the rhythm of the project. The customers will get a sense for how fast the developers are going. Based on that sense, the customers will be able to determine how long their project will take and how much it will cost.

Simple Design

An XP team makes their designs as simple and expressive as they can be. Furthermore, they narrow their focus to consider only the stories that are planned for the current iteration. They don't worry about stories to come. Instead, they migrate the design of the system, from iteration to iteration, to be the best design for the stories that the system currently implements.

This means that an XP team will probably not start with infrastructure. They probably won't select the database first. They probably won't select the middleware first. The team's first act will be to get the first batch of stories working in the *simplest way possible*. The team will only add the infrastructure when a story comes along that forces them to do so.

7. http://www.sciencedaily.com/releases/2000/12/001206144705.htm

The following three XP mantras guide the developer:

Consider the Simplest Thing That Could Possibly Work. XP teams always try to find the simplest possible design option for the current batch of stories. If we can make the current stories work with flat files, we might not use a database or EJB. If we can make the current stories work with a simple socket connection, we might not use an ORB or RMI. If we can make the current stories work without multithreading, we might not include mutithreading. We try to consider the simplest way to implement the current stories. Then we choose a solution that is as close to that simplicity as we can *practically* get.

You Aren't Going to Need It. Yeah, but we *know* we're going to need that database one day. We *know* we're going to need an ORB one day. We *know* we're going to have to support multiple users one day. So we need to put the hooks in for those things *now*, don't we?

An XP team seriously considers what will happen if they resist the temptation to add infrastructure before it is strictly needed. They start from the assumption that they aren't going to need that infrastructure. The team puts in the infrastructure, only if they have proof, or at least very compelling evidence, that putting in the infrastructure now will be more cost effective than waiting.

Once and Only Once. XPers don't tolerate code duplication. Wherever they find it, they eliminate it.

There are many sources of code duplication. The most obvious are those stretches of code that were captured with a mouse and plopped down in multiple places. When we find those, we eliminate them by creating a function or a base class. Sometimes two or more algorithms may be remarkably similar, and yet they differ in subtle ways. We turn those into functions or employ the TEMPLATE METHOD pattern.[8] Whatever the source of duplication, once discovered, we won't tolerate it.

The best way to eliminate redundancy is to create abstractions. After all, if two things are similar, there must be some abstraction that unifies them. Thus, the act of eliminating redundancy forces the team to create many abstractions and further reduce coupling.

Refactoring[9]

I cover this topic in more detail in Chapter 5. What follows is a brief overview.

Code tends to rot. As we add feature after feature and deal with bug after bug, the structure of the code degrades. Left unchecked, this degradation leads to a tangled, unmaintainable mess.

XP teams reverse this degradation through frequent refactoring. Refactoring is the practice of making a series of tiny transformations that improve the structure of the system without affecting its behavior. Each transformation is trivial, hardly worth doing. But together, they combine into significant transformations of the design and architecture of the system.

After each tiny transformation, we run the unit tests to make sure we haven't broken anything. Then we do the next transformation and the next and the next, running the tests after each. In this manner we keep the system working while transforming its design.

Refactoring is done continuously rather than at the end of the project, the end of the release, the end of the iteration, or even the end of the day. Refactoring is something we do every hour or every half hour. Through refactoring, we continuously keep the code as clean, simple, and expressive as possible.

Metaphor

Metaphor is the least understood of all the practices of XP. XPers are pragmatists at heart, and this lack of concrete definition makes us uncomfortable. Indeed, the proponents of XP have often discussed removing metaphor as a practice. And yet, in some sense, metaphor is one of the most important practices of all.

8. See Chapter 14, "Template Method & Strategy: Inheritance v. Delegation."
9. [Fowler99].

Think of a jigsaw puzzle. How do you know how the pieces go together? Clearly, each piece abuts others, and its shape must be perfectly complimentary to the pieces it touches. If you were blind and you had a very good sense of touch, you could put the puzzle together by diligently sifting through each piece and trying it in position after position.

But there is something more powerful than the shape of the pieces binding the puzzle together. There is a picture. The picture is the true guide. The picture is so powerful that if two adjacent pieces of the picture do not have complementary shapes, then you *know* that the puzzle maker made a mistake.

That is the metaphor. It's the big picture that ties the whole system together. It's the vision of the system that makes the location and shape of all the individual modules obvious. If a module's shape is inconsistent with the metaphor, then you know it is the module that is wrong.

Often a metaphor boils down to a system of names. The names provide a vocabulary for elements in the system and help to define their relationships.

For example, I once worked on a system that transmitted text to a screen at 60 characters per second. At that rate, a screen fill could take some time. So we'd allow the program that was generating the text to fill a buffer. When the buffer was full, we'd swap the program out to disk. When the buffer got close to empty, we'd swap the program back in and let it run more.

We spoke about this system in terms of dump trucks hauling garbage. The buffers were little trucks. The display screen was the dump. The program was the garbage producer. The names all fit together and helped us think about the system as a whole.

As another example, I once worked on a system that analyzed network traffic. Every thirty minutes, it would poll dozens of network adapters and pull down the monitoring data from them. Each network adapter gave us a small block of data composed of several individual variables. We called these blocks "slices." The slices were raw data that needed to be analyzed. The analysis program "cooked" the slices, so it was called "The Toaster." We called the individual variables within the slices, "crumbs." All in all, it was a useful and entertaining metaphor.

Conclusion

Extreme programming is a set of simple and concrete practices that combines into an agile development process. That process has been used on many teams with good results.

XP is a good general-purpose method for developing software. Many project teams will be able to adopt it as is. Many others will be able to adapt it by adding or modifying practices.

Bibliography

1. Dahl, Dijkstra. *Structured Programming*. New York: Hoare, Academic Press, 1972.
2. Conner, Daryl R. *Leading at the Edge of Chaos*. Wiley, 1998.
3. Cockburn, Alistair. *The Methodology Space*. Humans and Technology technical report HaT TR.97.03 (dated 97.10.03), `http://members.aol.com/acockburn/papers/methyspace/methyspace.htm`.
4. Beck, Kent. *Extreme Programming Explained: Embracing Change*. Reading, MA: Addison–Wesley, 1999.
5. Newkirk, James, and Robert C. Martin. *Extreme Programming in Practice*. Upper Saddle River, NJ: Addison–Wesley, 2001.
6. Williams, Laurie, Robert R. Kessler, Ward Cunningham, Ron Jeffries. *Strengthening the Case for Pair Programming*. IEEE Software, July–Aug. 2000.
7. Cockburn, Alistair, and Laurie Williams. *The Costs and Benefits of Pair Programming*. XP2000 Conference in Sardinia, reproduced in *Extreme Programming Examined*, Giancarlo Succi, Michele Marchesi. Addison–Wesley, 2001.
8. Nosek, J. T. *The Case for Collaborative Programming*. Communications of the ACM (1998): 105–108.
9. Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison–Wesley, 1999.