Chapter 2. Learning to Drive

I can remember clearly the day I first began learning to drive. My mother and I were driving up Interstate 5 near Chico, California; a straight, flat stretch of road where the highway stretches right to the horizon. My mom had me reach over from the passenger seat and hold the steering wheel. She let me get the feel of how the motion of the wheel affected the direction of the car. Then she told me, "Here's how you drive. Line the car up in the middle of the lane, straight toward the horizon."

I very carefully squinted straight down the road. I got the car smack dab in the middle of the lane, pointed right down the middle of the road. I was doing great. My mind wandered a little. . .

I jerked back to attention as the car hit the gravel. My mom (her courage now amazes me) gently got the car back straight on the road. My heart was pounding. Then she actually taught me about driving. "Driving is not about getting the car going in the right direction. Driving is about constantly paying attention, making a little correction this way, a little correction that way."

This is the paradigm for XP. Stay aware. Adapt. Change.

Everything in software changes. The requirements change. The design changes. The business changes. The technology changes. The team changes. The team members change. The problem isn't change, because change is going to happen; the problem, rather, is our inability to cope with change.

There are two levels at which the driving metaphor applies to XP. Customers drive the content of the system. The whole team drives the development process. XP lets you adapt by making frequent, small corrections; moving towards your goal with deployed software at short intervals. You don't wait a long time to find out if you were going the wrong way.

The customers drive the content of the system. Customers (internal or external) start with a general idea of what problems the system needs to solve. However, customers don't usually know exactly what the software should do. That's why software development is like driving, not like getting the car pointed straight down the road. The customers on the team need to keep in mind where on the horizon they want to go even as they decide, week-by-week, where the software should go *next*.

What each team does to express their values will be different from place to place and time to time and team to team. Just as the customers steer the content of the system, the whole team steers the development process, beginning with its current set of practices. As development continues, the team becomes aware of which of their practices enhance and which of their practices detract from their goals. Each practice is an experiment in improving effectiveness, communication, confidence, and productivity.

Chapter 3. Values, Principles, and Practices

What does it take to clearly communicate a new way of thinking about and doing software development? You can learn the basic techniques of gardening quickly from a book, but that doesn't make you a gardener. My friend Paul is a master gardener. I dig and plant and water and weed, but I am not a master gardener.

What are the differences between us? First, Paul knows more techniques than I do, and he's better at the techniques we both know. Technique matters because if you don't dig and plant things, you certainly aren't gardening. Call this level of knowledge and understanding *practices*, things you actually do. Practices are the things you do day-to-day. Specifying practices is useful because they are clear and objective. You either write a test before you change code or you don't. The practices are also useful because they give you a place to start. You can start writing tests before changing code, and gain benefit from doing so, long before you understand software development in a deeper way.

Even if I knew all the same gardening practices as Paul, I still wouldn't be a gardener. Paul has a highly developed sense of what is good and bad about gardening. He can look at a whole garden and get a gut sense of what's working and what isn't. Where I might be proud of my ability to correctly prune a branch, Paul might see that the whole tree should come out. He sees this not because he is a better pruner than I am, but because he has an overall sense of the forces at work in the garden. I have to work at what is simple and obvious to him.

Call this level of knowledge and understanding *values*. Values are the roots of the things we like and don't like in a situation. When a programmer says, "I don't want to estimate my tasks," he generally isn't talking about technique. He already estimates, but doesn't want to reveal what he really thinks for fear of providing a fixed point of judgement that will be used against him later. Better triple that estimate! Refusing to communicate estimates reveals something much deeper about how he sees the social forces in development. Perhaps he doesn't want to be accountable because he has been blamed unfairly in the past. In this case, the programmer values protection over communication. Values are the large-scale criteria we use to judge what we see, think, and do.

Making values explicit is important because without values, practices quickly become rote, activities performed for their own sake but lacking any purpose or direction. When I hear a programmer brush off a defect, I hear a failure of values, not technique. The defect itself might be a failure of technique, but the reluctance to learn from the defect shows that the programmer doesn't actually value learning and self-improvement as much as something else. This is not in the best interest of the program, the organization, or the programmer. Bringing values together with practices means that the programmer can perform a practice, in this case root-cause analysis, at effective times and for good reasons. Values bring purpose to practices.

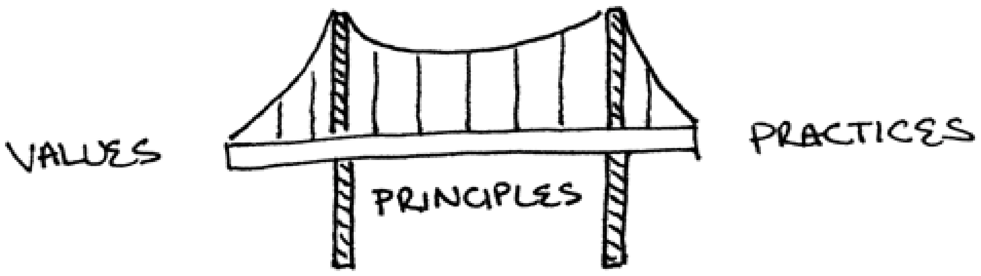
Practices are evidence of values. Values are expressed at such a high level that I could do just about anything in the name of a value. "I wrote this one-thousand-page document because *I value communication*." Maybe yes and maybe no. If a fifteen minute conversation once a day would have communicated more effectively than producing the document, then the document doesn't show that I value communication. Communicating in the most effective way I can shows I value communication.

Practices are clear. Everyone knows if I've attended the morning standup meetings. Whether I really value communication is fuzzy. Whether I maintain practices that enhance communication is concrete. Just as values bring purpose to practices, practices bring accountability to values.

Values and practices are an ocean apart. Values are universal. Ideally, my values as I work are exactly the same as my values in the rest of my life. Practices, however, are intensely situated. If I want feedback about whether I'm doing a good job programming, continuously building and testing my software makes sense. If I want feedback when I'm changing a diaper, "continuously building and testing" is absurd. The forces involved in the two activities are just too different. To get feedback about my diapering job, I have to pick the baby up when I'm done to see if the diaper falls off. I can't test halfway through. The value "feedback" is expressed in very different forms in the two activities of diapering and programming.

Bridging the gap between values and practices are *principles* (see Figure 1). Principles are domain-specific guidelines for life. Paul's knowledge as a gardener exceeds mine at the level of principles as well. I might know to plant marigolds next to strawberries, but Paul understands the principle of companion planting where adjacent plants make up for each others' weaknesses. Marigolds naturally repel some of the bugs that eat strawberries. Planting them together is a practice. Companion planting is the principle. In this book I present the values, principles, and practices of XP.

Figure 1.



This is the limit of what I can communicate in a book. It is a start but it isn't enough for you to master XP. No book of gardening, however complete, makes you a gardener. First you have to garden, then join the community of gardeners, then teach others to garden. Then you are a gardener.

So it is with XP. Reading this book won't make you an extreme programmer. That only comes with programming in the extreme style, participating in the community of people who share these values and at least some of your practices, and then sharing what you know with others.

You will benefit from studying and trying parts of XP. Learning to write tests before code is useful regardless of your values or the rest of your practices. However, there is as much difference between that and programming extreme as there is between my work in the garden and master gardening.

Chapter 4. Values

Paul, the master gardener, has an intuitive sense of what needs to be done next. He knows in his bones what matters and what doesn't. I might think perfectly straight rows are really important. I put a lot of effort into making my rows straight. Along comes Paul and says, "Why are you working so hard at making the rows straight? What you need is more compost." The difference between what I think is valuable and what is really valuable creates waste.

Everyone who touches software development has a sense of what matters. One person might think what really matters is carefully thinking through all conceivable design decisions before implementing. Another might think what really matters is not having any restrictions on his own personal freedom.

As Will Rogers said, "It ain't what you don't know that gets you in trouble. It's what you know that ain't so." The biggest problem I encounter in what people "just know" about software development is that they are focused on individual action. What actually matters is not how any given person behaves as much as how the individuals behave as part of a team and as part of an organization.

For example, people get passionate about coding style. While there are undoubtedly better styles and worse styles, the most important style issue is that the team chooses to work towards a common style. Idiosyncratic coding styles and the values revealed by them, individual freedom at all costs, don't help the team succeed.

If everyone on the team chooses to focus on what's important to the team, what is it they should focus on? XP

embraces five values to guide development:
communication, simplicity, feedback, courage, and respect.

Communication

What matters most in team software development is communication. When problems arise in development, most often someone already knows the solution; but that knowledge doesn't get through to someone with the power to make the change. This occurs internally when I ignore my intuition, but the effects are compounded when communicating between people.

Sometimes problems are caused by a lack of knowledge rather than a lack of communication. There's nothing you can do about these problems beforehand, since you didn't know. "Ctrl-shift-S is already assigned in Polish Windows. Who would have thought?" Once you find a surprising problem, communication can help you solve it. You can listen to people who have had similar problems in the past. You can talk as a team about how to make sure the problem doesn't recur.

Perhaps this sounds like a perpetual coffee klatch with everyone sitting around "caring and sharing" and no one doing anything. Other values held by the team keep this from happening. However, motion without communication is not progress.

When you encounter a problem, ask yourselves if the problem was caused by a lack of communication. What communication do you need now to address the problem? What communication do you need to keep yourself out of this trouble in the future?

Communication is important for creating a sense of team and effective cooperation. Communication, though, is not all you need for effective software development.

Simplicity

Simplicity is the most intensely intellectual of the XP values. To make a system simple enough to gracefully solve only today's problem is hard work. Yesterday's simple solution may be fine today, or it may look simplistic or complex. When you need to change to regain simplicity, you must find a way from where you are to where you want to be.

I ask people to think about the question, "What is the simplest thing that could possibly work?" Critics seem to miss the second half of the question. "Well, we have serious security and reliability constraints so we couldn't possibly make our system simple." I'm not asking you to think about what is too simple to work, just to bias your thinking toward eliminating wasted complexity. If security concerns dictate that you split your system across two processors where otherwise you could have used one, as far as I am concerned, that result is simple. The only better solution is if you could find a way to address the security concerns on a single processor.

Simplicity only makes sense in context. If I'm writing a parser with a team that understands parser generators, then using a parser generator is simple. If the team doesn't know anything about parsing and the language is simple, a recursive descent parser is simpler.

The values are intended to balance and support each other. Improving communication helps achieve simplicity by eliminating unneeded or deferrable requirements from today's concerns. Achieving simplicity gives you that much less to communicate about.

Feedback

No fixed direction remains valid for long; whether we are talking about the details of software development, the requirements of the system, or the architecture of the system. Directions set in advance of experience have an especially short half-life. Change is inevitable, but change creates the need for feedback.

I remember an all-day presentation I gave in Aarhus, Denmark. One front-row attendee's face got cloudier and cloudier as the day progressed. Finally he couldn't stand it. "Wouldn't it be easier just to do it right in the first place?" Of course it would, except for three things:

- We may not know how to do it "right". If we are solving a novel problem there may be several solutions that might work or there may be no clear solution at all.
- What's right for today may be wrong for tomorrow. Changes outside our control or ability to predict can easily invalidate yesterday's decisions.
- Doing everything "right" today might take so long that changing circumstances tomorrow invalidate today's solution before it is even finished.

Being satisfied with improvement rather than expecting instant perfection, we use feedback to get closer and closer to our goals. Feedback comes in many forms:

- Opinions about an idea, yours or your teammates'

- How the code looks when you implement the idea
- Whether the tests were easy to write
- Whether the tests run
- How the idea works once it has been deployed

XP teams strive to generate as much feedback as they can handle as quickly as possible. They try to shorten the feedback cycle to minutes or hours instead of weeks or months. The sooner you know, the sooner you can adapt.

It is possible to get too much feedback. If the team is ignoring important feedback; it needs to slow down, frustrating as that may be, until it can respond to the feedback. Then the team can address the underlying issues that caused the excess of feedback. For example, suppose you move to quarterly releases and suddenly have more defect reports than you can respond to before the next quarter's release. Slow down releases until you can handle the defect reports and still develop new functionality. Take the time to figure out why you are creating so many defects or why each defect takes so long to address. Once you've solved the basic problem; you can start releasing quarterly again, cranking up the feedback machine.

Feedback is a critical part of communication. "Is performance going to be a problem?" "I don't know. Let's write a little performance prototype and see." Feedback also contributes to simplicity. Which of three solutions will turn out to be simplest? Try all three and see. While implementing the same thing three times may seem wasteful, it may be the most efficient way to arrive at a solution whose simplicity you can live with. At the same time, the simpler the system, the easier it is to get feedback about it.

Courage

Courage is effective action in the face of fear. Some people have objected to using the word "courage", reserving it for what a patrolling soldier does when going through a darkened doorway. Without intending to diminish the kind of physical courage demonstrated by the soldier, it is certainly true that people involved in software development feel fear. It's how they handle their fear that dictates whether they are working as an effective part of a team.

Sometimes courage manifests as a bias to action. If you know what the problem is, do something about it. Sometimes courage manifests as patience. If you know there is a problem but you don't know what it is, it takes courage to wait for the real problem to emerge distinctly.

Courage as a primary value without counterbalancing values is dangerous. Doing something without regard for the consequences is not effective teamwork. Encourage teamwork by looking to the other values for guidance on what to do when afraid.

If courage alone is dangerous, in concert with the other values it is powerful. The courage to speak truths, pleasant or unpleasant, fosters communication and trust. The courage to discard failing solutions and seek new ones encourages simplicity. The courage to seek real, concrete answers creates feedback.

Respect

The previous four values point to one that lies below the surface of the other four: respect. If members of a team don't care about each other and what they are doing, XP won't work. If members of a team don't care about a project, nothing can save it.

Every person whose life is touched by software development has equal value as a human being. No one is intrinsically worth more than anyone else. For software development to simultaneously improve in humanity and productivity, the contributions of each person on the team need to be respected. I am important and so are you.