Part II: The Solution

> Now we have set the stage. We know what problem we have to solve, namely deciding how the basic activities of software development should take place—coding, testing, listening, and designing. We have a set of guiding values and principles to guide us as we choose strategies for each of these activities. And we have the flattened cost curve as our ace in the hole to simplify the strategies we choose.

Chapter 10. A Quick Overview

> *We will rely on the synergies between simple practices, practices that often were abandoned decades ago as impractical or naïve.*

The raw materials of our new software development discipline are

- The story about learning to drive
- The four values—communication, simplicity, feedback, and courage
- The principles
- The four basic activities—coding, testing, listening, and designing

Our job is to structure the four activities. Not only do we have to structure the activities, we have to do it in light of the long list of sometimes contradictory principles. And at the same time we have to try to improve the economic performance of software development enough so that someone will listen.

No problem.

Er...

As the purpose of this book is to explain how this could possibly work, I will quickly explain the major areas of practice in XP. In the next chapter, we will see how such ridiculously simple solutions could possibly work. Where a practice is weak, the strengths of other practices will cover for the weakness. Later chapters will cover some of the topics in more detail.

First, here are all the practices:

- *The Planning Game*—Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan.
- *Small releases*—Put a simple system into production quickly, then release new versions on a very short cycle.
- *Metaphor*—Guide all development with a simple shared story of how the whole system works.
- *Simple design*—The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
- *Testing*—Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished.

- *Refactoring*—Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.
- *Pair programming*—All production code is written with two programmers at one machine.
- *Collective ownership*—Anyone can change any code anywhere in the system at any time.
- *Continuous integration*—Integrate and build the system many times a day, every time a task is completed.
- *40 hour week*—Work no more than 40 hours a week as a rule. Never work overtime a second week in a row.
- *On-site customer*—Include a real, live user on the team, available full-time to answer questions.
- *Coding standards*—Programmers write all code in accordance with rules emphasizing communication through the code.

In this chapter we will quickly summarize what is involved in executing each practice. In the next chapter (How Could This Work?) we will examine the connections between the practices that allow the weaknesses of one practice to be overcome by the strengths of other practices.

The Planning Game

Neither business considerations nor technical considerations should be paramount. Software development is always an evolving dialog between the possible and the desirable. The nature of the dialog is that it changes both what is seen to be possible and what is seen to be desirable.

Business people need to decide about

- Scope—How much of a problem must be solved for the system to be valuable in production? The business person is in a position to understand how much is not enough and how much is too much.
- Priority—If you could only have A or B at first, which one do you want? The business person is in a position to determine this, much more so than a programmer.
- Composition of releases—How much or how little needs to be done before the business is better off with the software than without it? The programmer's intuition about this question can be wildly wrong.
- Dates of releases—What are important dates at which the presence of the software (or some of the software) would make a big difference?

Business can't make these decisions in a vacuum. Development needs to make the technical decisions that provide the raw material for the business decisions.

Technical people decide about

- Estimates—How long will a feature take to implement?
- Consequences—There are strategic business decisions that should be made only when informed about the technical consequences. Choice of a database is a good example. Business might rather work with a huge company than a

startup, but a factor of 2 in productivity may make the extra risk or discomfort worth it. Or not. Development needs to explain the consequences.

- Process—How will the work and the team be organized? The team needs to fit the culture in which it will operate, but you should write software well rather than preserve the irrationality of an enclosing culture.
- Detailed scheduling—Within a release, which stories will be done first? The programmers need the freedom to schedule the riskiest segments of development first, to reduce the overall risk of the project. Within that constraint, they still tend to move business priorities earlier in the process, reducing the chance that important stories will have to be dropped toward the end of the development of a release.

Small Releases

Every release should be as small as possible, containing the most valuable business requirements. The release has to make sense as a whole—that is, you can't implement half a feature and ship it, just to make the release cycle shorter.

It is far better to plan a month or two at a time than six months or a year at a time. A company shipping bulky software to customers might not be able to release this often. They should still reduce their cycle as much as possible.

Metaphor

Each XP software project is guided by a single overarching metaphor. Sometimes the metaphor is "naive," like a contract management system that is spoken of in terms of contracts and customers and endorsements. Sometimes the metaphor needs a little explanation, like saying the computer should appear as a desktop, or that pension calculation is like a spreadsheet. These are all metaphors, though, because we don't literally mean "the system is a spreadsheet." The metaphor just helps everyone on the project understand the basic elements and their relationships.

The words used to identify technical entities should be consistently taken from the chosen metaphor. As development proceeds and the metaphor matures, the whole team will find new inspiration from examining the metaphor.

The metaphor in XP replaces much of what other people call "architecture." The problem with calling the 10,000-meter view of the system an architecture is that architectures don't necessarily push the system into any sense of cohesion. An architecture is the big boxes and connections.

You could say, "Of course architecture badly done is bad." We need to emphasize the goal of architecture, which is to give everyone a coherent story within which to work, a story that can easily be shared by the business and technical folks. By asking for a metaphor we are likely to get an architecture that is easy to communicate and elaborate.

Simple Design

The right design for the software at any given time is the one that

1. Runs all the tests.
2. Has no duplicated logic. Be wary of hidden duplication like parallel class hierarchies.
3. States every intention important to the programmers.
4. Has the fewest possible classes and methods.

Every piece of design in the system must be able to justify its existence on these terms. Edware Tufte [1] has an exercise for graphic designers—design a graph however you want. Then, erase as long as you don't remove any information. Whatever is left when you can't erase any more is the right design for the graph. Simple design is like this—take out any design element that you can without violating rules 1, 2, and 3.

[1] Edward Tufte, *The Visual Display of Quantitative Information, Graphics Press,* 1992

This is opposite advice from what you generally hear: "Implement for today, design for tomorrow." If you believe that the future is uncertain, and you believe that you can cheaply change your mind, then putting in functionality on speculation is crazy. Put in what you need when you need it.

Testing

Any program feature without an automated test simply doesn't exist. Programmers write unit tests so that their confidence in the operation of the program can become part of the program itself. Customers write functional tests so that their confidence in the operation of the program can become part of the program, too. The result is a program that becomes more and more confident over time—it becomes more capable of accepting change, not less.

You don't have to write a test for every single method you write, only production methods that could possibly break. Sometimes you just want to find out if something is possible. You go explore for half an hour. Yes, it is possible. Now you throw away your code and start over with tests.

Refactoring

When implementing a program feature, the programmers always ask if there is a way of changing the existing program to make adding the feature simple. After they have added a feature, the programmers ask if they now can see how to make the program simpler, while still running all of the tests. This is called refactoring.

Note that this means that sometimes you do more work than absolutely necessary to get a feature running. But in working this way, you ensure that you can add the next feature with a reasonable amount of effort, and the next, and the next. You don't refactor on speculation, though; you refactor when the system asks you to. When the system requires that you duplicate code, it is asking for refactoring.

If a programmer sees a one-minute ugly way to get a test working and a ten-minute way to get it working with a simpler design, the correct choice is to spend the ten minutes. Fortunately, you can make even radical changes to the design of a system in small, low-risk steps:

Pair Programming

All production code is written with two people looking at one machine, with one keyboard and one mouse.

There are two roles in each pair. One partner, the one with the keyboard and the mouse, is thinking about the best way to implement this method right here. The other partner is thinking more strategically:

- Is this whole approach going to work?
- What are some other test cases that might not work yet?
- Is there some way to simplify the whole system so the current problem just disappears?

Pairing is dynamic. If two people pair in the morning, in the afternoon they might easily be paired with other folks. If you have responsibility for a task in an area that is unfamiliar to you, you might ask someone with recent experience to pair with you. More often, anyone on the team will do as a partner.

Collective Ownership

Anybody who sees an opportunity to add value to any portion of the code is required to do so at any time.

Contrast this to two other models of code ownership—no ownership and individual ownership. In the olden days, nobody owned any particular piece of code. If someone wanted to change some code, they did it to suit their own purpose, whether it fit well with what was already there or not. The result was chaos, especially with objects where the relationship between a line of code over here and a line of code over there was not easy to determine statically. The code grew quickly, but it also quickly grew unstable.

To get control of this situation, individual code ownership arose. The only person who could change a piece of code was its official owner. Anyone else who saw that the code needed changing had to submit their request to the owner. The result of strict ownership is that the code diverges from the team's understanding, as people are reluctant to interrupt the code owner. After all, they need the change now, not later. So the code remains stable, but it doesn't evolve as quickly as it should. Then the owner leaves ….

In XP, everybody takes responsibility for the whole of the system. Not everyone knows every part equally well, although everyone knows something about every part. If a pair is working and they see an opportunity to improve the code, they go ahead and improve it if it makes their life easier.

Continuous Integration

Code is integrated and tested after a few hours—a day of development at most. One simple way to do this is to have a machine dedicated to integration. When the machine is free, a pair with code to integrate sits down, loads the current release,

loads their changes (checking for and resolving any collisions), and runs the tests until they pass (100% correct).

Integrating one set of changes at a time works well because it is obvious who should fix a test that fails—we should, since we must have broken it, since the last pair left the tests at 100%. And if we can't get the tests to run at 100%, we should throw away what we did and start over, since we obviously didn't know enough to be programming that feature (although we likely do know enough now).

40-Hour Week

I want to be fresh and eager every morning, and tired and satisfied every night. On Friday, I want to be tired and satisfied enough that I feel good about two days to think about something other than work. Then on Monday I want to come in full of fire and ideas.

Whether this translates into precisely 40 hours per week at the work site is not terribly important. Different people have different tolerances for work. One person might be able to put in 35 concentrated hours, another 45. But no one can put in 60 hours a week for many weeks and still be fresh and creative and careful and confident. Don't do that.

Overtime is a symptom of a serious problem on the project. The XP rule is simple—you can't work a second week of overtime. For one week, fine, crank and put in some extra hours. If you come in Monday and say, "To meet our goals, we'll have to work late again," then you already have a problem that can't be solved by working more hours.

A related issue is vacation. Europeans often take vacations of two, three, or four straight weeks. Americans seldom take more than a few days at a time. If it were my company, I would insist that people take a two-week vacation every year, with at least another week or two available for shorter breaks.

On-Site Customer

A real customer must sit with the team, available to answer questions, resolve disputes, and set small-scale priorities. By "real customer" I mean someone who will really use the system when it is in production. If you are building a customer service system, the customer will be a customer service representative. If you are building a bond trading system, the customer will be a bond trader.

The big objection to this rule is that real users of the system under development are too valuable to give to the team. Managers will have to decide which is more valuable—having the software working sooner and better or having the output of one or two people. If having the system doesn't bring more value to the business than having one more person working, perhaps the system shouldn't be built.

And it's not as if the customer on the team can't get any work done. Even programmers can't generate 40 hours of questions each and every week. The on-site customer will have the disadvantage of being physically separated from other customers, but they will likely have time to do their normal work.

The downside of an on-site customer is if they spend hundreds of hours helping the programmers and then the project is canceled. Then you have lost the work they did, and you have also lost the work they could have done if they hadn't been contributing to a failing project. XP does everything possible to make sure that the project doesn't fail.

I worked on one project where we were grudgingly given a real customer, but "only for a little while." After the system shipped successfully and was obviously able to continue evolving, the managers on the customer side gave us three real customers. The company could have gotten more value out of the system with more business contribution.

Coding Standards

If you are going to have all these programmers changing from this part of the system to that part of the system, swapping partners a couple of times a day, and refactoring each other's code constantly, you simply cannot afford to have different sets of coding practices. With a little practice, it should become impossible to say who on the team wrote what code.

The standard should call for the least amount of work possible, consistent with the Once and Only Once rule (no duplicate code). The standard should emphasize communication. Finally, the standard must be adopted voluntarily by the whole team.