Chapter 11. How Could This Work?

> *The practices support each other. The weakness of one is covered by the strengths of others.*

Wait just a doggone minute. None of the practices described above is unique or original. They have all been used for as long as there have been programs to write. Most of these practices have been abandoned for more complicated, higher overhead practices, as their weaknesses have become apparent. Why isn't XP a simplistic approach to software? Before we go on, we had better convince ourselves that these simple practices won't kill us, just as they killed software projects decades ago.

The collapse of the exponential change cost curve brings all these practices back into play again. Each of the practices still has the same weaknesses as before, but what if those weaknesses were now made up for by the strengths of other practices? We might be able to get away with doing things simply.

This chapter presents another look at the practices, but this time focused on what usually makes the practice untenable, and showing how the other practices keep the bad effects of each from overwhelming the project. This chapter also shows how the whole XP story could possibly work.

The Planning Game

You couldn't possibly start development with only a rough plan. You couldn't constantly update the plan—that would take too long and upset the customers. Unless:

- The customers did the updating of the plan themselves, based on estimates provided by the programmers.
- You had enough of a plan at the beginning to give the customers a rough idea of what was possible over the next couple of years.
- You made short releases so any mistake in the plan would have a few weeks or months of impact at most.
- Your customer was sitting with the team, so they could spot potential changes and opportunities for improvement quickly.

Then perhaps you could start development with a simple plan, and continually refine it as you went along.

Short Releases

You couldn't possibly go into production after a few months. You certainly couldn't make new releases of the system on cycles ranging from daily to every couple of months. Unless:

- The Planning Game helped you work on the most valuable stories, so even a small system had business value.
- You were integrating continuously, so the cost of packaging a release was minimal.

- Your testing reduced the defect rate enough so you didn't have to go through a lengthy test cycle before allowing software to escape.
- You could make a simple design, sufficient for this release, not for all time.

Then perhaps you could make small releases, starting soon after development begins.

Metaphor

You couldn't possibly start development with just a metaphor. There isn't enough detail there, and besides, what if you're wrong? Unless:

- You quickly have concrete feedback from real code and tests about whether the metaphor is working in practice.
- Your customer is comfortable talking about the system in terms of the metaphor.
- You refactor to continually refine your understanding of what the metaphor means in practice.

Then perhaps you could start development with just a metaphor.

Simple Design

You couldn't possibly have just enough design for today's code. You would design yourself into a corner and then you'd be stuck, unable to continue evolving the system. Unless:

- You were used to refactoring, so making changes was not a worry.
- You had a clear overall metaphor so you were sure future design changes would tend to follow a convergent path.
- You were programming with a partner, so you were confident you were making a simple design, not a stupid design.

Then perhaps you could get away with doing the best possible job of making a design for today.

Testing

You couldn't possibly write all those tests. It would take too much time. Programmers won't write tests. Unless:

- The design is as simple as it can be, so writing tests isn't all that difficult.
- You are programming with a partner, so if you can't think of another test your partner can, and if your partner feels like blowing off the tests, you can gently rip the keyboard away.
- You feel good when you see the tests all running.
- Your customer feels good about the system when they see all of their tests running.

Then perhaps programmers and customers will write tests. Besides, if you don't write automated tests, the rest of XP doesn't work nearly as well.

Refactoring

You couldn't possibly refactor the design of the system all the time. It would take too long, it would be too hard to control, and it would most likely break the system. Unless:

- You are used to collective ownership, so you don't mind making changes wherever they are needed.
- You have coding standards, so you don't have to reformat before refactoring.
- You program in pairs, so you are more likely to have the courage to tackle a tough refactoring, and you are less likely to break something.
- You have a simple design, so the refactorings are easier.
- You have the tests, so you are less likely to break something without knowing it.
- You have continuous integration, so if you accidentally break something at a distance, or one of your refactorings conflicts with someone else's work, you know in a matter of hours.
- You are rested, so you have more courage and are less likely to make mistakes.

Then perhaps you could refactor whenever you saw the chance to make the system simpler, or reduce duplication, or communicate more clearly.

Pair Programming

You can't possibly write all the production code in pairs. It will be too slow. What if two people don't get along? Unless:

- The coding standards reduce the picayune squabbles.
- Everyone is fresh and rested, reducing further the chance of unprofitable ... uh ... discussions.
- The pairs write tests together, giving them a chance to align their understanding before tackling the meat of the implementation.
- The pairs have the metaphor to ground their decisions about naming and basic design.
- The pairs are working within a simple design, so they can both understand what is going on.

Then perhaps you could write all production code in pairs. Besides, if people program solo they are more likely to make mistakes, more likely to overdesign, and more likely to blow off the other practices, particularly under pressure.

Collective Ownership

You couldn't possibly have everybody potentially changing anything anywhere. Folks would be breaking stuff left and right, and the cost of integration would go up dramatically. Unless:

- You integrate after a short enough time, so the chances of conflicts go down.
- You write and run the tests, so the chance of breaking things accidentally goes down.
- You pair program, so you are less likely to break code, and programmers learn faster what they can profitably change.
- You adhere to coding standards, so you don't get into the dreaded Curly Brace Wars.

Then perhaps you could have anyone change code anywhere in the system when they see the chance to improve it. Besides, without collective ownership the rate of evolution of the design slows dramatically.

Continuous Integration

You couldn't possibly integrate after only a few hours of work. Integration takes far too long and there are too many conflicts and chances to accidentally break something. Unless:

- You can run the tests quickly so you know you haven't broken anything.
- You program in pairs, so there are half as many streams of changes to integrate.
- You refactor, so there are more smaller pieces, reducing the chance of conflicts.

Then perhaps you could integrate after a few hours. Besides, if you don't integrate quickly then the chance of conflicts rises and the cost of integration goes up steeply.

40-Hour Week

You couldn't possibly work 40-hour weeks. You can't create enough business value in 40 hours. Unless:

- The Planning Game is feeding you more valuable work to do.
- The combination of the Planning Game and testing reduces the frequency of nasty surprises, where you have more to do than you thought.
- The practices as a whole help you program at top speed, so there isn't any faster you can go.

Then perhaps you could produce enough business value in 40-hour weeks. Besides, if the team doesn't stay fresh and energetic, then they won't be able to execute the rest of the practices.

On-Site Customer

You couldn't possibly have a real customer on the team, sitting there full-time. They can produce far more value for the business elsewhere. Unless:

- They can produce value for the project by writing functional tests.
- They can produce value for the project by making small-scale priority and scope decisions for the programmers.

Then perhaps they can produce more value for the company by contributing to the project. Besides, if the team doesn't include a customer, they will have to add risk to the project by planning further in advance and coding without knowing exactly what tests they have to satisfy and what tests they can ignore.

Coding Standards

You couldn't possibly ask the team to code to a common standard. Programmers are deeply individualistic, and would quit rather than put their curly braces somewhere else. Unless:

- The whole of XP makes them more likely to be members of a winning team.

Then perhaps they would be willing to bend their style a little. Besides, without coding standards the additional friction slows pair programming and refactoring significantly.

Conclusion

Any one practice doesn't stand well on its own (with the possible exception of testing). They require the other practices to keep them in balance. Figure 4 is a diagram that summarizes the practices. A line between two practices means that the two practices reinforce each other. I didn't want to present this picture first, because it makes XP look complicated. The individual pieces are simple. The richness comes from the interactions of the parts.

Figure 4. The practices support each other