**UDDRAG FRA Kent Beck – Extreme Programming Explained**

# Extreme Programming Explained

Second Edition

*Embrace Change*

Kent Beck
with Cynthia Andres

Addison-Wesley
Boston

Publisher: John Wait
Editor in Chief: Don O'Hagan
Acquisitions Editor: Paul Petralia
Managing Editor: John Fuller
Project Editors: Julie Nahil and Kim Arney Mulcahy
Compositor: Kim Arney Mulcahy
Manufacturing Buyer: Carol Melville



*Library of Congress Cataloging-in-Publication Data*

Beck, Kent.
extreme programming explained: embrace change / Kent Beck with Cynthia Andres. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-27865-8 (alk. paper)
1. Computer software—Development. 2. eXtreme programming. I. Title.
QA76.76.D47B434 2004
005.1—dc22

2004057463



ISBN 0-321-27865-8
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
Tenth printing, January 2012

# Chapter 1. Risk: The Basic Problem

> *Software development fails to deliver, and fails to deliver value. This failure has huge economic and human impact. We need to find a new way to develop software.*

The basic problem of software development is risk. Here are some examples of risk:

- Schedule slips—the day for delivery comes, and you have to tell the customer that the software won't be ready for another six months.
- Project canceled—after numerous slips, the project is canceled without ever going into production.
- System goes sour—the software is successfully put into production, but after a couple of years the cost of making changes or the defect rate rises so much that the system must be replaced.
- Defect rate—the software is put into production, but the defect rate is so high that it isn't used.
- Business misunderstood—the software is put into production, but it doesn't solve the business problem that was originally posed.
- Business changes—the software is put into production, but the business problem it was designed to solve was replaced six months ago by another, more pressing, business problem.
- False feature rich—the software has a host of potentially interesting features, all of which were fun to program, but none of which makes the customer much money.
- Staff turnover—after two years, all the good programmers on the project begin to hate the program and leave.

In these pages you will read about Extreme Programming (XP), a software development discipline that addresses risk at all levels of the development process. XP is also very productive, produces high-quality software, and is a lot of fun to execute.

How does XP address the risks listed above?

- Schedule slips—XP calls for short release cycles, a few months at most, so the scope of any slip is limited. Within a release, XP uses one- to four-week iterations of customer-requested features for fine-grained feedback about progress. Within an iteration, XP plans with one- to three-day tasks, so the team can solve problems even during an iteration.

Finally, XP calls for implementing the highest priority features first, so any features that slip past the release will be of lower value.

- Project canceled—XP asks the customer to choose the smallest release that makes the most business sense, so there is less to go wrong before going into production and the value of the software is greatest.

- System goes sour—XP creates and maintains a comprehensive suite of tests, which are

- run and re-run after every change (several times a day), to ensure a quality baseline. XP always keeps the system in prime condition. Cruft is not allowed to accumulate.

- Defect rate—XP tests from the perspective of both programmers writing tests function-by-function and customers writing tests program-feature-by-program-feature.

- Business misunderstood—XP calls for the customer to be an integral part of the team. The specification of the project is continuously refined during development, so learning by the customer and the team can be reflected in the software.

- Business changes—XP shortens the release cycle, so there is less change during the development of a single release. During a release, the customer is welcome to substitute new functionality for functionality not yet completed. The team doesn't even notice if it is working on newly discovered functionality or features defined years ago.

- False feature rich—XP insists that only the highest priority tasks are addressed.

- Staff turnover—XP asks programmers to accept responsibility for estimating and completing their own work, gives them feedback about the actual time taken so their estimates can improve, and respects those estimates. The rules for who can make and change estimates are clear. Thus, there is less chance for a programmer to get frustrated by being asked to do the obviously impossible. XP also encourages human contact among the team, reducing the loneliness that is often at the heart of job dissatisfaction. Finally, XP incorporates an explicit model of staff turnover. New team members are encouraged to gradually accept more and more responsibility, and are assisted along the way by each other and by existing programmers.

## Our Mission

If we accept project risk as the problem to be solved, where are we going to look for the solution? What we need to do is invent a style of software development that addresses these risks. We need to communicate this discipline as clearly as possible to programmers, managers, and customers. We need to set out guidelines for adapting it to local conditions (that is, communicate what is fixed and what is variable).

That's what we cover in sections one and two of this book. We will go step by step through the aspects of the problem of creating a new style or discipline of development, and then we will solve the problem. From a set of basic assumptions, we will derive solutions that dictate how the various

activities in software development—planning, testing, development, design, and deployment—should occur.

# Chapter 4. Four Variables

> *We will control four variables in our projects—cost, time, quality, and scope. Of these, scope provides us the most valuable form of control.*

Here is a model of software development from the perspective of a system of control variables. In this model, there are four variables in software development:

- Cost
- Time
- Quality
- Scope

The way the software development game is played in this model is that external forces (customers, managers) get to pick the values of any three of the variables. The development team gets to pick the resultant value of the fourth variable.

Some managers and customers believe they can pick the value of all four variables. "You are *going* to get all these requirements done by the first of next month with exactly this team. And quality is job one here, so it will be up to our usual standards." When this happens, quality always goes out the window (this is generally up to the usual standards, though), since nobody does good work under too much stress. Also likely to go out of control is time. You get crappy software late.

The solution is to make the four variables visible. If everyone—programmers, customers, and managers—can see all four variables, they can consciously choose which variables to control. If they don't like the result implied for the fourth variable, they can change the inputs, or they can pick a different three variables to control.

## Interactions Between the Variables

*Cost*—More money can grease the skids a little, but too much money too soon creates more problems than it solves. On the other hand, give a project too little money and it won't be able to solve the customer's business problem.

*Time*—More time to deliver can improve quality and increase scope. Since feedback from systems in production is vastly higher quality than any other kind of feedback, giving a project too much time will hurt it. Give a project too little time and quality suffers, with scope, time, and cost not far behind.

*Quality*—Quality is terrible as a control variable. You can make very short-term gains (days or

weeks) by deliberately sacrificing quality, but the cost—human, business, and technical—is enormous.

*Scope*—Less scope makes it possible to deliver better quality (as long as the customer's business problem is still solved). It also lets you deliver sooner or cheaper.

There is not a simple relationship between the four variables. For example, you can't just get software faster by spending more money. As the saying goes, "Nine women cannot make a baby in one month." (And contrary to what I've heard from some managers, eighteen women still can't make a baby in one month.)

In many ways, cost is the most constrained variable. You can't just spend your way to quality, or scope, or short release cycles. In fact, at the beginning of a project, you can't spend much at all. The investment has to start small and grow over time. After a while, you can productively spend more and more money.

I had one client who said, "We have promised to deliver all of this functionality. To do that, we have to have 40 programmers."

I said, "You can't have 40 programmers on the first day. You have to start with one team. Then grow to two. Then four. In two years you can have 40 programmers, but not today."

They said, "You don't understand. We have to have 40 programmers." I said, "You can't have 40 programmers." They said, "We have to."

They didn't. I mean, they did. They hired the 40 programmers. Things didn't go well. The programmers left; they hired 40 more. Four years later they are just beginning to deliver value to the business, one small subproject at a time, and they nearly got canceled first.

All the constraints on cost can drive managers crazy. Especially if they are focused on an annual budgeting process, they are so used to driving everything from cost that they will make big mistakes ignoring the constraints on how much control cost gives you.

The other problem with cost is that higher costs often feed tangential goals, like status or prestige. "Of course, I have a project with 150 people (sniff, sniff)." This can lead to projects that fail because the manager wanted to look impressive. After all, how much status is there in staffing the same project with 10 programmers and delivering in half the time?

On the other hand, cost is deeply related to the other variables. Within the range of investment that can sensibly be made, by spending more money you can increase the scope, or you can move more deliberately and increase quality, or you can (to some extent) reduce time to market.

Spending money can also reduce friction—faster machines, more technical specialists, better offices.

The constraints on controlling projects by controlling time generally come from outside—the year

2000 being the most recent example. The end of the year; before the quarter starts; when the old system is scheduled to be shut off; a big trade show—these are some examples of external time constraints. So, the time variable is often out of the hands of the project manager and in the hands of the customer.

Quality is another strange variable. Often, by insisting on better quality you can get projects done sooner, or you can get more done in a given amount of time. This happened to me when I started writing unit tests (as described in Chapter 2, A Development Episode, page 7). As soon as I had my tests, I had so much more confidence in my code that I wrote faster, without stress. I could clean up my system more easily, which made further development easier. I've also seen this happen with teams. As soon as they start testing, or as soon as they agree on coding standards, they start going faster.

There is a strange relationship between internal and external quality. External quality is quality as measured by the customer. Internal quality is quality as measured by the programmers. Temporarily sacrificing internal quality to reduce time to market in hopes that external quality won't suffer too much is a tempting short-term play. And you can often get away with making a mess for a matter of weeks or months. Eventually, though, internal quality problems will catch up with you and make your software prohibitively expensive to maintain, or unable to reach a competitive level of external quality.

On the other hand, from time to time you can get done sooner by relaxing quality constraints. Once, I was working on a system to plug replace a legacy COBOL system. Our quality constraint was that we precisely reproduce the answers produced by the old system. As we got closer and closer to our release date, it became apparent that we could reproduce all the errors in the old system, but only by shipping much later. We went to the customers, showed them that our answers were more correct, and offered them the option of shipping on time if they wanted to believe our answers instead.

There is a human effect from quality. Everybody wants to do a good job, and they work much better if they feel they are doing good work. If you deliberately downgrade quality, your team might go faster at first, but soon the demoralization of producing crap will overwhelm any gains you temporarily made from not testing, or not reviewing, or not sticking to standards.

## Focus on Scope

Lots of people know about cost, quality, and time as control variables, but don't acknowledge the fourth. For software development, scope is the most important variable to be aware of. Neither the programmers nor the business people have more than a vague idea about what is valuable about the software under development. One of the most powerful decisions in project management is eliminating scope. If you actively manage scope, you can provide managers and customers with control over cost, quality, and time.

One of the great things about scope is that it is a variable that varies a lot. For decades, programmers have been whining, "The customers can't tell us what they want. When we give them what they say they want, they don't like it." This is an absolute truth of software development. The

requirements are never clear at first. Customers can never tell you exactly what they want.

The development of a piece of software changes its own requirements. As soon as the customers see the first release, they learn what they want in the second release...or what they really wanted in the first. And it's valuable learning, because it couldn't have possibly taken place based on speculation. It is learning that can only come from experience. But customers can't get there alone. They need people who can program, not as guides, but as companions.

What if we see the "softness" of requirements as an opportunity, not a problem? Then we can choose to see scope as the easiest of the four variables to control. Because it is so soft, we can shape it—a little this way, a little that way. If time gets tight toward a release date, there is always something that can be deferred to the next release. By not trying to do too much, we preserve our ability to produce the required quality on time.

If we created a discipline of development based on this model, we would fix the date, quality, and cost of a piece of software. We would look at the scope implied by the first three variables. Then, as development progressed, we would continually adjust the scope to match conditions as we found them.

This would have to be a process that tolerated change easily, because the project would change direction often. You wouldn't want to spend a lot on software that turned out not to be used. You wouldn't want to build a road you never drove on because you took another turn. Also, you would have to have a process that kept the cost of changes reasonable for the life of the system.

If you dropped important functionality at the end of every release cycle, the customer would soon get upset. To avoid this, XP uses two strategies:

1. You get lots of practice making estimates and feeding back the actual results. Better estimates reduce the probability that you will have to drop functionality.

2. You implement the customer's most important requirements first, so if further functionality has to be dropped it is less important than the functionality that is already running in the system.

# 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.

# 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.

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**