Embrace Change:
Extreme Programming Explained

Kent Beck, Daedalos Consulting

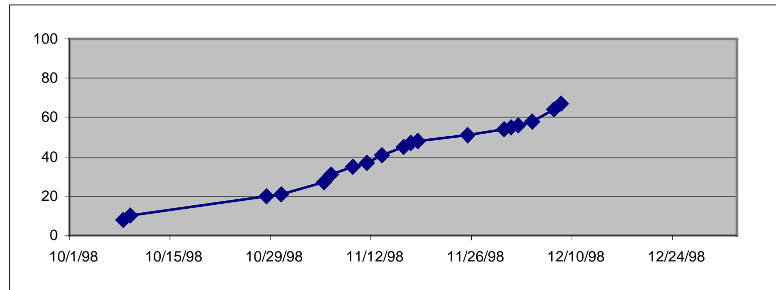


Table of Contents

Table of Contents 1
To Do 2
Preface 2
What is XP? 2
Acknowledgements 4
The Problem 4
The Basic Problem 4
Economics of Software Development 5
Four Variables 7
Cost of Change 9
The XP Paradigm 10
Four Values 11
Basic Principles 14
Back to Basics 16
The Solution 19
A Quick Overview 19
How Could This Work? 23
Splitting Business and Technical Responsibility 26
Planning Strategy 28
Facilities Strategy 33
Development Strategy 34
Design Strategy 39
Testing Strategy 44
Management Strategy 46
Implementing XP 48
My Story of XP 48
Lifecycle of an Ideal XP Project 50
Roles for People 53
20-80 Rule 58
Adopting XP 59
Retrofitting XP 60
What Makes XP Hard 62
When You Shouldn't Try XP 64
XP Business Opportunities 65
Insourcing 65
Completing the Circle 67
Annotated Bibliography 68

Copyright © 1998 Kent Beck, All rights

Glossary 69

To Do

Chapter hooks

Finish converting sections 1 and 2 to second person. Does second person work? Or does it sound whiny?

The other book talks about precisely how to do each thing- examples, exceptions, war stories. Be sure to take out how-to material. Does this focus work, or is it frustrating to only get part of the story?

Are there places to introduce learning strategies, or does that come in the how-to book?

Edit Planning Strategy to remove excess how-to material

Make sure planning strategy talks about customer on-site

Should I repeat the process of going through the principles one by one for each of the strategies?

Glossary

Bibliography

Preface

This is a book about Extreme Programming. It talks about the thinking behind Extreme Programming- its roots, philosophy, stories, myths. It is intended to help you make an informed decision about whether or not to use Extreme Programming on your project. Let me repeat that. If you read this book and correctly decide *not* to use Extreme Programming for your project, I will have met my goal just as much as if you correctly decide *to* use it. If you are using Extreme Programming, the description here of how and why XP works will also help you adjust it to match your local culture.

What this book isn't is a book about precisely how to do Extreme Programming. So, you won't read lots of checklists here, or see many examples, or lots of war stories. For that, you will have to go online, talk to some of the coaches mentioned here, wait for the topical, how-to books to follow, or just make up your own version.

A book of philosophy is little bit anti-Extreme Programming. XP (as I will call it in the rest of the book) values abstracting only from experience, not from speculation. Here is a book primarily of abstractions- principles, reasons, explanations- not real stuff. What gives?

Acceptance of XP is now in the hands of a group of people (you may be one) who are dissatisfied with software development as it is currently practiced. You want a better way to develop software, you want better relationships with your customers, you want happier, more stable, more productive developers. In short, you are looking for big rewards, and you aren't afraid to try new ideas to get them. But if you are going to take a risk, you want to be convinced that you aren't just being stupid.

XP tells you to do things differently than most other advice about developing object software. Sometimes XP's advice is absolutely contrary to accepted wisdom. I expect those choosing to use XP at this point to require damned good reasons for doing things differently, but if the reasons are there, to go right ahead. I wrote this book to give you those reasons.

What is XP?

What is XP? The short answer is that it is a methodology. It is distinguished from other methodologies by

- its reliance on automated tests written by developers and customers to monitor the progress of development and allow the system to evolve
- its reliance on oral communication and source code instead of written documentation,
- its reliance on a gradual design process that lasts as long as the system lasts,
- its incremental planning approach that quickly comes up with an overall plan which is expected to evolve through the life of the project,

 Copyright © 1998 Kent Beck, All rights

- its reliance on the close collaboration of programmers with ordinary skills,
- its reliance on practices that work with the short-term instincts of developers

I say that this is the short answer, because I don't like the word „methodology". I prefer to call XP a discipline of software development. It is a discipline because there are certain things that you have to do to be doing XP. You don't get to choose whether or not you will write tests- if you don't, you aren't extreme, end of discussion.

Don't expect a lot of trash talk about other methodologies here, though. This book tells you why XP does what it does. You must decide whether XP makes more sense than the advice you get elsewhere.

Does the world need another methodology? Don't we have enough already? I have asked myself these questions, oh, a few thousand times in the last few years. The answer I keep coming up with is that the world doesn't need another methodology if the ones we already have are working. But as I visit project after project struggling to get some value out of objects, I realize that the advice out there doesn't work for the people I talk to. So I'm going to add my two cents to the pot.

Is it time to talk about XP, yet? No. It hasn't been used on thousands of successful projects at all scales from one person to a thousand. There could easily be some gaping hole that makes the early results with XP entirely irreproducible. On the other hand, none of the ideas in XP are new. Most are as old as programming. The innovation of XP is putting them all under one umbrella and making sure they support each other to the greatest possible degree. From this standpoint, XP is nothing new at all. Except that developers don't practice many of the simple and effective ideas gathered here under the XP banner. And I think the projects that I see would be more successful if they did. So here you go.

The book is divided into three sections:

- The Problem- The chapters from „The Basic Problem" to „Back to Basics" set up the problem Extreme Programming is trying to solve, and criteria for evaluating the solution. Read this section to get an idea of the overall worldview of Extreme Programming.
- The Solution- The chapters from „A Quick Overview" to „Testing Strategy" turn the abstract ideas in the first section into concrete solutions. The emphasis is on the solution, but this section will not tell you exactly how you can emulate the solution, but rather talk about the general shape of the solution. It relates the elements of Extreme Programming back to the problems and principles introduced in the first section.
- Implementing XP- The chapters from „Insourcing" to „Completing the Circle" describe a variety of topics around implementing XP- how to adopt it, what is expected from the various people in an extreme project, how XP looks to the business folks.

The first two sections are written as if you and I were creating a new software development discipline together. We start by examining our basic assumptions about software development and move on through creating the discipline itself.

How I Wrote This Book

My first approach to writing about XP was to carefully outline a five book series covering every aspect of XP. Then all we (me and the rest of the XP gang- Ward Cunningham, Ron Jeffries, Ken Auer, and Martin Fowler at the moment) would have to do is fill in the blanks and we'd have all the books we'd ever need.

Hah! If I can't design a whole software system before I start implementing, how could I ever design a whole book series before I start writing? How unextreme. I'm a little embarrassed, but we all revert under pressure.

We started over. The group decided to write a first „How To XP" book and go from there. We planned on doing it in a style reminiscent of XP- writing a set of stories that describe the experience of reading and applying the book, then writing the book that satisfied those stories. We planned on kicking off the process with a three day writing orgy to help align our conception of the book.

If I was going to try to write 20-30 manuscript pages in three days, I figured I needed some practice. So I thought I would warm up by writing down a few explanations of how XP could possibly work. After I had written a few, an order began to emerge. Then a purpose- helping you decide if you should or shouldn't use XP. Then an outline. Then I had a list of what all I needed to do to finish the book. Then I clicked of the items on the list. That's how I wrote this book.

 Copyright © 1998 Kent Beck, All rights

Acknowledgements

I write in the first person here, not because these are my ideas, but rather because this is my perspective on these ideas. Most of the practices in XP are as old as programming. Ward Cunningham is my immediate source for much of what you will read here. In many ways I have spent the last ten years just trying to explain to other people what he does so naturally. Ron Jeffries, Martin Fowler, and Erich Gamma have helped me refine the ideas and experiment with them in practice. And none of this would happen if I hadn't watched my dad ply his craft all those years.

The Problem

This section sets the stage for Extreme Programming by discussing various aspects of the problem to be solved in inventing a new discipline of software development. By the end of the section we will have discussed the basic assumptions we will use as we choose practices covering the various aspects of software development- the driving metaphor, the four values, the principles derived from those values, and the activities to be structured by our new development discipline.

The Basic Problem

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

- The day for delivery comes, and you have to tell the customer that the software won't be ready for another six months
- After numerous slips, the project is cancelled without ever going into production
- The software is put into production but after a couple of years the cost or defect rate rises so much that it must be replaced
- The software is put into production but the defect rate is so high that it simply isn't used
- The software is put into production but it doesn't solve the business problem that was originally posed
- The software is put into production but the business problem it was originally designed to solve was made obsolete six months ago
- After two years, all the good developers on the project get fed up and leave

There is a huge problem getting software into production. A study of XXX shows that Y% of object projects fail ever to go into production??? But the list above shows that a simplistic goal of putting software into production is not enough to address the problem of software risk.

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. At least I think it does. You'll have to make your own judgement after reading the rest of the book and trying the ideas for yourself.

How does XP address the risks above?

- Schedule slips- XP calls for short delivery cycles, so the scope of any slip is limited. Within the development cycle, XP uses 2-4 week customer-visible iterations for even finer grained feedback about progress.
- Project cancelled - 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 to ensure a quality baseline. XP calls for continual evolution of the design so the cost of maintenance never rises sharply.
- Defect rate- XP tests from the perspective of both developers writing tests method-by-method and customers writing tests program feature-by-program feature.

 Copyright © 1998 Kent Beck, All rights

- Business misunderstood- XP calls for the customer to be an integral part of the development team. The specification of the project is continuously refined as development continues, so learning by the customer and the team can be reflected in the software.
- Business changes- XP shortens the delivery 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 developed.
- Turnover- XP asks developers to accept responsibility for estimating and completing their own work, and honors those estimates. The rules for making and changing estimates are clear. So, there is less chance for a developer to get frustrated by management incompetence. 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 turnover. New team members are encouraged to gradually accept more and more responsibility, and are assisted along the way by each other and by more senior developers.

Our Mission

If risk is the problem, what is the solution? What we need to do is invent a style of software development that tends to address the risks above. We need to communicate it as clearly as possible to developers. We need to set out guidelines for adapting it to local conditions (what is fixed and what is variable?)

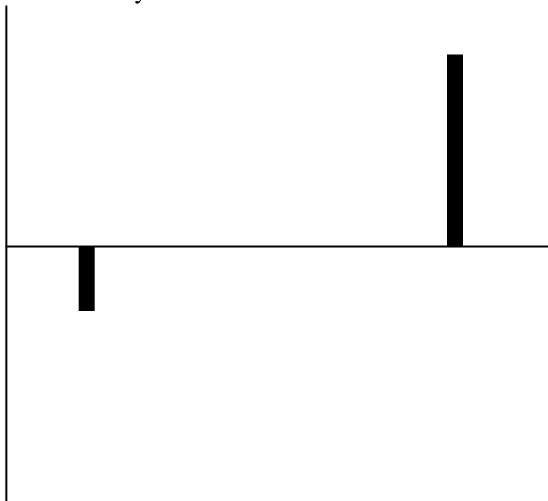
That's sections one and two of this book. We will go step by step through the aspects of the problem of creating a new style or discipline of development, and then we will solve the problem. From a set of basic assumptions, we will derive solutions that dictate how the various activities in software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

Economics of Software Development

What levers do we have to press in attempting to address the problem of projects that fail to deliver value? What is software worth after all? These are questions that can be difficult to ask, especially for a techie like me. But the people that pay for software to be delivered are sure going to ask them, and we'd better have an answer.

Let's start with an extremely simplified view of the economics of software development. Let's say we will spend one million Swiss francs today on developing a piece of software. In one year, we expect to receive three million Swiss francs from the software. How much is the project worth?

The simplest answer is that it is worth two million Swiss francs. I spend one million today. I receive three million in a year. Three minus one is two.



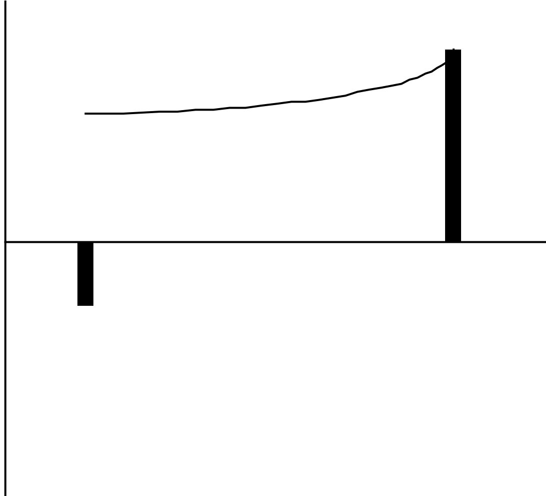
This is a simplistic answer, because time is money (and money is time). In business school, you learn to talk about the net present value of a cash flow. The net present value is the amount you would need today to equal the amount of the cashflow on the date of the cashflow, given that you can put the money in the bank and earn interest.

 Copyright © 1998 Kent Beck, All rights

Let's say we can earn 5% on our money in the bank. Then the one million francs we spend today is worth one million francs today. But the three million francs we receive in a year is really only worth 2.86 million francs, since if we had that much today, in a year we'd have the three million. So the value of the project is 1.86 million francs, still a great return on the investment.

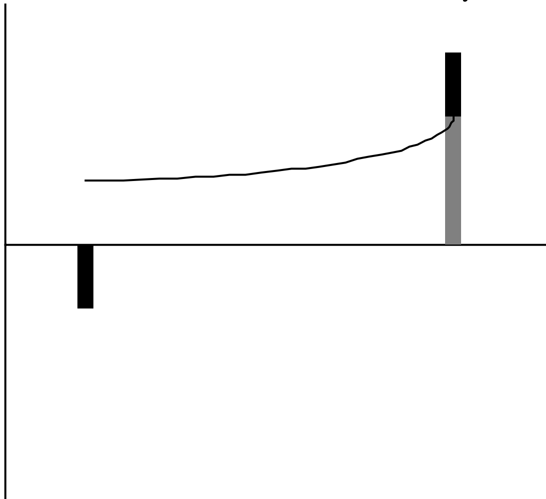
We need to add to the interest the „cost of carry". If you buy a ton of copper, not only do you pay interest on the money you used to buy the copper, you also pay for a warehouse to hold it. This is the cost of carry.

Software has a cost of carry, too. The more complicated your design, the more you have to spend in overhead communicating the design to everyone on the team, the more you have to spend on testing, the greater the chance of errors. So, while the curve is more or less the same shape as the curve for interest, the slope is steeper- there is more erosion of the future value of the software from the point of view of today.



This is still a simplistic answer, because software development is not certain. In fact, it is one of my assumptions that software development will never be certain, that managing the risk of development is the difference between success and failure. We must account for the uncertainty, too.

What if there is a 60% chance of receiving the income? Then we expect, on average, to receive 1.8 million francs in a year. When discounted, this leaves us with 1.71 million francs. So the value of the project is 0.71 million francs. This is still not bad, but it is a far cry from the 2 million francs we first calculated.



Consider now that most software projects last far longer than a year, so the effects of discounting the cashflows is more pronounced than in the simple example. Also, the chances of achieving a payback diminish over time. So a project that looks great at first glance- it only costs a million francs now but it has a potential market of hundreds of millions- can easily turn out not to make economic sense. If the interest rate is high, if the risk of mortality is high, if the investments precede the cashflows by too much, then even such a deal may have a negative value.

There are three ways you can increase the overall value of a project analyzed with this model:

- Push investment out so you don't have to pay so much interest and you are less likely to have to pay

 Copyright © 1998 Kent Beck, All rights

- Pull revenue in so that you get more interest and you are more like to receive the money
- Increase the chances of survival

Sometimes you can combine the strategies. For example, you can split a project into phases, which simultaneously pushes some of the investment into the future, pulls some of the payback closer, and increases the chance of the project surviving.

This model is by no means complete. The biggest cost of the risk of software development is not the money that you may or may not make. Often, the cost of risk is dominated by the cost of the lost opportunities. You are sure you're new system will be in production in a year, so you stop developing the old system. By the time it is obvious that the new system isn't going to ship, the old system is so far behind the competition that you have lost a significant share of the market.

However, the model makes a few pieces of strategy clear:

- You would like to invest money gradually, over time, rather than all at once.
- You would like to begin earning a payback from the system sooner rather than later.
- Above all, you should do anything possible to reduce the risk that the development will fail, since improving the mortality curve has the greatest effect on the value of the project.

At the limit, designing a software process with this model would lead to a project that went into production at the end of its first day. Each day you would invest a little and get a little more in return. Each day you would review the direction of the project, to be sure that you were always creating the most possible value.

You wouldn't develop blindly, though, thinking only of today. You would develop with an eye towards balancing the economics of today with the economics of the indefinite future.

Every day you would examine the investment you were about to make compared with the returns you were to get. You would compare the returns if you continued development and if you stopped investing. The day you made more money by stopping development, you would stop.

This model gives us a broad, strategic view of managing software development. The real world is far more complex. No top manager is willing to review a software project every day. No software project produces concrete value every day. The model above is good for making grand strategic decisions. We need another model, one that helps make more detailed decisions.

Four Variables

Here is a slightly more detailed model of software development. In this model, there are four variables in software development:

- Cost
- Time
- Quality
- Scope

The way the software development game is played is that external forces (customers, managers) get to pick the values of 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, developers, 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.

 Copyright © 1998 Kent Beck, All rights

Interactions Between the Variables

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, „9 women cannot make a baby in one month." (And contrary to what I've heard from some managers, 18 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 delivery cycles. In fact, at the beginning of a team, you can't spend much at all. The investment has to start small and grow over time. After a while, you can productively spend a bunch of money. Too much at the beginning will only spoil the project.

I had one client who said, „We have promised to deliver all of this functionality. To do that, we have to have 40 developers." I said, „You can't have 40 developers on the first day. You have to start with one team. Then grow to two. Then four. In two years you can have 40 developers, but not today." They said, „You don't understand. We have to have 40 developers." I said, „You can't have 40 developers." They said, „We have to." I fired this client soon afterwards, since we clearly weren't speaking the same language.

Brooks' Law is another constraint on cost. Spending more on a project that is late does make it later.

All the constraints on cost can drive managers nuts. 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 developers 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.

Controlling time gives you more control than controlling costs, because there are fewer internal constraints. More or less, you can deliver half as much in half the time, leaving cost and quality constant. Although, as the team learns, this isn't necessarily true.

The constraints on 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, these are some examples of external time constraints. So, often the time variable is 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 on a personal level when I started writing unit tests as outlined below. As soon as I had my tests, I had so much more confidence in my code that I wrote faster, without stress. I refactored more, 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 developers. There is always a tempting short term play, which is to temporarily ignore internal quality to reduce time to market in hopes that external quality won't suffer too much. And you can often get away with this move for a matter of months. Eventually, though, this focus 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 on the mainframe. 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 like they are doing there best. If you deliberately downgrade quality, your team might go faster at first, but soon the demoralization of producing crap will overwhelm any gain from not testing, or not reviewing, or not sticking to standards.

Copyright © 1998 Kent Beck, All rights

Focus on Scope

Lots of people know about the first three variables, but don't acknowledge the fourth. For software development, scope is the most important variable to be aware of. Neither party has more than a vague idea about what is valuable about the software under development. One of the most powerful „moves" in the book is eliminating scope that turns out to be less important than it seemed six months (or a month or a week) ago. If you actually manage scope, you can provide those nasty external forces with control over cost, quality, and time.

One of the great things about scope is that it is a variable that can vary. A lot. For decades, software developers 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." Get over 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. And the beauty of software development is that the development of a piece of software changes its own requirements. As soon as the customers see the first release, they learn. And it's valuable learning, because it couldn't have possibly taken place based on speculation.

What if we saw 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, you can shape it- a little this way, a little that way. If time gets tight towards a release date, there is always something that can be deferred to the next release, preserving your 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 four 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 change down for the life of the system.

Cost of Change

One of the universal assumptions of software engineering is that the cost of changing a program rises exponentially over time. I can remember sitting in a big linoleum-floored classroom as a college junior and seeing the professor write on the board:

$1	$10	$100	$1000	$10000	$100000
Requirements	Analysis	Design	Implementation	Testing	Production

If you catch a problem in the requirements stage, it costs $1 to fix. The same problem in analysis costs $10 to fix. In design, $100. Implementation, $1000. Testing, $10000. Production, oh baby, $100000.

I resolved then and there that I would never let a problem get through to production. No sirree, I was going to catch problems as soon as possible. I would work out every possible problem in advance. I would review and cross-check my code. No way was I going to cost my employer $100000.

So, what's the problem? The problem is that this curve is no longer valid. Or rather, that it doesn't match my experience and the experience of many other people.

The exponential cost curve is based on experimental evidence. But it is evidence of 30 years ago. The software development community has spent enormous resources in the intervening years trying to reduce the cost of change- better languages, better database technology, better programming practices, better environments and tools, new notations.

What would we do if all that investment paid off? What if all that work on languages and databases and whatnot actually got somewhere? What if the cost of change didn't rise exponentially over time, but rose much more slowly, eventually reaching an asymptote? How would we act then? What if tomorrow's software engineering professor writes this on the board:

$1	$3	$5	$30	$30
Day 1	Week 1	Month 1	Year 1	Decade 1

 Copyright © 1998 Kent Beck, All rights

This is one of the premises of XP. Heck, it is THE technical premise of XP. If the cost of change rose slowly over time, you would act completely differently than you do under the assumption that costs rise exponentially. You would make big decisions as late in the process as possible, to defer the cost of making the decisions and to have the greatest possible chance that they were right. You would only implement what you had to, in hopes that the needs you anticipate for tomorrow wouldn't come true. You would implement each requirement as if it were the last. You would introduce elements to the design only as they were needed to simplify the existing or about to exist code.

Keeping the cost of change low doesn't just happen. There are a set of technologies and practices that work to keep software pliable.

On the technology side, objects are really the key technology. Message sending is a powerful way to cheaply build many opportunities for change. Each message becomes a potential point for future modification, a modification that can take place without touching the existing code.

Object databases transfer this flexibility into the realm of permanent storage. With an object database, it is possible to create migrate objects in one format into objects in another format easily, since the code is inherent in the object, not separated as in earlier database technologies. Even if you can't find a way to migrate the objects, you can have to competing implementations of the same messages in the system at the same time.

This is not to say that you must have objects to have flexibility. I learned the fundamentals of XP by watching my dad write real time process control code in assembler. He developed a style that made it possible for him to continuously refine the design of his programs. However, my experience is that the cost of change rises more steeply without objects than with objects.

This is not to say that objects are enough. I have seen (and probably written, if truth were to be known) loads of code written with objects that no one wanted to touch. I identified several factors that made code easy to modify over time:

- Automated tests, so you know if you accidentally changed the existing behavior of the system.
- A simple design, with no extra design elements- ideas that weren't used yet but were expected to be used in the future.
- Practice modifying code, so when the time came to change the code, the team wasn't afraid of it.

With these elements in place- tests, simple design, and an attitude of constant refactoring- I measured the flattened curve above. A change that would have taken a minute before much coding had occurred took 30 minutes after we had been in production for two years. If the change were to a fundamental object, we might sweat a little during those 30 minutes. But by the time we had done it a few times, we would make the necessary changes, run the tests, migrate the objects in the database, and go home for the night. Next morning we'd check the error logs and they would be clean.

With this change in assumption comes the opportunity to take an entirely different approach to software engineering. It is every bit as disciplined as other approaches, but it is disciplined along another dimension. Instead of being careful to make big decisions early and little decisions later, we can create an approach to software development that makes each decision quickly, but backs each decision with automated tests, and that is prepared to change the software when you learn a better way to build it.

Creating such an approach won't be easy, though. We will have to re-examine our deepest assumptions about what makes for good software development. We can take the journey in stages. We'll start with a story, a story that will anchor everything else we do.

The XP Paradigm

Now we have the general shape of the problem- the tremendous cost of risk- and the resource needed to shape the solution- making change later rather than sooner. Now we need to begin to bring the solution into focus. The first thing we need is a metaphor, a shared story that we can turn to in times of stress or decision to help keep us on course.

I can remember clearly the day I first began learning to drive. My mother and I were driving up Interstate 5 near Chico, 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 motion of the wheel affected the direction of the car. Then she told me to line the car up in the middle of the lane. I was driving!

 Copyright © 1998 Kent Beck, All rights

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) got the car back straight. 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 I adopted for XP. There is no such thing as straight and level. Even if things seem to be going perfectly, you don‘t take your eyes off the road. Change is the only constant. Always be prepared to move a little this way, a little that way. Sometimes maybe you have to move in a completely different direction. That’s life as a software developer.

Everything in software changes. The requirements change. The design changes. The business changes. The technology changes. The team changes. The team members change.

The XP team becomes intellectual nomads, always prepared to quickly pack up the tents and follow the herd. The herd in this case might be a design that wants to go a different direction than anticipated, or a customer that wants to go a different direction than anticipated, or a team member who leaves, or a technology that suddenly gets hot, or a business climate that shifts.

Doesn’t belong here, but somewhere After a while, I got the feeling that the problem wasn’t the changes (which everybody bitched and moaned about), but rather the anticipation of change.

Like the nomads, the XP team gets used to traveling light. They don’t carry much in the way of baggage except what they need to keep producing value for the customer.

The moral of this story for the tactics of developing software is that you have to first be generally aware of where you are heading. With the goal in mind, you start moving and making little adjustments. You stay aware of when you are drifting a little, and you make another little adjustment.

The moral of this story for the strategy of developing software is a little more radical. If you followed this story, you would find the smallest piece of functionality that made the biggest difference for the customer and you would put it into production as quickly as possible. Then you would adjust your long term goal to match how the business actually reacted to your software. You would adjust your software to match how the business actually used your software. You would adjust your business to what turned out to be most valuable about the software.

The driving story also has a moral for the XP process itself. The four values of communication, simplicity, concrete feedback, and aggressiveness described below tell us how software development should feel. However, the practices to achieve that feeling will be different from place to place and time to time and person to person. Following the driving story you would adopt a simple set of practices that would give you the feeling you want. As development continues, you would constantly be aware of which practices enhance and which practices detract from your goal. Each practice would be an experiment, to be followed until proven inadequate.

Four Values

Reducing the metaphor to an initial set of practices requires that we agree on what we’re aiming for, what criteria we will use to weigh our decisions and judge our solutions.

Short term individual goals often conflict with long term social goals. Societies have learned to deal with this problem by developing shared sets of values, backed up by myths, rituals, punishments, and rewards. Without these values, humans tend to revert to their own short term best interest.

The four values of XP are:
- Communication
- Simplicity
- Feedback
- Aggressiveness

Communication

The first value of XP is communication. All of the problems I see with projects can be traced back to somebody not talking to somebody else about something important. Sometimes one developer doesn’t tell another about a critical change in the design. Sometimes a developer doesn’t ask the customer the right question, so a critical

 Copyright © 1998 Kent Beck, All rights

domain decision is blown. Sometimes a manager doesn’t ask a developer the right question, and project progress is misreported.

Bad communication doesn’t happen by chance. There are many circumstances that lead to a breakdown in communications. A developer tells a manager bad news and the manager punishes the developer. A customer tells the developer something important and the developer seems to ignore the information.

XP aims to keep the right communications flowing by employing many practices that can’t be done without communicating. They are practices that make short term sense, like unit testing, pair programming, and task estimation, but the effect of testing and pairing and estimating is that developers and customers and managers have to communicate.

This doesn’t mean that communications don’t sometimes get clogged in an XP project. We’re all people here, and people get scared, make mistakes, get distracted. XP employs a coach whose job it is to notice when people aren’t communicating and reintroduce them.

Simplicity

The second XP value is simplicity. I often ask my teams, „What is the simplest thing that could possibly work?“

Simplicity is not easy. It is the hardest thing in the world not to look towards the things you’ll need to implement tomorrow and next week and next month. But thinking ahead is really the fear of that exponential cost of change curve talking.

One the one side you have changes made early. They cost less than the same change made later, even if the curve is asymptotic rather than exponential. However, early changes cost more time earlier. They invoke more risk, because you don’t have the need for them right in your face telling you how they should be done. They also create the risk that they will go wrong between the time that you make them and the time that you need them. They also invoke the risk that your will learn that you don’t actually need them between when you make them and when you imagine that you will need them. They create inertia, because you have to keep them running while you are doing other things.

On the other hand you have the same changes, but made exactly when you need them. They cost more, sure, but not extravagantly more. And all the rest of the factors are on the side of change as needed- shorter time to get a certain functionality done, much less risk of getting the decisions wrong, no risk of obsolescence, no need to carry a change along which isn’t paying for itself.

XP is making a bet. It is betting that it is better to do a simple thing today and pay a little more tomorrow to change it if it needs it, than to do a more complicated thing today that may never be used anyway.

Simplicity and communication have a wonderful mutually supporting relationship. The more you communicate, the clearer you can see exactly what needs to be done and the more confidence you have about what really doesn’t need to be done. The simpler your system is, the less you have to communicate about, which leads to more complete communication, especially if you can make the system enough simpler that it requires fewer developers.

Feedback

The third value in XP is concrete feedback. I often ask my teams, „I don’t know, ask the system?“ or „have you written the test cases for that yet?“ It is so easy to fool yourself that concrete feedback about the current state of the system is absolutely priceless.

This value at the scale of minutes and days. The developers write unit tests for all the logic in the system that could possibly break. The developers have minute by minute concrete feedback about the state of their system. When customers write new stories, the developers immediately estimate them, so the customers have concrete feedback about the quality of their stories. The tracker watches the completion of the tasks in an iteration to give the whole team feedback about whether they are likely to finish everything they set out to do.

This value also works at the scale of weeks and months. The customers and testers write functional tests for all the operations of the system that they care about. They have concrete feedback about the current state of their system. The customers review the schedule every two or three weeks to see if the team’s overall velocity matches the plan. The system is put into production as soon as it makes sense, so the business can begin to „feel“ what the system is like in action and discover how it can best be exploited.

 Copyright © 1998 Kent Beck, All rights

This last needs a little explanation. One of the strategies in the planning game is that the team puts the most valuable requirements (we call them stories) into production as soon as possible. This gives the developers concrete feedback about the quality of their decisions and development process that never comes until the rubber meets the road. Some developers have never lived with a system in production. How can they possible learn to do a better job?

Most of the projects I visit seem to have exactly the opposite strategy. As soon as the system is in production, you can no longer make „interesting" changes, so keep the system in development as long as possible. I think this is unnatural. „In development" is a temporary state, one that the system will only be in for a small percentage of its life. Far better to give it independent life, to make it stand and breathe on its own. That way, if it doesn't make sense, you won't have spent much money and time on it. If it does make sense, it will be that much sooner that you will learn the lessons of the system and move on to more valuable things.

Concrete feedback works together with communication and simplicity. The more feedback you have, the easier it is to communicate. If someone has an objection to some code you have written and they hand you a test case that breaks it, that is worth a thousand hours of discussion about design aesthetics. If you are communicating clearly, you will know what to test and measure to learn about the system. Systems that are simple are easier to test. Writing the test gives you a focus for just how simple the system can be- until the tests run, you're not done, and when the tests all run, you're done.

Aggressiveness

Within the context of the first three values- communication, simplicity, and concrete feedback- then it's time to go like hell. If you aren't going absolutely your top speed, somebody else will be, and they will eat your lunch.

One of the stories I use to demonstrate aggressiveness in action is the time, about 80% through the engineering schedule of the first release of the C3 project, when the team discovered a fundamental flaw in the architecture. The functional test scores had been rising nicely, but then had flattened out far below where they needed to be. Fixing one defect caused another one. The problem was this architectural flaw.

The team did exactly the right thing. When they discovered that there was no way forward, they fixed the flaw. This broke half of the tests that had been running. A little concentrated effort later, however, and the tests scores were again headed towards completion. That was aggressive.

Another aggressive move that I see in my teams is throwing code away. You know how sometimes you work all day on something, and it isn't going very well, and the machine crashes? And how in the morning you come in and in half an hour replay all of the previous day's work but clean and simple. Okay- use it. If the end of the day is coming and the code is a little out of control, toss it. Maybe save the test cases, if you like the interface you've designed, but maybe not. Maybe just start over from scratch.

You have three design alternatives. Code a day's worth of each of them, just to see how they feel (in XP we call this a Spike). Toss them all and start over on the most promising one.

XP resembles a hill climbing algorithm. You get a simple design, then you make it a little more complex, then a little simpler, then a little more complex. What is to say that you won't ever develop yourself into a corner? Aggressiveness. Every once in a while someone on the team will have a crazy ass idea. They'll try it out. It will work (sometimes). Now you're climbing a whole new hill.

Now, if you didn't have the first three values in place, aggressiveness by itself is just plain hacking (in the pejorative sense of that word). However, when combined with communication, simplicity, and concrete feedback it becomes extremely valuable.

Communication support aggressiveness because it opens the possibility for more high-risk, high-reward experiments. „You don't like that? I hate that code. Let's see how much of it we can delete in an afternoon." Simplicity support aggressiveness because you can afford to be much more aggressive with a simple system. You are much less likely to break it. Aggressiveness support simplicity because as soon as you see the possibility of simplifying the system you try it. Concrete feedback supports aggressiveness because you feel much safer trying radical surgery on the code if you can push a button and see the tests turn green at the end (or not, in which case you throw the code away).

The Values in Practice

I asked the C3 team to tell me stories about their proudest moment on the project. I was hoping to get stories about big refactorings, or being saved by tests, or a happy customer. Instead, I got this:

 Copyright © 1998 Kent Beck, All rights

„The moment I was proudest of was when Eddie quit the team to spend more time with his family. The team gave him total respect. Nobody said a word about him leaving. Everybody just asked what they could do to help.“

This points to a deeper value, one that lies below the surface of all of the four- humanity.

All of this high-minded talk is well and good, but if there isn't some way to reduce it to practice, to reinforce it, to make the values a natural habit, then all we will have is yet another brave leap into the swamp of methodological good intentions. Next we have to have a more concrete guide leading us to practices that satisfy and embody the four values of communication, simplicity, feedback, and aggressiveness.

Basic Principles

Now we have the driving metaphor and the values of communication, simplicity, feedback, and aggressiveness. Next, we need to agree on some fundamental principles that we'll use as we reduce the metaphor and the values to practice. These principles will help us as we choose between alternatives. We will prefer an alternative that meets the principles more fully to one that doesn't. They are more concrete than the values, but each embodies the values. Where a value can be vague- one person's simple is another person's complex, a principle is more concrete- either you are playing to win or you're not.

Teach learning. Rather than make a bunch of doctrinaire statements like „thou must do testing like XYZ“, we will focus on teaching strategies for learning how much testing you should do. And how much design, and refactoring, and everything else. There are some ideas that we will be certain of, and those we will state with certainty. There will be others that we don't have quite so much confidence in, and those we will state as strategies with which the reader can learn their own answers.

Small initial investment. Too many resources is a recipe for disaster. Tight budgets force developers and customers to pare requirements and approaches. You can only do so much, and the focus that generates encourages you to do a good job of everything you do.

Resources can be too tight. If you don't have the resources to solve even one interesting problem, then the system you create is guaranteed not to be interesting. If you have someone dictating scope, dates, quality, and cost, then you are unlikely to be able to navigate to a successful conclusion. Mostly, though, I observed that everyone can get by with a lot less than they are comfortable with.

Play to win. It was always wonderful to watch John Wooden's UCLA dynasty teams. Usually they were crushing the opposition. However, even if the game was close going into the final minutes, UCLA was absolutely certain that they were going to win. After all, they had won so many, many times before. So, they were relaxed. They did what they needed to do. And they won again.

I remember an Oregon basketball game that provides a stark contrast. They were playing a nationally ranked Arizona team that was destined to send four players to the NBA. At half time, amazingly, Oregon was up by 12 points. Arizona couldn't do anything right, and Oregon's offence had them baffled. After the half, however, Oregon came out and tried to play as slowly as possible to reduce the number of points scored and preserve the victory. The strategy didn't work, of course. Arizona found ways to use their huge advantage in talent to win the game.

The difference is between playing to win and playing not to lose. Most of the software development I see is played not to lose. Lots of paper gets written. Lots of meetings are held. Everyone is trying to develop „by the book“, not because it makes any particular sense, but because they want to be able to say at the end that it wasn't their fault, they were following the process.

Software development played to win does everything that helps the team to win and doesn't do anything that doesn't help to win.

Rapid feedback. Learning psychology teaches that the duration between an action and its feedback is critical to learning. Animal experiments show that even small differences in the timing of feedback result in large differences in learning. A few more seconds and the mouse doesn't learn that the red button means food. So, one of the principles is to get feedback, interpret it, and put what is learned back into the system as quickly as possible. This is true of the business learning how best the system can contribute, and feeding back that learning in days or weeks instead of months or years. This is true of developers learning how best to design, implement and test the system, and feeding back that learning in seconds or minutes instead of days, weeks, or months.

 Copyright © 1998 Kent Beck, All rights

Concrete experiments. Every time you make a decision and you don't test it, there is some probability that the decision is wrong. The more decisions you make, the more these risks compound. Therefore, the result of a design session should be a series of experiments, not a finished product. The result of a discussion of requirements should be a series of experiments. Every abstract decision should be tested.

Assume Simplicity. Treat every problem as if it can be solved with ridiculous simplicity. The time you save on the 98% of problems for which this is true will give you ridiculous resources to apply to the other 2%. In many ways, this is the hardest principle for developers to swallow. We are told to plan for the future, to design for reuse. Instead, XP says to do a good job (tests, refactoring, communication) of solving today's job today and trust your ability to add complexity in the future where you need it.

Open, honest communication. This is almost such a motherhood statement that I left it out. Who wouldn't want to communicate openly and honestly? Everybody I visit, it seems. Developers have to be able to explaining the consequences of other people's decisions. They have to be able to tell each other where there are problems in the code. They have to be free to express their fears, and get support.

One of the first signs of trouble I look for on problems is any statement made after first looking around to see who is listening. If there are personal matters to be discussed, I can understand the need for privacy. But which of two object models to use is not a matter that is helped by a „top secret" stamp.

Working with people's instincts, not against them. Somewhere I heard the statistic that only 30% of all development projects even attempt to follow a published methodology. This fact gave me pause at the beginning of my work on XP. What is the point, if only 3 people in 10 would ever pay attention. Then I got to looking at what was in published methodologies.

The problem I saw was that the books contained all sorts of stuff that people don't like to do. The books on requirements made you fill out long forms, even if they didn't make sense. The books on analysis made you draw endless diagrams, with no feedback on whether you were right. Ditto the books on design. The books on testing made it clear that you were doomed if you didn't spend at least 110% of your time on testing.

No wonder people didn't follow the books. They aren't any fun. They don't make sense, even if they are persuasive.

So I resolved to go with people, not against them. People like winning. People like learning. People like interacting with other people. People like being part of a team. People like being in control. People like being trusted. People like doing a good job.

It's been tricky, designing a process where following short term self interest also serves long term team interest. If XP can't work with people's short term interest, however, it is doomed to the outer methodological darkness.

Accepted responsibility. No single action takes the life out of a team or a person more than being told what to do. Primate dominance displays only work so long in getting people to act like they are going along. Along the way, a person told what to do will find a thousand ways of expressing their frustration, most of them to the detriment of the team and many of them to the detriment of the person.

My alternative is that responsibility can only be accepted, it cannot be given. This does not mean that you always do exactly what you feel like doing. You are part of a team, and if the team comes to the conclusion that a certain task needs doing, someone will choose to do it, no matter how odious.

Incremental change. Big changes don't work. From the Hong Kong airport to the US air traffic control system, big changes just don't work. Even in Switzerland, where I live now, center of meticulous planning, they don't try to make big changes. Any problem is solved with a series of the smallest changes that make a difference.

You'll find this applied many ways in XP. The design changes a little at a time. The plan changes a little at a time. The team changes a little at a time.

Local adaptation. Whatever I tell you, you have to adapt to your local conditions. This is an application of accepted responsibility to development process. Adopting XP does not mean that I get to decide how you are going to develop. It means that you get to decide how to develop. I can tell you what I have found to work well. I can point out the consequences that I see from deviating. At the end of the day, however, this is your process. You have to decide on something today. You have to be aware of whether it still works tomorrow. You have to change and adapt. Don't read this thinking, „Finally, now I know how to develop." You should end up saying, „I have to decide all of this AND program?" Yes, you do. But it's worth it. I say that because I found it to be worth it.

 Copyright © 1998 Kent Beck, All rights

Embracing change. Change is the only constant. In any situation where you are deciding about development process, the best alternative is the one that preserves the most options while actually solving your most pressing problem.

Travel light. You can't expect to carry a lot of baggage and move fast. We should produce the fewest and simplest enduring deliverables possible, consistent with our other goals.

Quality work. We have to do a good job. Nobody likes working sloppy. If the four project development variables are scope, cost, time, and quality, then quality isn't really a free variable. The only possible values are „excellent" and „insanely excellent", depending on whether lives are at stake or not. Otherwise you don't enjoy your work, you don't work well, and the project goes down the drain.

Honest measurement (no false detail). Our quest for control has led us to measure, which is fine, but it has led us to measure at a level of detail that is not supported by our instruments. Better to say „this will take two weeks, more or less" than say, „14.176 hours," if you have no real way of knowing to this level of detail.

Living software. You can understand software as a living system. It is constantly growing and changing. Any attempt to see software as a thing which develops and then freezes will miss ripe opportunities for understanding and learning which come only after a system has been in production for years.

Back to Basics

Driving metaphor. Four values- communication, simplicity, feedback, and aggressiveness. A double handful of principles. Now we are writing to start building a discipline of software development. The first step is to decide on the scope. What is it that we will try to prescribe? What sorts of problems will we address and what sorts of problems will we ignore?

I remember when I first learned to program in BASIC. I had a couple of workbooks covering the fundamentals of programming. I went through them pretty quickly. When I had done that, I wanted to tackle a more challenging problem than the little exercises in the books. I decided I would write a Star Trek game, kind of like one I had played at the Lawrence Hall of Science in Berkeley, but cooler.

My process for writing the programs to solve the workbook exercises had been to stare at the problem for a few minutes, type in the code to solve it, then deal with whatever problems arose. So, I sat confidently down to write my game. Nothing came. I had no idea how to write an application bigger than 20 lines. So I stepped away and I tried to write down the whole program on paper before typing it in. I got three lines written before I got stuck again.

I needed to do something beyond programming. But I didn't know what else to do.

So, what if we went back to that state, but in the light of experience? What would we do? What would we try to get out of each activity as we experienced it afresh?

Coding

At the end of the day, there has to be a program. So, I nominate coding as the one activity we know we can't do without. Whether you draw diagrams that generate code or you type at a browser, you are making code.

What is it that we want to get out of code? The most important thing is learning. The way I learn is to have a thought, then test it out to see if it is a good thought. Code is the best way I know of to do this. Code isn't swayed by the power and logic of my rhetoric. Code isn't impressed by college degrees or large salaries. Code just sits there, happily doing exactly what you told it to do. If that isn't what you thought you told it to do, that's your problem.

When you code something up, you also have an opportunity to understand the best structure for the code. There are certain signs in the code that tell you that you don't yet understand the necessary structure.

Code also gives you a chance to communicate clearly and concisely. If you have an idea and explain it to me, I can easily misunderstand. If we code it together, though, I can see in the logic you write the precise shape of your ideas. Again, I see the shape of your ideas not as you see them in your head, but as they find expression to the outside world.

 Copyright © 1998 Kent Beck, All rights

This communication easily turns into learning. I see your idea and I get one of my own. I have trouble expressing it to you, so I turn to code to. Since it is a related idea, we use related code. You see that idea and have another.

Finally, code is the one artifact that development absolutely cannot live without. I've heard stories of systems where the source code was lost but they stayed in production. Sightings of such beasts has become increasingly rare, however. For a system to live, it must retain its source code.

Since we have the source code, we should use it for as many of the purposes of software engineering as possible. It turns out that code can be used to communicate- expressing tactical intent, describing algorithms, pointing to spots for possible future expansion and contraction. Code can also be used to express tests, tests that both objectively test the operation of the system and provide a valuable operational specification of the system at all levels.

Testing

I agree with the English ??? philosophers Locke, Berkeley, and Hume in this small area- software features that can't be measured simply don't exist. I am so good at fooling myself into believing that what I wrote is what I meant and that what I meant is what I should have meant that I don't trust anything I write until I have tests for it. The tests tell me if I did what I thought I did.

I have gotten to where I don't even want to start thinking about how to implement something until I already have the tests for it. That way I know not only what I did but I also know when I am done. Until the tests run I'm not done. When the tests run I am done.

Tests are both a resource and a responsibility. You don't get to write one test, make it run, and declare yourself finished. You are responsible for writing every test that you can imagine will break. After a while you will get good at reasoning about tests- if these two tests work, then this third test will work without my having to write it. Of course, this is exactly the same kind of reasoning that leads to bugs in programs, so you have to be careful about it. If problems show up later and they would have been uncovered had you written that third test, you have to be prepared to learn the lesson and write that third test next time.

I got along for years without writing repeatable tests for my systems. I know a lot of other folks who have done the same. So why don't I leave out testing in my list of essential activities, since it clearly isn't essential. I have two answers, one short term and one long term.

The long term answer is that tests keep the program alive longer (if they are run and maintained). When you have the tests, you can make more changes longer than you can without the tests. If you keep writing the tests, your confidence in the system increases over time.

One of the principles is to go with human nature and not against it. If all you could make was a long term argument for testing, you could forget it. Some people would do it out of a sense of duty or because someone was watching over their shoulder. As soon as the attention wavered or the pressure increased, no new tests would get written, the tests that were written wouldn't be run, and the whole thing would fall apart. So, if we want to go with human nature and we want the tests, we have to find a short term selfish reason for testing.

Fortunately, there is one. Programming when you have the tests is more fun than programming when you don't. I think it is more fun for me because I code with so much more confidence. I never have to entertain those nagging thoughts of „well, this is the right thing to do right now, but I wonder what I broke" Push the button. If the light turns green, you are ready to go to the next thing with renewed confidence.

I caught myself doing this in a public programming demonstration. Every time I would turn from the audience to begin programming again, I would push my testing button. I hadn't changed any code. Nothing in the environment had changed. I just wanted a little jolt of confidence. Seeing that the tests still ran gave me that.

Programming and testing together is also faster than just programming. I didn't expect this effect when I started, but I certainly noticed it and have heard it reported by lots of other people. You might gain productivity for half an hour by not testing. Once you have gotten used to testing, though, you will quickly notice the difference in productivity. For me, the gain in productivity comes from a reduction in the time spent debugging- I no longer spend an hour looking for a bug, I always find it in minutes (or I have a much bigger design problem).

I'll reiterate. Testing done badly becomes a set of rose colored glasses. You gain false confidence that your system is okay because the tests all run. You move on, little realizing that you have left a trap behind you, armed and ready to spring the next time you come that way.

 Copyright © 1998 Kent Beck, All rights

However, the other attitude doesn't make sense to me. I programmed with a guy who said, „If the test cases all run, the only thing it proves is that you don't have enough test cases." In practice, it wasn't true. If all the test cases ran before we put code into production, we almost never had problems the next day. If we had code that hadn't been tested, we generally had problems with it. So, it may be true that testing can never prove the absence of bugs, only the presence of features, but it is irrelevant. If the bug rate is low enough that you can act like you have proved the absence of bugs, then you can treat your program in a completely different way than if you are never quite sure how many bugs you might have.

The trick with testing is finding the level of defects you are willing to tolerate. If you can stand one customer complaint per month, then invest in testing and improve your testing process until you get to that level. Then, using that standard of testing, move forward as if the system is fine if the tests all run.

Oh, there is one other very good reason to test. You'll program faster if you test. This counterintuitive observation comes about because of two effects. First, when you test, especially when you test first, you are solving one problem at a time. First you decide what the interface will be, and what the externally visible behavior will be. You think (you eventually learn to think, I'm still working on this part) about the interface and only the interface. You purge your mind of all thoughts of the implementation. Because of that, you do a better job (read „quicker and less likely to need repair") of designing the interface. Then when you go to implement, all you do is make the test run. You aren't thinking about the interface. That's fixed. You aren't thinking about the next test case. That comes after a cup of coffee. Because you think only about a single test case at a time, you do a careful job of making it run. Once again, it happens quicker and with less chance of needing repair.

There are two audiences for the tests. The developers are confident that the a piece of logic runs when they write it. They need to make that confidence concrete in the form of tests so everyone else can share in their confidence. The customers need to be confident that the system will work from their perspective, too, so they need to prepare a set of tests that represent their confidence, „Well, I guess if you can compute all of these cases, the system must work."

Listening

Programmers don't know anything. Rather, programmers don't know anything that business people think is interesting. Hey, if those business people could do without programmers, they would throw us out in a second.

Where am I going with this? Well, if you resolve to test, you have to get the answers from somewhere. Since you (as a programmer) don't know anything, you have to ask someone else. They will tell you what the expected answers are, and what some of the unusual cases are from a business perspective.

If you are going to ask questions, then you'd better be prepared to listen to the answers. So listening is the third activity in software development.

Programmers listen in the large, too. They listen to what the customer says the business problem is. They help the customer to understand what is hard and what is easy, so it is an active kind of listening. The feedback they provide helps the customer understand their business problems better.

Just saying, „You should listen to each other and to the customer," doesn't help much. People try that and it doesn't work. We have to find a way to structure the communication so that the things that have to be communicated get communicated when they need to be communicated and in the amount of detail they need to be communicated. Similarly, the rules we develop also have to discourage communication that doesn't help, that is done before what is to be communicated is really understood, that is done in such great detail as to conceal the important part of the communication

Designing

Why isn't the above enough? Why can't you just listen, write a test case, make it run, listen, write a test case, make it run indefinitely? Because we know it doesn't work that way. You can do that for a while. In a forgiving language you may even be able to do that for a long while. Eventually, though, you get stuck. The only way to make the next test case run is to break another. Or the only way to make the test case run is far more trouble than it is worth. Entropy claims another victim.

The only way to avoid this is to design. Designing is creating a structure that organizes the logic in the system. Good design organizes the logic so that a change in one part of the system doesn't tend to require a change in another part of the system. Good design ensures that every piece of logic in the system has one and only one

 Copyright © 1998 Kent Beck, All rights

home. Good design puts the logic near the data it operates on. Good design allows the extension of the system with changes in one place.

Bad design is just the opposite. One conceptual change requires changes to many parts of the system. Logic has to be duplicated. Eventually, the cost of a bad design becomes overwhelming. You just can't remember any more where all the implicitly linked changes have to take place. You can't add new function without breaking existing function.

Complexity is another source of bad design. If a design requires four layers of indirection to find out what is really happening, and if those layers don't provide any particular functional or explanatory purpose, then the design is bad.

So, the final activity we have to structure in our new discipline is designing. We have to provide a context in which good designs are created, bad designs are fixed, and the current design is learned by everyone who needs to learn it.

Conclusion

So you code because if you don't code, you haven't done anything. You test because if you don't test, you don't know when you are done coding. You listen because if you don't listen you don't know what to code or what to test. And you design so you can keep coding and testing and listening indefinitely. That's it. Those are the activities we have to help structure:

- Coding
- Testing
- Listening
- Designing

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- planning, testing, development, and design. We have a set of guiding values and principles to guide us as we choose strategies for each of these activities. And we have to flattened cost curve as our ace in the hole to simplify the strategies we choose.

A Quick Overview

Metaphor, values, principles, the four activities of coding, testing, listening, and designing. Now we are ready to say what we are going to do about the activities. Here is a quick look at the strategies we will employ in XP.

Possibly add coding standards, incremental planning,???

So, the job of XP is to structure the four activities above- coding, testing, listening, and designing. 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.

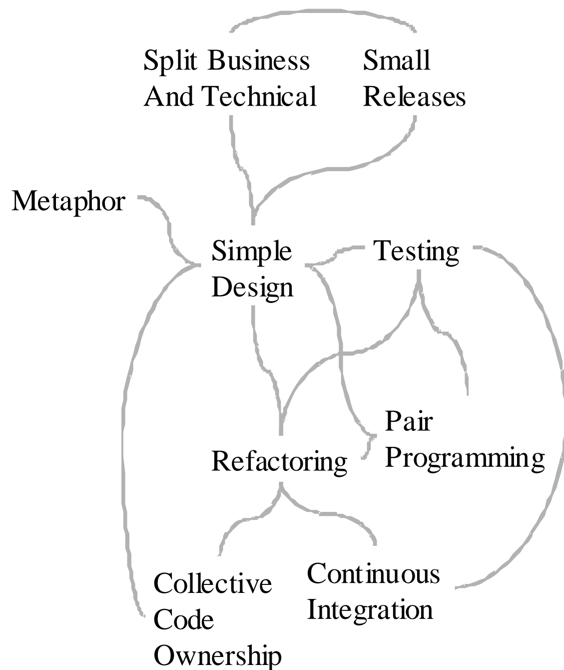
No problem.

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. (Not to give away the story or anything, but the basic idea is that the strength of several other of the practices will make up for the weakness of any single practice). Later chapters will cover some of the topics in more detail.

Here is a diagram that summarizes the practices:

 Copyright © 1998 Kent Beck, All rights



Split Business and Technical Decisions

Neither business considerations not 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 today in order to be „good enough“? 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 developer.
- 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 developer’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 my 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 developers 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 towards the end of the development of a release.

 Copyright © 1998 Kent Beck, All rights

Why? You can't make the best software without the best decisions, and you can't have the best decisions if the wrong people make the decisions.

Small Releases

Every release should be as small as possible, containing the most valuable business requirements. The release has to make sense as a story- 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.

Why? Because the feedback you get from being in production is ten times louder and more accurate than any kind of feedback you get during development. The faster you learn, the faster you get better. Also, each release provides a natural inflection point at which to reflect on and if necessary change the project's direction.

Metaphor

The software should be 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 management is like a spreadsheet.

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

Why? You need a common story that causes development to tend to align rather than diverge, but you would like it to be a story that can be put together quickly and allowed to ripen over time, rather than having to spell it out before you have any experience.

Simple Design

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

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

Every piece of design in the system should be able to absolutely justify its existence on these terms. If you took it out, you would have to violate point 1 or 2 above.

This is quite 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. If you find you need something and there is already code there, put it in anyway.

Why? You'll get a simple design done sooner. You'll do a better job because there is less work to do.

Testing

Any program feature without an automated test simply doesn't exist. Developers 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.

Why? Without the feedback of the tests you really have no idea whether you are making progress or not. You might be right on target. You might not be half as done as you think you are. In addition, tests make the developers more productive. The functional tests also draw the customers into the development process in a concrete way. They are required to say precisely what they mean, and what they don't mean.

 Copyright © 1998 Kent Beck, All rights

Refactoring

When implementing a program feature, the developers always consider before they start if there is a way of changing the existing program to make the adding of the feature simple. After they have added a feature, the developers always consider if they now see how to make the program simpler while still running all of the tests.

Note that this means that sometimes you do more work than absolutely necessary to get a feature running. But in doing it, you ensure that you can add the next feature with a reasonable amount of effort, and the next, and the next.

At the limit this means that if a programmer sees a one minute ugly way to get a test working and a one month way to get it working with a simpler design, the correct choice is to spend the month. Fortunately this is almost never the choice.

Why? Programs should be able to sustain development indefinitely. Without refactoring, though, you soon would find yourself unable to proceed. Either you would want to add a feature and simply couldn't without breaking an existing feature, or you couldn't add the new feature without paying an exorbitant price.

Pair Programming

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

Why? Two people get more done in one day this way than if they work at separate machines. The quality is far superior. Information rapidly disperses among the team if the pairs change frequently. And it's a heck of a lot more fun.

Collective Code Ownership

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

Contrast this to the previous two models- no ownership and individual ownership.

In the beginning, 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 may not be easy to determine statically. The code grew quickly, but it also quickly became unstable.

To get control of this situation, people invented individual code ownership. 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 this is generally that the code changes much more slowly than the experience, as people are reluctant to interrupt the owner, and after all, they need the change now, not later. So the code remains stable, but it doesn't grow as quickly as it should.

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, but if a developer is working and sees an opportunity to improve an object, they go ahead and improve it if it makes their life easier.

Why? It is often not until the third or fourth use of a piece of code that the most desirable form for the code becomes clear. It is at that moment of insight, „Why in the world couldn't they have just made a message XXX, then all of these users could have been so much simpler?“ that you want the developers to just fix the code and all the users.

Continuous Integration

Code is integrated and tested after a few hours, perhaps 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).

Why? Integration gives you lots of feedback about where you are with respect to the rest of the team. It also gives the pair a sense of closure about their work- there, it's in the build and we know it runs.

Copyright © 1998 Kent Beck, All rights

How Could This Work?

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 the have been abandoned for more complicated, higher overhead practices as their weaknesses have become apparent. Why isn't XP a simplistic, Pollyanna 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 defeat of the exponential 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.

So, here is another look at the same practices, but this time focused on what usually makes the practice untenable and how the other practices keep the bad effects from overwhelming the project. Read this chapter if you want to get a sense that the whole story could possibly work, that XP isn't just dangerously simplistic software development.

Split Business and Technical Decisions

Splitting decision making between business and technical people seems like a recipe for disaster. After all, if someone isn't in charge, aren't there endless fights?

The first solution to reducing the chance of arguments is to make it clear that business is in charge of development. The big strategic decisions are made by the business people for business reasons. However, those decisions aren't made in a vacuum. They are informed by the estimates of the developers.

The second solution to reducing arguing is to make the rules for who does what as clear as possible. The Planning Game is one way to make the rules clear. There are players, and pieces, and allowable moves. If someone tries to make a move they aren't allowed to make, the other players can remind them of the rules.

If there isn't trust between developers and customers, arguments are far more likely. Small Releases goes a long way toward alleviating this problem, as it lets both sides share victory early. It gives the customer an early chance to affect the process, to get proof that the developers are actually going to listen. The testing process, especially functional tests defined by the customer, also helps create a relationship of trust that short circuits much of the wrangling that would otherwise make split decision making ineffective.

Small Releases

If all you did with your software development process was make the releases monthly instead of annually, you would incur huge costs. Each release costs a fixed amount to build, test, and distribute.

The first effect is that as you make more and more releases, and you are aware of what makes them easy and what makes them hard, you get better at releasing software. You build tools to help. You develop standard sets of practice that smooth the process.

The tests go a long way towards reducing the cost of releasing code to customers. There isn't ever a large and dangerous „Testing Phase" after development ceases. I have been in teams where the testing phase was deliberately padded to account for the possibility that something might go wrong. You don't need to do that in an XP project, since the tests are being written and made to run as you go along.

Continuous Integration also makes small releases go smoother. Every integration is like a little build for release, and these integrations happen several times a day. At nearly every moment you have software you wouldn't mind shipping to customers, if they were willing to live with the functionality implemented so far. This also goes far towards making the „real" release process, when it comes, into a non-event.

Metaphor

A metaphor seems to be too simple a thing to base the entire design of a system on. It doesn't seem to offer enough details to guide development. Couldn't everyone have their own interpretation of the metaphor? Couldn't the design of the system diverge?

 Copyright © 1998 Kent Beck, All rights

Pair Programming is the biggest practice preventing design drift. In the morning, you and are working on code in one part of the system. In the afternoon, we are both working with other people, relating what we learned in the morning to our tasks of the afternoon. Pair Programming goes a long way towards making sure that everyone has the same basic understanding of the system.

Also, if you keep the design simple, with no features that don't absolutely have to be there, then you need less guidance before you start. There are simply fewer decisions that could possibly go wrong.

What if the metaphor is wrong? What if you got a year into development and realized that the metaphor you had chosen had fundamentally limited you? Well, if you are quickly putting small releases of the most important features into production, you are likely to learn about any such limitations much sooner than a year. And, if you have to switch metaphors, which will likely involve changing a bunch of class names and method names and variable names, you do it. You have all of your experience with refactoring to rely on, and when you are done, you push the button to make sure all the tests still run. So even the unlikely disaster isn't so bad.

Simple Design

A really simple design will have to change. The design as it stands is unlikely to absorb the next requirements smoothly. Something will have to be added.

The first point is that this is not unique to simple designs. Complicated designs, done before development or allowed to grow organically, also have to change. And changing a simple design is easier than changing a complicated design.

Second, all the tests you have written and all the practice with refactoring ensure that the when the time comes for change, it won't be prohibitively expensive. If you do it all the time, it won't even be particularly disruptive-you will change the design a little every day.

The tests help with the simple design in another way. Every developer has a tendency to look ahead, to add complexity „just in case". The tests help you combat this tendency. You only need enough design to run all the tests, and to satisfy the simple style rule „Say everything once and only once." I commonly find myself saying, „Oh, well if all I had to do was THAT then I could just design it like THIS." And more times than not, that's all the design I ever need for that area.

Another point is that as the design matures, the areas where the design is likely to need more flexibility are precisely those areas where it is likely to gain flexibility. „Oh, I see, if this was an object instead of a method, then we could do these three things more simply." And then along comes the fourth variation and it turns out to be easy.

Refactoring and Collective Code Ownership help ensure that the design sheds excess complexity quickly. ???

Testing

The testing strategy can't work because you couldn't possibly get programmers to write their own tests. And even if you did, they would be too expensive to maintain and run. And wouldn't all that time spent testing slow down development? And customers writing tests? Puhlease...

First, unit testing by developers improves productivity. The reduced stress because you know exactly where you stand at any moment helps developers go from smoothly from one task to another without ever stopping to wonder if they have broken something.

This feeling of confidence is reinforced every time you get one more test to run. It is reinforced again when you integrate and run all the tests. Erich Gamma coined the phrase „Test Infected" for this addiction. It is the feeling that you can't go home if all the tests aren't running.

Pair Programming also reinforces testing. Even if I am feeling like slacking off on the tests, you won't let me get away with it. You will insist that I write the tests, and if I don't you will grab the keyboard and write them yourself.

Customers come to love the tests. The tests represent their confidence that the system is ready to go. The more tests they have, the more confident they feel. This, too, becomes addictive.

The Simple Design makes writing the tests easier, because there is less to test with a simple design than with a complex one.

Copyright © 1998 Kent Beck, All rights

Refactoring

Refactoring couldn't work because it takes longer to change code than to design it right in the first place. And aren't you always risking breaking something?

First, if you practice refactoring, and do it in pairs, the cost is not nearly as high as you might imagine (or you might have experienced in the past). If you do it all the time, you get good at it.

The tests have a huge role to play in refactoring. There are few better feelings in software development than having a powerful insight into how your program should be structured, changing the structure, pushing the button, and watching the tests all run. You have just added great value to your program, because the insights that come late are always the least obvious. You have staved off entropy for one more day. And you know that your program will be better for you and the rest of the team to live with tomorrow.

Refactoring tends to lead you all over the system- you add a parameter to a method and several implementers must change. You have to practice Collective Code Ownership for this to work. If you had to go and get 5 people's permission before you made the change, you would just find some workaround instead, and that valuable moment would be lost.

Continuous Integration also plays into reducing the cost of refactoring. If you only ever refactor for a couple of hours at a time, then your chances of colliding with what someone else is doing are greatly reduced. Even if there is a collision, you will both find out in a matter of hours, so the problem will be easy to fix.

Pair programming makes refactoring go faster and greatly reduces its risk, but probably the most important contribution of pair programming to refactoring is courage. If we are programming together and I see something that might pay off, I am much more likely to try it if you encourage me.

Pair Programming

You can't possibly write all the production code in pairs. Won't it will go slower? My manager won't ever allow us to do that. And what if two people don't get along?

The productivity of pairs is greater than the productivity of the same two people working separately. Now, this relies on a number of factors, like the responsiveness of the development environment. It is certainly easier to pair program in Smalltalk than in Java, and much easier in Java than in C++. However, there is still productivity to be had from programming two to a machine.

The Simple Design ensures that there isn't too much to communicate about. If every time a pair sat down, they had to spend an hour reviewing the design before they began, it would never work. If they both have a good general idea of the design, and one of them has detailed knowledge of the area to be modified, they can get to work right away.

Unit testing helps Pair Programming by focusing both people on the same task. The first problem to solve when you begin to pair is to figure out what problem you are solving. The tests provide a concrete focus for this task. You can begin implementing after you've implemented the first test. Until the tests all run, you aren't done. Conversely, when all the tests you can imagine writing run, then you're finished.

Continuous Integration gives the pair a sense of closure. When you have the tests for a task running, you integrate your code with the current release and run the tests. When the tests all run, you release your code. This gives the pair a natural end to their programming episode. Now they can go their separate ways feeling that they have made concrete progress.

Collective Code Ownership

You can't just go changing code all over the system. You might break something. And the developers will be constantly tripping over each other.

The tests are the first line of defense. As the tests get better and better at catching errors, the chance that you will accidentally break something drops lower and lower. At some point, the chance of accidentally breaking something is low enough that you can act like it never happens. And then you can go much faster and make changes to many more parts of the system. But you can only do this if you write the tests.

Pair programming exerts a powerful stabilizing effect on collective code ownership. Where one person might miss an unintended side effect, two people together are much less likely to make the same mistake. The C3

 Copyright © 1998 Kent Beck, All rights

project noticed that in the 5 months before they first went into production, the only code that introduced unintended side effects was code that was written by one person. Not that it wasn't possible for a pair to screw up. They just never did.

With all those changes taking place all over the system, you would also be sunk without Continuous Integration. If a pair spent a week making changes all over the system, and another spent the same week making their changes, there would be a host of collisions when the time came to integrate the changes. If instead the pair integrates at most a day's worth of changes or better a few hours worth of changes, then chance of collision becomes small enough that the problem becomes tractable again.

Continuous Integration

You can't integrate all the time. It takes too long.

There is certainly a base level of tool support you need to implement continuous integration. Your configuration management tool needs to be able to build a new system in a few minutes, and to be able to detect collisions between developers. Without this you will tend to build less and less often, and encounter more and more of the problems above that are addressed by integrating a few hours of development at a time.

However, once you have those tools, and you have the tests, integration isn't a chore, it is an important part of the psychological cycle of development. First you explore what a task means in terms of what behavior is expected and where in the system the changes need to be made. Then you make the changes and test them, perhaps refactoring related code along the way. Then you bring what you learned back to the released system. Once it your changes are integrated and the tests all run, you can walk away knowing that you are done with that task. You can move along to the next one with confidence.

Continuous integration certainly works better with Simple Design and with Refactoring, because there tends to be much less bulk of code, so there is less strain on the tools. Pair programming also helps continuous integration by ensuring that there are half as many streams of changes coming into the integration process. You don't have 10 people needed to release changes in a day, you have 5 pairs.

Conclusion

My intention in this chapter was to show the objections to how XP approaches major areas of software development, and to show how these objections are counteracted by the strengths of other practices. This is all in service of answering the quite reasonable question, „Yes, but how can this possibly work?“

Splitting Business and Technical Responsibility

There are two common failure modes in the relationship between Business and Development. If either Business or Development gains too much power, the project suffers.

Business

If Business has the power, they feel fit to dictate all four variables to Development. Here is what you will do. Here is when it will be done. No, you can't have any new workstations. And it better be of the highest quality or you're in trouble, buster.

In this scenario, Business always specifies too much. Some of the items on the list of requirements are absolutely essential. But some are not. And if Development doesn't have any power, they can't object, they can't force Business to choose which is which. So Development dutifully goes work, heads down, on the impossible task they have been given.

It seems to be in the nature of the less important requirements that they entail the greatest risk. They are typically the poorest understood, so there is great risk of the requirements changing all during development. Somehow, they also tend to be technically riskier.

The result of the „Business in Charge“ scenario, then, is that the project takes on too much effort and way, way too much risk for too little return.

 Copyright © 1998 Kent Beck, All rights

Development

Now, when Development gets their turn, you would think life would get better. But it doesn't. The net effect is exactly the same.

When Development is in charge, they put in place all the process and technology that they never had time for when „those suits" were pushing them around. They install new tools, new languages, new methodologies. And the tools, languages, and technologies are chosen because they are interesting and cutting edge. Cutting edge implies risk (if we haven't learned that by now, when will we?)

So, the net result of the „Development in Charge" scenario is too much effort and way, way too much risk for too little return.

What to Do?

The solution is to somehow split the responsibility and power between Business and Development. Business people should make the decisions for which they are suited. Developers should make the decisions for which they are suited. Each parties decisions should inform the other's. Neither party should be able to unilaterally decide anything.

Maintaining this kind of political balancing act might seem well-nigh impossible. If the world can't do it in the Balkans, what chance do you have? Well, if all you had was the vague goal of „balancing political power", you would have no chance. The first strong personality that came along would upset the balance. Fortunately, the goal can be far more concrete than that.

First, a story. If someone asks me whether I want the Ferrari or the mini-van, I am almost certain to choose the Ferrari. It will inevitably be more fun. However, as soon as someone says, „Do you want the Ferrari for 200.000 francs or the mini-van for 40.000?" I can begin to make an informed decision. Adding new requirements like „...and I need to be able to carry 5 kids" or „...and it has to go 200 KM in an hour" clear the picture further. There are cases where either decision makes sense, but you can't possibly make a good decision based only glossy photographs. You need to know what resources you have, what constraints you have, and how much each one costs.

Following this model, business people should choose:

- The scope or timing of releases
- The relative priorities of proposed features
- The exact scope of proposed features

The development organization must contribute to these decisions:

- Estimates of the time required to implement various features
- Estimates of the consequences of various technical alternatives
- Development also needs to decide on a development process that suits their personalities, their business environment, and their company culture. No single list of „here's how you write software" can possibly fit every situation. In fact, no single list can possibly fit any situation, since the situation is always in flux.
- Development has to choose what set of practices they will use to begin with, and the process by which they will review the effects of those practices and experiment with changes. This is rather like the US Constitution, which sets out a basic philosophy, a basic set of rules (the Bill of Rights, the first 13 amendments), and rules for changing the rules (adding new amendments).

Since business decisions take place all through the life of a project, giving business people responsibility for business decisions implies that a customer is as much a part of an XP team as a developer. In particular, for best results they should sit with the rest of the team and be available full or nearly-full time.

 Copyright © 1998 Kent Beck, All rights

Choice of Database

While the choice of a database might seem at first to be a purely technical decision, it is actually a business decision. The customer will have to live with the database vendor for many years, and has to be comfortable with the relationship at a business level even more than at a technical level.

This doesn't mean that the choice of database should occur in a technical vacuum. There are enormous technical consequences of the various database paradigms. Business cannot make a good decision about a database without understanding these consequences.

What if it's hard?

Most of the time, the decisions that come out of this process are surprisingly simple. The business people say, „I had no idea that was so expensive. Just do this one third of it. That will do fine for now."

Sometimes, though, it doesn't work out that way. Sometimes, the smallest, most valuable chunk of development is large and risky from the developers' perspective. When this happens, you can't blink. You will have to be careful. You can afford few mistakes. You may have to pull in more outside resources. But when the time comes to go over the top of the trench, then you really earn your money. You do everything you can to encourage smaller scope. You do everything you can think of to reduce risk. But then you just go for it.

Another way of saying this is that the split of power between business and development is not an excuse to avoid tough jobs. Quite the contrary. It is a way of sorting out those jobs that must be tough from those jobs that you just haven't figured out how to make simple yet. Most of the time the work will be simpler than you first imagined. When it isn't, you do it anyway, because that is what you get paid for.

Planning Strategy

Planning is the process of guessing what it will be like to develop a piece of software with a customer. Here are some of the purposes of planning:

- Bring the team together
- Decide on scope and priorities
- Estimate cost and schedule
- Give everyone involved confidence that the system can actually be done
- Provide a benchmark for feedback

Let's review the principles that affect planning. (Some of them are the general principles in the chapter „Principles". Others are specific to planning.):

- Do only the planning you need for the next horizon- At any given level of detail, only plan to the next horizon- the next release, the end of the next iteration. This doesn't mean that you can't do long range planning. You can, just not in great detail.
- Accepted responsibility- responsibility can only be accepted, not given. This means that there is no such thing as top down planning in XP. The manager can't go to the team and say, „Here's the pile of stuff we have to do and here's how long it will take." The project manager has to ask the team to take responsibility for doing the work. And then listen to the answer.
- The person responsible for implementing gets to estimate- If the team takes responsibility for getting something done, they get to say how long it will take. If an individual on the team takes responsibility for getting something done, they get to say how long it will take.
- Ignore dependencies between parts- plan as if the parts of development can be switched around at will. As long as you are careful to implement the highest business priorities first, this simple rule will tend not to get you into trouble. „How much for the coffee?" „The coffee is 25 cents, but refills are free." „Just give me a refill, then." This doesn't tend to happen.

 Copyright © 1998 Kent Beck, All rights

- Planning for priorities vs. planning for development- be aware of the purposes of planning. Planning so the customer can establish priorities needs much less detail to be valuable than planning for detailed implementation

Teamwork

Often, writing the plan is the first time the whole team as come together to work on a deliverable. The planning process sets up the power and communication relationships between developers, managers, and customers.

The communication part of planning proceeds by setting up a common vocabulary that everyone on the project can use and assume will be understood by everyone else. The key here is to have the simplest set of concepts that capture what everyone really has to know, and to give them names that work well together.

XP takes a strong position on the political power relationships in the team. One set of political relationships is the strict separation of technical and business decisions. Technical people make technical decisions and business people make business decisions. You may be wondering how, in a developer-customer relationship that probably already has a history of a certain split, you can possibly legislate who is to decide what. The answer lies in the rituals and artifacts used in XP, as outlined below. If everyone agrees to follow the rules, business people will tend to make business decisions and technical people will tend to make technical decisions.

XP can go wrong in this, as in any other facet, especially where it is crossing established habits and relationships. You will have to watch the planning process closely to maintain the separation of business and technical decisions and to maintain the balance of power among the developers.

The second set of political relationships is between the developers. XP argues against a stratification of developers into „architects“ and „programmers“. The best of all possible worlds is if every developer is making good decisions at all scales. The system metaphor, pair programming, relentless refactoring, all tend to give every developer the opportunity to make decisions at all scales.

Again, this is the a process that needs to be nurtured. The developers on the team will naturally grow leaders, but the leaders must be sensitive to the effects of their position, and use it to encourage good decision making on the part of everyone else, instead of encouraging dependence.

Scope and Priorities

The second purpose of planning is to set the scope and priorities of the development. Really the question isn't „scope and priorities of THE development“, but „scope and priorities for the first release“. Systems live a long time. Scope and priorities change. To deal rationally with this situation, you have to have a horizon past which you choose not to look.

So, the first question is „what all should the system do?“ In exploring this, the team should be as free and creative and wild as possible, since it is often the problems that we aren't yet looking straight at that provide the greatest value. XP uses storytelling as a metaphor and technique for eliciting possible problems for the system to solve.

„We paid half of the plant bonuses one day early and the whole plant went on strike for two weeks and cost the company $10 million.“ There is a clear lesson here for the developer who is listening. This story translates directly or indirectly into a handful of detailed requirements that clearly have a serious business impact.

One of the tough balancing acts in planning is giving customers free rein to tell you whatever they feel like telling you versus making sure you don't leave out anything important. What I like about storytelling as a metaphor for setting priorities is that it makes sure you will hear everything important. It is precisely the incidents that stick in someone's mind that reveal the most important problems to solve.

This is not to say that developers or customers will feel satisfied with this process. I have had customers who swore up and down that they must have forgotten something terribly important, even after systems had gone into production. This is a little like the classic question of how do you make sure you've thought of everything you haven't thought of. The answer is that you can't. Do the best job you can, then have faith that if something comes up later, you can deal with it. You'll make a lot more progress that way than if you fret and stress about every little detail.

The storytelling metaphor is carefully chosen because it preserves the early balance of political power in the hands of the customer. Diving into a formalized, detailed notation early in the process puts all the power in the hands of those who know how to manipulate the notation, typically programmers. But for development to

 Copyright © 1998 Kent Beck, All rights

succeed in its business goals, the customer must always have the feeling that they are driving the process, not that they are being driven by the process.

So, you listen to all these stories. Then what? Then you distill them onto Story Cards. On the Chrysler payroll project we had 153 story cards going into our initial planning session. *Example*

You should have a hundred-ish stories going into the scheduling process. A thousand cards are physically too hard to handle. Ten cards don't give enough scope for making tradeoffs.

This may seem to be a ridiculously slim amount of information from which to commit to a schedule. In some perfect world, you would like to know exactly what you are getting into before you start. However, we are trying to simultaneously solve a bunch of problems:

- Keep the customer involved in the process- adding information, making tradeoffs
- Keep the process as short as possible
- Keep the weight of our artifacts from hindering change later
- Make good technical decisions (primarily estimation and infrastructure at this point)

A pile of a hundred cards, each with something the system has to do, balances these problems. They are simple and unimposing, so the customer is comfortable writing on the and waving them around. The amount of information that can be written on them is limited, so they tend to be done quickly. And if they are quickly reviewed by developers, they will rapidly converge to just enough information from which to estimate effort.

Estimation and Commitment

The next purpose of planning is to decide what is going to be done by when. The XP process for this is to assign costs to each story, which will be used by the customer to decide what will go into the first release.

Here we see another application of the principle of rapid and concrete feedback. The question to be answered by the customer is „What is the smallest system that makes business sense?“ XP puts that system into production, and then everybody can learn from their experience with it. The only way you would ever have a three year XP project is to have six six-month projects back to back. Anything more generates too much risk and not enough learning.

While the customers are writing the stories, the developers are actively experimenting with the technology and the architecture. Developers try coding with various metaphors to see which will support the concepts needed. They try out the various pieces of new technology they will be using to decide what parts are safe to use and what parts are not. They try to translate the stories into test cases and then code. Using their experiences, they help the customers refine the story-writing process. Finally, they practice all the rituals that they will use throughout development- estimation and feedback, pair programming, the testing strategies, refactoring.

The purpose of the developer exploration is to become comfortable with each other, number one. They also have to gain confidence in what are likely to be new development practices, so that under stress those practices will still be followed. The dialog with the customers over the preferred content of the story cards is to begin developing a relationship of trust which again will sustain the project when the inevitable stresses begin to build (the principle of quick and concrete feedback, again).

The conclusion of this part of the process is a meeting in which the customers and developers have a dialog to set priorities based on the estimates of the developers.

First, the customers sort the stories into three piles- „must have or the system doesn't make sense“, „provides significant business payoff“, and „nice to have“. While this sorting is going on, the customers can discuss the meaning and relative value of the stories and the developers can ask questions about the meaning and relationship of the stories. You write the final customer priority on each card.

The purpose of this is to get the customers thinking about priorities, that some things are more important than others. Often, customers are so used to developers (seemingly arbitrarily) discarding requirements that their initial response is to say that everything is equally important, that everything has to be done, and right away at that. This is never true. There are always some stories that are more important, and some that can safely be deferred. By reducing the size of the first deliverable, everybody gets feedback sooner, reducing the risk and increasing the value of the development.

 Copyright © 1998 Kent Beck, All rights

Next, the developers sort the cards by risk. Again, there are three piles- „know how to implement“, „know pretty much how to implement“, „no idea“. Developers can discuss the stories among themselves, sharing their experiences while prototyping.

This exercise gets the developers focused on just how much they know as a team. Even if everyone isn’t interested or experienced with everything that has to be done, all the developers can see in the composition of the piles just how much they know about the system to be built. With the customers there, they can also see how much of the development will simply be covering known territory.

If you get into this meeting and all of the cards end up as high-priority and high-risk, you’re screwed. But if you’ve run the exploration phase with clear communication and lots of feedback there is little chance of this happening. A week before the meeting, if you can see that it might happen, run through parts of the process privately, like just sorting by risk. If the high-risk pile comes out huge, spend the next week doing whatever you can to make it smaller.

Now you are ready for the estimation of the whole system. The result will be that there will be far too much to do in the time available. There always is. Don’t worry about this.

Here’s how it works. The developers look at the cards and estimate how long each one would take to implement in ideal engineering weeks. That is, with no interruptions, all the prerequisites done and all the information you needed about the details of the story, how many weeks would it take for one person to implement the story? This should spark discussion about the best way to implement certain features, what the story really means, how big the story really should be, whether it can be split or joined with other stories. Take the estimates of all the stories, add them up, multiply by a load factor (the relationship between estimated time and calendar time, measured during exploration above), and divide by the number of developers. That gives you a number of calendar weeks. Write the finish date on the board. (Remember that some amount of productization will be required over and above the engineering before the system is ready for production).

The customer will go nuts. It can’t possibly take this long, you guys are idiots, I’ve already promised it in a third that time. Now the dance begins. Remember the dance between the desired and the possible that I described in chapter 2? Here it is, made manifest. The customer can’t change the estimates, of course, since no one can estimate for anyone else. However, they can change the stories. They can choose not to include certain stories in the next release. They can split a story, have the developer re-estimate the pieces, and only keep part of the original story in the next release.

It might take minutes, it might take hours, but eventually the customer will come to a set of stories they can just barely live with and a date they can just barely live with. It will take courage not to change estimates just because the customer doesn’t like them. Don’t. If they can explain why the story is much smaller than the developers originally thought, fine. Then change the estimate.

This is a crucial moment in the project. It is the first time that the customers and the developers are working toward the same concrete goal- a date and scope everyone can live with. It also gets everyone thinking simple- what is a simpler implementation that could get this done faster? What is a simpler story that the customer could still live with?

Making up the commitment schedule

The Planning Game

Maybe this goes in the how to book?

Here’s another look at the same process. I purposely abstracted the people involved to two participants- Business and Development. This can help to remove some of the unhelpful emotional heat from the discussion of plans. Instead of, „Joe, you bastard, you promised me this by Friday,“ the game style of presentation says, „Development learned something. It needs help from Business in responding in the best way.“

Business doesn’t like Development. Relations between the people who need systems and the people who build systems is so strained, they often resemble the relations between centuries-old enemies in some backward part of the world. Mistrust, accusations, and subtle and indirect maneuvering all abound. You can’t develop decent software in this kind of environment.

The best environment is one of mutual trust. Each party respects the other. Each party believes that the other has their best interest at heart, and the interests of the larger community. Each party is willing to let the other do their job, joining the skills, experience and perspective of both.

 Copyright © 1998 Kent Beck, All rights

You can’t legislate this kind of relationship. You can’t just say, „We know we’ve screwed up. We’re terribly sorry. It won’t happen again. Let’s work completely differently, starting right after lunch.“ The world, and people, just don’t work that way. Under stress, people revert to earlier behavior, no matter how badly that behavior has worked in the past.

What is needed on the way to a mature, mutually respectful relationship is a set of rules to govern how the relationship is conducted- who gets to make which decisions, when the decisions will be made, how those decisions will be recorded.

Never forget, however, that the rules of the game are an aid, a step towards the relationship you really want with your customers. The rules can never capture the subtlety, flexibility, and passion of real human relations. Without some set of rules, however, you can’t begin to improve the situation. Once the rules are in place and your relationship is improving, then you can begin to modify the rules to make development go smoother, and eventually abandon the rules altogether as an explicit thing.

First, however, you have to play by the rules. Here they are:

The Goal

The goal of the game is to maximize the value of software produced by the team. From the value of the software, you have to deduct the cost of its development, and the risk incurred during development.

The Strategy

The strategy for the team is to invest as little as possible to put the most valuable functionality into production as quickly as possible, but only in conjunction with the programming and design strategies designed to reduce risk. In the light of the technology and business lessons of this first system, the team quickly puts the new most-valuable functionality into production. And so on.

The Pieces

The pieces in the planning game are the Story Cards.

The Players

The players in the planning game are Development and Business.

The Moves

Exploration Phase

The purpose of the gathering phase is to give both players an appreciation for what all the system should do eventually.

Write a Story- Business writes a story describing something the system needs to do. The stories are written on index cards, with a name and a short paragraph describing the purpose of the story.

Estimate a Story- Development takes a story and estimates, in Ideal Engineering Time, how long the story will take to implement. If Development can’t implement the story, it can ask Business for clarification, or to split the story.

Split a Story- If Development can’t estimate a whole story, or if Business realizes that part of a story is more important than the rest, Business can split a story into two or more stories.

Commitment Phase

The purpose of the commitment phase is for Business to choose the scope and date of the release, and for Development to confidently commit to delivering it.

 Copyright © 1998 Kent Beck, All rights

Sort by Value- Business sorts the stories into three piles- those without which the system will not function, those that are less essential but provide significant business value, and those that would be nice to have.

Sort by Risk- Development sorts the stories into three piles- those that they can estimate precisely, those that they can estimate reasonably well, and those that they cannot estimate at all.

Set Velocity- Development tells Business how fast it can development, in Ideal Engineering Weeks per calendar month, for example.

Choose Scope- Business chooses the set of cards in the release, either by setting a date for engineering to be complete and choosing cards based on their estimate and the project velocity, or by choosing the cards and calculating the date.

Create the Plan- Development divides the stories in the release into piles, with 2-4 weeks worth of stories in each pile. The system must run, completely but embryonically, at the end of the first pile. Valuable and risky stories should occur early in development. No one pile should contain too much risk.

Steering Phase

The purpose of the maintenance phase is to update the plan based on what is learned by Development and Business.

Recovery- If Development realizes that it has overestimated its velocity, it can ask Business to Choose Scope and the Create the Plan again.

New Story- If Business realizes it needs a new story, it can write the story. Development estimates the story. Then Business removes stories with the equivalent estimate from the remaining plan and inserts the new story.

Re-estimate- If Development feels that the plan no longer provides an accurate map of development, it can re-estimate all of the remaining stories and Set Velocity again.

Facilities Strategy

Insert a picture of the C3 space, and maybe a diagram of an open cubicle layout.

If you don’t have a reasonable place to work, your project won’t be successful. It wasn’t until Peopleware that I saw a software project management book spend more than a sentence or two on the physical environment. But the difference between a good space for the team and a bad space for the team is immediate and dramatic.

It was a watershed in my development as a consultant the time I was asked to review the object oriented design blah blah blah for a project. I looked at the system and sure enough it was a mess. Then I noticed where everyone was sitting. The team was four senior developers. They all had corner offices on four corners of a medium sized building. I told them they should move the offices together. Wow! I was brought in because of my knowledge of Smalltalk and objects, and the most valuable suggestion I had was that they should rearrange the furniture.

Facilities is a tough job in any case. The facilities planner is judged on how little money they spend, and how much flexibility they retain. The person using the facilities wants to work closely with the rest of the team. At the same time, though, they need privacy from which to phone for doctor’s appointments.

XP wants to err on the side of too much public space. XP is a communal software development discipline. The team members need to be able to see each other, to hear shouted one-off questions, to „accidentally“ hear conversations to which they have vital contributions.

XP can strain facilities. Common office layouts don’t work well for XP. Putting your computer in a corner, for example, doesn’t work, because it is impossible for two people to sit side-by-side and program. Ordinary cubicle wall heights don’t work well- walls between cubicles should be half height or eliminated entirely. At the same time, the team should be separated from other teams.

Probably the best setup is an open bullpen, with little cubbies around the outside of the space. The team members can keep their personal items in these cubbies, go to them to make phone calls, and spend time at them when they don’t want to be interrupted. The rest of the team needs to respect the „virtual“ privacy of someone sitting in their cubby. Put the biggest, fastest development machines on tables in the middle of the space (cubbies might or might not contain machines). This way, if someone wants to program, they will naturally be drawn to

 Copyright © 1998 Kent Beck, All rights

the open, public space. From here everyone can see what is happening, pairs can form easily, and each pair can draw from the energy of the other pairs who are also developing at the same time.

If you can, reserve a little of the nicest space off to one side for a communal space. Put in an espresso maker, couches, some toys, something to draw people there. Often, the most effective way to get unstuck while you are developing is to step away for a moment. If there is a pleasant space to step away to, you are more likely to do it when you need it.

The aggressiveness value finds its expression in the XP attitude towards facilities. If the corporate attitude towards facilities is at odds with the team's attitude, then the team wins. If the computers are in the wrong place, they are moved. If the partitions are in the way, they are taken down. If the lights are too bright, they are taken out. If the phones are too loud, one day, mysteriously, they are all found to have cotton stuffed in the bells.

I showed up at a bank on the first day and found that we had been given ugly old one-person desks to work at. The desks had two sets of metal drawers on either side of a little slot into which you could just slide your legs. This was clearly not going to work. We looked around until we found an industrial strength screwdriver, then we took off one set of drawers. Now two people could sit side-by-side at each desk.

All this screwing around with the furniture can get you in trouble. The facilities management people can get downright angry to find that someone has been moving desks around without their permission or involvement (never mind that a request for a change can take weeks or months to fulfil). I say, „Too bad." I have software to write, and if getting rid of a partition helps me write that software better, I'm going to do it. If the organization can't stand that much initiative, then I don't want to work there, anyway.

Taking control of your physical environment sends a powerful message to the team. They are not going to let irrational opposing interests in the organization get in the way of success. Taking control of their physical environment is the first step towards taking control of how they work overall.

Facilities are worthy of constant experimentation (the feedback value at work). After all, the organization spent a gazillion dollars for all that flexible office furniture. All that money would be wasted if you didn't flex the furniture a little. What if these two people's cubbies were closer? Further? What if the integration machine was in the middle? In the corner? Try it. Whatever works, stays. Whatever doesn't is sacrificed to the next experiment.

Development Strategy

XP uses the metaphor of programming for its activities- that is, everything that you looks in some way like programming. It is in development strategy that the metaphor is most clear- programming is like programming. However, like all the rest of XP, XP development is deceptively simple. All the pieces are simple enough to explain, but executing them is hard. Fear intrudes. Old habits return under pressure.

The development strategy begins with Iteration Planning, a process much like the Planning Game, but in miniature. Continuous integration reduces development conflicts and creates a natural end to a development episode. Collective code ownership encourages the whole team to make the whole system better. Finally, pair programming ties the whole process together. First, here is an idealized development episode combining all these practices.

A Development Episode

I look at my stack of task cards. The top one says „Export Quarter-to-date Withholding". At this morning's stand up meeting, I remember you said you had finished the quarter-to-date calculation. I ask if you have time to help with the export. „Sure," you say.

We spend a couple of minutes discussing the work you did yesterday. You talk about the bins you added, what the tests are like, maybe a little about how you noticed yesterday that pair programming worked better when you moved the monitor back a foot.

You ask, „What are the test cases for this task?" I say, „When we run the export station, the values in the export record should match the values in the bins." „Which fields have to be populated?" you ask. „I don't know. Let's ask Eddie." We interrupt Eddie for 30 seconds. He explains the five fields that he knows about that are related to quarter-to-date.

 Copyright © 1998 Kent Beck, All rights

We go look at the structure of some of the existing export test cases. We find one that is almost what we need. By abstracting a superclass we can implement our test case easily. We do the refactoring. We run the existing tests. They all run.

We notice that several other export test cases could take advantage of the superclass we just created. We want to see some results on the task, so we just write down „Retrofit AbstractExportTest“ on our to-do card.

Now we write the test case. Since we just made the test case superclass, writing the new test case is easy. We are done in a few minutes. About half way through, I say, „I can even see how we will implement this. We can...“ „Let’s get the test case finished first,“ you interrupt. While we’re writing the test case, ideas for three variations come to mind. You write them on the to-do card.

We finish the test case and run it. It fails. Naturally. We haven’t implemented anything yet. „Wait a minute,“ you say, „yesterday, Ralph and I were working on a Calculator in the morning. We write five test cases that we thought would break. All but one of them ran first thing.“

We bring up a debugger on the test case. We look at the objects we have to compute with. I write the code. We notice a couple more test cases we should write. We put them on the to-do card. The test case runs.

We go to the next test case, and the next. I implement them. You notice that the code could be made simpler. You try to explain to me how to simplify. After a couple of attempts, I push the keyboard over to you. You refactor the code. You run the test cases. They pass. You implement the next couple of test cases.

After a while we look at the to-do card and the only item on it is restructuring the other test cases. Things have gone smoothly, so we go ahead and restructure them, making sure they run when we finish.

Now the card is empty. We notice that the integration machine is free. We load the latest release. Then we load our changes. Then we run all the test cases. One fails. „That’s strange. It’s been almost a month since I’ve had a test case break during integration,“ you say. No problem. We debug the test case. Then we run the whole suite again. This time it passes. We release our code.

Whew! That’s the whole development cycle. I’ll highlight below some things you might want to pay attention to about the cycle. For now, notice that:

- Development occurs in pairs.
- Development isn’t restricted to one small area, but instead adds value wherever possible.
- Development is driven by tests. Until the tests run, you aren’t done. When the tests run, you are done.
- Integration immediately follows development, including integration testing.

Iteration planning

The Planning Game gives the customer quick feedback about the progress of the project toward its next set of goals. *This doesn’t make sense.* Because the iterations are so short, however, there is not a lot of time to recover from screwups. To that end, there needs to be intra-iteration planning.

The constraints on planning for three weeks of development are much the same as those on planning for four or six months of development. You don’t want to spend too much time planning, since reality will never consent to match the plan exactly. You want rapid feedback on how you are doing, so a third of the way through you will know whether you are in trouble. You want the individuals responsible for delivering something to also be responsible for estimating it. You want pressure to limit the scope of development to what is really needed. You want a process that doesn’t generate so much pressure that people do things that turn out to be stupid, just to meet the needs of a short term plan.

The practice is called the Iteration Planning Game. It is similar to the Planning Game in that cards are used as the pieces. This time, though, the cards are Task Cards instead of Story Cards. The players are all the individual developers, since you can count on less friction between members of the team than you are likely to get between developers and customers. The time scale is shorter- the whole game should take less than half of the first Monday of an iteration. The phases and moves are similar-

 Copyright © 1998 Kent Beck, All rights

Exploration

Write a Task- Take the stories for the iteration and turn them into tasks. Generally the tasks are smaller than the whole story, because you can't implement a whole story in a couple of days. Sometimes one task will support several stories. Sometimes a task won't directly relate to any particular story- like migrating to a new version of system software.

Split a Task/Combine Tasks- If you can't estimate a task at a few days, break it down into smaller tasks. If several tasks each take an hour, combine them to form a larger task.

Commitment

Accept a Task- A developers accepts responsibility for a task.

Estimate a Task- The responsible developer estimates the number of ideal engineering days to implement each task. Often this is conditional on getting help from another developer who may be more familiar with code to be modified. Tasks that take more than ???N days must be split.

Set Load Factors- Each developer chooses their load factor for the iteration- the percentage of time they will spend actually developing.

Balancing- Each developer adds up their task estimates and multiplies by their load factor. Developers who turn out to be overcommitted must give up some tasks. If the whole team is overcommitted, they must find a way to get back on balance- see Recovery, below.

Steering

Implement a Task- A developer takes a task card, finds a partner, writes the test cases for the task, makes them all work, then integrates and releases the new code when the universal test suite runs.

Record Progress- Every two or three days the tracker asks each developer how long they have spent on each of their tasks and how many days they have left.

Recovery- Developers who turn out to be overcommitted ask for help- reducing the scope of some tasks, asking the customer to reduce the scope of some stories, shedding non-essential tasks, getting more or better help, and as a last resort asking the customer to defer some stories to a later iteration.

Verify Story- As soon as the functional tests are ready and the tasks for a story are complete, the functional tests are run to verify that the story works. Interesting cases brought to life during implementation can be added to the functional test suite.

The differences between iteration planning and release planning are primarily that you can tolerate more wiggling in the iteration schedule than in the commitment schedule. This is because the developers can always create a relationship of deeper trust with each other than they can with the customers. For example, if one of three weeks have passed in an iteration and progress has been too slow, it is entirely possible to stop for a day for a major collaborative refactoring that is needed for everyone's progress. No developer is going to get the feeling that the whole project is falling apart at that point (not after a little experience). If the customer was to see such seemingly drastic changes on a daily basis, however, they would quickly become nervous.

In a way this seems like lying- you are concealing some of the process of development from the customer. It is important that this doesn't become true. You don't deliberately conceal anything. If the customer wants to sit through a whole day of refactoring, well, they probably have more valuable things to do, but they are certainly welcome. I like to think of it as an extension to the principle of splitting business and technical decisions. There are changes at a certain level of detail that are no longer the province of Business- developers know how to micro-manage their time better than any business person could.

One difference between the Planning Game and the Iteration Planning Game is that developers sign up for the tasks before they estimate. The team implicitly takes collective responsibility for the stories, so the estimates should be made by the team collectively. Individual developers accept responsibilities for tasks, so they must estimate the tasks themselves.

Another difference with iteration planning is that some of the tasks are not directly related to the needs of the customer. If someone needs to strengthen the tools for integration, and it is enough work that it can't easily be

Copyright © 1998 Kent Beck, All rights

hidden in the cracks of development, then it becomes a task of its own, scheduled and prioritized with all the other tasks.

Let's look back at the constraints on the iteration planning process and how the strategy above meets them:

- You don't want to spend too much time planning, since reality will never consent to match the plan exactly. Half a day out of 15 is not so much overhead. Of course, if you could reduce that time it would be better, but it is not so much time.
- You want rapid feedback on how you are doing, so a third of the way through you will know whether you are in trouble. The questions the Tracker asks give you a fair idea half way through the iteration if you are on track or not. That is enough time to react locally to problems, without asking the customer to make changes.
- You want the individuals responsible for delivering something to also be responsible for estimating it. As long as the developers sign up for tasks before they estimate, this works.
- You want pressure to limit the scope of development to what is really needed. It always feels strange to say that you can only do 7.5 days of work in three weeks. However, as your estimates get better and better, you will find that it is absolutely true. That feeling that you aren't really working so terribly hard makes you want to do more tasks. But you know you want to maintain standards and quality as you do so (and you have a partner looking at the same screen to make sure you do maintain quality). So you have a tendency to do as little as you can and still honestly be able to say you've completed the task.
- You want a process that doesn't generate so much pressure that people do things that turn out to be stupid, just to meet the needs of a short term plan. Again, this goes back to saying that you can do 7.5 days of work. You just can't take on too many tasks. If you do one iteration, there will be ample, public feedback that you shouldn't have tried to do so much. So you won't do it again. This results in your committing to do about as much as you can actually do, and do it with quality.

On smaller projects, I have eliminated iteration planning. It is certainly necessary for coordinating the work of 10 developers. It is certainly not necessary for coordinating the work of 2. Somewhere in there you will find that the need to coordinate makes the extra effort involved in formal iteration planning worth it.

Continuous integration

Another part of the development strategy is continuous integration. No code sits un-integrated for more than a couple of hours. At the end of a development episode, the code is integrated with the latest release and all the tests must run at 100%.

At the limit of continuous integration, every time you changed a method, the change would instantly be reflected in everyone else's code. Aside from the infrastructure and bandwidth required to support such a style, this wouldn't actually work well. While you are developing you want to pretend that you are the only developer on the project. You want to march ahead at full speed, ignoring the relationship of the changes you make to the changes anyone else happens to be making. Having changes happen out of your immediate control would shatter this illusion.

Integrating after a few hours (certainly no more than a day) gives many of the benefits of both style- single developer and instantaneous integration. While you are developing, you can act like you and your partner are the only pair on the project. You can make changes wherever you want to. Then you switch hats. As integrators, you become aware (the tools tell you) where there are collisions in the definition of classes or methods. By running the tests you become aware of semantic collisions.

If integration took a couple of hours, it would not be possible to work in this style. It is important to have tools that support a fast integration/build/test cycle. You also need a reasonably complete test suite that runs in a few minutes. And the effort of fixing collisions can't be too great.

This turns out not to be a problem. Constant refactoring has the effect of breaking the system into lots of little objects and lots of little methods. This lowers the chance that two pairs of developers will change the same class or method at the same time. If they do, the effort required to reconcile the changes is small, because each only represents a few hours of development.

Another important reason to accept the costs of continuous integration is that it dramatically reduces the risk of the project. If two people have different ideas about the appearance or operation of a piece of code, you will know in hours. You will never spend days chasing a bug that was created some time in the last few weeks. And

 Copyright © 1998 Kent Beck, All rights

all that practice at integration comes in very handy when it comes to creating the final project. The „production build“ is no big deal. Everyone on the team could do it in their sleep by the time it comes around, because they have been doing it every day for months.

Continuous integration also provides a valuable human benefit during development. In the middle of working on a task you have a hundred things on your mind. By working until there is a natural break- there are no more small items on the to-do card- and then integrating, you provide a rhythm to development. Learn/test/code/release. It’s almost like breathing. You form an idea, you express it, you add it to the system. Now your mind is clear, ready for the next idea.

From time to time, continuous integration forces you to split the implementation of a task into two episodes. We accept the overhead this causes, the need to remember what was done already and what remains to be done. In the interim, you may have an insight into what caused the first episode to go so slowly. You start the next episode with some refactoring, and the rest of the second episode goes much more smoothly.

Collective code ownership

Collective code ownership is this seemingly crazy idea that anyone can change any piece of code in the system at any time. Without the tests, you’d simply be dead trying to do this. With the tests, and the kind of quality of tests you get after a few months of writing lots of tests, you can get away with this. Well, you can if you only ever integrate a few hours worth of changes at a time. Which, of course, you only ever do.

One of the effects of collective code ownership is that complex code does not tend to live very long. Because everyone is used to looking all over the system, such code will be found sooner rather than later. And when it is found, someone will try to simplify it. If the simplification doesn‘t work, as evidenced by the tests failing, then the code will be thrown away. Even if this happens, there will be someone other than the original pair who understands why the code might have to be complex. More often than not, however, the simplification works, or at least part of it works.

Collective code ownership tends to prevent complex code from entering the system in the first place. If you know that someone else will soon (a few hours) be looking at what you are writing at the moment, you will think twice before putting in a complexity you can’t immediately support.

Collective code ownership increases you feeling of personal power on a project. On an XP project you are never stuck with someone else’s stupidity. You see something in the way, you get it out of the way. If you choose to live with something for the moment because it is expedient, that is your business. But you’re never stuck. So you never get the sense that, „I could get my work done, if only I didn’t have to deal with these other idiots.“ One less frustration. One step closer to clear thinking.

Collective code ownership also tends to spread knowledge of the system around the team. It is unlikely that there will ever be a part of the system that only two people know (it has to be at least a pair, which is already better than the usual situation where one smart programmer holds everyone else hostage). This further reduces project risk.

Pair Programming

Pair programming really deserves its own book. It is a subtle skill, one that you can spend the rest of your life getting good at. For the purposes of this chapter, we’ll just look at why it works for XP.

First, a couple of words about what pair programming isn’t. It isn’t one person programming while another person watches. Just watching someone program is about as interesting as watching grass die in a desert, except the outcome is not in doubt in the desert. Pair programming is a dialog between two people trying to simultaneously program and understand together how to program better. It is a conversation, assisted by and focused on a computer.

Pair programming is also not a tutoring session. Sometimes pairs contain one partner with much more experience than the other partner. When this is true, the first few sessions will look a lot like tutoring. The junior partner will ask lots of questions, and type very little. Very quickly, though, the junior partner will start catching stupid little mistakes, like unbalanced parentheses. The senior partner will notice the help. After a few more weeks, the junior partner will start picking up the larger patterns that the senior partner is using, and notice deviations from those patterns.

 Copyright © 1998 Kent Beck, All rights

In a couple of months, typically, the gap between the partners is not nearly so noticeable as it was at first. The junior partner is typing more regularly. The pair notices that each of them has strengths and weaknesses. Productivity, quality, and satisfaction rise. Well, anyway, that's how it's supposed to go.

Pair programming is not about being joined at the hip. If you look at the ideal development episode earlier in the chapter, you will notice that the first thing I did was ask for help. Sometimes you are looking for a particular partner when you start a task. More commonly, though, you just find someone who is available. And, if you both have tasks to do, you agree to work the morning on one task and the afternoon on the other.

What if two people just don't get along? They don't have to pair. It may make scheduling the rest of the pairings more awkward, but if the interpersonal problem is bad enough, that will be better. What if someone just refuses to pair? Well, either they are working on isolated bits of the system that don't really impact anyone else, in which case it might be okay for them not to pair, or they are working right in the middle of the system, in which case the team should be aware of the effect of the lack of pairing. If the loner's code causes more problems than everyone else's, they can choose to learn to work like the rest of the team, or they can choose to look for work outside of the team. XP is not for everybody, and not everybody is cut out for XP.

So, why does pair programming work for XP? Well, the first value is communication, and there are few forms of communication more intense than face-to-face. So, pair programming works for XP because it encourages communication. I like to think of the analogy of a pool of water. When an important new bit of information is learned by someone on the team, it is like putting a drop of dye in the water. Because of the pairs switching around all the time, the information rapidly diffuses throughout the team just as the dye spreads throughout the pool. Unlike the dye, however, the information becomes richer and more intense as it spreads and is enriched by the experience and insight of everyone on the team.

Pair programming also works because it is more productive to work two people to a machine than two people at two machines. The pair together will get more functionality done together than they would have separately. I can't claim to have scientific evidence of this, but I have seen it and experienced it often enough to believe it.

Even if you weren't more productive, you would still want to pair, because the resulting code is so much higher quality. While one partner is busy typing, the other partner is thinking at a more strategic level- where is this line of development going? Will it run into a dead end? Is there a better overall strategy? Is there an opportunity to refactor?

Another powerful feature of pair programming is that some of the practices just wouldn't work without it. Under stress people revert. They will skip writing tests. They will put off refactoring. They will avoid integrating. With your partner watching, though, chances are that even if you feel like blowing off one of these practices, your partner won't. This is not to say that pairs don't ever make process mistakes. They certainly do, or you wouldn't need the coach. But the chances of ignoring your commitment to the rest of the team is much smaller in pairs than it is when you are working alone.

The conversational nature of pair programming also enhances the software development process. You quickly learn to talk at many different levels- this code here, code like this elsewhere in the system, development episodes like this in the past, systems like this from years past, the practices you are using and how they can be made better.

Design Strategy

In many way, this is the most difficult chapter to write. The stage is set- we have the values, the principles, the activities. All that's left is to apply what we know to the problem of designing.

What if the result sounds crazy? What if the advice we come up with is exactly the opposite of what you find in nearly any other software engineering book, paper, lecture, pamphlet, or video?

I suppose if the advice of XP were exactly like the advice of every other methodology, it would be of limited value. I mean, everybody else could be saying „design for the future, prepare for reuse, look ahead as far as possible" because it is good advice. On the other hand, the projects I visit that try to follow this advice generally aren't doing very well. Some undoubtedly do, but the ones I see generally don't. So the other option is that everybody else is wrong, or rather, in the context of the rest of XP, XP may have a better answer.

I won't keep you in suspense- the design strategy in XP is always to have the simplest design that runs the current test suite.

Now, that didn't hurt so bad. What's wrong with simplicity? What's wrong with test suites?

 Copyright © 1998 Kent Beck, All rights

The Simplest Thing that Could Possibly Work

Let's take a step back and come on this answer a little at a time. All four values play into this strategy:

- Communication- A complicated design is harder to communicate than a simple one. We should therefore create a design strategy that comes up with the simplest possible design, consistent with the rest of our goals. On the other hand, we should create a design strategy that comes up with communicative designs, where the elements of the design communicate important aspects of the system to a reader.
- Simplicity- We should have a design strategy that will produce simple designs, but the strategy itself should be simple. This doesn't mean that it needs to be easy to execute. Good design is never easy. But the expression of the strategy should be simple.
- Feedback- One of the problems I always had designing was that I didn't know when I was right or wrong. The longer I went on designing, the worse this problem became. A simple design solves this by being done so quickly. The next thing to do is code it and see how the code appears.
- Aggressiveness- What could be more aggressive than stopping after a little bit of design, confident that when the time comes, you can add more, when and as needed?

So, following the values we have to create a design strategy which results in a design that is simple. We should quickly find a way to verify its quality and feed back what we learn into the design. And we should crank the cycle time for this whole process down as short as possible.

The principles also work into the design strategy. Here are some:

- Small initial investment- We should make the smallest possible investment in the design before getting payback for it.
- Assume simplicity- We should assume that the simplest design we can imagine possibly working actually will work. This will give us time to do a thorough job when the simplest design doesn't work. In the meantime, we won't have to carry along the cost of extra complexity.
- Working with people's instincts, not against them- I'm afraid we'll have to fall down a little on this principle. As developers, we get in the habit of anticipating problems. When they appear later, we're happy. When they don't appear (much more often, in my experience), we don't notice. So the design strategy will have to go sideways of this behavior. Fortunately, most folks can unlearn the habit of borrowing trouble, as my grandmother called it. Unfortunately, the smarter you are, the harder it will be to unlearn.
- Incremental change- The design strategy will work by gradual change. We will design a little at a time. There will never be a time when the system „is designed". It will always be subject to change, although there will be parts of the system that remain quiet for a while.
- Travel light- The design strategy should produce no „extra" design. There should be enough to suit our current purposes (the need to do Quality Work), but no more. If we Embrace Change, we will be willing to start simple and continually refine.

This leads us to the following design strategy-

1. Start with a test, so we will know when we are done.
2. Design just enough to get that test running.
3. Repeat
4. If you ever see the chance to make the design simpler, do it. See the section „Simplest?" for a definition of the principles that drive this.

This strategy may look ridiculously simple. It is simple. It is not ridiculous. It is capable of creating large, sophisticated systems. However, it is not easy.

When learning to execute the design strategy, developers typically go through several phases:

Rejection- The strategy can't possibly work, and they go on as before. Pair programming and coaching provides pressure to get out of this rut. Often what turns the tide is an experience of guessing at a complicated design,

 Copyright © 1998 Kent Beck, All rights

even being sure it is needed, doing a simple one instead, and then seeing that the feared complexities never appear.

Confusion- If the way I used to design doesn't work any more, what does? Developers typically thrash around for a new style, based on simplicity.

Acceptance- After a while, developers find their new way. Their designs become simpler and simpler over time. The more experienced they are, the longer the first two stages last, in general, but the better the results at the end. Take that last one again- once working in this style, the more experienced a developer is, the simpler the designs they create.

How does it work?

It's in execution that the strategy will feel strange. We'll pick up the first test case. We'll say, „If all we had to do was implement this test case, then we would only need one object with two methods." We'll implement the object and the two methods. And we'll be done. Our whole design is one object. For about a minute.

Then we'll pick up the next test case. Well, we could just hack in a solution, or we could restructure the existing one object into two objects. Then implementing the test case would involve replacing one of the objects. So, we restructure first, run the first test case to make sure it works, then implement the next test case.

After a day or two of this, the system is big enough that we can imagine two teams working on it without worrying stepping on each other all the time. So we get two pairs implementing test cases at the same time and periodically (a few hours at a time) integrating their changes. Another day or two and the system can support the whole team developing in this style.

From time to time, the team will get the feeling that the crud has been creeping up behind them. Perhaps they have measured a consistent deviation from their estimates. Or maybe their stomachs knot up when they know they have to change a certain part of the system. In any case, somebody calls, „Time out." The team gets together for a day and restructures the system as a whole.

And that's it- that's how you design extreme. In the XP view, design is not drawing a bunch of pictures and then implementing the system to conform to the pictures. That would be pointing the car. The steering metaphor points the way to a different style of design- get the car started, then point it a little this way, then a little that way, then this way again.

What is simplest?

So, the definition of the best design is the simplest design that runs all the test cases. The effectiveness of this definition turns on what we mean by simplest?

Is the simplest design the one with the fewest classes? This would lead to objects that were too big to be effective. Is the simplest design the one with the fewest methods? This would lead to big methods and duplication. Is the simplest design the one with the fewest lines of code? This would lead to compression for compression's sake and a loss of communication.

Here is what I mean by simplest. I boil it down to four constraints, in priority order:

1. The design must communicate everything you want to communicate
2. The design must contain no duplicate code
3. The design should have the fewest possible classes
4. The design should have the fewest possible methods

The purpose of the design of the system is first to communicate the intent of the developers and second to provide a place for the logic of the system to live. The constraints above provide a framework within which to satisfy these two requirements.

If you view the design as a communication medium, then you will have objects or methods for every important concept. You will choose the names of classes and methods to work together (??? Where is the stuff about metaphor???)

 Copyright © 1998 Kent Beck, All rights

Constrained as you are to communicate, then you must find a way to eliminate all the duplicated logic in the system. This is the hardest part of design for me, because you have to first find the duplication, and then you have to find a way to eliminate it. Eliminating duplication naturally leads you to create lots of little objects and lots of little methods, because otherwise there will inevitably be duplication.

But you don't just create new objects or methods for the fun of it. If you ever find yourself with a class that does nothing and communicates nothing or a method that does nothing and communicates nothing, then you delete it.

Another way of looking at this process is as erasure. You have a system that runs the test cases. You delete everything that doesn't have a purpose- either a communication purpose or a computational purpose. What you are left with is the simplest design that could possibly work.

How could this work?

The traditional strategy for reducing the cost of software over time is to reduce the probability and cost of rework. XP goes exactly backwards of these. Rather than reduce the frequency of rework, XP revels in rework. A day without refactoring is like a day without sunshine. How could this possibly cost less?

The key is that risk is money just as much as time is money. If you put in a design feature today and you use it tomorrow, you win, because you pay less to put it in today. The chapter The Economics of Software, however, suggests that this evaluation is not complete. If the difference between today's cost and tomorrow's cost is small enough, or if there is long enough between them (for some value of today and tomorrow), then the sum of today's investment and the interest you pay is greater than the investment tomorrow. In that case, you would better off waiting.

Design isn't free. Another aspect of this situation is that when you put more design in today, you increase the overhead of the system. There is more to test, more to understand, more to explain. So every day you don't just pay interest on the money you spent, you also pay a little Design Tax. With this in mind, the difference in today's investment and tomorrow's investment can be much greater and it is still a good idea to design tomorrow for tomorrow's problems.

As if this weren't enough, the killer is risk. As the Economics of Software pointed out, you can't just evaluate the cost of something that happens tomorrow. You also have to evaluate the probability of it happening. Now, I love to guess and be right just as much as anybody but what I discovered when I started paying attention was that I didn't guess right nearly as often as I thought. Often the fabulous design that I created a year ago had nearly no correct guesses. I had to rework every bit of the design before I was done, sometimes two or three times.

So, the cost of making a design decision today is the cost of making the decision plus the interest we pay on that money plus the inertia it adds to the system. The benefit of making a design decision today is the expected value of the decision being profitably used in the future, where the probability of profitable use is low.

We could easily conclude that we should never make a design decision today that we don't need it today. In fact, that is what XP concludes. „Sufficient to the day the troubles thereof."

Now, several factors make the above evaluation null and void. If the cost of making the change tomorrow is very much higher, then we should make the decision today on the off chance that we are right. If the inertia of the design is low enough (for example, you have really, really smart people), then the benefits of just-in-time design are less. If you are a really, really good guesser, then you could go ahead and design everything today. For the rest of us, however, I don't see any alternative to the conclusion that today's design should be done today and tomorrow's design should be done tomorrow.

Preparing for change

I often hear the objection, „Yes, but by carefully examining possible paths of future change and putting in hooks to accommodate those changes, our software is prepared for change." To which I reply, „There are other ways to be prepared for change." The following is from Michael Feathers:

??? Insert XNF ???

In the American Civil War there were generals like Thomas who worked out every possible detail before he set out on a campaign. He gathered all the supplies he might possibly need. He waited until he had perfect intelligence about the enemy dispositions. Then off he went, a few miles a day, inexorably grinding towards his goal. Until a quick cavalry strike in his rear paralyzed him.

 Copyright © 1998 Kent Beck, All rights

Thomas was prepared for change. Any circumstance that arose, he was ready for. But he moved slow and gave the enemy plenty of time to react.

There were also generals like Sherman, who set off on campaign by cutting loose from their base of supplies, taking with them only what they knew they couldn't live without and trusting that they could gather everything else they needed on their way. If there was a strong defense, Sherman by-passed it. If there were cavalry in his rear, he ignored them.

Sherman was prepared for change, too, but in a way that allowed him to move fast. And he ended up driving the enemy, not the other way around.

Role of pictures in design

But what about all those pretty pictures? Some people really do think better about their designs in terms of pictures instead of code. How do you help them make their contribution to the design?

First, there is nothing wrong with designing software using explicit pictures instead of a purely mental model of the system. Trouble drawing the pictures can give you subtle clues about the health of a design- you find it impossible to reduce the number of elements in the picture to manageable levels, there is an obvious asymmetry, there are many more lines than there are boxes. All of these are clues that come best from a graphical representation of the design.

Another strength of designing with pictures is how fast you can go. You can compare and contrast three designs using pictures in the time it would take you to code one.

The trouble with pictures is that they can't crash. They give you certain kinds of feedback about the design, but they insulate you from other kinds of feedback. Unfortunately, the feedback they insulate your from is exactly the feedback that teaches you most- Will this run the test cases? Does this support simple code? This is feedback you can only get from coding.

So, on the one hand we can go fast when we design with pictures. On the other, we are at risk when we design with pictures. We need a strategy that let's us take advantage of the strength of designing with pictures that also reduces its weaknesses.

But we are not alone. We have the principles to guide us. Let's see-

- Small initial investment- suggests that we draw a few pictures at a time
- Play to win- suggests that we don't use pictures out of fear (for example because we want to put off the day we admit we don't know what the design should be)
- Rapid feedback- suggests that we quickly find out if our pictures were on target or not
- Concrete experiments- suggests that we use something concrete, like code, to get our feedback
- Working with people's instincts- suggests that we encourage pictures of those who work best with pictures
- Embracing change and Travel Light- suggest that we don't save the pictures once they have had their effect on the code, since the decisions they represent will probably change tomorrow anyway

So, the strategy is that anyone can design with pictures all they want, but as soon as a question is raised that can be answered with code, the designers must turn to code for the answer. The pictures aren't saved. For example, the pictures could be drawn on a whiteboard. Wishing you could save the whiteboard is a sure sign the design hasn't been communicated, either to the team or to the system.

Using this advice, you would never spend more than 10-15 minutes drawing pictures. Then you will know what question you want to ask of the system. Then you could draw a few more pictures.

The same advice applies to other non-code design notations, like CRC cards. Do a few minutes of it, enough to illuminate a question, then turn to the system to reduce the risk that you have been fooling yourself.

System Architecture

???What is missing? System architecture…

 Copyright © 1998 Kent Beck, All rights

Testing Strategy

Oh yuck. Nobody wants to talk about testing. Testing is the ugly step-child of software development. The problem is, everybody knows that testing is important. Everybody knows they don't do enough testing. And we feel it- our projects don't go as well as they should and we feel like more testing might help the problem. But then we read a testing book and instantly get bogged down in the host of kinds and ways of testing. There's no way we could do all that and still get any development done.

Here's what XP testing is like. Every time a developer writes some code, they think it is going to work. So every time they think some code is going to work, they take that confidence out of the ether and turn it into an artifact that goes into the program. The confidence is there for their own use. And because it is there in the program, everyone else can use that confidence, too.

The same story works for the customer. Every time they think of something concrete the program should do, they turn it into another piece of confidence that goes into the program. Now their confidence is in there with the developers' confidence. The program just gets more and more confident.

Now, a testing person would look at XP testing and sneer. This is not the work of someone who loves testing. Quite the contrary. This is the work of someone who loves getting programs working. So, you should write the tests that help get programs working and keep programs working. Nothing more.

Remember the principle „Work with human nature, not against it." That is the fundamental mistake in the testing books I've read. They start with the premise that testing is at the center of development. You must do this test and that test and oh yes this other one, too. If we want developers and customers to write tests, we had better make the process as painless as possible, realizing that the tests are there as instrumentation, and it is the behavior of the system being instrumented that everyone cares about, not the tests themselves. If it was possible to develop without tests, we would dump all the tests in a minute.

The tests that you must write in XP are isolated and automatic.

First, each test doesn't interact with the others you write. That way you avoid the problem that one test fails and causes a hundred other failures. Nothing discourages testing more than false negatives. You get this adrenaline rush when you arrive in the morning and find a pile of defects. When it turns out to be no big deal, it's a big let down. Are you going to pay careful attention to the tests after this has happened five or ten times? No way.

The test are also automatic. Tests are most valuable when the stress level rises, when people are working too much, when human judgement starts to fail. So the tests must be automatic- returning a thumbs up/thumbs down indication of whether the system is behaving.

It is impossible to test absolutely everything, without the tests being as complicated and error-prone as the code. It is suicide to test nothing (in this sense of isolated, automatic tests). So, of all the things you can imagine testing, what should you test?

You should test things that might break. If code is so simple that it can't possibly break, and you measure that the code in question doesn't actually break in practice, then you shouldn't write a test for it. If I told you to test absolutely everything, pretty soon you would realize that most of the tests you were writing were valueless, and you would stop writing them. But, if you were at all like me, you stop writing all tests- „This testing stuff is for the birds."

Testing is a bet. The bet pays off when your expectations are violated. The happy way a test can pay off is when a test works that you didn't expect to work. Then you're done implementing that test. The sad way a test can pay off is when a test breaks that you expected to work. In either case, you learn something. And software development is learning. The more you learn, the better you develop.

So, if you could, you would only write those tests that pay off. Since you can't know which tests would pay off (if you did, then you would already know and you wouldn't be learning anything), you write tests that might pay off. As you test, you reflect on which kinds of tests tend to pay off and which don't, and you write more of the ones that do pay off, and fewer of the ones that don't.

Who Writes Tests?

As I said at the beginning of the chapter, the tests come from two sources:

- developers

 Copyright © 1998 Kent Beck, All rights

- customers

The developers write tests method-by-method. Here are the circumstances in which a developer writes a test:

- As you are about to write a method, if the interface is at all unclear, you write a test before you write the method.
- If the interface is clear, but you imagine that the implementation will be the least bit complicated, you write a test before you write the method.
- If you think of an unusual circumstance in which the code should work as written, you write a test to communicate the circumstance.
- If you find a problem later, you write a test that isolates the problem.
- If you are about to refactor some code, and you aren't sure how it's supposed to behave, and there isn't already a test for the aspect of the behavior in question, you write a test first.

The developer-written unit tests always run at 100%. If one of the unit tests is broken, no one on the team has a more important job than fixing the tests. Because, if a test is broken, you have an unknown amount of work to do to fix it. It might only take a minute. But it might take a month. You don't know. And because the developers control the writing and execution of the unit tests, they can keep the tests completely in sync.

The customers write tests story-by-story. The question they need to ask themselves is, „What would have to be checked before I would confident that this story was done?" Each scenario they come up with turns into a test, in this case a functional test.

The functional tests don't run at 100% all of the time. Because they come from a different source than the code itself, I haven't figured out a way to synchronize the tests and the code. So, while the measure of the unit tests is binary- 100% or bust- the measure of functional tests is by necessity based on percentages. Over time you expect the functional test scores to rise to near 100%. As you get close to a release, the customer will need to categorize the failing functional tests. Some will be more important to fix than others.

Customers typically can't write functional tests alone. They need the specialized help of the kind of devious, paranoid mind that makes a good tester. That's why an XP team of any size carries at least one dedicated tester. The tester's job is to translate the sometimes vague testing ideas of the customer into real, automatic, isolated tests. The tester also uses the customer-inspired tests as the starting point for variations that are likely to break the software.

Even if you have a dedicated tester, someone whose joy comes from breaking software that is supposed to be done already, they work within the same economic framework as developers writing tests. The tester is placing bets, hoping for a test that succeeds when it should fail or that fails when it should succeed. So the tester is also learning to write better and better tests over time, tests that are more likely to pay off. The tester is certainly not there to just churn out as many tests as possible.

Other Tests

While the unit and functional tests are the heart of the XP testing strategy, there are other tests that may make sense from time to time. An XP team will recognize when they are going astray and a new kind of test could help. They might write any of the following kind of tests (or any of the other tests you can find in a testing book):

- Parallel test- a test designed to prove that the new system works exactly like the old system. Rather, the test shows how the new system differs from the old system, so a business person can make the business decision of when the difference is small enough that they can put the new system into production.
- Stress test- a test design to simulate the worst possible load. Stress tests are good for complex systems where the performance characteristics are not easily predictable.
- Monkey test- a test designed to make sure the system acts sensibly in the face of nonsensical input.

 Copyright © 1998 Kent Beck, All rights

Management Strategy

I've left this section to last, because it really only makes sense once you understand the context in which it will be executed. As with the rest of the strategies, it is ridiculously naïve if taken in isolation. With the rest of the practices providing checks and balances, however, it makes sense.

On the one hand, you would like the manager to make all the decisions. There is no communication overhead, because there is only one person. There is one person to be responsible to upper management.

We know this strategy doesn't work, because no one person knows enough to do a good job of making all the decisions. Management strategies that are balanced towards centralized control are also difficult to execute, because they require lots of overhead on the part of those being managed.

On the other hand, the opposite strategy doesn't work. You can't just let everyone go off and do what they want without any oversight. People inevitably get off on tangents. Someone needs to have a bigger view of the project, and to be able to influence the project when it gets off course.

Once again, we can fall back on the principles to help us navigate between these two extremes:

- Accepted responsibility- suggests that it is the manager's job to highlight what needs to be done, not to assign work.
- Quality work- suggests that the relationship between managers and developers needs to be based on trust, because the developers want to do a good job. On the other hand, this doesn't mean the manager does nothing. However, there is a big difference between the attitude that „I am trying to get these guys to do a decent job“ and „I get to help these guys do an even better job“.
- Incremental change- suggests that the manager provides guidance all along, not a big policy manual at the beginning
- Local adaptation- suggests that the manager needs to take the lead in adapting XP to local conditions, to be aware of how the XP culture clashes with the company culture and to find a way to resolve the misfit.
- Travel light- suggests that the manager doesn't impose a lot of overhead- long all-hands meetings, lengthy status reports. Whatever the manager requires of the developers shouldn't take much time to fulfil.
- Honest measurement- suggests that whatever metrics the manager gathers should be at realistic levels of accuracy. Don't try to account for every second if your watch only has a minute hand.

The strategy that emerges from this evaluation is more like de-centralized decision making than centralized control. The manager's job is to run the Planning Game, to collect metrics and make sure they are seen by those being measured, and occasionally to intervene in situations that prove that they can't be resolved in a distributed way.

Metrics

The basic XP management tool is the metric. For example, the ratio between estimated development time and calendar time is the basic measure for running the Planning Game. It lets the team set the Project Velocity. If the ratio rises (less calendar time for a given estimated amount of development), it can mean that the team process is working well. Or, it can mean that the team isn't doing enough besides fulfil requirements, and that a price will be paid long term.

The medium of the metric is the Big Visible Chart. Rather than send email to everyone that they learn to ignore, the manager periodically (no less than weekly) updates a chart that is placed in a prominent position. Often, this is all the intervention that is needed. You think there aren't enough tests being written? Put a chart of the number of tests up, and update it every day.

Don't have too many metrics, and be prepared to retire metrics that have served their purpose. Three or four measures are typically all a team can stand at one time.

A particular danger of metrics is that they tend to go stale over time. In particular, any metric that is approaching 100% is likely to be useless. For unit test scores, which must be 100%, this advice doesn't apply, but then the unit test score is more like an assumption than a metric. You can't count on 97% functional test scores to mean

 Copyright © 1998 Kent Beck, All rights

that you have 3% of the effort remaining, however. If a metric gets close to 100%, replace it with another that starts comfortably down in the single digits.

This is not to suggest that you can manage and XP project „by the numbers“. Instead, the numbers are a way of gently and non-coercively communicating the need for change. The XP manager’s most sensitive barometer of the need for change is awareness of his or her own feelings, physical and emotional. If your stomach knots when you get in the car in the morning, something is wrong with your project and it’s your job to effect the change.

Coaching

What most folks think of as management is divided into two roles in XP: the coach and the tracker. Coaching is primarily concerned with the technical execution (and evolution) of the process. The ideal coach is a good communicator, not easily panicked, technically skilled (although this is not an absolute requirement), and confident. Often, you want to use as coach the person who on other teams would have been the lead developer or system architect. However, the role is very different.

The phrases „lead developer“ and „system architect“ conjure up visions of isolated geniuses making the important decisions on the project. The coach is just the opposite. The measure of a coach is how few technical decisions he or she makes. The job is rather to get everybody else making good decisions.

The coach doesn’t take responsibility for many development tasks. Rather, the job duties are:

- Be available as a development partner, particularly for new developers taking responsibility for the first time or for difficult technical tasks
- See long term refactoring goals, and encourage small scale refactorings to address parts of these goals
- Help developers with individual technical skills, like testing, formatting, and refactoring
- Explain the process to upper level managers

But perhaps the most important job for the coach is the acquisition of toys and food. XP projects seem to attract toys. Lots are of the ordinary brain-teaser type recommended by lateral thinking consultants everywhere. But every once in a while the coach will have the opportunity to profoundly influence development by buying just the right toy, and staying alive to this possibility is one of the coach’s greatest responsibilities. For example, on the Chrysler project design meetings were going on for hours without resolution. So I bought an ordinary kitchen timer and decreed that no design meeting could be longer than 10 minutes. I don’t believe the timer was ever used, but its visible presence reminded everyone to be aware of when a discussion had ceased being useful and had turned into a process for avoiding going and writing some code to get the answer for sure.

Food, also, is a hallmark of XP projects. There is something powerful about breaking bread with someone. You have an entirely different discussion with them if you are chewing at the same time. So XP projects always have food laying around.

Tracking

Tracking is the other major component of management in XP. You can make all the estimates you want, but if you don’t measure what really happens against what you predicted would happen, you won’t ever learn.

It is the job of the tracker to gather whatever metrics are being gathered at the moment and make sure the team is aware of what was actually measured (and reminded of what was predicted or what is desired).

Running the Planning Game is a part of tracking. The person tracking needs to know the rules of the game cold, and be prepared to enforce them, even in emotional situations (planning is always emotional).

Tracking needs to happen without lots of overhead. If the person gathering actual development time is asking developers for their status twice a day, the developers will soon run away rather than face the grilling. Instead, the person tracking should experiment with just how little measurement they can do and still be effective. Gathering real development data twice a week is plenty. More measurement won’t give you better results.

Intervention

Managing an XP team isn’t all fetching donuts and tossing Frisbees. There are times when problems simply can’t be solved by the emergent brilliance of the team, encouraged by a loving and watchful manager. At times

 Copyright © 1998 Kent Beck, All rights

like this, the XP manager must be comfortable stepping, making decisions, even unpopular ones, and seeing the consequences through to the end.

First, though, the manager must search carefully to discover if there was something they should have been aware of or done earlier to avoid the problem entirely. The time for intervention is not the time for donning white armor and leaping on a charger. Rather, it is a time to come to the team and say, „I don't know how I let it get like this, but now I have to do XXX." Humility is the rule of the day for an intervention.

One of the matters serious enough for intervention is personnel changes. If a team member just isn't working out, the manager needs to ask them to leave. And decisions like this are better done sooner rather than later. As soon as you can't think of any scenario in which the offender would be a help rather than a hindrance, you should make the move. Waiting will only make the problem worse.

A slightly more pleasant duty is intervening when the team's process needs changing. It isn't the manager's job to dictate what is to change and how, generally, but to point out the need for change. The team should come up with one or more experiments to run. Then the manager reports back on the measured changes caused by the experiment.

The final interventionist duty of the manager is killing the project. The team is likely to never want to stop. They would never quit on their own. But the day will come when further engineering investment in the current system is less attractive than some other alternative, like starting a replacement project. The manager is responsible for always being aware of when this threshold is crossed, and when the threshold is crossed, for informing upper management of the need for the change. This is likely a moment for a mix of gentleness and firmness.

Implementing XP

In this section we will discuss various topics around the theme of putting the strategies from the last section into practice. Once you choose such a radically simplified set of strategies as we found in the last section, you suddenly have much more flexibility to play with. You can use this flexibility for many purposes, but you need to be aware that it exists and what possibilities it opens up for you.

My Story of XP

It may help you to understand XP a little more if you see how I came to it, so here's my part of the story. XP is much bigger than just me, so you should ask the other folks their stories, too.

First let me say that nothing in XP is new. All of the practices are nearly as old as programming. My contribution is to put them all under one umbrella, explore how they can work best together, and give them, modestly speaking, the coolest name for a methodology in history.

The heart of the practices are things I watched my dad do when I was a teenager. My dad wrote process control systems with the then-new 8-bit microprocessors. I watched him start with simple little designs, test them to death, and rewrite any part that started getting complicated.

In college, I was a crummy programmer. I did just fine at all the math-related topics, but I couldn't code my way out of the proverbial wet McDonalds's sack.

When I got out of college, I went to work at Tektronix Labs, which had one of the best available implementations of Smalltalk. There I hooked up with Ward Cunningham. I could make Smalltalk do cool things, but I didn't understand it. Ward understood it. I started watching him program a program called Plumbin' that he developed for an advanced course. After a day or two I was telling him about missing periods and unbalance parentheses. Pretty soon thereafter we moved his machine so we could sit next to each other. We then started a run that produced 15 interesting research products in 15 months. We would just start coding to understand some vague idea we had. If it went poorly, we would stop and start on something else.

At the same time I rediscovered Christopher Alexander. One of his statements that influenced me was in the preface to the second edition of Notes on the Synthesis of Form. In it, he railed against ungrounded abstract thinking. He quotes ???, „Sociologists talk about sociological method. Physicists talk about physics." This gelled a lot of frustration I had been feeling about methodologists and computer scientists with their heads in the clouds. I vowed to talk about „stuff" and give up ungrounded abstract thinking.

 Copyright © 1998 Kent Beck, All rights

Another influential aspect of Alexander is his plea to redistribute decision making power between architects and clients. I saw software engineers wielding power over users in a way very much like what Alexander described as the power relationship between architect and customer. Alexander wanted to get the decision making closer to the person whose life was affected by the decision, and who knew more subtleties about the decision than anyone else, the customer. At the same time, there were certain things that architects did best, and they should be allowed to do them, often at the expense of setting up constraints for the customer.

After I left Tek I spent six or seven years slinging Smalltalk code. I grew increasingly disenchanted with the cleverness valued in the Smalltalk community. The things that worked for me commercially were always simple simple things.

I had a seminal meeting with Joel Spiegel at a breakfast meeting at the Phoenix OOPSLA in 1990. We were talking about development and research. I said that I really appreciated research jobs because there was so little accountability. Joel gently explained the two really bad features of not having accountability:

1. If times get tough, it is great to be able to point to a revenue stream and say, „I created that."
2. The quality and quantity of feedback you get from a system in production is entirely different from what you get during development. This was the beginnings of the philosophy of getting the system into production as quickly as possible. If you really want to do research, that is if you really want to learn, you have to put your systems into production.

When I started my own consulting company, I was burnt out. I had been working months of hundred hour weeks to make enough money to reduce the risk of starting my own company. After I had visited my first few clients, I realized that they needed entirely different advice. I had a story- patterns, pair programming, cleaning code up- but not enough of a story to be satisfied with or to explain to the clients. So I started over.

I really started over. The first thing I did was teach myself to touch type all over again. I realized that I was looking at the keys from time to time, and that it interrupted the flow of my thinking. So I laboriously closed my eyes and practiced just typing for hours.

Next to go out the window were my Smalltalk skills. I was a pretty good Smalltalk programmer, but I knew that I couldn't explain how to do what I did. So I set aside a month. The first day I resolved not to type a single character if I couldn't explain exactly why it was that character and not another. I was reminded of the centipede that suddenly began paying attention to his feet. It was torture. What had been a fairly free, natural, and enjoyable experience became agony. I would write one word, then spend two hours explaining, in the form of a pattern, what I had just done. At the end of a couple of days I was ready to give up. At the end of a week I still wasn't having any fun, but I was already programming faster and cleaner than I had before. Most of the time I was using patterns that I had already written. I would only have to add one or two patterns a day.

At the end of the month, I was much faster and cleaner than I had been before. My programs came rolling off my fingers in a way I couldn't have imagined before. Ward and I had always been deeply aware of our own processes when we programmed together, and we talked about them often in the course of a day. Now I had finally gotten beyond the internal continuation of that discussion. Now I had answered for myself all of the questions we had posed way back when, and I could just use the answers without thinking about the questions any more.

The result of that exercise was the Smalltalk Best Practice Patterns. It actually started out to be a book about everything in the entire programming universe. I clung to that notion for a while, until I finally realized that it would never be done. Cutting the book down to be just about communicating through code (during a call with Ward) was a compelling example of scope management, a theme that runs through the Planning Game.

Just after I had rebuilt my programming chops from scratch, I was fortunate enough to hook up with a project at Hewitt Associates. Unlike almost every project I have ever been involved in, they brought me in at the beginning. I was asked to help develop the architecture of the system, and to teach them about development processes. A day before flying out to Chicago I was in a panic about what to say about testing. I decided to turn back to the workspace-based style of testing that Ward and I had done. We had a formalized way of using a workspace. Well, if the whole workspace was an object, what would it be? A Test. What about the variables? The variables in the workspace. And on and on, until I had the testing framework.

Hewitt was the first time I'd been asked to lead a whole project. Until then I thought of myself as a programmer who was also asked to give advice. Now I was being asked to „be in charge", at least a little bit. I was forced to think of what I knew in a larger context, and to think of the advice I gave in terms of how it helped or hurt the productivity of the whole team, not just individuals. I also got a lot of confidence out of the gig, because the project was successful and on-time and has since made a huge difference in Hewitt's business.

 Copyright © 1998 Kent Beck, All rights

When I was done with the testing framework, the whole thing was so small and simple that I was a little embarrassed. Sure, it was okay to give to a client who knew me already. But sending it out to the world was scary. What would the Smalltalk world think about a purported framework with 4 classes and 30 methods? So, I sent the framework to Hal Hildebrand, one of Smalltalk's hottest programmers. He had written a fabulous database called Tensegrity. I went off to Chicago and thought nothing more about it.

When I got back there was an excited voicemail from Hal. I called him immediately. He had been having trouble with Tensegrity, but because much of it was such low level code, he didn't have any tools to debug it with. Totally frustrated, he decided to just start writing tests for all of his lowest level data structures. He spent a week writing the tests, discovering several defects in the process. When he couldn't think of any more tests to write, he tried the operations that had been broken before. They all just worked.

This story was a revelation to me, how the emergent effect of small-scale tests could be large scale stability. I had read books on testing, and I thought that the amount of work they recommended was ridiculous. But if I could just do little small scale testing and mostly get large scale stability, I figure I could convince myself and other people to do that.

Fast forward a year. One day I got a call asking me to review a project at Chrysler. They had some performance problems. When I got there, it turned out that they had huge people problems, and performance was the tiniest tip of the iceberg. I did my usual consulting trick of listening and repeating. Over coffee the morning of my second day there one of the developers said, „The best thing for this project would be a crashed disk." Everyone laughed nervously. Not me. The next day I was in the office of the CIO telling her that I thought her best option was to give everybody a week off, throw away the code, and start over. She said, „Okay. You're in charge."

Now, I hadn't wanted to be in charge. I had just wanted to give the best advice I could. Having precipitated the crisis, though, I felt duty-bound to see the team out of it. In the plane on the way home, though, I was shaking so badly that I couldn't drink.

So, I had all these influences- Ward's fluid development style and his Episodes pattern language for project management, Hal's experience of emergent stability through unit testing, Alexander's clear separation of political power, my belief in the power of communicative code, the beginnings of the formalization of refactoring. In addition I had studied a little of the Scrum methodology, evolutionary biology, chaos theory, and the power of rituals and totems.

I returned to Chrysler a couple of weeks later. The first thing I did was interview everyone on the team to find out what they wanted to do and what they thought they should do. After asking the first interviewee, Bob Coe, the project manager, these questions, I explained how the project would be run. I just made XP up, or rather drew the threads together to make it up, on the spot- stories, commitment schedule, iteration schedule, testing, refactoring, pair programming, the planning game. Then I interviewed the next person, and explained the process. Then another. Another. At the end of the day I had explained the process 12 times and was getting pretty convinced myself that it would work. When Bob came back that evening with a spreadsheet for tracking the tasks in an iteration I knew we would be successful.

C3 went great. It was lots of fun, and a great challenge. I often had to act like I was confident when I wasn't. I was confident in my methods, but I wasn't confident in myself. I had to muddle through. I had gotten them into this. I was responsible for getting them out. Probably the best thing that happened is that we went into production three months after our initial schedule. If we had precisely hit our date, I would have been insufferable. That slip hurt a lot. I had made this fabulous process, gotten the team to buy into it, they had made it even better than I had imagined it, and we were still 20% late. I still had a lot to learn.

The story gets a little muddled here. The events are a little too recent to make up a coherent story about them. Chrysler had left me cautiously confident that I could help people develop software better. Since then I have had a couple of years with some hits and some misses, convincing me that I still do have a lot to learn. I am absolutely convinced about the general direction however- travel light, build well for today, constantly push for simplicity, push responsibility out, and above all test the shit out of everything you touch.

Lifecycle of an Ideal XP Project

This chapter gives you an idea of the overall story of an XP project. It is idealized- you may have gotten the idea by now that no two XP projects could (or should) ever be exactly alike. What I hope you will get from this chapter is an idea of the overall flow of a project.

 Copyright © 1998 Kent Beck, All rights

Exploration

Pre-production is an unnatural state for a system and should be gotten out of the way as quickly as possible. What was the phrase I heard recently? „To go into production is to die." XP says exactly the opposite. Not to be in production is to be spending money without making money. Now, it may just be my wallet, but I don't find the outgo/no income state to be very comfortable very long.

Before you can go into production, however, you have to believe that you can go into production. You have to have enough confidence in your tools that you believe you can get the program finished. You have to believe that once the code is done, you can run it day in and day out. You have to believe that you have (or can learn) the skills you need. The team members need to learn to trust each other.

The exploration phase is where all of this comes together. You are done with exploration when the customer is confident that there is more than enough material on the Story Cards to make a good first release and the developers are confident that they can't estimate any better without actually implementing the system.

During exploration the developers are using every piece of technology they are going to be using in the production system. They are actively exploring possibilities for the system architecture. They do this by spending a week or two building a system like what they will build eventually, but doing it three or four ways. Different pairs can try the system different ways and compare, or you could have two pairs try the system the same way and see what differences emerge.

If a week won't suffice to get a particular piece of technology up and running, then I would have to classify that technology as a risk. It doesn't mean that you shouldn't use it. But you should explore it more carefully, and consider alternatives.

You might want to consider bringing in technology specialists during exploration so that your experiments aren't hampered by stupid little things that could be handled easily by someone who already has been there. By wary about blindly accepting advice for the eventual use of the technology, however. Experts sometimes develop habits that are based on value systems not entirely in tune with extreme programming. The team will have to be comfortable with the practices they choose. Saying „the expert said so," isn't very satisfying when the project is spiraling out of control.

The developers should also experiment with the performance limits of the technology they are going to use. If at all possible, they should simulate realistic loads with the production hardware and network. You don't have to have the whole system done to write a load simulator. You can get a lot of mileage by just calculating, for example, how many bytes per second your network will have to support and then running an experiment to see if it can provide the necessary bandwidth.

The developers should also experiment with architectural ideas- how do you build a system for multiple levels of undo? Well, implement it three different ways for a day and see which one feels best. These little architectural explorations are most important when you find the user coming up with stories that you have no idea how to implement.

The developers should estimate every programming task they embark on during exploration. When a task is done, they should report on the actual calendar time required for the task. This practice with estimation will raise the team's confidence in their estimates when the time comes to make a public commitment.

While the team is practicing with the technology, the customer is practicing writing tests. Don't expect this to go completely smoothly. The stories at first won't be what you need. The key is getting the customers lots of quick feedback so they can learn quickly to specify what the developers need and not specify what the developers don't need. The key question is „Can the developers confidently estimate the effort required by the story?" Sometimes the story needs to be written differently, sometimes the developers need to go off and experiment for a bit.

If you have a team that already knows their technology and each other, the exploration phase could be as short as a few weeks. With a team that is completely new to a technology or domain, you might have to spend a few months. If exploration took longer than this, I might look for a small but real project that they could complete easily to lend urgency to the process.

 Copyright © 1998 Kent Beck, All rights

Planning

The purpose of the planning phase is for the customers and developers to confidently agree on a date by which the smallest, most valuable stories will be done. See the Planning Game for a way to run this. If you prepare during exploration, planning (the production of the commitment schedule) should take a day or two.

The plan for the first release should be between two and six months long. Shorter than that and you won't be able to solve any significant business problems (probably- if you can, great!). Longer than that and there is too much risk.

Iterations to first release

The commitment schedule is broken into 2-4 week iterations. Each iteration will produce a set of functional test cases for each of the stories scheduled for that iteration.

While you are clicking through the iterations, you are looking for deviations from the plan. Is everything taking twice as long as you had thought? Half as long? Are the test cases getting done on time? Are you having fun?

When you detect deviations from the plan, then you need to change something. Maybe the plan needs to change- add or remove stories or change their scope. Maybe the process needs changing- you find better ways of working your technology, or better ways of working XP.

Ideally, at the end of every iteration, the customer will have completed the functional tests and they will all run. Make a little ceremony out of the end of each iteration- buy pizza, shoot off fireworks, have the customer sign the completed story cards. Hey- you have just shipped quality software on time. It may only be three weeks worth, but it is still an accomplishment and it is worth celebrating.

At the end of the last iteration, you are ready to go into production.

Productionizing

The end game of a release sees a tightening up of the feedback cycle. Instead of three week iterations, you may go to one week iterations. You may have a daily stand-up meeting so everybody knows what everybody else is working on.

Typically there will be some process for certifying the software as ready to go into production. Be prepared to implement new tests to prove your fitness for production. Parallel testing is often applied at this stage.

You may also need to tune the performance of the system during this phase. I haven't said much about performance tuning in this book. I believe very strongly in the motto „Make it run, make it right, make it fast.“ Late in the game is the perfect time to tune, because you will have as much knowledge as possible embedded in the design of the system, you will have the most realistic possible estimates of the production load on the system, and you are likely to have the production hardware available.

During productionizing, you will slow down the pace at which you evolve the software. It isn't that software stops evolving, but rather that risk becomes more important in your evaluation of whether a change deserves to go into this release. However, be aware that the more experience you have with a system, the more insight you will have into how it should be designed. If you begin finding lots of ideas that you can't justify putting into the system for this release, make a visible list so everybody can see where you will be going after this release goes into production.

When the software actually goes into production throw a big party. Many projects never go into production. Having yours go live is a reason to celebrate. If you're not a little scared at the same time, you're crazy, but the party can help you blow off a little of the excess tension that is bound to have built up.

Subsequent releases

Every release begins with an exploration phase. You may try big refactorings that you were afraid of late in the previous development cycle. You may try new technology that you intend to add in the next release, or migrate to newer versions of the technology you are already using. You may experiment with new architectural ideas. The customer may try writing wacky new stories in search of a big winner for the business.

 Copyright © 1998 Kent Beck, All rights

Developing a system that is in production is not at all the same as developing a system that isn't yet in production. You are more careful of the changes you make. You have to be prepared to interrupt development to react to production problems. You have live data that you have to be careful to migrate as you change the design. If pre-production weren't so dangerous, you'd keep from going into production forever.

Being in production is likely to change your development velocity. Be conservative with your new estimates. While you are exploring, measure the effect production support has on your development activities. I have seen an increase in the ratio of ideal engineering time to calendar time of 50% after having gone into production (from 2 calendar days per engineering day to 3). Don't guess, though, measure.

Be prepared to change the team structure slightly to deal with production. You may want to take turns manning the „help desk", so most of the developers don't have to deal with production interruptions most of the time. Be careful to rotate all the developers through the position- there are things you learn from support production that you just can't learn any other way. On the other hand, it isn't as much fun as developing.

Put newly developed software into production as you go along. You may know that parts of the software won't be executed. Put it into the production system anyway. I have been on projects where this cycle was daily or weekly, but in any case you shouldn't leave code laying around for longer than an iteration. The timing depends on how much verification and migration cost. The last thing you need when you are at the end of a development cycle is to integrate a big gob of code, which „couldn't possibly" break anything. If you keep the production code base and the development code base nearly in sync, you will have much earlier warning of integration problems.

Death

Dying well is as important as living well. This is as true for XP as for people.

If the customer can't come up with any new stories, then it is time to put the system into mothballs. Now is the time to write a 5-10 page tour of the system, the kind of document you wish you would find when you came to change something in 5 years.

That's the good reason to die- the customer is happy with the system and can't think of anything they would like to add for the foreseeable future. (I've never experienced this, but I've heard about it, so I included it here).

There is a not so good reason to die- the system just isn't delivering. The customer needs features and you just can't add them economically. The defect rate creeps up to where it is intolerable.

This is the entropic death you have fought against for so long. XP is not magic. Entropy eventually catches XP projects, too. You just hope that it happens much later rather than sooner.

In any case, we have already posited the impossible- the system needs to die. It should happen with everybody's eyes open. The team should be aware of the economics of the situation. They, the customers, and the managers should be able to agree that the team, and the system, just can't deliver what is needed.

Then it is time for a fond farewell. Throw a party. Invite everyone who has worked on the system to come back and reminisce. Take the opportunity to try to plot the seeds of the system's downfall, so you'll know better what to look for in the future. Imagine with the team how they would run things differently next time.

Roles for People

A sports team works best when there are certain roles that someone takes responsibility for. In football you have the goalie, the striker, the sweeper, and so on. In basketball you have a point guard, a center, a shooting guard, and so on.

A player taking one of these positions accepts a certain set o responsibilities- setting up teammates for scoring, preventing the other team from scoring, perhaps managing a certain portion of the field. Some of the roles are nearly solitary. Other require that the player correct the mistakes of teammates, or manage their interactions.

These roles become customary, and sometimes even embedded in the rules of the game, precisely because they work. At some time, probably every other combination of responsibilities has been tried. The ones you see today are there because they worked and the other ones didn't.

 Copyright © 1998 Kent Beck, All rights

Good coaches are effective at getting players to work well at their position. They spot deviations from the usual practice of the position and either help the player correct the deviation, or understand why it is acceptable for that player to do things a little different.

However, the great coach knows that the positions are merely customary, not laws of nature. From time to time, the game changes or the players change enough so that a new position becomes possible or an old one becomes obsolete. Great coaches are always looking for what advantages could be had be creating new positions and eliminating existing ones.

Another facility of great sports coaches is their ability to mold the system to their players, instead of the other way around. If you have a system that works fabulously if you have quick players, and the team that shows up at the first workout is big and strong instead, then you will do better creating a new system that lets the team's talents shine. Lots of coaches can't do this. Instead, they get so focused on the beauty of „the system“ that they can't see that it isn't working.

All of this is leading up to a big warning in what follows- here are some roles that are found to have worked well on previous projects. If you have people who don't fit the roles, change the roles. Don't try to change the people (well, not too much). Don't act like there isn't a problem. If a role says, „This person must be willing to take large chances,“ and you have a watchmaker instead, you have to find a different division of the responsibilities that will meet your goals without the role in question being filled by a risk taker.

For example, I was talking to a manager about a team of his. A developers was also the customer. I said that couldn't possibly work, since the developer has to execute the process and make technical decisions and be sure to defer business decisions to the customer (see the Planning Game).

No, no, no. This guy is a real bond trader, he just happens to know how to program, too. The other bond traders all like and respect him, and they are willing to trust him and confide in him. He has a solid vision for where the system is headed. The other developers separate when he is speaking as the customer and when he is making technical decisions.

Okay. The rules here say that a developer can't be the customer. In this case, the rules don't apply. What still applies is the separation of technical and business decisions. The whole team, the developer/customer, and especially the coach, must be aware of which hat the developer/customer is wearing at any given time. And the coach needs to be aware that no matter how well the arrangement has worked in the past, if they run into trouble, the dual role is a likely cause of problems.

Developer

The developer is the heart of XP. Actually, if developers could always make decisions that carefully balanced short-term and long-term priorities, there would be no need for any other technical people on the project besides developers. Of course, if the customer didn't absolutely have to have to software in order to keep the business running, there would be no need for the developers, so it wouldn't do to get too big headed about being the vital developer.

On the surface, being an XP developer looks a lot like being a developer within other software development disciplines. You spend your time working with programs, making them bigger, simpler, faster. Beneath the surface, the focus is quite different. You job isn't over when the computer understands what to do. Your first value is communication with other people. If the program runs, but there is some vital component of communication left to be done, you aren't done. You write tests that demonstrate some vital aspect of the software. You break the program into more smaller pieces, or merge pieces that are too small into larger, more coherent pieces. You find a system of names that more accurately reflects your intent.

This may sound like a high-minded pursuit of perfection. It is anything but. You try to develop the most valuable software for the customer, but not to develop anything that isn't valuable. If you can reduce the size of the problem enough, then you can afford to be careful with the work you do on what remains. Then, you are careful by habit.

There are skills that you must possess as an XP developer that are not needed or at least not emphasized in other styles of development. Pair programming is a learnable skill, but one often at odds with the tendencies of the sort of people who typically get into programming. Perhaps I should state this less equivocally- nerds aren't good at talking. Now, there are certainly exceptions to this, and it is possible to learn to talk with other folks, but the fact is that you will have to communicate and coordinate closely with other developers in order to be successful.

 Copyright © 1998 Kent Beck, All rights

Another skill needed by the extreme developer is the habit of simplicity. When the customer says, „You must do this and this and this,“ you have to be prepared to which of those items is really necessary and how much of each. Simplicity also extends to the code you write. A developer with every last analysis and design pattern ready to hand will not be likely to succeed with XP. Of course, you can do a better job if you have more tools in your toolbox than if you have fewer, but it is much more important to have a handful of tools that you know when not to use, than to know everything about everything and risk using too much solution.

You will need skills that are more technically oriented as well. You have to be able to program reasonably well. You have to be able to refactor, which is a skill with at least as much depth and subtlety as programming in the first place. You have to be able to unit test your code, which like refactoring requires taste and judgement to apply well.

You have to be willing to set aside the feeling of individual ownership of some portion of the system in favor of partial ownership of the whole system. If someone changes code that you wrote, in whatever part of the system, you have to trust the changes and learn from them. Of course, if they are wrong-headed, you are responsible for going and making things better.

Above all, you must be prepared to acknowledge your fears. Everybody is afraid- afraid of looking dumb, afraid of being thought useless, afraid of growing obsolete, afraid of not being good enough. That has been the hardest job of all for me- to give up being a guru and be willing to start over. Without courage, XP just simply doesn’t work. You would spend all of your time trying desperately not to fail. Instead, if you are willing, with the help of your team, to acknowledge your fears, then you can get on with the business of having fun writing great software.

Customer

The customer is the other half of the essential duality of extreme programming. The programmer knows how to program. The customer knows what to program. Well, not at first, of course, but the customer is willing to learn just as much as the developer is.

Being an XP customer is not easy. There are skills you have to learn, like writing good stories, and an attitude that will make you successful. Most of all, though, you have to become comfortable influencing a project without being able to control it. Forces outside your control will shape what actually gets built just as much as the decisions you make. Changes in business conditions, technology, the composition and capability of the team, all of these have a huge impact on what software gets delivered.

You will have to make decisions. This is the hardest skill for some of the customers I have worked with. They are used to IT not delivering half of what they promised, and for what is delivered to be half wrong. They have learned never to give an inch to IT, since they are bound to be disappointed anyway. XP won’t work with such a customer. If you are an XP customer, the team needs you to say with confidence, „This is more important than that,“ „This much of this story is enough,“ „These stories together are just enough.“ And when times get tough, and they always get tough, the team needs you to be able to change your mind, „Well, I guess we don’t absolutely have to have this until next quarter.“ Being able to make decisions like this will save your team at times, and reduce their stress enough so that they can do their best for you.

The best customers are those who will actually use the system being developed, but who also have a certain perspective on the problem to be solved. If you are one of these customers, you will have to be aware of when you are thinking a certain way because that is how things have always been done, rather than because of some essential quality in the problem. If you are a step or two removed from actually using the system, you will have to work extra hard to be sure that you accurately represent the needs of the real users. ???Something in here about how organizations push to have this separation, even when it isn’t actually necessary???

You will have to learn how to write stories. This may seem like an impossible task at first, but the team will give you the gift of copious feedback on the first few you write, and you will rapidly learn how much ground to cover in each story, and what information to include and exclude to guide planning.

You will have to learn to write functional tests. If you are the customer for an application with a mathematical basis, you will have an easier job- a few minutes or hours with a spreadsheet will suffice to create a test case. Or perhaps your team will build you a tool to make entering new test cases easy. Programs with a formulaic basis (like workflow, for example), also need functional tests. You will have to work closely with the team to learn what kind of things it is helpful to test, and what kind of tests are just redundant. Some teams may even assign you technical help for choosing, writing, and running the tests. Remember that your goal is to write those tests that let you say, „Well, if these run, then I’m confident the system will run.“

 Copyright © 1998 Kent Beck, All rights

Finally, you will have to demonstrate courage. There is a way from today to where you want to be. This team can help you find it, if you will help them find it.

Tester

Developers are legendary for fooling themselves about the quality of their work. A tester's job is to be the cold bucket of water snapping awake the now-dripping head of the deluded developer.

Testers need a suspicious and devious nature to be successful. In a way this prevents their ever becoming part of the team in the way that the developers are part of the team. You have to be able to live with this separateness to be an XP tester.

Since a lot of testing responsibility lies on the shoulders of the developers, you are focused on the customer. You are responsible for helping the customer choose and write functional tests. If the functional tests aren't part of the integration suite, you are responsible for running the functional tests regularly and posting the results in a prominent place.

Some XP teams are too small to require or support a separate tester. If you are on such a team, you must be painfully aware of when you are wearing your developer hat and when you are wearing your tester hat. Never mix the two responsibilities. As a tester you are trying to break the system by playing within the rules of acceptable input. You have to swallow your fondness for and confidence in the code, to see it through paranoid-tinted glasses. And you will never do as good a job as a dedicated tester would do. You may be able to do a good enough job, though.

You are not responsible for keeping the tests running though all the flipping and flopping inevitably endured by an XP system. The developers should refactor the tests at the same time they refactor the code, leaving the system in no worse shape when they are done than when they began.

Tracker

As a tracker, you are the conscience of the team (think Jiminy Cricket, but with better clothes).

Doing good estimates is a matter of practice and feedback. You have to make lots of estimates, and then notice how reality conformed to your guesses. Your job to close the loop on feedback. The next time the team is making estimates, you need to be able to say, „Two thirds of our estimates last time were at least 50% too high." On an individual basis, you need to be able to say, „Your task estimates are either way too high or way too low." The next estimates to come out are still the responsibility of the people who have to implement whatever is being estimated, but you have given them the feedback so that when they come out, they can be better than last time.

You are also responsible for keeping an eye on the big picture. Half way through an iteration you should be able to tell the team whether they are going to make it if the follow the current course or if they need to change something. A couple of iterations into a commitment schedule you should be able to tell the team whether they are going to make the next release without making big changes.

You are the team historian. You keep a log of functional test scores. You keep a log of defects reported, who accepted responsibility for each, what test cases were added on each defect's behalf.

The skill you need to cultivate most is the ability to collect the information you need without disturbing the whole process more than necessary. You want to disturb the process a little, to keep people aware of how much time they actually spent on a task in a way they might not be aware if you didn't ask. But you can't be such as a pain in the neck that people avoid answering you.

Coach

As coach, you are responsible for the process as a whole. You notice when people are deviating from the team's process and bring this to the team's attention. You remain calm when everyone else is panicking, remembering that in the next two weeks you an only get two week's worth of work done or less, and either two week's worth is enough or it isn't.

Everyone on an XP team is responsible for understanding their application of XP to some extent. You are responsible for understanding it much more deeply- what are alternative practices that might help the current set of problems, how are other teams using XP, what are the ideas behind XP and how do they relate to the current situation.

 Copyright © 1998 Kent Beck, All rights

The most difficult thing I have found about being a coach is that you work best when you work indirectly. If you see a mistake in the design, first you have to decide whether it is important enough that you should intervene at all. Every time you guide the team, you make them that much less self-reliant. Too much steering and they lose the ability to work without you, resulting in lowered productivity, lowered quality, lowered morale. So, first you have to decide whether the problem you see is enough of a problem that you need to accept the risk of intervening.

If you decide you really do know better than the team, then you have to make your point as unobtrusively as possible. For example, it is far better to suggest a test case that can only be implemented cleanly by fixing the design, than it is to just go and fix the design yourself. But it is a skill to learn how not to just directly say what you see, but to say it in such a way that the team sees it, too.

Sometimes, however, you must be direct, direct to the point of rudeness. Confident, aggressive developers are valuable precisely because they are confident and aggressive. However, this leaves them vulnerable to a certain kind of blindness, and the only cure is plain speaking. When you have let a situation deteriorate that the gentle hand on the yoke can no longer work, you have to be prepared to grab the yoke with both hands and steer. But only long enough to get the team back on track. Then you have to let one hand drop again.

I want to say something here about skills coaching. I am always in the position of teaching the skills of XP-simple design, refactoring, testing. But I don't think this is necessarily part of the job description of coach. If you had a team that was technically self-sufficient, but needed help with their process, you could coach without being a techno-whiz. You would still have to convince the propeller-heads that they should listen to you.

Consultant

XP projects don't spawn a lot of specialists. Since everyone is pairing with everyone else, and the pairs float around so much, and anyone can accept responsibility for a task if they want to, there is little chance that dark holes will develop where only one or two people understand the system.

This is a strength, because the team is extremely flexible, but it is also a weakness, because from time to time the team needs deep technical knowledge. The emphasis on simplicity of design reduces the occurrence of the need for the pointy hat, but it will absolutely happen from time to time.

When it does, the team needs a consultant. Chances are, if you are a consultant you won't be used to working extreme. You are likely to view what the team does with a certain amount of skepticism. But the team should be extremely clear about the problem they need to solve. They will be able to provide you with tests to show exactly when it has been solved (in fact they will insist on the tests).

What they won't do is let you go off and solve the problem by yourself. If the team needs deep technical knowledge in an area once, they are likely to need it again. Their goal is to get you to teach them how to solve their own problem. So, one or two team members will sit with you as you solve the problem. They will likely ask you lots of questions. They will challenge your design and assumptions, to see if they can't find something simpler that will still work.

And when you are done, they will most likely throw away everything you have done and do it over themselves. Don't be insulted. They do this to themselves every day a little bit, and probably once a month they throw away a day's work.

Big Boss

If you're the big boss, what the team needs most from you is courage, confidence, and occasionally insisting that they do what they say they do. It is likely to be difficult for you to work with the team at first. They are going to insist on you checking up on them frequently. They are going to explain the consequences of changes in the situation. Like, if you don't get them the new tester they asked for, they will explain exactly what they think that will do to the schedule. If you don't like their answer, they will invite you to reduce the scope of the project.

Coming from an XP team, this constitutes honest communication. They aren't whining, really they aren't. They want you to know as soon as possible when things are going differently than the plan said they would go, so you have as much time to react as possible.

The team needs you to have courage, because what they do will sometimes seem crazy, especially if you have a background in software development. Some of the ideas you will recognize and approve of, like the strong emphasis on testing. Some don't seem to make sense at first, like pair programming being a more productive way to program and constantly refining the design being a lower risk way to design. But watch and see what they

 Copyright © 1998 Kent Beck, All rights

produce. If it doesn't work, you can step in. If it does, you're golden, because you will have a team that is working productively, that keeps its customers happy, and that does everything they can not to ever surprise you.

This doesn't mean that the team won't screw up from time to time. They will. You will look at what they are doing, and it won't make sense to you, and you'll ask them to explain it, and the explanation won't make sense. That's when the team is relying on you to make them stop and take a look at what they are doing. You got to your position for a reason. The team wants to put that skill to work for them when they need it. And, frankly, to keep it out of the way when they don't need it.

20-80 Rule

Software developers are used to dealing with the 80-20 rule- 80% of the benefit comes from 20% of the work. XP makes use of this rule itself- put the most valuable 20% of functionality into production, do the most valuable 20% of the design, rely on the 80-20 rule to defer optimization.

The 80-20 rule is not universal, however. Not all systems obey it. For the 80-20 rule to apply, the system in question must have controls that are relatively independent of each other. For example, when I tune the performance of a system, each possible place I could tune generally has little effect on the other places I could tune. I never find myself in a situation where I tune the biggest time hog, only to find that because of that tuning I can't tune the next one. So, in a system with independent controls, some of them are bound to be more important than others.

Why do I tell this story? Because I get a lot of people who look at XP and say, „We do that. We do absolutely everything you say. Not the tests. But we do everything you say." And that's not extreme. They aren't getting the benefits of what I am talking about. Not even close.

How could this be? Well, the 80-20 rule must not apply. Then what does apply?

Ward Cunningham tells the story a book that got him onto advanced ski slopes called „Skiing Like the Racers". Half the book is about tuning your boots, getting them set up just right so you can feel the mountain and be on balance. And then the book says, „but you will only see 20% of the improvement when you have done 80% of these exercises". It goes on to explain that there is a huge difference between being on balance and being off balance. If you are a little off balance, you may as well be a lot off balance. And it is a host of little factors, like getting your boots just right, that allow you to be right on balance. If any one is off, you'll be off.

I think that XP is like that. I have tried to highlight throughout this book how the practices and the principles work together with each other to create a synergy that is greater than the sum of the parts. It's not just that you do testing, it's that you are testing a simple system, and it got simple because you had a pair programming partner who challenged you to refactor and reminded you to write more tests and patted you on the back when you got rid of complexity and...

This poses a dilemma. Am I saying that XP is all or nothing? Do you have to follow these practices to the letter or risk not seeing any improvement? What if I'm wrong? What if your intuition about some of the crazy advice in this book is right? What if you really do know better for your organization?

If I'm right, if XP is kind of all or nothing, then you will have to take a risk to get the full benefit. But here's how I'd hedge my risk if I were you. I would adopt one or two of the practices. Unit testing and the planning game are two that almost always make sense. Try them out for a while. See if you get advantage from them. Then imagine what it would be like if you got five times or ten times the advantage. I was right about those first couple. Might I be right about the rest of the story.

Putting your toe in makes sense from a lot of perspectives. You will be gain confidence that you can read this book and put it into practice. Your development process will get enough better that you will have a little slack with which implement the rest of the practices.

Or not. You may try the planning game and discover that it just doesn't work with your kind of boss or your kind of customers or your kind of company. Then you'll know not to try the rest of it, because it is sure not to work.

On the other hand, you may be crazy enough to try this all at once. If you do, be prepared for a pretty rough road. Your fears are bound to surface, and at the worst times. You are sure to slip up in writing tests. There will be things you just can't figure out how to put into practice. Be patient with yourself. You have taken a large measure of what society says makes you valuable and thrown it away, with the hope that you can replace it with something even more valuable. That can't possibly be easy.

 Copyright © 1998 Kent Beck, All rights

Adopting XP

How do you adopt XP? Does this belong here or in the other book? Could I answer the question „why do you adopt XP“ or „how does the XP adoption process work“?

There is a simple answer to the question, „Okay. I'm convinced. How do I start?“ Just start doing it. Everything. Start with the stories. Continue on through refactoring, testing, pair programming. Ship release after fabulously successful release.

Wait. Isn't one of the principles „ Incremental Change“? Jumping right into a whole new development method isn't incremental. It's catastrophic. It would be an enormous risk to take on a new development culture and a bunch of new techniques at the same time. Chances are you would crash and burn. If you were in a desperate enough situation, jumping in with both feet might be your best option, but I wouldn't recommend it if there was any other reasonable option.

Okay, so what is the incremental approach to adopting XP? Well, you would start with one technique, say unit testing, and adopt it. You would get the team together regularly to talk about their experience of testing, so the team could be unifying its vision of its own process. Once unit testing was in place, you could adopt refactoring. Then pair programming. Then the Planning Game. Then incremental design. And on and on until you had your own version of all of the practices and your own version of XP.

The problem with the entirely incremental approach is two-fold. First, it takes a long time. If you really adopt one practice at a time you might spend two years getting to XP. Now, two years isn't really a terribly long time to effect a change in culture. However, lots of folks are judged on much shorter time scales.

The second problem with incrementally adopting XP is the 20-80 rule. You will certainly see some benefit from unit testing, some from refactoring, some from evolutionary design, some from the planning game, some from functional tests. However, if the chapter on the 20-80 rule is to be believed (I believe it, but then I wrote it), you won't see dramatic improvement until you're nearly done. And dramatic improvement is the only rational justification for making such a large change as adopting XP.

So, there must be some middle way between one-toe-at-a-time and head-first. Here are the elements of a strategy (not THE strategy, just A strategy):

1. Find a coach
2. Learn a few practices
3. Then dive in

I suggest that you first engage a coach, someone who has been there before and applied XP. Unfortunately, at the time of this writing, this limits the coaching population to about 10 people worldwide. This will change quickly (I sincerely hope this will change quickly- if not, ignore everything I said here). If you can't hire a coach, grow your own. Designate your best, most communicative developer to stop writing code and learn about and experiment with XP. Now, this newly minted coach won't be able to write any code- they'll be busy learning about coaching- but if they can make a big enough difference for everyone else, you won't mind.

If your coach has only read about XP, never participated in an extreme project before, you will have a lot to learn together. Actually, that's a polite way of saying that you will have to watch the coach carefully. Remember, the job of the coach is to encourage consensus and alignment, not to impose. A new coach is likely to be insecure at first. If they are saying something that doesn't make sense to you, it is likely because it doesn't make sense.

While you are looking for (or growing) a coach, get the team together to learn the practices. Start with unit testing or refactoring or coding standards or pair programming. Do everything else the way you already do it, but add the practice. Make sure you spend time talking about the practice, too. (No offence intended, but I have noticed that developers generally need practice communicating). Get one practice comfortably in place before you start on the next one.

There will come a point in this process where you are receiving benefit from the practices that you have already adopted and your coach will be ready to start coaching in earnest. If this point in time also coincides with the beginning of development for a new release, so much the better. So, all the pieces are in place- confidence that you can do better, leadership, a clear goal. Then dive in. It will be scary. But this is the time to do it. And if you do it, you can begin reaping the benefits of the 20-80 rule.

 Copyright © 1998 Kent Beck, All rights

However, if the team just can’t get together on adopting any new practice, or you can’t find a coach, forget it. Pick and choose from among the practices that make sense to you. Encourage other team members to do the same. But the time may just not be ripe for your team to adopt XP. Or it may never be ripe. That’s fine. XP is certainly not for everybody. There’s no stigma in not being able to do XP, or rather for XP not to be right for your team.

I have talked to lots of teams that claim that they are already doing XP. None of the practices are new and unique, so I’d be shocked if XP was the first time anyone ever unit tested or refactored. However, XP is an unlikely enough combination that I would be surprised if anyone had done it in exactly this way before. Certainly none of those teams I talked to that claim to already be doing XP were doing what I would call XP. There is always a „but“- „We’re doing exactly what you talk about. Exactly. Everything. Not the testing. But everything else. It’s exactly XP.“

If your team is in this position- you seem to already being doing pretty much everything in this book- congratulations! Take a careful look at each of your practices. You may have the opportunity to quickly get a large improvement by adopting one new practice or modifying one of your existing practices. Or you may already be doing software better and XP has nothing to teach you.

I wish I had better advice. At this point, few enough teams have adopted XP that there just isn’t enough experience to give you better advice about adopting it. And this means that XP is a risk. In two or three years, I expect the transition of teams to XP to be well understood and fairly routine (if still painful). If you can wait a couple of years, by all means do it. Or adopt a couple of new practices and then wait and see. But I’m assuming that if you have gotten this far into the book, it is because you need big medicine. So, get your toes used to the water. Then dive in. Preparation and commitment. Courage and skill.

Retrofitting XP

Adopting XP with a new team is a challenge. Adopting it with an existing team and existing code base is even harder. You have all the existing challenges- learning the skills, coaching, making the process your own. You also have the immediate pressure of keeping production software running. The software is unlikely to be written to your new standards. It is likely to be more complex than it needs to be. It is unlikely to be tested to the degree you would like. On a new team, you can select only those people who are willing to try XP. An existing team is likely to have some skeptics. And on top of that, all the desks are already set up and you can’t even pair program.

You will have to take more time to retrofit XP on a project than you would to adopt on the equivalent new team. That’s the bad news. The good news is that there are some risks that a greenfield XP development has to face that you won’t have to face. You will never be in the risky position of thinking you have a good idea for software but not really knowing. You will never be in the risky position of making lots of decisions without the immediate and brutal feedback you get from real customers.

I talk to lots of teams that say, „Oh yes, we are already doing XP. Everything but that testing stuff. And we have a 200 page requirements document. But everything else is exactly how we do it.“ That’s why this chapter is set up the way it is. If you are already doing the same practiced advocated by XP, you can ignore it. If there is some new practice that you want to pick up, check out the section devoted to that practice.

So, how can adopt XP with an existing team on software that is already in production? You will have to modify the adoption strategy in the following areas:

- Testing
- Design
- Planning
- Management
- Development

Testing

Testing is perhaps the most frustrating area when you are shifting existing code to XP. The code you wrote before you had tests is a little scary. You never know quite where you stand. Will this change be safe? You’re not sure.

 Copyright © 1998 Kent Beck, All rights

As soon as you start writing the tests, the picture changes. You have confidence in the new code. You don't mind making changes, in fact, it's kind of fun.

Shifting between old code and new code is like night and day. You will find yourself avoiding the old code. You have to resist this tendency. The only way to gain control in this situation is to bring all the code forward. Otherwise you ugly things will grow in the dark. You will have risks of unknown magnitude.

It is tempting in this situation to just go back and write the tests for all the existing code. I wouldn't do this. Instead, write the tests on demand. When you need to add functionality to untested code, write tests for its current functionality first. When you need to fix a bug, write a test first. When you need to refactor, write tests first.

What you will find is that development goes slowly at first. You will spend much more time writing tests than you do in normal XP. However, the parts of the system that you visit all the time, the parts that attract attention and new features, will quickly be thoroughly tested. Quickly, the parts of the system that are used most will feel like they were written with XP.

Design

The picture transitioning to XP design is much like testing. You will notice that the new code feels completely different than the old code. You will want to fix everything at once. Don't. Take it a bit at a time. As you add new functionality, be prepared to refactor first. You are always prepared to refactor first XP development, but you will have actually have to do it more often as you are transitioning to XP.

Early on in the process, have the team identify some large scale refactoring goals. There may be a particularly tangled inheritance hierarchy, or a piece of functionality scattered across the system that you want to unify. Set these goals, put them on cards, and display them prominently. When you can say the big refactoring is done (it may take a year or more of nibbling), have a big party. Ceremoniously burn the card. Eat and drink well.

The effect of this strategy is much like the effect of the demand-driven testing strategy. Those parts of the system that you visit all the time in your development activities will soon feel just like the code that you are writing now. The overhead of extra refactorings will soon fade.

Planning

You will have to convert your existing requirements information to story cards. You will have to educate your customer about the rules of the game. The customer will have to decide what constitutes the next release.

The biggest challenge (and opportunity) of switching to XP planning is teaching your customer how much more they can get from the team. They probably haven't had the experience of a development team that welcomes requirements changes. It takes a while to get used to how much more the customer can get from the team.

Management

One of the most difficult transitions is getting used to XP management. XP management is a game of indirection and influence. If you are a manager, you will probably catch yourself making decisions that should be made by developers or customers. If you do, don't panic. Just remind yourself and everyone else present that you are learning. And then ask the right person to make the decision and let you know what they decided.

Developers suddenly confronted with new responsibilities are unlikely to do a great job immediately. As a manager, you must be careful during the transition period to remind everyone of their rules.

The feeling will be a little like transitioning the design or the tests. At first, it will feel awkward. You will know that you aren't going at full speed. As you pay attention to the situations that occur day in and day out, you (and the developers and the customers) will learn how to handle them smoothly. You will quickly begin feeling comfortable with your new process. From time to time, however, a situation will arise that you haven't „done extreme" before. When this happens, take a step back. Remind the team of the rules, and values and principles. Then decide what to do.

One of the most difficult aspects of managing a galloping shift to XP is deciding that a team member isn't working out. In this situation, you are always better off without them. And you should make the change as soon as you are sure the situation isn't going to get any better.

 Copyright © 1998 Kent Beck, All rights

Development

The first thing you have to do is get those desks set up right. No kidding. Re-read the material about pair programming. Set up your desks so two people can sit side by side and shift the keyboard back and forth without having to move.

On the one hand, you should be more rigid about pair programming than you would normally have to be. Pair programming may be uncomfortable at first. Force yourself to do it, even if you don't feel like it. On the other hand, take a break sometimes. Go off by yourself and code for a couple of hours. Throw away the results, of course, but don't destroy your joy in programming just to be able to say you paired for 30 hours in one week.

Nibble away at the testing and design problems. Bring all the code you touch up to your agreed-on coding standards. You will be surprised how much you can learn from even this simple activity.

In Trouble?

Some of you reading this will have an existing team but your software isn't yet in production. Your project may be in a lot of trouble. XP may look like a possible salvation.

Don't count on it. If you had used XP from the beginning, you might (or might not) have avoided your current situation. However, if switching horses midstream is tough, switching from a drowning horse is ten times as tough. Emotions are going to be high. Morale is going to be low.

If the choice is switch to XP or be fired, first realize that your chances aren't very good. Under stress you revert to old habits. And you already have lots of stress. So your chances of successfully making the switch are drastically reduced. So make a more modest goal for yourself than saving the whole project. Take it one day at a time. Celebrate how much you can learn about testing, or managing indirectly, or how beautiful you can make a design, or how much code you can delete. Perhaps enough order will emerge from the chaos that you won't mind coming in the next day.

However, if you're going to do switch a troubled project to XP, don't do it piecemeal. Half measures are going to leave everybody in more or less the same state that they were before, and we know that doesn't work.

Carefully evaluate the current code base. Would you be better off without it? If so, flush it. All of it. Have a big bonfire and burn the old tapes. Take a week off.

What Makes XP Hard

When people hear me talk about XP they say, „But you make it sound so simple." Well, that's because it is simple. It doesn't take a Ph.D. to contribute to an XP project (in fact, the Ph.D.s sometimes have the most trouble).

What XP isn't is it isn't easy.

Let's run that again. XP is simple, but it isn't easy? Exactly. The practices that make up XP can be learned by anyone who has convinced someone else to pay them to program. That isn't the hard part. The hard part is putting all the pieces together, and then keeping them in balance. The pieces tend to support each other, but there are many problems, concerns, fears, events, and mistakes that can throw the process off balance. The whole reason you would „sacrifice" a senior technical person to be coach is because the problem of keeping the process on balance is so difficult.

I don't want to frighten you. Not more than necessary. Most software development groups could execute XP (see the next chapter, When You Shouldn't Try XP for exceptions). But only if they take it seriously.

Here are some of the things I've found hard about XP, both when I am applying it for my own code and when I am coaching teams adopting it. I don't want you to go borrowing trouble, but when the transition to XP is going rough (and there will be days, I promise), you should know that you are not alone. You're having a hard time because what you are doing is hard.

It's hard to do simple things. It seems crazy, but sometimes it is easier to do something more complicated than to do something simple. This is particularly true when you have been successful doing the complicated thing in the past. Learning to see the world in the simplest possible terms is a skill and a challenge. The challenge is that you may have to change your value system. Instead of being impressed when someone (like you, for instance) gets

 Copyright © 1998 Kent Beck, All rights

something complicated to work, you have to learn to be dissatisfied with complexity, not to rest until you can't imagine anything simpler working.

It's hard to admit you don't know. I make my living by knowing stuff that other people don't know. This makes it a personal challenge to adopt XP, a discipline based on the premise that you can develop only as fast as you learn. And if you're learning, that means you didn't know before. It will be frightening to go to the customer and ask them to explain what are to them the most elementary concepts. It will be frightening to turn to your programming partner and admit that there are basic things about computer science that you frankly never quite got when you were in school. Or that you've forgotten since then.

It's hard to collaborate. Our whole education system is tuned to individual achievement. If you work with someone on a project, the teacher calls it cheating and you are punished. The reward systems in most companies, with individual evaluations and raises (often cast as a zero sum game), also encourages individual thinking. You will likely have to learn new people skills, interacting as closely with your team as you will in XP.

It's hard to break down emotional walls. The smooth running of an XP project relies on the smooth expression of emotions. If someone is getting pissed off and not talking about it, it won't be long before the team starts to under perform. But we learn to separate our emotional lives and our business lives. But the team can't function effectively if communication is not kept flowing, fears are acknowledged, anger discharged, joy shared.

If this makes XP sound like a Big Sur, touchy-feely, brandy-sipping, hot-tubbing experience, well, I don't think of it that way. I've tried to developer software pretending I didn't have any emotions and demanding distance from my co-workers. It didn't work. I talk about how I feel and I listen when others talk about how they feel, and the process goes much more smoothly.

The practices are so sideways to what we hear and say and maybe even have been successful with in the past. One of the big difficulties is just how contrary XP sounds. I'm often afraid when I first meet a new manager that I will sound radical or crazy or impractical. However, I don't know a better way to develop software, so I eventually get over it. Be prepared to have people react strongly when you explain XP, however.

Little problems can have huge effects. I think of the checks and balances of XP as being quite robust, the process able to tolerate lots of variation. I have found that small things can often make huge differences. Once on the Chrysler payroll project the team was having trouble hitting their estimates. Iteration after iteration a story or two would slip out. I'm embarrassed to say it took me three or four months to diagnose the problem. I heard someone talking about „First Tuesday Syndrome." I asked what that was. „The feeling you get the day after the iteration planning meeting when you come in, look at your stories, and realize you have no idea how to implement them in the estimated time."

I had originally specified the process as:

1. Sign up for tasks
2. Estimate your tasks
3. Rebalance if someone is overcommitted

The team had wanted to avoid the third step, so they changed the process to:

1. Estimate tasks collectively
2. Sign up for tasks

The problem was that the person who accepted responsibility for a task didn't own the estimate. They would come in the next day and say, „Why is this going to take three days? I don't even know what is involved." You may be able to guess that this isn't the most productive state for a developer. People were losing a day or two to First Tuesday Syndrome every iteration. No wonder they weren't hitting their targets.

I tell this story to illustrate that small problems with the process can have large effects. I don't mean to say that you should do everything exactly as I say or you'll be sorry, buster. You still have to accept responsibility for your own process. But what makes XP hard is exactly this- by accepting responsibility for your own development process, you are accepting responsibility for being aware and fixing it when there is a problem.

The steering metaphor goes sideways to the car-pointing metaphor prevalent in lots of organizations. A final difficulty, and one that can easily sink and XP project, is that the steering metaphor is just not acceptable in many company cultures. Early warning of problems is seen as a sign of weakness or complaining. You will have

 Copyright © 1998 Kent Beck, All rights

to have the courage of your convictions when your company wants you (sometimes very badly) to act contrary to the process you have chosen for yourself.

When You Shouldn't Try XP

Let me make it clear that I am talking about the whole ball of wax here. There are practices in XP that are a good idea regardless of what you think about the whole picture. You should do them. Period. Testing is a good example. The Planning Game probably works, even if you spend more time on estimation and up-front design. As the 20-80 rule states, there is a huge difference between all of it and not all of it.

And all of XP, frankly, is not a story that can be told everywhere. XP is not a story that should be told everywhere. There are times and places and people and customers that would explode an XP project like a cheap balloon. And it is important not to use XP for those project, as important not to use XP where it is bound to fail as it is important to use it where it provides real advantages. That's what this book is supposed to be all about- deciding when to use XP and when not to use XP.

That said, I won't be telling you „Don't use XP to build missile nosecones." I haven't ever built missile nosecone software, so I don't know what it is like. So I can't tell you that XP will work. But I also can't tell you that it won't work. If you write missile nosecone software, you can decide for yourself whether XP might or might not work.

I have failed with XP enough to know some of the ways it certainly doesn't work, however. Take this as a list of places (and times and customers and people) I know that don't do well with XP.

The biggest barrier to the success of an XP project is culture. Not national culture, although that has an effect, too, but business culture. Any business that operates on the „pointing the car" paradigm is going to have a real rough time with a team that insists on steering.

A variant of „pointing the car" is the big specification. If a customer or manager insists on a complete specification or analysis or design before they begin the small matter of programming, then there is bound to be friction between the team's culture and the customer or manager's culture. The project may still be able to be successful using XP, but it won't be easy. You will be asking the customer or manager to trade a document that gave them a feeling of control for a dialog (the Planning Game) that requires them to be continuously engaged. That can be scary to a person who is already overcommitted.

On the other hand, I worked with a bank customer that just loved big piles of paper. They insisted throughout the project that we would have to „document" the system. We kept telling them that of course, when the customer wanted to make the tradeoff to get less functionality and more paper, we would be glad to oblige. We heard about this „documentation" for months. As the project went forward, and it became clear just how valuable the tests were for keeping the system stable and for communicating the intended use of the objects, the pronouncements about documentation became quieter and quieter, although they were still there. In the end, what the development manager said he really wanted was a four page introduction to the main objects in the system. As far as he was concerned, anyone who couldn't find the rest of what they needed to know from the code and the tests had no business touching the code.

Another culture that is not conducive to XP is one in which you are required to put in long hours to prove your „commitment to the company". You can't execute XP tired. If the amount produced by a team working at top speed isn't enough for your company, then XP isn't your solution. Overtime on an XP project are a sure sign that something is wrong with the process, and you'd better fix what's wrong.

Really smart developers sometimes have a hard time with XP. Sometimes the smart people have the hardest time trading the „Guess Right" game for close communication and continuous evolution. I would be very wary now if I came into a room full of Ph.D. computer scientists and was told they wanted to work extreme. If I didn't run away immediately, I would at least want to bring a couple of stout sticks with me.

Size clearly matters. You couldn't run an XP project with 100 developers. Nor 50. Nor 20, probably. 10 is definitely do-able. Around 3 or 4 developers you can safely shed some of the practices that are focused on developer coordination, like the Iteration Plan. The amount of functionality to be produced and the number of people producing it don't have any sort of simple linear relationship. If you have a big project, you might want to experiment with XP- try it for a month with a small team- to see how fast you might be able to develop.

You shouldn't use XP if you are using a technology with an inherently exponential cost curve. For example, if you are developing the Nth mainframe system to use the same relational database and you aren't absolutely sure the database schema is exactly what you need, now and forever, you shouldn't use XP. XP relies on keeping the

 Copyright © 1998 Kent Beck, All rights

code clean and simple. If you make the code complicated to avoid modifying 200 existing applications, pretty soon you will lose the flexibility that brought you to XP in the first place.

Another technology barrier to XP is environments where there is a long time needed to gain feedback. For example, if your system takes 24 hours to compile and link, you will have a hard time integrating, building, and testing several times a day.

I've seen environments where it was simply impossible to realistically test software- you are in production on a million dollar machine that is operating at capacity and there simply isn't another million dollars around. Or there are so many combinations of possible problems that you can't run any meaningful test suite in less than a day. In this case, you are absolutely right to trade thought for testing. But at that point you aren't doing XP any more. When I programmed in that sort of environment, I never felt free to evolve the design of the software, I had to build in flexibility up front. You can still build fine software this way, but you shouldn't be using XP to do it.

Remember the story of the senior people with the corner offices? If you have the wrong physical environment, XP can't work. One big room with little cubbies around the outside and powerful machines on tables in the middle is about the best environment I know. Ward tells the story of the WyCash project where they had individual offices. However, the offices were big enough for two people to work in comfortably, so when folks wanted to pair, they would just go to one office or the other. If you absolutely can't move the desks, or the noise level prevents conversation, or you can't be close enough for serendipitous communication, you won't be able to execute XP at anything like its full potential.

What doesn't work for sure? If you have developers on two floors, forget it. If you have developers widely separated on one floor, forget it. Geographically separated- you could probably do this if you really had two teams working on related projects with limited interaction, but I would still want to have started them out as one team which grew and split.

Finally, there is absolutely no way you can do XP with a baby screaming in the room. Trust me on this one.

XP Business Opportunities

One of the things that hindered the software patterns community is that there is no clear business model. You write the patterns, you publish them (maybe making a few bucks in royalties, maybe not), and that's it.

XP is different. If XP is successful, there will be lots of ways of making money from it:

Tools- There will be a big market for tools for refactoring code and data. Object databases have a big advantage in schema evolution over earlier technology. Development environments that support refactoring will have a competitive advantage.

Consulting- There will be a market for coaches. I have coached teams as much as full time (I think this is overkill) and as little as three days a month (with frequent phone calls and email and a person I trusted there every day). There will also be a genuine consulting market for XP troubleshooters who can come in and quickly give a team ideas for being more effective (in the old style I would have said „fixing the team“, but the team owns their own process, so no one else can possibly fix them).

Training- There will be a market for training the skills needed to execute XP- the Planning Game, pair programming, refactoring, testing, coding for communication. There will also be more specialized training for the various roles- how to be a good customer, tester, tracker, or coach.

Role Specialists- There will be a market for specialists in the roles. It is tough to be a good XP developer or tester or tracker. People with experience in real XP projects who are willing to do it again, and not succumb to the lure of coaching or the lecture circuit (to say nothing of those idiots that write books), will be at a premium.

Change-driven business- The real business opportunity in XP is for a business person that realizes that they can make the flexibility of their software development process into a competitive weapon. The ultimate opportunity is not making better software, but making better business.

Insourcing

There is another side to the principle that says that responsibility can only be accepted, it can never be given. This principle also makes it clear that there is a kind of „Conservation of Responsibility“ law at work in the

 Copyright © 1998 Kent Beck, All rights

universe. Important responsibilities that are not accepted by anyone end up flying around and hurting people and organizations.

This becomes particularly important in outsourced development. The typical outsourcing arrangement sees the customer pass off responsibility for the system to a „expert technology provider“. The provider accepts responsibility for developing the first release of the system. So far, so good.

Assuming the provider gets as far as delivering the system, what happens next shows the danger of unaccepted responsibility. The provider delivers the system. The customer accepts responsibility for it.

Wait a minute. The customer didn't really accept responsibility for the system. They only accepted responsibility for operating the system as it was delivered. But since software is never done, who accepted responsibility for further development? You'd have to say that the customer did, since they own the software, and they are financial responsible for further development. But while the company may have accepted responsibility, the people haven't. They don't have any idea how to proceed with development.

The customer can build all kinds of clauses into the contract insisting that the provider deliver voluminous documentation, train customer employees, and build with fabulous future flexibility in mind. The simple fact is that the provider's goals and the customer's goals are directly at odds. The provider wants to get done and get paid with as little effort and risk as possible. The customer wants the most functionality and the greatest possible preparation for the future.

We've already established that the provider has accepted responsibility for the system, so guess whose goals win? The provider holds the whip hand in the relationship.

Some people have observed the problems with outsourcing and given up on the idea of outsourced fixed-price contracts altogether. I think this is overreaction. There is definitely a problem to solve- companies are not capable of writing the systems critical to their business, and providers have the technical know-how to do build those systems. However, the nature of the contracts has to be changed for the relationship to work.

Providers‘ and customers‘ interests are not aligned. How would XP address this puzzle? Back to the principles. Small initial investment and rapid feedback suggest that we contract a series of releases, not just a single system. This pushes the provider towards paying attention to quality, since they know they have to live with the consequences of their actions.

Embracing change suggests that neither provider nor customer should get too excited if the content of future releases changes. This is a little like the commitment schedule at the next higher scale of time. The beginning of each release provides a natural point at which to re-estimate and reschedule the remaining stories.

Concrete experiments suggests that the customer only pay for working software. This will push the release cycle shorter, which reduces the risk for both parties.

Open, honest communication is a pre-requisite for this process. As the provider learns what is easy and what is hard, the customer will get better at getting the most value in the available time. As the customer learns what is most valuable about the system, they can easily feed that information back into future releases.

Accepted responsibility suggests that the customer not give up all responsibility for the system. The customer always accepts responsibility for the system as a whole. The provider accepts responsibility for technical leadership and for providing the bulk of the developers at first.

Incremental change suggests that through the releases, the customer should gradually take more and more development responsibility. The provider might employ 75% of the people on the team during the development of the first release, dropping to 50% for the second release, 25% for the third, and 10% for the fourth.

Travel light suggests that defensive-oriented deliverables, like extensive paper documentation, be eliminated. This will help ensure that even though the release cycle is short, significant business functionality is released each time. It will also reduce the size of the team needed, which reduces the scope of the communication problems.

Quality work suggests that even in an outsourced project, people still want to do well. In an atmosphere of developing trust, the developers will be given the time to do a good job of only those things that are needed.

Living software also suggests that the development be divided into releases. Since the software will continue to grow and change after the provider is no longer involved in the project, the project should begin on that basis.

 Copyright © 1998 Kent Beck, All rights

The business model that emerges from these observations looks a little like outsourcing, but with some critical differences.

There is a fixed price contract for fixed deliverables.	You expect to learn and change the deliverables over time. The time and cost of the contract don't change, unless the contract no longer makes sense.
Development is performed by the provider's people.	Release after release, development responsibility is gradually transferred to developers directly employed by the customer.
Development uses the latest technology.	But not so far out on the bleeding edge that no one understands the resulting system.
Responsibility for further development is eventually turned over to the customer.	The provider comes back for periodic reviews.
The provider specifies the development process.	The process is simple enough that it can be run by the customer (the Planning Game, tracking metrics).

Insourcing has advantages for the provider:

- Predictable revenue stream
- Gradual shift from one revenue source to another
- Long term relationship with the customer

Of course, the provider would have to work awfully hard use an insourcing arrangement to put useless bodies on a project and suck arbitrary amounts of money from a customer. Since the customer is always there, and always contributing, if the project needs more resources the customer can decide who should provide them.

Insourcing has advantages for the customer:

- Predictable costs
- Reduced risk because they are always in charge of the process
- Reduced risk because they gradually learn to continue development without constant help

The biggest problem with insourcing is that it uses the steering metaphor for a business relationship. Many businesses like to treat their relationships as if they work on the pointing the car metaphor. Both the customer and provider in an insourcing contract must commit to spend effort maintaining the contract itself, as well as the software to be delivered on that contract.

Completing the Circle

I used to be a big fan of meta-circular thinking. I was terribly impressed when someone could implement a programming language in itself or apply concepts at scales from minutes to years.

After a while, however, I became disenchanted with such thinking approaches. It seemed to me that most people resorted to piling abstraction on abstraction to avoid finding out if their ideas worked in any concrete context. I directed my thoughts and my strategy in quite the opposite direction. I wanted to talk about the real stuff, and just the real stuff. Abstraction was for propeller headed scientists.

My personal take on patterns was very much in this mode. Patterns certainly are an abstraction from the programs they describe. However, they are an abstraction born of experience and programs.

I recently gave myself permission to start using abstraction and circular descriptions again. I would like to say that I had matured to the point where I was no longer scared of the consequences. In fact, I was desperate. I was unable to describe XP any other way. Without first describing the principles and values, then showing how they apply at all levels of scale within XP, I just sounded crazy. I had no choice but to describe the first principles from which I operated, then show how all these startling conclusions grew naturally. It was that or get thrown into the loony bin.

 Copyright © 1998 Kent Beck, All rights

Having come this far in the rediscovery of the power of abstraction, I found that I could go even further. I could think about how to apply XP to itself. The same principles that apply to developing software systems also apply to developing development processes. Let's look at what some of the principles mean in the context of deploying and evolving XP.

Teach learning- We won't set out exactly what to do in every circumstance. Some of the practices will be the same everywhere. Mostly, however, we will focus on teaching people how to learn the answers for themselves.

Small initial investment- I didn't wait until XP was perfectly worked out and then write a single large tome explaining everything. I wrote this book. I expect to follow it up shortly with a practical, how-to book that talks a little bit about all the most important topics. Then we will see where people have problems. Where they have problems, we'll do some experiments, teach some courses, then write another little book.

Play to win- There is a risk to publishing about XP now. We don't have academic studies behind us, a long and diversified track record. XP at the moment is vulnerable to all sorts of criticism (some of it no doubt justified). However, waiting until all the planets are exactly aligned before saying anything publicly would be playing not to lose. The play to win strategy is to say what we know as clearly as we can as soon as we are confident, then planning on learning along with everyone else.

Rapid feedback- I experimented with all the XP strategies on myself before I taught them to anyone else. Then I tried them on a small scale with small groups, paying attention to what worked and what didn't. I encouraged other people (for example on Ward Cunningham's Wiki Wiki Web) to experiment, too, and to report their results. Now we have a double handful of projects that have tried XP in some form. We have already learned a lot. I expect to continue this process with my own coaching, and by forming a community of XPers who can share their experiences.

Concrete experiments- This book makes XP sound quite, well if not logical, at least it has some sort of theory behind it. The basic values were always there, but most the strategies grew out of concrete situations. XP comes from experience with real projects putting software into production for real customers. The theory came along later as a way of explaining and refining.

Assume simplicity- I assumed that simple, understandable strategies would work for the major areas of effort in software development- planning, coding, testing, designing, integration. None of the strategies would impress experts in the area. But by assuming that simple strategies will work, XP brings high-performance software development into reach for blue-collar programmers.

Open, honest communication- I have tried to be clear here about what I have experienced, what I know in my heart to be true, what I have observed, and what I merely suspect. If this results in less than the perfect confidence of some books on software development, I have to accept that.

Work with people's instincts, not against them- I wouldn't have chosen to write this book first. My inclination was to write the how-to book first. However, since fear of such a superficially radical approach is the major barrier currently to the acceptance of XP, I wrote this book to address that fear.

Travel light- Again, I didn't write the 700 page XP Bible. I developed strategies for the activities that I found to be problems. I wrote down the essential information about those strategies. As XP continues to evolve, there won't be a huge problem with splinter groups, „I do East Coast XP, you pig," because there is little enough written that everyone knows that everyone does it a little different.

Quality work- I have done my best with all the strategies in XP. In particular, I haven't compromised any of the strategies to make them easier to swallow. I am learning how to communicate them in less startling ways, but that isn't the same as compromising the content. The result is a message that I am proud to communicate. I don't know how better to develop software than this.

Embracing change- My own experience of developing XP has been one of constant change. I learn so much from every project, from every talk I give, from every paper I write. At first I tried to outline „Everything You Could Want to Know", but the material changed so much that I gave up. I embraced change by figuring out the smallest book that could possibly work, by teaching XP a little differently each time I coach a team applying it, by talking to lots of folks and changing my explanations a little each time.

Thanks for coming along on this journey. I hope I'll see you writing great software. Or bombing down the slopes. Ciao!

 Copyright © 1998 Kent Beck, All rights

Annotated Bibliography

The Illusion of Life: Disney Animation- This book is interesting from an XP perspective because it describes how the team structure at Disney evolved over the years to deal with changing business and technology.

Attacks, Rommel- This book has great attitude. Find the quote about dying or charging.

The Powerhounds Guide to Alta- More great attitude. I don't like the „if you can't die, it sucks" mentality. Quote the quote about „failure of will or technique will have serious, possibly fatal, consequences."

The Timeless Way of Building- The book this book is most like, at least in spirit.

Glossary

System metaphor

Pair programming

Refactoring

Refactoring

Ideal engineering time

Load factor

 Copyright © 1998 Kent Beck, All rights