

Chapter 1

To Reviewers

You hold in your hands a draft of *Planning Extreme Programming*, to be published in Fall 2000 by Addison-Wesley. Please read and comment.

What we are interested in:

- ✧ Chapter order
- ✧ Suggestions for appropriate layout for Bob Martin's fabulous Rufus and Rupert story. We intend to intersperse bits of the stories with the chapters.
- ✧ More clever quotes to open chapters.
- ✧ Experiences with the techniques here or variations that worked or didn't work. We plan to include little sidebars here and there containing anecdotes, attributed or not as you choose. This is your chance for fame (without fortune)!
- ✧ Redundancies where we say over and over all the time the same thing.
- ✧ Out and out stupidity.

What we aren't interested in:

- ✧ Typos. We have a copy editor. If you can't help yourself, go ahead, but our time will be used more efficiently if you leave our crappy spelling and grammar to the professionals.

Known items yet to be done:

- ✧ Opening quotes for all chapters
- ✧ Preces for all chapters
- ✧ Design of front and back inside covers (suggestions welcome)
- ✧ More pictures

Send comments to planningxp@earthlink.com. We prefer annotated PDF, but we'll take straight email (note page numbers in your suggestions) or written-on paper. (Send paper to P.O. Box 128, Merlin, OR 97532 USA).

Feel free to pass on the draft to anyone who might read it and comment. We aren't terribly worried that you will use the text in its current form and not buy the book when it comes out. If you do, the mistakes we have deliberately seeded herein will come back to haunt you. Better all around if you just buy the book when it comes out, hey?

Thanks for your help,
Kent, Martin, and Bob

Copyright (c) 2000, Kent Beck, Martin Fowler, and Robert Martin,
All rights reserved

Chapter 2

Preface

This is a book about planning software projects. We are writing it mostly to project managers—those who have to plan and track the correspondence of the planning with reality. We also are writing it to programmers and customers, who also have a vital role to play in planning and developing software.

Planning is not about predicting the future. When you make a plan for developing a piece of software, development is not going to go like that. Not ever. Your customers wouldn't even be happy if it did, because by the time software gets there, the customers don't want what was planned, they want something different.

Eisenhower is credited with saying, “Plans are useless. Planning is vital.” We agree. That's why this isn't a book about plans, it's about planning. And planning is so valuable and important, so vital, that it deserves to go on a little every day, as long as development lasts. If you follow the advice in this book, you are going to have a new problem to solve every day—planning—but we won't apologize for that, because without planning, software development inevitably goes off the rails.

This isn't a book about the whole of project management. We don't cover typical project manager jobs such as personnel evaluation, recruiting, and budgeting. We have stuck to the parts of the process we know—getting everybody on the team pointed in one direction, discovering when this is no longer true, and restoring harmony.

Extreme Programming (XP) is the programming discipline (you can use the ‘m’ word if you want to, we'd rather not, thank you) we are evolving to address the problems of quickly delivering quality software, and then evolving it to meet changing business needs. XP isn't just about planning. It covers all aspects of small team software develop-

ment—design, testing, implementation, deployment, maintenance. However, planning is a key piece of the XP puzzle. (For more about XP, read “Extreme Programming Explained: Embrace Change”.)

XP develops long projects by breaking them into a sequence of self-contained 1-3 week projects. During each mini-project (iteration),

- ◇ Customers pick the features to be added.
- ◇ Programmers add the features.
- ◇ Programmers and customers write and maintain automated tests to demonstrate the presence of these features.
- ◇ Programmers evolve the design of the system to gracefully support the features.
- ◇ Programmers finish the features so they are completely ready to be deployed.

Without careful planning, the process falls apart. The team must choose the best possible features to implement. The team must react as positively as possible to the inevitable setbacks. The team must not overcommit, or they will slow down. The team must not under commit, or the customer won’t get value for their money. The job of the daily planner is to help keep the team on track in all these areas.

We come by our project planning ideas by necessity. As consultants, we typically see projects that are mostly dead. They typically aren’t doing any planning, or they are doing too much of the wrong sort. We invented these planning techniques as the least project planning that could possibly help in these situations. You will have to take what’s here and extend and adapt it to your situation. But if you have no planning, or planning that is strangling your project, what’s here works for us.

I have a cunning plan.

Baldrick

To Reviewers 1

Preface 3

Why Plan? 13

Why We Should Plan 14

What we need in planning 16

The Planning Trap 17

Rufus 19

Rufus and Rupert 19

Rupert 32

42

Fear 43

Unacknowledged fear is the source of all engineering failure. 44

The Customer Bill of Rights. 45

Programmer Bill of Rights 45

Driving Software e47

The Problem 51

Balancing Power 53

Top Down Overview 57

Bottom Up Overview w61

Balloon Story 63

Too Much to Do 63

Not "Not Enough Time" 64

Cost 67

Four Variables 67

Quality 69

Time and Scope e70

Shopping For Stories 71

Yesterday's Weather 73

The Story 74

How it works 74

Scoping a Project 75

Making the Big Plan 77

What, Me Worry? 78

Release Planning 79

Who does Release Planning 81

How Stable is the Release Plan? 81

How Far in Advance Do You Plan? 81

How do you Keep the Release Plan? 82

How much can you put into a release? 82

Release Planning Chapters 83

Short Releases 85

Release Planning Variations 85

Long Releases 86

Small Stories 86

86

Writing Stories 87

Principles of Good Stories 88

Feedback from Estimation 90

Prioritizing User Stories 90

Sizing User Stories 91

Testability 91

Splitting user stories 92

User Story Adornments 93

The story writing process 93

When are you done writing user stories? 94

Disposition of user stories 94

Examples of Stories 97

Estimation 99

Estimating the Size of a Story 100

Estimating How Much You can do in an
Iteration 101

The Meaning of Ideal Time 102

Improving your Estimates 104

Ordering the Stories 105

Business Value 106

Technical Risk 108

Worst Things First 108

Performance 109

Negotiating between the two 110

Example Release Plan 111

Measuring Velocity 115

Release Planning Events 115

Changing the Priorities of Stories 116

Adding a story 116

Rebuild the Release Plan 116

Making the First Plan 119

The First Plan 119

Choosing your Iteration Length 120

Iteration Planning 123

Never slip the Date 124

Listing the tasks for an iteration 127

Iteration Planning Meeting 127

Technical Tasks 128

Measuring the velocity of a developer 129

Signing up and estimating Tasks 129

Scut Work 131

Too much to do 131

Too Little To Do 132

Example Iteration Plan 133

Iteration Progress Check 135

Tracking an Iteration 135

When a Programmer finds they aren't going to make it 136

When a Programmer has Extra Time 137

Finding you have Too Much to Do 138

Example Iteration Tracking 141

Stand up Meeting 145

End Game 147

Deployment 148

Documentation 148

Recovery 151

Principles 152

Recovering an Iteration 154

Recovering a Release 155

Regaining Your Balance 155

Visible Graphs 157

Choosing Which Graphs to Show 158

Functional Tests Defined and Passing 158

Production Code Bulk, vs. Test Code Bulk 159

Successful Builds 161

Relative Bug Density 162

Story Progress 163

System Performance 163

How to use the Graphs 164

Your Graphs 164

Dealing with Bugs 165

Dealing with bug reports on deployed software 167

Production Support Team 167

Dealing with critical bugs 168

The Customer 169

Finding a Customer 170

Guiding the Customer 170

The Seed 173

Ready To Commit 175

What about research? 176

Coming 179

Going 179

Changes to the Team 179

Splitting the team 180

People growing 180

Tools 181

Fixed scope isn't fixed 183

Outsourced XP 183

Negotiable Scope Contracts 184

Customer 187

In House Development 187

Contracts 188

Shrink Wrap 189

Missing estimates 191

Red Flags 191

Customers won't make decisions 192

Defect reports 193

Not going end to end 193

Failing daily builds 193

Customer won't finish 194

Your Own Process 195

Bibliography 197

Chapter 3

Why Plan?

We plan to ensure that we are always doing the most important thing left to do, to coordinate effectively with other people, and to quickly respond to unexpected events.

When Kent was about ten, he went fly fishing for the first time in the Idaho panhandle. After a fruitless, sweaty day in pursuit of brook trout, he and his friends headed for home. After half an hour stumbling through dense trees, they realized they were lost. He started to panic—fast breathing, tunnel vision, chills. Then someone suggested a plan—they would walk up hill until they hit a logging road they knew was up there. Instantly, the panic disappeared.

Kent was struck at the time of the importance of having a plan. Without the plan, he was going to do something stupid, or just go catatonic. With the plan he was calm again.

Plans in software can work the same way. If you know you have a tight deadline, but you make a plan and the plan says you can make the deadline, then you'll start on your first task with a sense of urgency, but still working as well as possible. After all, you have enough time. This is exactly the behavior that is most likely to cause the plan to come true. Panic leads to fatigue, defects, and communication breakdowns.

But we've also seen plans lead to trouble. They can be a huge time sink dragging days out of people who'd rather be doing something productive, they can be used as a stick to beat people with, and worst of all they can conceal trouble until it's too late to deal with it.

Why We Should Plan



First, we don't plan so we can predict the future. Business and software are changing too rapidly for prediction to be possible. Even if it was possible to predict what we needed in three years, it wouldn't necessarily help because between now and then we need so many different things.

The more obvious it is that you should do something, the more important it is to ask why. Clearly you must do some planning when tackling a serious software development project. Therefore before you start planning you have to understand why you need to it. Without that answer how can you tell if you succeed?

We plan because:

- ✧ We need to ensure that we are always working on the most important thing we need to do.
- ✧ We need to coordinate with other people
- ✧ When unexpected events occur we need to understand the consequences for the first two.

The first is the obvious reason for planning. There's nothing more frustrating than working hard on a part of the system, only to find that it doesn't really matter and gets scrapped in the next release of the sys-

tem. Time spent doing one thing is time not spent doing something else, so if that something else is important then we may fail.

Say it's two o'clock and we're in Boston. We want to drive up to Acadia, but we'd also like to get haircuts and hit Freeport for camping gear. Last time we drove up to Acadia it took us five hours with no stops. So we see some options. If we shoot straight up to Acadia we can be there by seven. If we want to stop for dinner on the way, say an hour, and be there by eight. To get haircuts we'd need another hour, so that would be nine. Visiting Freeport is another hour. We look at what's most important to us, if we want to be fed, equipped, not too late. and care less about our appearance we might decide to drop the haircut. A plan helps us see our options.

Coordination is why everyone else wants us to plan. We get a call from our wives to meet for dinner in Bar Harbor. Since it's two we know we can do it if we drive right up, stop in Freeport, and be there around eight. Software is full of such coordination. Marketing announcements, financial period ends, or management promises. Planning allows us to get an idea of what is reasonable.

But planning is only as good as the estimates that the plans are based on, and estimates always come second to actuals. If we hit a horrible traffic jam all the planning in the world can't make that dinner date. The real world has this horrible habit of intruding on the best laid plans.

But planning still helps us since it allows us to consider the effects of the unexpected event. Leaving at two we hit bad traffic and don't get to Portland until four-thirty. We know we usually get there after an hour, so our experience (and plan) tells us to call our friends to put dinner back to half-past eight and drop the visit to Freeport. Planning allows us both to adjust what we do and to coordinate with others. But the key value is to do this as soon as you know the effect of the event. Our wives would much rather know about our delay at four-thirty than at eight, and it would be really annoying to spend time in Freeport and only later realize that we've really screwed up dinner with our Cindies. (We don't even want to contemplate the consequences of that, in comparison software failures are minor events....)

What we need in planning

Planning is something that people do at various scales. You might plan your day's activities. The team plans out its tasks for the iteration. Development and business lay out a plan for the next year. Senior managers develop plans for a large organization. When carrying out the plan you have to understand the scale in which you plan. If you are driving from Boston to Acadia, you won't plan every curve in the road, but you will want to figure out which roads to take and when to change from one to another. You're not going to expect to arrive to the minute, but we know there is some limit of lateness that requires the apologetic phone call.

In order to carry out the coordination it's vital to have an accurate picture of how far you are along the plan. On a road trip this is fairly straightforward. You can measure mileage, take into account the nature of the roads, and come up with a rough schedule with significant points along the way. If you are very late arriving at Portland, you can easily tell, and thus estimate the delay in reaching Bar Harbor. Software's virtual nature again conspires against this property. With all the degrees of freedom it can be very difficult to find out whether you are 70% done or 30% done. It's like taking a road trip where you don't know whether you've gone 30 miles or 300 miles. Without any frame of reference you feel uncomfortable. If your dinner date doesn't know how far you've gone, they're uncomfortable too.

Therefore any software planning technique must try to regain this visibility, so everyone involved in the project can really see how far along a project is. This means that you need clear milestones, ones that cannot be fudged, and clearly represent progress. They must also be things that everyone involved in the project, including the customer, can understand and learn to trust.

Plans are about figuring out a likely course of events, and figuring the consequences of the inevitable changes. We need different plans, at different scales. Yet all the plans must be both simple to build and simple to keep up to date. Large and complex plans are not helpful because they cost too much to build and maintain. Since plans involve coordination, they must be comprehensible to everyone who is affected by the plan -- another reason for simplicity.

Finally they must be honest, and make it difficult to anyone, including development, to be fooled by reports of progress unrelated to reality.

The Planning Trap

It's the final paragraph that gives us a hint as to why planning can be a trap. This is because there is another reason why people plan:

- ◇ To demonstrate they are in control of events

You'll notice our pejorative use of the third person for this reason. Controlling events is an oxymoron: you can't control events, you can only control their consequences. And even then the amount of control you have is limited. Events cause plans to change. Once you hit that traffic jam then either dinner or Freeport are affected. You can't just carry on with the plan and pretend everything is the same. That would be stupid.

Yet we've seen this happen plenty of times. If things don't go according to plan, then the planner is afraid they will be blamed. That fear causes the determination to say that the plan is still on track. They might admit to themselves that the plan is off track, but if they make the plan complicated enough they can even hide that. The key thing is to say to the outside world that everything is still going according to plan.

But now the plan is diverging from reality and turning into an illusion. Worse still, the planner spends energy trying to maintain the illusion. Developers gradually lose motivation- if the plan's an illusion than why try to follow it?

The hope is that it will all sort itself out in the end. Occasionally that may happen. More often the gap between illusion and reality grows until at some point the illusion is unsustainable. At this point things get ugly. The customer is angry because they have made their own plans based on the illusion, perhaps made some promises that they can't keep. The programmers are angry because they've worked hard, done as well as any programmers could do, but now are being shouted at for not doing the impossible and making the illusion real.

Event happen and plans change. If things are going exactly according to plan, that's usually a sign of trouble. The worst thing that can hap-

pen to a project is the divergence between the plan and reality. So don't fall into that trap. Keep your plans honest, and expect them to always change.

Chapter 4

Rufus and Rupert

Rufus

Your name is Bob. The date is January 3rd, 2001, and your head still aches from the recent millennial revelry. You are sitting in a conference room with several managers and a group of your peers. You are a project team leader. Your boss is there, and he has brought along all of his team leaders. His boss called the meeting.

"We have a new project to develop." Says your bosses boss. Call him BB. The points in his hair are so long that they scrape the ceiling. Your boss' points are just starting to grow, but he eagerly awaits the day when he can leave Brylcream stains on the acoustic tiles. BB describes the essence of the new market they have identified and the product they want to develop to exploit this market.

"We must have this new project up and working by fourth quarter - October first." BB demands. "Nothing is of higher priority; so we are cancelling your current project."

The reaction in the room is stunned silence. Months of work are simply going to be thrown away. Slowly, a murmur of objection begins to circulate around the conference table.

His points give off an evil green glow as BB meets the eyes of everyone in the room. One by one that insidious stare reduces each attendee to quivering lumps of proto-

plasm. It is clear that he will brook no discussion on this matter.

Once silence has been restored, BB says: "We need to begin immediately. How long will it take you to do the analysis?"

You raise your hand. Your boss tries to stop you, but his spitwad misses you and you are unaware of his efforts.

"Sir, we can't tell you how long the analysis will take until we have some requirements."

"The requirements document won't be ready for three or four weeks." BB says, his points vibrating with frustration. "So, pretend that you have the requirements in front of you now. How long will you require for analysis?"

No one breathes. Everyone looks around at everybody else to see if they have some idea.

"If analysis takes any longer than April first, then we have a problem. Can you finish the analysis by then?"

Your boss visibly gathers his courage building to the ejaculation: "We'll find a way, sir!" His points grow 3mm; and your headache increases by two Tylenol.

"Good." BB Smiles. "Now, how long will it take to do the design?"

"Sir," you say. Your boss visibly pales. He is clearly worried that his 3mms are at risk. "Without an analysis, it will not be possible to tell you how long design will take."

BB's expression shifts beyond austere. "PRETEND, you have the analysis already!" He says, while fixing you with his vacant beady little eyes. "How long will it take you to do the design?"

Two Tylenol are not going to cut it. Your boss, in a desperate attempt to save his new growth babbles: "Well, sir, with only six months left to complete the project, design had better take no longer than three months."

"I'm glad you agree, Smithers!" BB says, beaming. Your boss relaxes. He knows his points are secure. After awhile he starts lightly humming the Brylcream jingle.

BB continues, "So, analysis will be complete by April 1st, Design will be complete by July 1st, and that gives you three months to implement the project. This meeting is an example of how well our new consensus and empowerment policies are working. Now, get out there and start working. I'll expect to see TQM plans and QIT assignments on my desk by next week. Oh, and don't forget your cross-functional team meetings and reports will be needed for next month's quality audit."

"Forget the Tylenol." You think to yourself as you return to your cubicle. "I need bourbon."

Visibly excited, your boss comes over to you and says, "Gosh, what a great meeting. I think we're really going to do some world shaking with this project." You nod in agreement, too disgusted to do anything else.

"Oh," your boss continues, "I almost forgot." He hands you a thirty page document. "Remember that the SEI are coming to do an evaluation next week. This is the evaluation guide. You need to read through it, memorize it, and then shred it. It tells you how to answer any questions that the SEI auditors ask you. It also tells you what parts of the building you are allowed to take them to, and what parts to avoid. We are determined to be a CMM level 3 organization by June!"

You and your peers start working on the analysis of the new project. This is difficult because you have no requirements. But, from the 10-minute introduction given by BB on that fateful morning, you have some idea of what the product is supposed to do.

Corporate process demands that you begin by creating a use case document. You and your team begin enumerating use cases and drawing oval and stick diagrams.

Philosophical debates break out amongst the team. There is disagreement as to whether certain use cases should be connected with <<extends>> or <<includes>> relationships. Competing models are created, but nobody knows how to evaluate them. The debate continues, effectively paralyzing progress.

After a week, somebody finds the iceberg.com website that recommends disposing entirely of <<extends>> and <<includes>> and replacing them with <<precedes>> and <<uses>>. The documents on this website, authored by Don Sengroiux, describes a method known as Stalwart-analysis which claims to be a step by step method for translating use-cases into design diagrams.

More competing use-case models are created using this new scheme; but again, nobody agrees on how to evaluate them. And the thrashing continues.

More and more, the use-case meetings are driven by emotion rather than reason. If it weren't for the fact that you don't have requirements, you'd be pretty upset by the lack of progress you are making.

The requirements document arrives on the 15th of February. And then again on the 20th, 25th, and every week thereafter. Each new version contradicts the previous. Clearly the marketing folks who are writing the requirements, empowered though they might be, are not finding consensus.

At the same time, several new competing use-case templates have been proposed by the various team members. Each presents its own particularly creative way of delaying progress. The debates rage on.

On March 1st, Percival Putrignence, the process proctor, succeeds in integrating all the competing use-case forms and templates into a single, all-encompassing form. Just the blank form is fifteen pages long. He has managed to include every field that appeared on all the competing templates. He also presents a 159 page document describing how to fill out the use-case form. All current use cases must be rewritten according to the new standard.

You marvel to yourself that it now requires fifteen pages of fill-in-the-blank, and essay questions, to answer the question: "What should the system do when the user hit's return."

The corporate process (authored by L. E. Ott, famed author of "Holistic analysis: A progressive dialectic for software engineers.") insists that you discover all primary use-cases, 87% of all secondary use cases, and 36.274% of all tertiary use cases before you can complete analysis and enter the design phase. You have no idea what a tertiary use-case is. So in an attempt to meet this requirement you try to get your use-case document reviewed by the marketing department. Maybe they know what a tertiary use-case is.

Unfortunately the marketing folks are too busy with sales support to talk to you. Indeed, since the project started, you have not been able to get a single meeting with marketing. The best they have been able to do is provide a never ending stream of changing and contradictory requirements documents.

While one team has been spinning endlessly on the use-case document, another has been working out the domain model. Endless variations of UML documents are pouring out of this team. Every week the model is reworked. The team members can't decide on whether

to use <<interfaces>> or <<types>> in the model. A huge disagreement has been raging on the proper syntax and application of OCL. Other's in the team just got back from a five day class on "catabolism", and have been producing incredibly detailed and arcane diagrams that nobody else can fathom.

On March 27th, with one week to go before analysis is to be complete, you have produced a sea of documents and diagrams; but are no closer to a cogent analysis of the problem than you were on January third.

...

And then, a miracle happens.

...

On Saturday, April 1st you check you email from home. You see a memo from your boss to BB. It states unequivocally that you are done with the analysis!

You phone your boss and complain. "How could you have told BB that we were done with the analysis?"

"Have you looked at a calendar lately?" he responds, "It's April 1st!"

The irony of that date does not escape you. "But we have so much more to think about. So much more to analyze!, we haven't even decided whether to use <<extends>> or <<precedes>>!"

"Where is your evidence that you are not done?" inquires your boss impatiently.

"Whaaa...."

But he cuts you off. "Analysis can go on forever, it has to be stopped at some point. And since this is the date it was scheduled to stop, it has been stopped. Now, on Monday I want you to gather up all existing analysis materials and put them into a public folder. Release that

folder to Percival so that he can log it in the CM system by Monday afternoon. Then get busy and start designing."

As you hang up the phone, you begin to consider the benefits of keeping a bottle of bourbon in your bottom desk drawer.

They threw a party to celebrate the on-time completion of the analysis phase. BB gave a colon stirring speech on empowerment. And your boss, another 3mm taller, congratulated his team on the incredible show of unity and teamwork. Finally, the CIO takes the stage and tells everyone that the SEI audit went very well, and thanks everyone for studying and shredding the evaluation guides that were passed out. Level three now seems assured, and will be awarded by June.

(Scuttlebut has it that managers at the level of BB and above are to receive significant bonuses once the SEI awards level 3.)

As the weeks flow by, you and your team work on the design of the system. Of course you find that the analysis that the design is supposedly based upon is flawed ... no, useless... no, worse than useless. But when you tell your boss that you need to go back and work some more on the analysis to shore up its weaker sections, he simply states: "The analysis phase is over. The only allowable activity is design. Now get back to it."

So, you and your team hack the design as best you can, unsure of whether the requirements have been properly analyzed or not. Of course it really doesn't matter much since the requirements document is still thrashing with weekly revisions, and the marketing department still refuses to meet with you.

The design is a nightmare. Your boss recently mis-read a book named "The Finish-line" in which the author, Mark DeThomaso, blithely suggested that design documents should be taken down to code level detail.

"If we are going to be working at that level of detail," you ask, "why don't we just write the code instead?"

"Because then you wouldn't be designing, of course. And the only allowable activity in the design phase is design!"

"Besides," he continues, "we have just purchased a company wide license for Dandylion! This tool enables "Round the Horn Engineering!" You are to transfer all design diagrams into this tool. It will automatically generate our code for us! It will also keep the design diagrams in sync with the code!"

Your boss hands you a brightly colored shrink-wrapped box containing the Dandylion distribution. You accept it numbly, and shamle off to your cubicle. Twelve hours, eight crashes, a disk reformatting, and eight shots of 151 later, you finally have the tool installed on your server. You consider the week your team will lose while attending Dandylion training. Then you smile and think, "Any week I'm not here, is a good week."

Design diagram after design diagram is created by your team. Dandylion makes it very hard to draw these diagrams. There are dozens and dozens of deeply nested dialog boxes with funny text fields and check boxes that must all be filled in correctly. And then there's the problem of moving classes between packages...

At first these diagram are driven from the use cases. But the requirements are changing so often that the use-cases rapidly become meaningless.

Debates rage about whether Visitor or Decorator design patterns should be employed. One developer refuses to use Visitor in any form claiming that it's not a properly object-oriented construct. Another refuses to use multiple inheritance since it is the spawn of the devil.

Review meetings rapidly degenerate into debates about the meaning of Object Orientation, the definition of analysis vs. design, or when to use aggregation vs. association.

Midway through the design cycle, the marketing folks announce that they have rethought the focus of the system. Their new requirements document is completely restructured. They have eliminated several major feature areas, and replaced them with feature areas that they anticipate customer surveys will show to be more appropriate.

You tell your boss that these changes mean that you need to reanalyze and redesign much of the system. But he says: "The analysis phase is over. The only allowable activity is design. Now get back to it."

You suggest that it might be better to create a simple prototype to show to the marketing folks, and even some potential customers. But your boss says: "The analysis phase is over. The only allowable activity is design. Now get back to it."

Hack, hack, hack, hack. You try to create some kind of a design document that might actually reflect the new requirements documents. However, the revolution of the requirements has not caused them to stop thrashing. Indeed, if anything, the wild oscillations of the requirements document have only increased in frequency and amplitude. You slog your way through them.

On June 15th, the Dandyllion database gets corrupted. Apparently the corruption has been progressive. Small errors in the DB accumulated over the months into bigger and bigger errors. Eventually the CASE tool just stopped working. Of course the slowly encroaching corruption is present on all the backups.

Calls to the Dandyllion technical support line go unanswered for several days. Finally you receive a brief email from Dandyllion, informing you that this is a known problem, and the solution is to purchase the new version (which they promise will be ready some time next quarter) and then re-enter all the diagrams by hand.

...

Then, on July 1st another miracle happens! You are done with the design!

Rather than go to your boss and complain, you stock your middle desk drawer with some vodka.

They threw a party to celebrate the on-time completion of the design phase, and their graduation to CMM level 3. This time you find BB's speech so stirring that you have to use the restroom before it begins.

There are new banners and plaques all over your workplace. They show pictures of eagles and mountain climbers, and they talk about teamwork and empowerment. They read better after a few scotches. That reminds you that you need to clear out your file cabinet to make room for the brandy.

You and your team begin to code. But you rapidly discover that the design is lacking in some significant areas. Actually it's lacking any significance at all. You convene a design session in one of the conference rooms to try to work through some of the nastier problems. But your boss

catches you at it and disbands the meeting saying: "The design phase is over. The only allowable activity is coding. Now get back to it."

The code generated by Dandylion is really hideous. It turns out that you and your team were using association and aggregation the wrong way after all. All the generated code has to be edited to correct these flaws. Editing this code is extremely difficult because it has been instrumented with ugly comment blocks that have special syntax that Dandylion needs in order to keep the diagrams in sync with the code. If you accidentally alter one of these comments, then the diagrams will be regenerated incorrectly. It turns out that "Round the Horn Engineering" requires an awful lot of effort.

The more you try to keep the code compatible with Dandylion, the more errors Dandylion generates. In the end, you give up and decide to keep the diagrams up to date manually. A second later you decide there's no point in keeping the diagrams up to date at all. Besides, who has time?

Your boss hires a consultant to build tools to count the number of lines of code that are being produced. He puts a big thermometer graph on the wall with the number 1,000,000 on the top. Every day he extends the red line to show how many lines have been added.

Three days after the thermometer appears on the wall, your boss stops you in the hall. "That graph isn't growing fast enough. We need to have a million lines done by October 1st."

"We aren't even sh-sh-shure that the proshect will require a m-million linezh." You blather.

"We have to have a million lines done by October 1st." your boss reiterates. His points have grown again, and the

Grecian formula he uses on them creates an aura of authority and competence. "Are you sure your comment blocks are big enough?"

Then, in a flash of managerial insight he says: "I have it! I want you to institute a new policy amongst the engineers. No line of code is to be longer than 20 characters. Any such line must be split into two or more -- preferably more. All existing code needs to be reworked to this standard. That'll get our line count up!"

You decide not to tell him that this will require two unscheduled man months. You decide not to tell him anything at all. You decide that intravenous injections of pure ethanol are the only solution. You make the appropriate arrangements.

Hack, hack, hack, and hack. You and your team madly code away. By August 1st your boss, frowning at the thermometer on the wall institutes a mandatory 50-hour work-week.

Hack, hack, hack, and hack. By September 1st, the thermometer is at 1.2 million lines and your boss asks you to write a report describing why you exceeded the coding budget by 20%. He institutes mandatory Saturdays and demands that the project be brought back down to a million lines. You start a campaign of re-merging lines.

Hack, hack, hack, and hack. Tempers are flaring; people are quitting; QA is raining trouble reports down on you. Customers are demanding installation and user manuals, salesmen are demanding advance demonstrations for special customers; the requirements document is still thrashing, the marketing folks are complaining that the product isn't anything like they specified, and the liquor store won't accept your credit card anymore.

Something has to give. On September 15th BB calls a meeting.

As he enters the room, his points are emitting clouds of steam. When he speaks, the bass overtones of his carefully manicured voice cause the pit of your stomach to roll over. "The QA manager has told me that this project has less than 50% of the required features implemented. He has also informed me that the system crashes all the time, yields wrong results, and is hideously slow. He has also complained that he cannot keep up with the continuous train of daily releases; each more buggy than the last!"

He stops for a few seconds, visibly trying to compose himself. "The QA manager estimates that, at this rate of development, we won't be able to ship the product until December!"

Actually, you think it's more like March, but you don't say anything.

"December!" BB roars. People duck their heads as though he were pointing an assault rifle at them. "December is absolutely out of the question. Team leaders, I want new estimates on my desk in the morning. I am hereby mandating 65-hour workweeks until this project is complete. And it better be complete by Nov. 1st."

As he leaves the conference room he is heard to mutter: "Empowerment - Bad!"

Your boss is bald; his points are mounted on BB's wall. The fluorescent lights reflecting off his pate momentarily dazzle you.

"Do you have anything to drink?" he asks. Having just finished your last bottle of Boone's Farm, you pull a bottle of Thunderbird from your bookshelf and pour it into his

coffee mug. "What's it going to take to get this project done?" he asks.

"We need to freeze the requirements, analyze them, design them, and then implement them." You say callously.

"By Nov. 1st?" your boss exclaims incredulously. "No way! Just get back to coding the damned thing." He storms out, scratching his vacant head.

A few days later you find that your boss has been transferred to the corporate research division. Turnover has skyrocketed. Customers, informed at the last minute that their orders cannot be fulfilled on time, have begun to cancel their orders. Marketing is reevaluating whether or not this product aligns with the overall goals of the company, etc., etc. Memos fly, heads roll, policies change, and things are, overall, pretty grim.

Finally, by March, after far too many 65-hour weeks, a very shaky version of the software is ready. In the field, bug discovery rates are high, and the technical support staff are at their wit's end trying to cope with the complaints and demands of the irate customers. Nobody is happy.

In April, BB decides to buy his way out of the problem by licensing a product produced by Rupert industries and redistributing it. The customers are mollified, the marketing folks are smug, and you are laid off.

Rupert

Rupert Industries. Project: ~Alpha~

Your name is Robert. The date is January 3rd, 2001. The quiet hours spent with your family this holiday have left you refreshed and ready for work. You are sitting in a con-

ference room with your team of professionals. The manager of the division called the meeting.

"We have some ideas for a new project" says the division manager. Call him Russ. He is a high strung British chap with more energy than a fusion reactor. He is ambitious and driven; but understands the value of a team.

Russ describes the essence of the new market opportunity the company has identified, and introduces you to Jay, the marketing manager who is responsible for defining the products that will address it.

Addressing you, Jay says: "We'd like to start defining our first product offering as soon as possible. When can you and you team meet with me?"

You reply: "We'll be done with the current iteration of our project this Friday. We can spare a few hours for you between now and then. After that, we'll take a few people from the team and dedicate them to you. We'll begin hiring their replacements, and the new people for your team immediately."

"Great", says Russ, "But I want you to understand that it is critical that we have something to exhibit at the trade show coming up this July. If we can't be there with something significant, we'll lose the opportunity."

"I understand." you reply. "I don't yet know what it is that you have in mind, but I'm sure we can have something by July. I just can't tell you what that something will be right now. In any case, you and Jay are going to have complete control over what we developers do, so you can rest assured that by July you'll have the most important things that can be accomplished in that time ready to exhibit."

Russ nods in satisfaction. He knows how this works. Your team has always kept him advised and allowed him to steer their development. He has the utmost confidence that your team will work on the most important things first; and that they will produce a high quality product.

~ ~ ~

"So Robert," says Jay at their first meeting, "How does your team feel about being split up?"

"We'll miss working with each other", you answer, "but some of were getting pretty tired of that last project and

are looking forward to a change. So, what are you guys cooking up?”

Jay beams. “You know how much trouble our customers currently have...” And he spends a half hour or so describing the problem and possible solution.

“OK, wait a second” you respond. “I need to be clear about this.” And so you and Jay talk about how this system might work. Some of Jay’s ideas aren’t fully formed. You suggest possible solutions. He likes some of them. You continue discussing.

During the discussion, as each new topic is addressed, Jay writes user story cards. Each card represents something that the new system has to do. The cards accumulate on the table and are spread out in front of you. Both you and Jay point at them, and pick them up, and make notes on them as you discuss the stories. The cards are powerful mnemonic devices that you can use to represent complex ideas that are barely formed.

At the end of the meeting you say: “OK, I’ve got a general idea of what you want. I’m going to talk to the team about it. I imagine there are some experiments they’ll want to run with various database structures and presentation formats. Next time we meet, it’ll be as a group, and we’ll start identifying the most important features of the system.

A week later your nascent team meets with Jay. They spread the existing user story cards out on the table and begin to get into some of the details of the system.

The meeting is very dynamic. Jay presents the stories in the order of their importance. There is much discussion about each one. The developers are concerned about keeping the stories small enough to estimate and test. So they continually ask Jay to split one story into several smaller stories. Jay is concerned that each story has a clear business value and priority, so as he splits them, he makes sure this stays true.

The stories accumulate on the table. Jay writes them, but the developers make notes on them as needed. Nobody tries to capture everything that is said; the cards are not meant to capture everything; they are just reminders of the conversation.

As the developers become more comfortable with the stories, they begin writing estimates on them. These estimates are crude and budgetary, but they give Jay an idea of what the story will cost.

At the end of the meeting, it is clear that there are many more stories that could be discussed. It is also clear that the most important stories have been addressed, and that they represent several months worth of work. Jay closes the meeting by taking the cards with him and promising to have a proposal for the first release in the morning.

~ ~ ~

The next morning you reconvene the meeting. Jay chooses five cards and places them on the table.

“According to your estimates, these cards represent about one perfect team-week’s worth of work. The last iteration of the previous project managed to get one perfect team-week done in three real weeks. If we can get these five stories done in three weeks, we’ll be able to demonstrate them to Russ. That will make him feel very comfortable about our progress.”

Jay is pushing it. The sheepish look on his face lets you know that he knows it too. You reply, “Jay, this is a new team, working on a new project. It’s a bit presumptuous to expect that our velocity will be the same as the previous team’s. However, I met with the team yesterday afternoon, and we all agreed that our initial velocity should, in fact, be set to one perfect-week for every three real-weeks. So you’ve lucked out on this one.”

“Just remember,” you continue, “that the story estimates and the story velocity are very tentative at this point. We’ll learn more when we plan the iteration, and even more when we implement it.”

Jay looks over his glasses at you as if to say “Who’s the boss around here anyway.”, and then smiles and says “Yeah, don’t worry, I know the drill by now.”

Jay then puts fifteen more cards on the table. He says, “If we can get all these cards done by the end of March, we can turn the system over to our beta test customers. And we’ll get good feedback from them.”

You reply, “OK, so we’ve got our first iteration defined; and we have the stories for the next three iterations after that. These four iterations will make our first release.”

“So,” says Jay, “Can you really do these five stories in the next three weeks?”

“I don’t know for sure Jay,” you reply, “Let’s break them down into tasks and see what we get.”

So Jay, you, and your team spend the next several hours taking each of the five stories that Jay chose for the first iteration and breaking them down into small tasks. The developers quickly realize that some of the tasks can be shared between stories, and that other tasks have commonalities that can probably be taken advantage of. It is clear that potential designs are popping into the developers’ heads. From time to time they form little discussion knots and scribble UML diagrams on some cards.

Soon, the whiteboard is filled with the tasks that, once completed, will implement the five stories for this iteration. You start the sign up process by saying: “OK, let’s sign up for these tasks.”

“I’ll take the initial database generation.” says Pete, “That’s what I did on the last project, and this doesn’t look very different. I estimate it at two of my perfect man-days.”

“OK, well then I’ll take the login screen.” says Joe.

“Aw darn,” says Elmo, the junior member of the team, “I’ve never done a GUI, and I kinda wanted to try that one.”

“Ah the impatience of youth.” Joe says sagely, with a wink in your direction, “You can assist me with it, young Jedi.” To Jay: “I think it’ll take me about three of my perfect man-days.”

One by one the developers sign up for tasks and estimate them in terms of their own perfect man-days. Both you and Jay know that it is best to let the developers volunteer for tasks, than it is to assign the tasks to them. You also know full well that you daren’t challenge any of the developer’s estimates. You know these guys, and you trust them. You know they are going to do the very best they can.

The developers know that they can't sign up for more perfect man-days than they finished in the last iteration they worked on. Once each developer has filled his schedule for the iteration, they stop signing up for tasks.

Eventually, all the developers have stopped signing up for tasks. But, of course, there are still tasks left on the board.

"I was worried that might happen." you say, "OK, there's only one thing to do, Jay. We've got too much to do in this iteration. What stories or tasks can we remove."

Jay sighs. He knows that this is the only option. Working overtime at the beginning of a project is insane; and projects where he's tried it have not fared well.

So Jay starts to remove the least important functionality. "Well, we really don't need the login screen just yet. We can simply start the system in the logged in state."

"Rats!" cries Elmo. "I really wanted to do that."

"Patience, Grasshopper." says Joe. "Those who wait for the bees to leave the hive, will not have lips too swollen to relish the honey."

Elmo looks confused.

Everyone looks confused.

"So...", Jay continues, "I think we can also do away with..."

And so, bit by bit the list of tasks shrinks. Developers who lose a task, sign up for one of the remaining ones.

The negotiation is not painless. Several times Jay exhibits obvious frustration and impatience. Once, when tensions are especially high, Elmo volunteers to "Work extra hard to make up some of the missing time." You are about to correct him when, fortunately, Joe looks him in the eye and says, "When once you proceed down the dark path, forever will it dominate your destiny."

In the end, an iteration acceptable to Jay is reached. It's not what Jay wanted. Indeed, it is significantly less. But it's something the team feels that they can achieve in the next three weeks. And, after all, it still addresses the most important things that Jay wanted in the iteration.

"So, Jay," you say when thing had quieted down a bit. "When can we expect acceptance tests from you?"

Jay sighs. This is the other side of the coin. For every story the development team implements, Jay must supply a suite of acceptance tests that prove that they work. And the team needs these long before the end of the iteration, since they will certainly point out differences in the way Jay and the developers imagine the system's behavior.

"I'll get you some example test scripts today." Jay promises. "I'll add to them every day after that. You'll have the entire suite by the middle of the iteration."

~~~

The iteration begins on Monday morning with a flurry of CRC sessions. By mid-morning all the developers have assembled into pairs and are rapidly coding away.

"And now, my young apprentice," Joe says to Elmo, "you shall learn the mysteries of Test First Design!"

"Wow, that sounds pretty rad." Elmo replies. "How do you do it?"

Joe beams. It's clear that he has been anticipating this moment. "Laddy-buck, what does the code do right now?"

"Huh?", replied Elmo, "It doesn't do anything at all, there is no code."

"So, consider our task, can you think of something the code should do?"

"Sure." Elmo said with youthful surety, "First, it should connect to the database."

"And thereupon, what must needs be required to connecteth the database?"

"You sure talk wierd." laughed Elmo. "I think we'd have to get the database object from some registry and call the Connect() method."

"Ah. Astute young wizard. Thou perceivest correctly that we requireth an object within which we can cacheth the database object."

"Is 'cacheth' really a word?"

"It is when I say it! So, what test can we write that we know the database registry should pass?"

Elmo sighs. he knows he'll just have to play along. "We should be able to create a database object and pass it to the registry in a Store() method. And then we should

be able to pull it out of the registry with a Get() method and make sure it's the same object."

"Oh, well said, my pre-pubescent sprite!"

"Hay!"

"So, now, let's write a test function that proves your case."

"But shouldn't we write the database object and registry object first?"

"Ah, you've much to learn my young impatient one. Just write the test first."

"But it won't even compile!"

"Are you sure? What if it did?"

"Uh..."

"Just write the test, Elmo. Trust me."

And so Joe, Elmo, and all the other developers began to code their tasks, one test case at a time. The room in which they worked was a-buzz with the conversations between the pairs. The murmur was punctuated by an occasional high-five when a pair managed to finish a task or a difficult test case.

As development proceeded, the developers changed partners once or twice a day. Each developer got to see what all the others were doing, and so knowledge of the code spread generally throughout the team.

Whenever a pair finished something significant; whether a whole task, or just an important part of a task, they integrated what they had with the rest of the system. Thus, the code base grew daily, and integration difficulties were minimized.

The developers communicated with Jay on a daily basis. They'd go to him whenever they had a question about the functionality of the system, or the interpretation of an acceptance test case.

Jay, good as his word, supplied the team with a steady stream of acceptance test scripts. The team read these carefully and thereby gained a much better understanding of what Jay expected the system to do.

By the beginning of the second week, there was enough functionality to demonstrate to Jay. Jay watched eagerly as the demonstration passed test-case after test-case.

"This is really cool." Jay said as the demonstration finally ended. "But this doesn't seem like one third of the tasks. Is your velocity slower than anticipated?"

You grimace. You'd been waiting for a good time to mention this to Jay; but now Jay was forcing the issue.

"Yes, unfortunately we are going slower than we had expected. The new application server we are using is turning out to be a pain to configure. Also it takes forever to reboot; and we have to reboot it whenever we make even the slightest change to its configuration."

Jay eyes you with suspicion. The stress of last Monday's negotiations had still not entirely dissipated. He says: "And what does this mean to our schedule? We can't slip it again, we just can't. Russ will have a fit! He'll haul us all into the woodshed and ream us some new ones."

You look Jay right in the eyes. There's no pleasant way to give someone news like this. So you just blurt out, "Look, if things keep going like their going, then we're not going to be done with everything by next Friday. Now it's possible that we'll figure out a way to go faster. But, frankly, I wouldn't depend upon that. You should start thinking about one or two tasks that could be removed from the iteration without ruining the demonstration for Russ. Come hell or high water we are going to give that demonstration on Friday, and I don't think you want us to choose which tasks to omit."

"Aw for -- goodness sakes!" Jay barely manages to stifle yelling that last word as he stalks away shaking his head.

Not for the first time you say to yourself: "Nobody ever promised me project management would be easy." You are pretty sure it won't be the last time either.

Actually, things went a bit better than you had hoped. The team did, in fact, have to drop one task from the iteration; but Jay had chosen wisely, and the demonstration for Russ went without a hitch.

Russ was not impressed with the progress, but neither was he dismayed. He simply said, "This is pretty good. But remember, we have to be able to demonstrate this system at the trade show in July; and at this rate it doesn't look like you'll have all that much to show."

Jay, whose attitude had improved dramatically with the completion of the iteration, responded to Russ by saying: "Russ, this team is working hard, and well. When July comes around I am confident that we'll have something significant to demonstrate. It won't be everything, and some of it may be smoke and mirrors, but we'll have something."

Painful though the last iteration was, it had calibrated your velocity numbers. The next iteration went much better. Not because your team got more done than in the last iteration, but simply because they didn't have to remove any tasks or stories in the middle of the iteration.

By the start of the fourth iteration, a natural rhythm has been established. Jay, you, and the team know exactly what to expect from each other. The team is running hard, but the pace is sustainable. You are confident that the team can keep up this pace for a year or more.

The number of surprises in the schedule diminishes to near zero; however the number of surprises in the requirements does not. Jay and Russ frequently look over the growing system and make recommendations or changes to the existing functionality. But all parties realize that these changes take time, and must be scheduled. So the changes do not cause anyone's expectations to be violated.

In March there is a major demonstration of the system to the board of directors. The system is very limited, and is not yet in a form good enough to take to the trade show; but progress is steady, and the board is reasonably impressed.

The second release goes even smoother than the first. By now the team has figured out a way to automate Jay's acceptance test scripts. They have also refactored the design of the system to the point where it is really easy to add new features and change old ones.

The second release was done by the end of June, and was taken to the trade show. It had less in it than Jay and Russ would have liked; but it did demonstrate the most important features of the system. Though customers at the trade show noticed that certain features were missing, overall they were very impressed. You, Russ, and Jay

all returned from the trade show with smiles on your faces. You all felt as though this project was a winner.

Indeed, many months later you are contacted by Rufus Inc. They had been working on a system like this for their internal operations. They have cancelled the development of that system after a death-march project; and are negotiating to license your technology for their environment.

Indeed, things are looking up!

Chapter 5

Fear

*Courage! What makes a King out of a slave?
Courage! What makes the flag on the mast to wave?
Courage! What makes the elephant charge his tusk, in the misty mist or the
dusky dusk?
What makes the muskrat guard his musk?
Courage! What makes the sphinx the seventh wonder?
Courage! What makes the dawn come up like thunder?
Courage! What makes the Hottentot so hot? What puts the "ape" in apri-
cot? What have they got that I ain't got?*

-- Cowardly Lion

Why do we need a software process? For the same reason that we need laws, governments, and taxes: Fear.

The American Declaration of Independence says:

*That among these [rights] are life, liberty, and the pursuit of happiness.
That to secure these rights, governments are instituted among men, deriv-
ing their just powers from the consent of the governed.¹*

Though the profundity of these words may distract us, consider the word “secure”. We institute governments because we are afraid of losing our rights.

By the same token, we institute software processes because we are afraid.

Customers are afraid that:

1. Microsoft word complains about the grammar...

- ◇ They won't get what they asked for.
- ◇ They'll ask for the wrong thing.
- ◇ They'll pay too much for too little.
- ◇ They must surrender control of their career to techies who don't care.
- ◇ They won't ever see a meaningful plan.
- ◇ The plans they do see will be fairy-tales.
- ◇ They won't know what's going on.
- ◇ They'll be held to their first decisions and won't be able to react to changes in the business.
- ◇ No one will tell them the truth.

Developers are afraid too. They fear that:

- ◇ They will be told to do more than they know how to do.
- ◇ They will be told to do things that don't make sense.
- ◇ They are too stupid.
- ◇ They are falling behind technically.
- ◇ They will be given responsibility without authority.
- ◇ They won't be given clear definitions of what needs to be done.
- ◇ That they'll have to sacrifice quality for deadlines.
- ◇ That they'll have to solve hard problems without help.
- ◇ That they won't have enough time to succeed.

Unacknowledged fear is the source of all engineering failure.

If these fears are not put on the table and dealt with, then developers and customer each try to protect themselves by building walls. They refuse to share critical information:

“If I tell the engineers about this, they'll spend months trying to figure it out instead of doing what I need.”

“If I tell the customer how quickly I got this done, he'll expect me to do everything that fast.”

They exaggerate, tell half-truths, lie, cover-up, and work at cross purposes. They build huge useless political and procedural structures aimed at protection instead of success.

In order to be successful, a development process must be instituted among customers and developers that secures certain inalienable rights. Among these are:

The Customer Bill of Rights.

- ◇ You have the right to an overall plan, to know what can be accomplished, when, and at what cost.
- ◇ You have the right to get the most possible value out of every programming week.
- ◇ You have the right to see progress in a running system, proven to work by passing repeatable tests that you specify.
- ◇ You have the right to change your mind, to substitute functionality, and to change priorities without paying exorbitant costs.
- ◇ You have the right to be informed of schedule changes, in time to choose how to reduce scope to restore the original date. You can cancel at any time and be left with a useful working system reflecting investment to date.

Programmer Bill of Rights

- ◇ You have the right to know what is needed, with clear declarations of priority.
- ◇ You have the right to produce quality work at all times.
- ◇ You have the right to ask for and receive help from peers, superiors, and customers.
- ◇ You have the right to make, and update your own estimates.
- ◇ You have the right to accept your responsibilities instead of having them assigned to you.

If we are going to develop well, we must create a culture that makes it possible for programmers and customer to acknowledge their fears and accept their rights and responsibilities. Without such guarantees, we cannot be courageous. We huddle in fear behind fortress walls, building them ever stronger, adding ever more weight to the develop-

ment processes we have adopted. We continually add cannonades and battlements, documents and reviews, procedures and signoffs, moats with crocodiles, torture chambers, and huge pots of boiling oil.

But when our fears are acknowledged and our rights are accepted, then we can be courageous. We can set goals that are hard to reach, and collaborate to make those goals. We can tear down the structures that we built out of fear and that impede us. We will have the courage to do only what is necessary and no more, to spend our time on what's important, rather than on protecting ourselves.

Chapter 6

Driving Software

The driving story featured prominently in Embrace Change, but it is central to XP, so we repeat it here. If you read EC, you'll only want to read this chapter to see if we've somehow managed to make the story a bit more dramatic.



It was a beautiful sunny day. Kent and his mom were driving along a straight stretch of I-5 near Chico. He was about 12 years old.

“It’s about time you learn how to drive,” said Mom.

“Really?” Excitement bubbled in Kent’s chest.

“Yes. Now, what I want you to do is get the car right in between the lines and pointed absolutely straight,” said Mom.

“I can do that.” Kent very carefully lines up the star on the beige Mercedes 240D dead straight to the horizon. His eyebrows raise a little at just how easy this driving thing really is. After a moment, his eyes drift to a roadside sign.

ggggrrrrrrccchhhh (hey, you try to write down the sound combining wheels on gravel and a pre-adolescent yelp). Kent’s mouth goes dry, his heart pounds.

“Okay,” says Mom, concealing a smile, “that’s not how you drive a car. Driving a car is not about getting the car pointed in the right direction. Driving a car is about constantly making little corrections. You drift a little this way, you steer a little that way. This way, that way, as long as you are driving.”

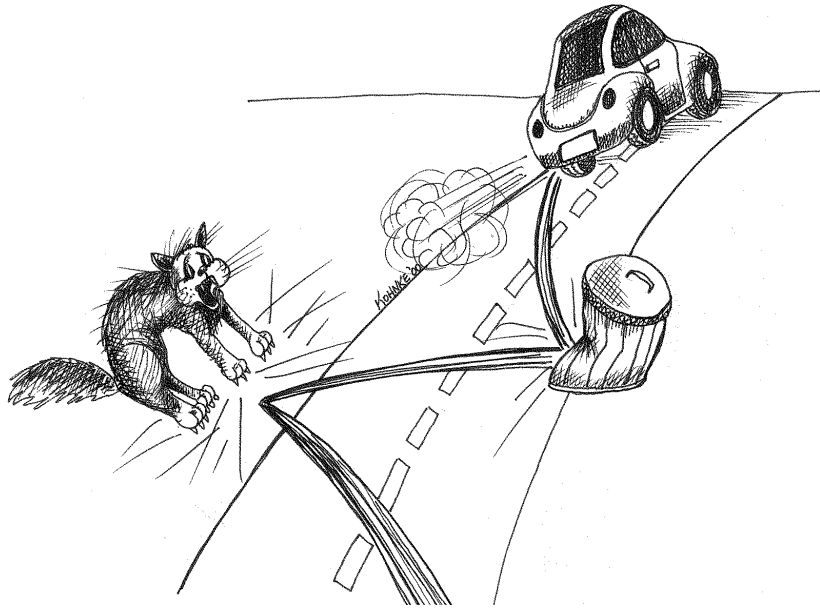
You don’t drive software development by getting your project pointed in the right direction (The Plan). You drive software development by seeing that you are drifting a little this way and steering a little that way. This way, that way, as long as you develop the software.

One very vocal opponent of XP once used the phrase: “Ready...Fire...Aim!” The intent was clearly pejorative. How can you hit a target unless you aim first? The point, however, is that we are not trying to hit a target. Instead, we are trying to maximize the benefit of a process.

The driving metaphor helps us once again. Your first act as you get into the car is not turning the wheel so that it points towards your destination. Your first act is usually to turn on the ignition. Indeed, the initial direction of motion has little to do with your destination, and much more to do with your local circumstances. You might want to back out of your garage before heading for Peoria. Though you probably have a destination in mind, and probably also have a route planned, that route and destination are always subject to change. The radio may warn you of heavy traffic, causing you to change your route. Your spouse may call on the cell phone and ask you to pick up some milk, causing you to modify your destination.

Software development is a process. It can go well, or it can go badly. To keep it going well we must continually direct it. To direct it we must frequently assess the direction it is going, compare this to the direction

we want it to go, and then make careful adjustments. Thus, good project management can be characterized by: “Ready... Fire... Aim... Aim... Aim... Aim... Aim...”.



Chapter 7

The Problem

Gentlemen, if we do not succeed, we risk failure.

-- Dan Quayle.

What is the problem that planning is supposed to solve? Or, to ask this another way: What symptoms of project failure can we blame on poor planning?

Projects sometimes fail long before they deliver anything. At some point they may be determined to be too expensive to continue. Or perhaps they took too long to develop and the business need evaporated. Or perhaps the requirements change so often that the developers can never finish one thing without having to stop and start all over on something new. Certainly these are planning failures.

Projects sometimes deliver the wrong goods. In such cases customers are disappointed with the project because it does not behave the way they expected. Perhaps it is too slow, or too clumsy. Or perhaps it crashes or freezes a lot. Perhaps it simply doesn't solve the problem the way they thought it would. Or perhaps it just takes too long, or costs too much, to make the necessary changes that track the business. Certainly these are also planning failures.

There are two ways to approach prevention of these planning failures. We can plan not to lose, or we can plan to win. The two are not identical. Planning not to lose is defensive; while planning to win is aggressive.

If we decide to plan not to lose, we take a defensive posture in which we expend huge amounts of effort trying to prevent and track errors. This will lead us to a very heavyweight planning process in which we try to plan everything up front in as much detail as possible. Such a process

will have many review steps, sign-offs, authorizations, and phase gates. Such a planning process is highly adept at making sure that blame can be assigned when something fails; but takes no direct steps towards making sure that the right system is delivered at a reasonable cost.

Such a heavyweight process can work, so long as the customers and developers trust each other, and work together as a team. However, the process itself, with its checks and balances, does not engender a team spirit. Rather, since it tracks accountability, team members tend to react defensively, and to find ways to avoid, transfer, or obfuscate accountability.

As a result estimates grow, review steps increase, paperwork multiplies, and it takes forever to get anything done. Career goals become linked to how well you can avoid accountability for failure, rather than how well you can succeed.

When we plan to win, however, we do not concern ourselves with errors and accountability. Rather we assume that everyone wants to win. We do the most important things first, as quickly as we can. We get rapid feedback from our customers. Such a plan acknowledges that errors will occur, but also plans to correct them quickly through feedback.

When we plan to win we take direct steps to ensure that we are building the right system at the best possible cost. Every action we take goes towards that end. Instead of trying to plan everything up front, we plan just the next few steps; and then allow customer feedback to correct our trajectory. In this way, we get off the mark quickly, and then continuously correct our direction. Errors are unimportant because they will be corrected quickly.

So, the problem that planning is supposed to solve is simply, *to build the right system at the right cost*. If we take a defensive posture by planning not to lose, we will be able to hold people accountable for any failures; but at an enormous cost. If we take an aggressive posture and plan to win, we will be unafraid to make errors, and will continuously correct them to meet our goals.

Chapter 8

Balancing Power

“I want to have a baby.”

“You can’t have a baby. You’re a man!”

“Don’t you oppress me!”

--The Judean People’s Front, The Life of Brian

The key to project management is balancing power between the customers and the programmers. Done right, software project management has:

- ✧ Business people making business decisions
- ✧ Technical people making technical decisions

Isn’t this just like saying that Wensleydale is the Queen of Cheeses? Of course business people make business decisions.

What about this one?

“We think this system will take six months to develop.”

“You have three months.”

“What can we take out?”

“Nothing. Everything has to be there.”

Guessing how long something is likely to take to program is a purely technical decision. The programmers pool their experience on similar projects, stir their understanding of how the new system is different, and pick a number off a dartboard. No, actually it’s a little more systematic than that (see Estimating). However, tough as estimating is, the

programmers are in a much better position to guess than anyone else. So estimating is a technical decision.

In the dialog above, taken from an actual doomed project, estimating was done by a business person for business reasons. The resulting estimate, that the work in question will take three months, cannot possibly have been better than the programmers' estimate of six months. However, without intervention, everyone went on to the next stage of planning based on grossly inaccurate information. The resulting plan, no matter how cheap, flexible, and communicable, was, simply put, tripe.

If occasionally business people make technical decisions, at least technical people don't make business decisions.

Uh, how about this one?

"I have ten things to do. I know I'll only get five of them done. I'll work on this DCOM / CORBA infrastructure first. It looks cool."

Stop. Choosing the relative priority between features is a business decision. Whether another user interface feature is more important than another report column is a business decision. The customer takes what they know of the market, combines it with their experience of similar systems, then picks a feature off a dartboard. No, actually it can be a little more systematic than that (sometimes it is, sometimes it isn't). Tough as it is to guess which feature to do next, the customer is in a much better position than the programmers to make this decision.

Business decisions in planning are:

- ◇ Dates
- ◇ Scope
- ◇ Priority

Technical decisions in planning are:

- ◇ Estimates

If we have the right people making the decisions, planning will go as well as possible—we'll be able to deal with our disasters. We'll do it by reducing the number of disasters as much as possible, by finding out about the disasters as quickly as possible, and by maintaining as many options as possible as long as possible.

Balancing political power may seem like a tall order for a simple project manager. If we can't do it in the Balkans after a couple of millennia of concerted effort, what chance do you have?

It's not so bad as it sounds. Our solution to balancing power is to create a simple set of rules that if followed tend to cause the technical people to make the technical decisions and the business people to make the business decisions. If we do a little of this planning game every day, we have the chance to catch and address problems as quickly as possible.

Chapter 9

Top Down Overview

We're going to outline XP two ways. If you who like an overall understanding before going into details, read this chapter first. If you like to go from smaller to larger scale, read the next chapter before reading this one.

Plant in the spring, harvest in the fall. The world works on cycles. Software development is no different. The planning challenge is that there are two cycles that we need to accommodate and synchronize--the business cycle and the development cycle.

The business cycle revolves around the activities of business:

- ◇ Press releases
- ◇ Software release, manufacturing and installation
- ◇ Training
- ◇ Billing

In the old days this cycle was a leisurely 2-3 years long. Recently, the cycle has tightened enormously, driven by widespread telecommunications and technical advances in the delivery of software. Still, the business cycle is at least months long. We will call one of these 1-6 month turns of the business crank a **release**.

The development cycle has always been shorter than the business cycle. The intent of having a shorter cycle was to correct projects in mid-course. Sometimes the interim deliverables documented certain kinds of decisions, as in the requirements, analysis, and design documents of a waterfall. Sometimes they were partly functional systems or sub-systems, as in incremental development.

The contraction of the business cycle requires a similar contraction of the development cycle. If we release every few months, then if we want

to be able to make mid-course corrections we must shrink the development cycle to no more than a few weeks. We will call one of these 1-3 week development cycles an iteration.

The problem with a few week cycle is that it is impossible to complete anything in a few weeks. You can't complete the analysis, or build the infrastructure, or set up the framework. We have to use some other measure of progress. Since the customers are paying for the software, we'll use a measure they understand--the story. A story represents a feature the customer wants in the software, a story they would like to be able to tell their friends about this great system they are using.

For a few stories to fit into an iteration, the stories have to be fairly small. A story should take a programmer a few days to a few weeks to implement.

What is the result of an iteration? Any activity that comes between finishing the engineering of a release and delivering it to customers represents a risk. We hate risk. So the result of an iteration must be a fully tested, production ready system.

This may sound impossible. But the first time you go through an iteration, the number of stories is small. Surely you can completely test and verify a system that small. If you invest in making the verification automatic, when you finish the second iteration the incremental cost of verifying both sets of stories is again small. If you never "get behind on your payments", if the result of each iteration is ready to be used, then the cost of maintaining readiness remains reasonable.

Add Kent's picture of a bunch of stories narrowing to a release, narrowing to an iteration, narrowing to tasks, narrowing to tests

The following seems nearly redundant with the text above

The highest level of plan we have is the release plan (Chapter 16). The release plan simply states which user stories are expected to be delivered in which release. We like to release early and often, so our releases are as small as we can make them. We may release every month, or every few months. Occasionally there are valid reasons for releasing every six months to a year, particularly with shrink-wrap software, but we try to push back on that as much as possible.

Our releases, however, often come too infrequently for us to get proper feedback on our progress, so we like to divide our releases into iterations. Again we plan by assigning user stories to iterations. An iter-

ation is two weeks (for some small value of two). Each iteration is a “pretend release”, and allows us to see where we are on the way to an actual release.

Release planning covers assigning user stories to releases and iterations. It’s done jointly by the customer and development, but the customer is holding the steering wheel. It’s a public plan that many people will look at to see what’s happening, but like all XP plans it’s one that changes frequently.

The next level of plan is the iteration plan (Chapter 23). This breaks down the user stories into smaller development tasks, each task is about 1-4 days of development effort. Iteration planning is started at the beginning of an iteration and only covers what is done during that single iteration. The iteration plan breaks down the user stories into development tasks and developers choose tasks to act on. The iteration plan is done by development with the customer advising.

The iteration plan is tracked as it goes on. If anything comes up during iteration planning that could affect the release plan, the release plan is updated with that information. This may cause the allocation of stories to iterations to change - a task that involves the customer.

When a developer starts a task, she will usually break it into smaller pieces driven by tests. This is a personal plan which is not shared around the team and is out of the scope of this book.

Chapter 10

Bottom Up Overview

Let's start with the favorite whipping boy of methodologists, the waterfall:

Picture of the waterfall

The waterfall presents you with a whole bunch of big problems to solves—specification, design, implementation, testing, integration, deployment, training, documentation. For all its faults, the waterfall accurately captures the fact that to ship software, you must make decisions at all of these levels before you are done. So, what if we shrunk the waterfall to microscopic size, say 2 weeks? We would still have to solve all the problems we had to solve before—specification, design, implementation, testing, integration, deployment, training, documentation—but now they would be small problems, not big problems, and some of them might disappear entirely.

Picture of the waterfall shrinking down to nearly nothing

We'll even rotate our little guy to emphasize that inside our little two week waterfall (let's call it an iteration, just to be conventional) there won't be phases, but we will still have to solve all the problems the phases were there to solve. Each problem will be solved throughout the iteration.

Iteration rotated

Our cute little iteration contains all the elements of full scale development. At the end of it, we have shippable software, ready to deploy. It just doesn't contain many features. Maybe even just one feature. So we will have to do another iteration and another and another (hence the name, we suppose). Each iteration will contain a few more features. Each will be more sophisticated than the last.

Iterations stretching out right and down

One more thing and the picture is complete. One of the purposes of planning is we always want to work on the most valuable thing possible at any given time. We can't pick features at random and expect them to be most valuable. We have to begin development by taking a quick look at everything that might be valuable, putting all our cards on the table. At the beginning of each iteration the business (remember the balance of power) will pick the most valuable features for the next iteration.

Thin slice of analysis above the iterations

Now we have a process where planning can do its proper job. The process makes sure we get off to as good a start as possible by laying out the whole of development. The process mitigates requirements risk by picking new requirements every few weeks. The process mitigates implementation risk by breaking planning up into small enough pieces that when one blows up, it will affect the overall plan as quickly and visibly as possible.

In short the trick to dealing with waterfalls is to make them short. As any paddler knows, it's better to kayak the Rogue River than Niagara Falls.

Chapter 11

Too Much to Do

Balloon Story

Imagine you have a balloon:

Picture of an empty balloon. One end labelled "time" the other "bugs"

This is perfect. We can get done quickly and we have great quality. Then along comes a customer and tells us they want to actually do something.

Picture of a customer blowing into the balloon, inflating the "time" end

Sigh. Now our picture is ruined, because we can't get done quickly. Hey, this is a balloon, of course we can get done quickly. Just squeeze the time out.

Picture of a hand squeezing "time" and "bugs" expanding

Uh oh, when we hurry up, we get more bugs. Here's the really nasty part, when bugs get bigger, time gets bigger, too, and the more we squeeze the bugs, the bigger the time gets.

Picture of the balloon fully inflated

What's even worse is that if we don't squeeze the bugs out, time grows even faster.

This conundrum is what makes project planning such a joy. Going faster makes you go slower, and the loop creates positive feedback, so that eventually you go so slow creating such crummy software that the project dies.

What's a planner to do?

Not "Not Enough Time"

We were on a project together once that the team had rescued from impending oblivion. Towards the first release there came a time when it was clear to everyone that we weren't going to make the release date. One day we had a stand-up meeting to discuss the problem. We went around the circle answering the question, "What is preventing us from going into production?"

"I don't have enough time."

"I don't have enough time."

"I don't have enough time."

...

Nobody had enough time. But there was no obvious answer. Everybody went home.

Over Korean food that night (we always ate Korean, it seems to be good for the brain), we were talking about the meeting. Suddenly a cosmic ray collided with a piece of kimchee and we saw the real problem.

The next morning we called another stand-up.

"Repeat after me—I have too much to do."

Shrugs all around, but what the heck.

"I have too much to do."

"I have too much to do."

"I have too much to do."

And so on around the circle until we got to Richard.

"I have too much to do. What is your point?"

The point is that when you don't have enough time you are out of luck. You can't make more time. Not enough time is a position of hopelessness. And hopelessness breeds frustration, mistakes, burnout, and failure.

Having too much to do, however, is a situation we are all know. When you have too much to do you can:

- ◇ Prioritize and not do some things
- ◇ Reduce the size of some of the things you do
- ◇ Ask someone else to do some things

Having too much to do breeds hope. We may not like being there, but at least we know what to do.

Chapter 12

Four Variables

We use four variables to help us think about how to control a project: cost, quality, time and scope. They are inter-related and effect each other in ways that are not obvious.

We've all heard statements like "cost, time, quality: pick any two". Plenty of people have ways in which they talk about how there are these variables involved in getting something done, and that you can't control them all at once. Well we choose a set of variables too, and we find them a good set to work with. They are:

- ◇ Cost
- ◇ Quality
- ◇ Time
- ◇ Scope

We like to think of them as four levers on some big victorian steam machine. The four levers control the machine (which is our project, of course). If you move any lever the others move. You can lock any lever you like, but if you lock three levers you cannot move the fourth.

The catch, however is that the effect of moving a lever is both delayed and non-linear. You can't just double the cost, hold everything else the same and halve the time. So each lever gets it's own little instruction manual. The good news is that the manual wasn't written by a second rate victorian novelist.

Cost

If you look at the cost lever carefully you quickly see that it's actually several mostly independent levers. Moving any of them can increase or

reduce your costs, but each lever has a different effect on the three other primary levers.

The most powerful lever is that of people. You increase this lever by putting more people on the project. This lever, however, suffers from having both a non-linear effect and a long delay.

The non-linearity comes from the communication overhead of having more people. Doubling your team doesn't make you go twice as fast because it increases the amount of communication that needs to go on. There isn't really much guidance we can give you on this, partly because there isn't the data and partly because so many other factors have an effect. All you can do is add some people and see the effect.

The trouble is, you'll have to wait to really see the effect, since the result causes several changes that take time to play out. The immediate effect is the alarming sight of nothing happening, or even worse a slowing down. When a new person joins a running team he will initially be of little value because he doesn't know the system nor the team. Indeed he can slow things down because he drains time from other people as they teach him about these things. The more people you add, the bigger this slow down effect is. Add enough people and the project can come to a big crunching halt. This is the origin for Brook's Law ("adding people to a late project just makes it later").

So we advise to add just a few people to a team, and for at least an iteration, don't expect any speed up. Over time you will see some benefit but it may take several iterations before you see a result.

Remember that extreme programming has a limit on how many people can be in the team. We fix the upper limit at about a dozen developers. Beyond that you need a different process. However many of the XP practices, including this planning approach, will still be useful.

There are other ways to spend money. Spending on tools can be like adding people. You get a slow down as people learn how to use the tools. Only when they become comfortable with it will you know how good the result is.

Some money can have a very good return: faster computers, bigger monitors. Don't be afraid to spend money to keep motivation up. Well motivated developers are much more effective than people whose motivation sags.

Overtime doesn't help. Although in the very short term it does speed up the machine, if you do it for any length of time you will get bitten badly. Often the big killer is motivation. It's much better to have a motivated developer work seven hours than a demotivated developer work ten. However even if the developers want to do long hours it's not a good idea. Long hours make people tired, tired people make mistakes, and mistakes take time to fix. We've both gone into clients in the morning and spent all day chasing a bug that was put in at ten o'clock the previous night. Particularly with young silicon-valley teams we have to work hard to get people not do overtime. If they really have no life get them to play computer games in the evening instead: it's much more productive to have castles mown down by trebuchets than it is to slip bugs into complicated software.

Quality

Quality is really two levers: external and internal quality. External quality is the quality perceived by the customer. This includes bugs, but may also include non-functional requirements such as how the GUI looks and how fast the software is.

For non-functional things try to move them over to scope. Make a story for something like "make the UI more pleasing" or "get average claim processing time to under 300ms". As we'll see later scope is the best lever to operate.

Bugs are often also a scoping issue. Often you may want to trade off defects for features. We'll talk about this more in Chapter 30.

The other lever is internal quality. This reflects the quality of the internals of the system: how well it is designed, how good the internal tests are, etc. This is a very dangerous lever to play with. If you allow internal quality to drop you'll get a small immediate increase in speed, rapidly followed by a much bigger decrease in speed. As a result you must keep an eagle eye on this lever and make sure it is always up as far as it can go. In time nothing kills speed more effectively than poor internal quality. That's why extreme programming puts so much attention on practices like testing and refactoring to keep internal quality high.

Time and Scope

This is what this book is about: how to manipulate the time and scope levers.

Chapter 13

Shopping For Stories

We heard someone complain recently that there aren't any laws of physics in software. They meant that in software nothing is impossible. Every programmer has had the experience of estimating a piece of work at six weeks, only to have the pronouncement come down, "But it has to be done in three."

If there were laws of physics for software, this just wouldn't happen. No one goes to SwissAir and asks them to fly a 747 from Zurich to San Francisco in four hours. The laws of physics prevent it, and no amount of bullying or bribing or "stretch goals" can change that.

The problem with having no laws of physics is that programmers are from time to time asked to do the impossible. In trying, they end up doing much less than they could if they had a difficult but possible problem to solve.

So, we need a planning style that

- ✧ preserves the programmers' confidence the plan is possible,
- ✧ preserves the customers' confidence they are getting as much as they can,
- ✧ costs as little to edit as possible (since we'll be using planning to steer we'll be planning often)

Here it is: what if planning for a piece of software was like shopping? When you go grocery shopping for the week, you have a budget. You go into the store and look around at the items, their prices, and you think about what you need to accomplish. If you are feeding a horde of teenagers, you tend towards rice and beans. If the boss is coming over for dinner, you get steak for one night and go easy the rest of the week.

The elements of the analogy are:

- ◇ The items
- ◇ The prices
- ◇ The budget
- ◇ The constraints

Planning for Extreme Programming uses the analogy thusly:

- ◇ The items are the features required of the software (described as stories)
- ◇ The prices are the estimates on each story
- ◇ The budget is the team's measured progress in terms of these estimates
- ◇ The constraints are the business and technology constraints as they are discovered by the customer

The shopping analogy can carry us a little further.

- ◇ Sales—If reports turn out to be easier to implement than expected, that's having a sale on reports. "Attention software shoppers. Reports are going two for one on aisle 14."
- ◇ Rain check—If you have to discard a new wizard in the middle of a release to save the end date, that's taking a rain check. "IOU one wizard."
- ◇ Price hikes—If graphics are harder to add than expected, the prices go up. "Due to circumstances beyond our control, graphics are now \$1.49/pound."

Any time we have to decide what to do we will go shopping. Who chooses, how big the items are, and who sets the prices will all vary, but the strategy is the same. We will shop for \$5,000,000 worth of software and we will shop for next week's tasks.

Chapter 14

Yesterday's Weather

You can't put five pounds of shit in a ten pound bag.

--anyone who has tried

How big is the bag? This shopping metaphor is all well and good, but what is the budget? How much do you commit to doing in the next N months?

If you commit to too much, development proceeds under a cloud. The programmers know they are doomed. They don't do their best work. They don't communicate clearly. The political sophisticates play Schedule Chicken, where the first person to point out the impossibility of the task ahead is labelled a loser, not a "team player".

Kent keenly remembers the end of a project review. He asked, "What one question would you like me to ask upper management?" The response, "Why are they saying we will be successful when every single programmer knows we are going to fail?"

Okay, we don't want to do that. Neither do we want to undercommit. If it turns out we can go twice as fast as we thought we could, the business will take a while to catch up. The press releases won't mention half of the cool new features. Sales won't understand what all is in the product. And it is a matter of pride for programmers to put out 100%.

How can we navigate such an emotional and business minefield? How can we come up with a complicated rule-making apparatus that accurately captures and balances all the technical and emotional information available?

We don't, surprise, surprise. Instead, we opt for a simple rule that works pretty well in most circumstances:

◇ Say you'll do as much tomorrow as you actually got done today

The Story

Apocryphal story--Some country's weather service (not yours, but perhaps ours) spent a bazillion dollars on a sophisticated new weather prediction system. Lights flashed, cards spewed, tapes spun and out came predictions that were about 70% accurate. The people who authorized spending the bazillion dollars were quite impressed.

Then one day some noticed a simpler way to get the same accuracy. Every day predict that tomorrow's weather will be exactly the same as today's.

That's why we call our rule Yesterday's Weather.

How it works

Assume for the moment that each feature you are going to implement takes the same amount of effort (see Chapter 1 for what we really do). If we did 5 features last month and we're asked how much we can do this month, we say "5". If we have gotten 3 features done every two weeks for a while and we're asked how much we can do in the next six months, we say "3 features / iteration * 2 iterations / month * 6 months = 36 features".

Here are some of the emergent properties of this rule:

- ◇ We won't habitually over-estimate our abilities, since we have to go against actuals. If we over-estimate once, we won't be able to the next time.
- ◇ If we are over-committed, we will tend to finish some items in any given time period instead of half finishing them all, since it is so embarrassing to tell the customer they can have zero next time.
- ◇ On the other hand, if we have a disastrous period, we are guaranteed a little breathing space to recover.
- ◇ The customer learns to trust the number, because it is so easy to explain.
- ◇ Our estimates automatically track all kinds of changes--changes to the team, new technology, dramatic changes in product direction.

Chapter 15

Scoping a Project

Is it bigger than a breadbox?

Let's say you're the only one on the project so far, what's the first step? How can you use shopping to bring a project into existence?

- ✧ Items- Big pieces of functionality (we call pieces of functionality "stories" in XP)
- ✧ Prices- Rough estimates of the time to implement each item
- ✧ Budget- Roughly how many people you have to work on the project
- ✧ Constraints- Supplied by someone with business knowledge

The purpose of this first plan is to quickly answer the question, "Does the project make any sense at all?" Often these sanity plans are made before there are any technical people on the project at all. Don't worry about getting perfect numbers. If the project makes any sense then you'll invest enough to get you a plan you have some confidence in.

What if we were to implement a space-age travel system? (For the full story of the system see "Examples of Stories" on page 97.) We might have big items like:

- ✧ Book a space flight
- ✧ Book a hotel
- ✧ Check itinerary
- ✧ Adventure trips
- ✧ Planetary weather
- ✧ Holographic planetary simulation
- ✧ Cross species orientation

◇ Auto-translator

Before we can assign prices to them, we have to know a little more about the system. We ask a few questions:

- ◇ How many reservations do we need to handle?
- ◇ How much of the time do we need to have the system available?
- ◇ What kind of machines will be used to access the system?

With that in mind, we can guess at how long a team of 10 would need to implement each feature:

- ◇ Book a space flight - 2 months
- ◇ Book a hotel - 1 month
- ◇ Check itinerary - 1 month
- ◇ Adventure trips - 2 months
- ◇ Holographic planetary simulation - 6 months
- ◇ Cross species orientation - 4 months
- ◇ Auto-translator - 8 months

We make some simplifying assumptions as we go along.

- ◇ The items are completely independent of each other
- ◇ We will develop the necessary infrastructure along with the story, but only the infrastructure absolutely needed for that story

We know these assumptions aren't exactly accurate, but then again neither is anything else. If we were trying to predict the future, this would worry us. Since we aren't, it doesn't.

So, the bottom line is that we can implement the system in 24 months.

Then the shouting starts. "We have to go to market in six months, tops, or we're dead." Yes, we understand. "If you can't do it, we'll hire someone who can." You should do that, but perhaps we can talk a little first. "You programmers can't tell me what to do." Of course not, but perhaps you would like to know what you can't do.

Now the negotiation starts. "What if we just made a booking system first? We'd need the first three stories. That's four months. But we can't launch without the holographic simulation. What can you give me in two months?..."

A few hours or days or weeks later, we have a rough plan from which we can move forward.

Making the Big Plan

Picture with a big fuzzy cloud on the left with a big question mark in it, a "transformation" arrow, and on the right 3-4 boxes of various size with smaller question marks, a horizontal line, and below that a smaller cloud.

The purpose of the big plan is to answer the question, "Should we invest more?" Figure N shows the three ways we address this question:

- ◇ Break the problem into pieces
- ◇ Bring the pieces into focus by estimating them
- ◇ Defer less valuable pieces

Start with a conversation about the system (this works best if you involve at least one other person). As you talk, write down your thoughts, one per index card. If your thoughts get too detailed, stop writing until you get abstract again.

Some cards will contain business functionality. These are stories. Lay these out in the middle of a big table. Some of the cards will contain ideas that are context- throughput, reliability, budget, sketches of happy customers. Set these to one side.

Now you need to estimate how long each story would take your team (just guess at a size at first) to implement. Give yourself plenty of padding. There will be plenty of time for stone cold reality later. Bask in the glow of infinite possibilities for the moment.

If your estimates are too small (like days or weeks), you slipped into detail land. Put those cards to one side and start over. If you can't imagine being able to estimate a story ("Easy to Use" is the classic example), put it to one side. Better yet, think about some specific things that would make the system easy to use, and turn them into stories (e.g. "Personal Profiles").

You can only estimate from experience. What if you don't have any experience? Then you'd better fake it. Write a little prototype. Ask a friend who knows. Invite a programmer into the conversation.

Move fast. You're sketching here, trying to quickly capture a picture of the whole system. Don't spend more than a few hours on your first rough plan.

What, Me Worry?

Kent has a client where they based their business plan on a set of big stories much like the ones above (different topic, naturally, since space travel seemed a bit dicey, even to venture capitalists). When the team began implementing, they measured their progress at about 40% of the original plan. Amazingly, this is perfectly okay!

Nobody is happy that they are going slower than the plan, but the team's progress is not putting the company at risk. Far from it. Because they can demonstrate their progress regularly, and because they can react as the market changes, investors and customers believe the company will continue to do great things. About half the functionality actually implemented was in the original plan, and the other half has been brought in or invented in response to real customers.

Gotta love the Internet. Where else does accomplishing 20% of your original plan make you a superstar?

Chapter 16

Release Planning

Release planning allocates stories to releases over a time horizon of a number of months - typically focusing on a public release.



The big plan helped us decide that it wasn't patently stupid to invest in the project. Now we need to synchronize the project with the business. We have to synchronize two aspects of the project:

- ◇ Date
- ◇ Scope

Often, important dates for a project come from outside the company:

- ◇ The date on the contract
- ◇ COMDEX
- ◇ When the VC money will run out

Even if the date of the next release is internally generated, it will be set for business reasons. You want to release often to stay ahead of your competition. If you release too often, you won't ever have enough new functionality to merit a press release, a new round of sales calls, or champagne for the programmers.

Our units of time are the release and the iteration. A release is the amount of time you work on the software before you put it into production. We're aggressive about going into production early and often, because it provides such good feedback. So we want to release at most every few months. If your release lengths are longer than a few months see Chapter 17.

Iterations are two week cycles, which we use for controlling progress within a release. Each iteration finishes with a fully integrated, tested, ready-to-release system. However most organizations aren't prepared to release every iteration, so you need the larger unit of time. If you can then you may not need the notion of release at all, we'll talk about that in Chapter 17.

Our unit of work-to-be-done is the user story (or just story). A user story is a chunk of function that makes sense to the customer, and is small enough that you can do several in one iteration. A lot of people use the terms feature or requirement item for this. We use "story" because it helps people to focus on the notion that the users describe what they need.

Release planning is then simply allocating user stories to releases and iterations.

Picture with the right side of the big plan picture on the left, a transformation arrow, and on the right 20 smaller boxes with smaller question marks in them, a horizontal line, and a few boxes below the line.

Figure N shows the big plan turning into a release plan. The strategies you will use are similar to making the big plan in the first place:

- ◇ Break the big stories into smaller stories
- ◇ Sharpen the focus on the stories by estimating how long each will take
- ◇ Defer less valuable stories until what is left fits in the time available

Who does Release Planning?

Release planning is a joint effort between the customer and the programmers. The customer drives release planning and the programmers help navigate. The customer chooses which stories to place in the release and which stories to implement later, while the developers programmers provide the estimates required to make a sensible allocation.

The customer

- ◇ defines the user stories
- ◇ decides what business value the stories have
- ◇ decides what stories to build in this release

The programmers

- ◇ estimate how long it will take to build each story
- ◇ warn the customer about significant technical risks
- ◇ measure their team progress to provide the customer with an overall budget

How Stable is the Release Plan?

Not at all.

The only thing we know for certain about a plan is that things won't go according to it. So release planning happens all the time. Every time the customer changes their mind about the requirements and their priority, this changes the plan. Every time the developers learn something new about the speed of doing things, this changes the plan.

The plan is therefore just a snapshot of the current view of what things will be done well. This snapshot helps people get an idea of what to expect, but is no statement of certainty. It will be revised frequently. Everyone: developers, customers, and management need to accept this constant change.

How Far in Advance Do You Plan?

How far in advance do you build a release plan for? We know that the longer ahead we plan, the less accurate we will be, so there's little point going into great detail for years into the future. We prefer to plan one or two iterations in advance and one or two releases in advance.

The one or two iterations means that the programmers clearly need to know what stories are in the iteration they are currently working on. It's also useful to know what's in the next iteration. Beyond that the iteration allocation is not so useful.

However the business needs to know what is currently in this release, and it's useful to know what's in the release after that.

The real decider for how long in advance you should plan is the cost of keeping the plan up to date versus the benefit you get when you know that plans are inherently unstable. You have to honestly assess the value compared to the volatility of the plans.

How do you Keep the Release Plan?

Our preferred form of release plan is a set of cards. Each card represents a user story, and contains the essential information to describe what the story is about. You group the cards together to show which stories are in this release. Lay out stories with adhesive on a wall, or pin them up on a cork board. Wrap future stories with a rubber band and stick them safely in a drawer.

We like cards because they are simple, physical devices that encourage everyone to manipulate them. It's always that little bit harder for people for people to see and manipulate things that are stored in a computer.

However if you want to store this in a computer, go ahead. Just do it in a simple way, preferably in a way that just a little too simple. A simple spreadsheet often does the job best. We've often noticed that people who use complicated project management packages are prone to spending time fiddling with the package that they should be spending communicating with other people.

How much can you put into a release?

If you have stories, iterations and releases, you need to know how many stories you can do in each iteration and release. We use the term **velocity** to represent how much effort there is available in a release. We estimate the velocity of the team and estimate the amount of effort required for each story (see Chapter 19 for details on estimation).

Then when planning the simple rule is that the sum of the effort for all the stories in an iteration cannot exceed the velocity. Effectively the velocity is the budget for your shopping trip. Each iteration you can only buy as much as your budget allows. Similarly the amount of stories you can do in a release is the velocity times the amount of iterations in the release.

Release Planning Chapters

The next few chapters discuss the various elements of release planning in more detail.

- ◇ Chapter 19 talks about how you estimate how much it will take to implement a story and to figure out what the velocity of the team is.
- ◇ Chapter 18 discusses how you write stories so that they allow you to break up the features of the system into useful chunks. It also talks about why the stories don't need to be too detailed.
- ◇ Chapter 20 contains advice on what order you should do your stories, in short do the high value stories first but with half an eye on technical risks.
- ◇ Chapter 21 talks about the various events that cause you to do more release planning in the middle of a release.
- ◇ Chapter 22 talks about how you come up with the first plan, which is both always the hardest and the least accurate.

You'll notice that we begin by talking about how you plan in the middle of the project first, and later talk about how you come up with a first plan. You may find this frustrating as you probably will need a first plan first. But we noticed that with continuous planning, the first plan is actually quite an oddity. So we set the process up to make later plans easier, the first plan is a special case.

Even if you start with a first plan, it's important to be familiar with what planning will look like when you get going. In particular this is important because you'll find planning will get both easier and more accurate once you have a few iterations under your belt.

Chapter 17

Release Planning Variations

Everywhere XP is adopted, it undergoes rapid evolution (see Chapter 40). Here are a few variations we have encountered.

Short Releases

Sometimes you can release much more often, maybe every iteration. This can happen for in-house development, and also for ASP situations where your users are distant but using thin clients and you have close control over the server.

Most of this is good news. Each iteration is ready for production, and going into production with each iteration is perfectly feasible. This usually means that you need to have high confidence in your tests and a very automated build process, but if you are doing XP you should have these anyway.

With short releases you don't really need any notion of a release at all. You can work iteration to iteration, planning one or two iterations ahead. This allows you to be very responsive to the customer, giving the customer very good and close control of the process with rapid feedback to see the results.

However there is a danger. The danger is that the customer will lose strategic vision of where the software needs to go. In this case the customer spends so much time choosing features for the short term, that they miss important long term needs.

So here the release can come back as a longer term milestone. By the end of the next quarter we hope to be here, and by the end of the following quarter we want to be there. So even if you release into production every iteration, still think about planning out a release on quarterly

scale. Even though the resulting plan isn't very helpful, the act of planning those longer releases is vital. (Didn't some General say something like that?)

Long Releases

What happens if you can only release once a year?

Our first reaction to this reality is to question it. Maybe there is some way to release more frequently.

A common case for long releases is in replacing an existing system. Since you have to do all the things the old system had to do, you can't release something with only some of the features.

In this case look for a way to let the old and the new co-exist. Manage some contracts with old system and some with the new, gradually moving the contracts over with each release. This may involve extra work to migrate data between the systems and to integrate the user interfaces, but the resulting drop in risk is usually worth it.

Another case is with shrink-wrap software. Many users just will not want to upgrade their software every quarter, nor will the marketing department cope with the resulting flux. In this case look for a way to send intermediate releases to those customers that may be more interested in these versions. Call them service packs or something. That way at least some of your users will use the system in production, and you'll get the feedback you need.

Frequent releases are good, but if you can't then you don't have to abandon XP completely. In this case you'll need to create interim releases that are only available internally. These will be enough for the friendly users to play with in controlled conditions and to provide a focus for planning purposes.

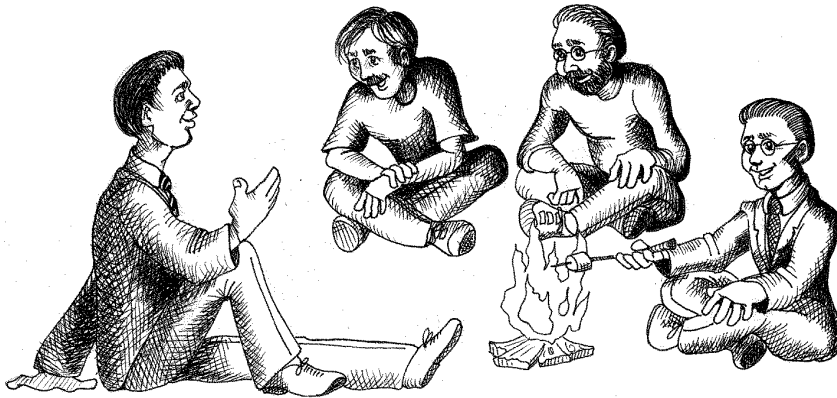
Small Stories

Some teams like to have more smaller stories. Instead of four or five two week stories, they will plan 25 two day stories. This gives the customer finer control over the activities of the team, at the cost of some of the flexibility of the team and more involvement by the customer.

Chapter 18

Writing Stories

The story is the unit of functionality in an XP project. We demonstrate progress by delivering tested, integrated code that implements a story. A story should be understandable to customers and developers, testable, valuable to the customer, and take about 1-3 ideal weeks to build.



A user story is a simple description of a feature in the software the customer is willing to pay for. The emphasis on the word *simple* is essential. There are many books out there that will go into great detail about "requirements engineering", use case design and similar topics. These present some useful ideas, but as ever XP looks for an approach which is the simplest approach that could possibly work.

You don't need nearly as much information about a feature to give a plan-worthy estimate and to make priority decisions about it relative to its mates as you do to actually implement the feature. XP takes advan-

tage of this by only writing down the extra detail when implementing the story, and leaving the rest of the requirements process light and flexible.

Principles of Good Stories

Stories must be understandable to the customer. It's no good making the requirements so difficult to write and organize that you need years of training in requirements engineering to be able to understand them. So the form for a story is plain english (or whatever your local language is.) Everyone can speak natural language, anything else is just unnatural.

We like to write user stories on index cards. This keeps them terse, and also makes them easy to manipulate during planning sessions. Developers and customers can pass a card back and forth, put it in a certain location on the table, put it back in the deck, pin it up on a cork board etc. For the purposes of planning is much easier to manipulate concepts on cards than it is to manipulate a printed list of items.

(If you're determined to put them into a computer, do so in a way that you can easily print them out on cards using standard printer card stock.)

The best user story is a sentence or two that describes something important to the customer. For example:

The system should check the spelling of all words entered in the comments field.

The shorter the story the better. The story represents a concept, and is not a detailed specification. A user story is nothing more than an agreement that the customer and developers will talk together about a feature. Remember, hideous written detail is not necessary if the customer can work with the programmers while they are programming. The key thing is that there should be no significant surprises when the time comes to do the iteration plan and build of the story.

It's not that you don't need all of those details. You just don't need them all up front. When the stories are built, you can get these details verbally, or with a document. Our favorite way of sorting out details is functional test cases. But whatever you do, do it then, because you won't really understand the details until then. Of course this leaves you

with some uncertainty, but we've not found that more detail avoids the uncertainty, all it does is give the illusion of certainty — which we think is worse.

Each story must provide something of value to the customer. If the customer isn't getting value, why would they want it? This means that any technical infrastructure must be built in conjunction with the stories, and must be developed to support what the stories need. This helps to avoid projects spending weeks providing clever infrastructures that aren't really needed.

Developers do not write user stories. Any “good idea” or “neat new feature” that a developer dreams up will not appear on a user story unless the customer agrees that the idea is good, or that the feature is neat. The customer has the sole responsibility for supplying the team with user stories, and no one can usurp this responsibility.

One of the hardest things about stories is how big to make them. Stories need to be of a size that you can build a few of them in each iteration. This size works give you the ability to steer by shifting use cases between iterations. In practice this means that each story should be around 1–4 ideal weeks of effort. It also means that the developers should be able to estimate how long it will take to do a story. If they can't, it usually means the story needs to be broken down into smaller parts.

This also implies that the stories require communication between the customer and developers. The customer must write the story, the developers then estimate the story. The two parties must collaborate and communicate to do this.

Stories should be independent of each other. This allows us the freedom to build them in any order. This is, of course, impossible. But in practice we find that if we pretend it is possible, most of the time we get away with it. We do get dependencies, but rarely will they cause a problem with the planning process. If there's an effect that would change the estimates depending on what order the stories are done, then we can just write that on the card. We don't find we get too many of them.

When the story gets built it will be important to be able to tell that it works. So each story must be testable. This doesn't mean writing the

test case right now, but it does mean that you should be able to figure out how to test it.

Feedback from Estimation

In Chapter 19 we talk about how you estimate a story. You should start getting estimates as soon as you start writing stories. The main reason you want to do this is to get some feedback as to the right level of detail for estimation.

It is very common for a user to write a story that cannot be easily estimated. For example:

It should be easy for the user to diagnose a failure.

What does this mean? How can we estimate it? Since we don't know what it means, we cannot estimate it. Thus, we ask the customer to explain what would make diagnosing a failure easy, and what has made it hard in the past. We ask the customer to describe what operations they envision that would make the process easy. For example:

Regardless of the state of the system, the user should be able to do something to the icon of a failed device that starts the diagnosis process.

The customer had made several things clear. First, "easy to diagnose" really means "easy to start the diagnosis process". Secondly, he envisions device icons on the screen. He also thinks it's important that the users be able to do things to those icons. This gives us enough context to estimate the story.

Programmers don't need infinite detail in order to estimate; they just need the customer to translate his needs into something concrete that they can take action on.

Prioritizing User Stories

It doesn't really help to ask for absolute rankings of stories like "high/medium/low". All the stories end up being high, or the customer wouldn't have written them, now would they? Instead, the customer needs to prepare to answer the question, "What do you need now, and what will we implement later?"

Stories often mix priorities. For example:

Test results must be displayed within 10 seconds after requesting the test, and must contain the dispatch statement and all measured and derived readings.

What the customer really considers to be of high priority within the story is the appearance of the dispatch statement within 10 seconds. If it takes longer for the measured and derived readings to appear, that's not a problem. Under the press of limited time, though, the customer will discover what's important and what isn't.

It is almost impossible to find this kind of baggage early on. What usually happens is that the programmers will challenge the priority of individual elements of a story during iteration planning, or even development. Once the baggage is detected, regardless of when, the story can be split.

Sizing User Stories

A good user story represents between one and three ideal weeks of work. Stories that are estimated much above three weeks should be split. Stories that are estimated much below one week should be merged. (Paper clips are effective tools for merging several cards into a single unit.)

Testability

Eventually, the customer will have to specify acceptance tests whose execution will determine whether the user stories have been successfully implemented. Thus, all user stories must be testable. The programmers and the customer alike must agree that there are discrete tests that will demonstrate that the user story is successfully implemented.

Acceptance testing is also our answer to traceability. We don't think it's worth trying to trace requirements to classes in the system that implement the requirement. That's too much paper work and even if you have a tool, it's still too much effort to keep it all up to date. Instead we prefer the simpler trace between story and the functional tests. If the acceptance tests work, we can safely assume that some code exists that maps to the story. If we change that code later in such a way that breaks the story, then the test will fail. This is the ultimate in traceability.

Splitting user stories

Splitting a user story into two or more smaller user stories is extremely common. The mechanism is simple. The customer divides the concept into two or more smaller concepts, write the new cards, and throws the old card away. The developers then estimate the new stories, and the customers prioritize them.

The dispatch statement must be displayed within 10 seconds after requesting the test.

Test results, other than the dispatch statement, must be displayed within 30 seconds after requesting the test, and must contain all measured and derived readings.

Many events can trigger a split.

- ◇ When you're writing the initial stories the developers may say a story is too large. This may mean it's more than four ideal weeks of effort, or simply just too big to estimate.
- ◇ When you're doing release planning you find you cannot fit all of a story into an iteration. Then you can split the story so that part of it can be done in the iteration.
- ◇ As you're tracking an iteration you realize you have too much to do. To cut scope you may split a story.

To split a story a customer begins by suggesting a split. The customer should make the split along priority lines, factoring out some less vital work that can wait till the next iteration. Once the customer has made the trial split, the developers then estimate the two parts as if they were separate stories.

Of course this is not likely to work out just right first time. So if the split leaves the lower priority part too small then the customer needs to move more work into the lower priority piece. If the low priority piece is too big then the customer may move some work to the higher priority piece, or they could leave it as it is and move some other work into the iteration to make up for the difference.

You'll go back and forth a few times on this, but if everyone is in the room together it won't take too long.

User Story Adornments

We prefer to keep our user stories uncluttered. They contain only three pieces of data: The story itself, its estimate, and its priority. There is no doubt that other data items could be invented. The list of possibilities is nearly endless. But we recommend that you refrain from “wouldn’t it be nice ifs” and “it might be convenient ifs”. An uncluttered user story, written on a card, is easier to manipulate, and will save more time and effort than the cost of the occasional lack of convenient data.

The story writing process

It is tempting to write down a simple little cookbook, “Great User Stories in Just 30 Minutes.” It would make your customer’s job much easier. All except that part where we can’t do it, any more than you can write a cookbook for great literary story telling. The best we can do is tell you how to spot bad stories, and suggest you spend a lot of time going back and forth between customer and programmer while you’re feeling your way.

The process of writing stories is iterative; requiring lots of feedback. Customers will propose a story to the programmers. The programmers will consider whether or not the story can be tested, estimated, and whether or not it is of appropriate size. If the story is cannot be estimated, the programmers will ask the customer to clarify the story. If the story is too big, the developers will ask the customer to split the story.

Stories should be written a few at a time. The programmers should sit down with the customer, figure out somewhere between 2-5 stories, and then stop. The programmers then work on developing the estimates of those stories. While the programmers are working on these estimates, they will certainly want to talk to the customer about the details and issues. So the programmers and the customer are in constant communication.

In some cases the estimates may be obvious. Perhaps the story is similar to others that have already been completed. In other cases the story may be very difficult to estimate, and may require exploratory programming by one or more of the team. Such exploration rarely takes longer

than a day, and you should chuck out any code that you wrote while doing the exploratory programming.

Once the first few stories have been estimated, the programmers and the customers will have a better understanding of what makes for good stories. An image of the system is beginning to form in their minds. From this superior viewpoint, they are ready to write the next few user stories and repeat the process.

Don't be too impressed with your stories. Writing the stories is not the point. Communicating is the point. We've seen too many requirements documents that are written down, but don't involve communication. Remember that just writing something down (or just speaking it) is not enough to communicate. For communication to work the listener (or reader) must also understand what was said.

When are you done writing user stories?

Never; or at least not until the project is cancelled. Software projects are living entities. They grow and change over time. While user story writing will be most prolific during the early phases of the project, it will never end.

When have you done enough user stories to begin developing? When customer is certain that the most important user stories have been identified, and you have enough for 2-3 months worth of development.

Usually the start of the project contains a lot of story writing as you try to get a broad feel of the desired functionality of the system. This list of stories will drive the initial release plan.

However list of stories is only an initial list. As the project goes on stories will be added, dropped, and split all the time. Not just is this normal, it is the whole point of XP. The aim to deliver software that matches the requirements at delivery, not to match the requirements as they were at the beginning of the project.

Disposition of user stories

Once a user story has been implemented, and its functional tests are running, the story itself serves no further purpose. The development team might think of tiling their floors or papering their walls with them; but discarding them is probably the best option. Remember that

they are encoded in far more detail and accuracy in the acceptance tests, so no information will be lost if the cards themselves are destroyed.

Examples of Stories

"Tell me story..."

-every child at some point

One of the problems about talking about stories is giving examples. We can't usually show examples from real projects because of client confidentiality. This presents a bit of a problem.

For this book, however, we got lucky. Kent was clearing some woods from his farm in Oregon when he came across a reverse time capsule. The capsule contained project information about a travel booking project that we did in the twenty-fourth century. (Or is that "will do", trans-temporal tense can get very confusing.)

One of the things inside this time capsule were story cards for the project.

Find Lowest Fare

Present to the customer the ten lowest fares for a particular route

Show available flights

Show possible flights (with connections) between any two planets.

Sort available flights by convenience

When you're showing the flights, sort them by convenience: time of journey, number of changes, closeness to desired departure and arrival time

Purchase ticket

Purchase ticket charging to a credit card. Check credit card validity when doing this. Also check broad immigration rules (no Oolavoos to go to Traal etc.)

Customer profile

Keep customer details for quick reference e.g.: credit card info, home address, dietary and gravitational needs.

Review itineraries

Show all tanneries the customer has in the system.

Cancel itinerary

Customer cancels an itinerary. Cancel all flights, hotels etc.

Print immigration paperwork

Prints paperwork required to leave and arrive at a planet. Only for the easier planets (e.g. not Vogon).

Show Hotels

Show hotels near a place.

Show hotel availability

Show hotels available for the period indicated by the spaceflight itinerary.

Sophisticated Hotel Search

Allow customer to search for hotels using more than dates and location. This would include facilities, level of service, costs, and guide recommendations.

Book a Hotel

Book a hotel. Charge to credit card, check credit card validity.

Hotel / Spaceline programs

Show hotels that have joint sales agreements with the spaceline the customer is using. Show the prices including the discounts with these programs. Only for spacelines that are actively collaborating with us at this stage.

Airplane hire

Allow the customer to hire an airplane while they are on a planet. Link the dates in from the space flight. Enhance the customer profile to include airplane preferences (insurance selection, manual vs. automatic etc.)

The stories show by example the principles we've shown so far. Each one is short, too short to act as a detailed specification for programming. However they give enough information to act as this common expectation. For anyone who has used a web based travel reservation system, they do give a good picture of what is needed at each stage. For people without that background, you may well need a little more information, but we should still avoid great detail at this stage.

Notice how there is no discussion at all of dependencies. You could well argue that to do the *Sophisticated Hotel Search* you would need to do *Show hotel availability*. The order in which you tackle them might affect the estimates. But essentially we don't worry about it. If it does make a significant difference to the estimates, and that makes an effect to the planning process then a simple note will do the trick then.

Chapter 19

Estimation

Is it bigger than a breadbox?

--20 Questions

Base estimation on finding a similar piece of work that you've already done. It'll take about the same amount of time as that.

How do you come up with estimates? We've seen a lot of words devoted to this topic. We've seen quite a few mathematical formulae. The best of these are based on the lines of code to be written. These can tell you how much time it will take to develop so many thousand lines of code. This is particularly useful since it's so easy to estimate precisely how many lines of code a solution will be before you start writing it. (Spotted the sarcasm yet?)

Let's be clear, estimation is a black art at best. You're not going to get accurate figures however hard you try. You can, however, get good enough numbers with a little bit of effort, and you can get better numbers over time.

There are two keys to effective estimation

- ◇ Keep it simple
- ◇ Use what happened in the past

The best guide to estimating the future is to look for something that happened in the past that was about the same as the future thing. Then just assume that history will repeat itself, as it often does. If there's a significant difference between then and now, then use a very simple way to factor it in. Don't try to be too sophisticated, estimates will never be anything other than approximate, however hard you try.

Estimating the Size of a Story

A simple, yet effective way to estimate the size of a story is to look for a similar story that you've already delivered. Then look at your records to see how long it took to build. Then assume the new story will take the same amount of effort. "Oh, another report. Reports always take us a week."

What if all the stories were the same size? Then estimating them would be easy. They are all, ehem, the same size. All that you would have to do is see how many stories the team completed each iteration and you would know how much you could get done.

We don't know how to make all the stories the same size. However, relative sizes work just as well as absolute sizes for planning purposes. Our approach is to look at the previous stories that got built. If you find one that looks to be equivalent to the one you are considering, give it the same estimate. "Oh, another report. Reports always take us a week."

If you can't find one the same size, look for something half as big or twice as big. Multiply or divide as appropriate. Do not worry that Martin probably failed to get an A in his Further Maths A level because of his unfortunate habit to multiply by two when he should have divided. You're a lot cleverer than he is.

It doesn't actually matter what units you express the estimate in. The only important thing is that you use a consistent unit between what you did and what you estimate for the future. In this book we use ideal weeks. Ideal weeks are the number of weeks that the story would take to one pair to implement, if the programmers could dedicate 100% of their time to it.

(You'll notice a slight flaw in this approach. How do you estimate when you haven't built anything yet and you don't have anything to compare it to? See Chapter 22 for a thoroughly unsatisfactory solution to this problem.)

Estimation is a team effort. The team discusses the story, considers how long it may take to implement, and decides upon an estimate. It may be that the team members will disagree regarding the estimate. Some may think the story is hard and will take a long time to develop. Others may think it is easy, and will take a short time to develop. We

follow the rule: “Optimism wins”. That is, if after reasonable discussion the disagreement persists, we choose the shortest estimate.

Remember, estimates are not commitments. And a couple of bad estimates are not disasters. What we are aiming for is to continuously improve our ability to make estimates. By choosing the most optimistic estimate we accomplish two things. We keep tension in the estimates so that they don’t grow hideously long; and we keep tension in the team so that the team learns not to be too optimistic. Team members, whose optimism burned the team once, will learn to temper that optimism.

Another issue that worries many people is dependencies between the stories. As we say in Chapter 20, you can mostly ignore dependencies. However mostly doesn’t mean always. You will get some cases where you say “flooping the thingummy will take six weeks, but if we do it after we burble the foobar it’ll only take four weeks”. In this case use the appropriate number for flooping depending on its position and make a note of the assumption. You will only have a few of these.

Periodically you will re-estimate every story, which gives you a chance to incorporate additional information like dependencies that have been erased or technologies that turn out to be difficult (or easy, we suppose).

Estimating How Much You can do in an Iteration

You can think of each iteration as a box, each of which can hold a limited number of wine bottles. The key question to any oenophile is how many bottles can you fit in a box? You could measure the box, measure the bottles and do some geometric analysis, you could form a committee of august persons, or you could just try it and see.

We like the latter approach. At the end of each iteration we measure how much stuff got done. We assume that, on average, we’ll get the same amount of stuff done next time. Life being what it is the amount of stuff will change from iteration to iteration, but we should see a relatively steady number over time.

Counting stories is not quite so easy as counting wine bottles, as not all stories are of the same size. So we have to measure the size of each story. At the end of each iteration we look at all the stories that got done, and we record how many weeks of ideal time it took to do each story. We then add up all the ideal time in all the stories, and that tells

us how much ideal time there is in each iteration, we call this figure the velocity of the team.

Following this simple rule violates one of the dictums of project management, "Work expands to fill the available space." The project manager on one of Kent's projects recently allowed as to how this just wasn't true, since the team had delivered 37 days worth of stories after having initially committed to 23, and done it with two people sick the entire time.

Be very wary about adjusting velocity to cope with changes in a team's size. As we discussed in Chapter 12 changing the composition of a team has both a non-linear and delayed effect on velocity. So we prefer to not predict the effect, but instead record velocities and see how they change. This is particularly true as you add people, since you rarely know how long it will take for the addition to take hold.

We also use velocity for individual developers. A programmer with a velocity of 10 ideal days can sign up for 10 ideal days of work in each iteration. Most developers will have the same velocity. However if someone is working part time, or is new to the team they will have a lower velocity.

You have to be careful not to attach too much meaning to velocity. Say you have two teams of the same size with the same iteration lengths but different velocities. What does this mean?

The answer is all sorts of things tangled up together. It might mean one team is more talented, or that one team had better tools. But it might also mean that one team tended to use more optimistic estimates than the other and needed a smaller velocity to compensate. In the end all of this stuff about ideal time is one almighty fudge to compensate for the difficulty of estimating software development.

The Meaning of Ideal Time

There's been a fair bit of discussion in the XP community about how what units of effort we should use.

In many ways the simplest unit would be calendar effort, which is based on calendar time.

Calendar time is the familiar passage of time, modified to handle working days. So if you are working Mondays to Fridays then four calendar weeks is equal to 20 calendar days.

Calendar effort is amount of people times calendar time. So a team of six people has 30 calendar development days of effort available per calendar week. In four weeks they would have 24 calendar development weeks of effort available. If one person on the team worked half time, they would have 22 calendar development weeks of effort available in that same four week period.

Most people measure all tasks in terms of calendar effort. This makes sense because it's easy to measure. However it makes estimating harder. The key to estimating is to consider people working at a reasonable sense of efficiency. Typically this means they don't get interrupted and distracted by the typical distractions that affect everyone all the time. These even out over the long haul but can have a big effect in short periods of a week or two. As such they really make a mess of historical data that is the backbone of a good estimation system.

So in XP we come up with a second kind of time: ideal time. Ideal time is time without interruption, where you can concentrate on your work and you feel fully productive. We measure and estimate using ideal time, because that allows us to compare tasks without worrying about interruptions. If we look at a task and see it is about as complicated as one that took 2 ideal days last week, we can estimate it will take two ideal days this week. The elapsed time could well be very different, but that is something we monitor separately.

We use the term ideal time, but really it's ideal effort. A team of six people might have 10 ideal development days of effort available a week. Typically when people talk of task lengths they say "that'll take 3 ideal days". What they really mean is that it will take "three ideal development days", but that's too much of a mouthful.

The notion of ideal time has really little to do with time. Indeed some people like to use something like story points, task points or gummy bears to measure the effort for stories and tasks. This works since the only important thing is that you have the same unit for the actual stories you measured in the past as you use for your estimates.

We like ideal weeks because it has some correspondence to the familiar, yet the word ideal is there to remind us that things aren't perfect. It also can help early on when doing your first plan (Chapter 22).

(If you've read some older stuff on XP you'll come across the term load factor. Load factor is the ratio of the calendar effort in an iteration and the velocity. So a team with 5 people using two week iterations has 10 elapsed programming weeks per iteration. If it's velocity is 4 then it has a load factor of 2.5 ($10 / 4$). We used to use load factor a lot in planning, but since learned that it's easier to just use velocity, so now we don't use load factor any more.)

Improving your Estimates

The good news about this estimation process is that tends to improve. As you do it more often, as you build up more actual figures, everyone in the team gets better at estimation. Encourage this process. Don't expect too much of early estimates, but do expect them to improve.

For this to work you need to track your actuals. Make sure you record how long it took to do a story. The record shouldn't be any more precise than your planning precision, so to the nearest ideal week will be fine. Building up a good set of actuals is essential to improving the planning process.

Chapter 20

Ordering the Stories

The most important stories to do first are the ones that contain the highest business value. It is also valuable to do high risk stories early on. Beware of sequential dependencies, most of the time they are false.

One of the key parts of planning is deciding what order you should do things. There is never any shortage of things to do. What should we do today?

Sometimes software is just too complicated, so instead let's consider something that is both more mundane and more essential. In cooking you answer this question by looking at the timing and sequencing in the recipe. Often there is quite a strong ordering involved: you have to fry the onions golden before you put in the chicken.

This notion of ordering is the dominant aspect of most kinds of planning. On big engineering projects this is what leads to the practice of dependency analysis and Pert charts. Dependencies between tasks dominate planning. Your mission, should you choose to accept it, is to find the best way of navigating through the dependencies.

To plan in this style you need to layout a list of tasks and capture the dependencies between the activities. By looking at these and looking at the duration of the activities you can figure out the key activities on the critical path where a delay will effect the whole project. If you do a course on project management you'll usually spend a good bit of time on this kind of analysis, together with such topics as resource leveling. You may well use a computer tool to help deal with all of this.

All of these techniques are pretty much useless on an XP project. They are useless because dependencies between tasks do not figure very

much in the picture. Most of the time there is no necessary ordering between the stories. You can build them in any order you like. We don't have a proof for this, but like most software pundits that never stopped us from a brave assertion. Our evidence is simply that nearly every time someone comes up with an ordering constraint, we can shoot it down.

A good example is a project Martin ran into for doing costing analysis for a factory system. The cost models were complicated and very new, so were not well understood. They ran on a large amount of data. The project team reasoned that since the models needed the data, the software to enter the data needed to be written first. This meant building a lot of GUI screens to enter and validate the data.

That was a false dependency. The data could be put into the database using raw SQL. This would allow them to work with the costing models and figure out what the models were really about. Until they understood the models better, they were hardly able to know what data they needed. Running the models was the point of the application after all, the data was only there to serve them.

We believe that most ordering dependencies are false. Of course most aren't all, but the cases where you do get dependencies they are easy to deal with on a case by case basis, either inside of an iteration where you have control, or by modifying estimates midstream. They don't affect the overall plan much.

A logical, yet naïve, reader would see this lack of constraints as a benefit. But often it isn't. Instead we see people who are looking for dependencies, and are so determined to find them they find dependencies that don't exist. After all if there is no ordering constraint, how do we choose what order to do things?

Business Value

So we need different factors for our choice. The first of these is business value. Do those stories that yield the most benefit to the customers. This seems so obvious that we hardly feel the need to point out why. But just in case, we'll do it anyway.

We want to get a release to the customer as soon as possible. We want this release to be as valuable to the customer as possible. That way the customer will like us and keep feeding us cookies. So we give them the things they want most. That way we can release quickly and the cus-

tomer feels the benefit. Should everything go to pot at the end of the schedule, it's okay, because the stories at risk are less important than the stories we have already completed.

Even if we can't release too quickly the customer will be happier if we do the most valuable things first. It shows we are listening really trying to solve their problems. It also may prompt to the customer to go for an earlier release once they see the value of what appears. Early releases scare people who do prototyping, because prototypes aren't production quality. But XP never builds anything it can't be proud of, so we can deliver at any moment.

How do we measure the business value of a story? The short answer is that we don't, that is the developers don't. Business value is a decision entirely within the realm of the business people. It's their business, so they know what the value is. If they're wrong, well they're paying for it. If you have enough dollars you have the right to wrong. The developers can suggest alternatives and ask questions. But in the end business value is always a business call.

How carefully should you figure out their values? You don't need to get it to the dollar, indeed it usually isn't worth trying to put a dollar value on it at all. The relative values are what counts, and you only need a rough sense of the relativity.

This is enough to start with. As you play out the release plan, you'll allocate the stories into iterations, and that will lead to a finer determination of business value. But in order to do that the business needs some more information.

Imagine you're standing in front of the new car that you've been salivating over for years. Finally the old heap has thrown a rod and you can reward yourself with nearly a year of that new car smell. You're finalizing the deal and the salesman asks "would you like the optional sun roof with that sir?" What do you answer?

Unless you're either incredibly cheap, or incredibly rich (or incredibly against fresh air) you'll want one piece of information. How much is that window in the Rover? If the cost is \$10 you'd probably take it. If it's \$10,000 you'll probably pass. But if the range is that big, you can't decide without the cost.

For most software, the range is that big.

So who gets to allocate the cost? Development states the cost, and they state it by estimating how much effort it will take, see Chapter 19.

Technical Risk

As development looks at the stories they will inevitably start thinking about how they will build them. As they do this they will run a gamut of feelings; all the way from “piece of piss” to “can’t be done”. These feelings are important, because they are the manifestation of where the project could go off the rails. When developers feel nervous, everyone should listen.

The nervousness comes from a number of sources.

- ◇ Development may not be confident that they can estimate the story to the required precision. This could take a programmer one week, or it could take twenty. That’s too big a range. It’s not enough to split the difference and say ten. That may be a start, the best you can do, but you need to know more. The risk here is that the estimate could be off, and off badly.
- ◇ Development fears that working on this story late could cause a lot of rework on the existing code base. Of course we XPers like to believe the cost of change curve is flat. But while we are extreme, we’re not stupid. Most of the time we can get away with that assumption, but sometimes we can’t. Those cases when we can’t are risky, and need to be dealt with earlier.

A story may depend on some third-party piece of software. Software developers are very distrustful of marketing blurb. The software may say it can do something, but can it really? Until we know for sure, there is risk.

Worst Things First

Why do the high risk items first? High risk usually means hardest. Why not work on easier stuff early on and work your way up to the hard stuff?

The primary reason is that risk implies uncertainty in the schedule, that is the cost, of the story. If some features are going to cost a lot more than development thinks, then business needs to know as soon as possible. These surprises are going to happen, because software devel-

opment is inherently a surprising activity. The important thing is to discover them while you still have time to do something about it. That's why so much of XP's planning is about uncovering risk early on.

Also tackling hard things early does wonders to reduce fear. If you're working on something easy, and know there's something hard later, there is always nagging doubt. The doubt saps confidence, it leads people to over-design in order to make things easier later, it gnaws at the relationship between development and customers, and the relationships within the team. So it's better to know the truth early. Most of the time it's easier than people think, especially when simplicity is the driving force. When it isn't at least you know how bad it is, the team knows how bad it is, and business can be told how bad it is. Once you know the score, you can figure out what to do about it.

One exception to this rule is early in the process with a newly formed or untrained team. If you have a team that hasn't worked together before, the biggest risk is that they won't find it easy to work together. So working on something easier in the first iteration makes sense. But as soon as they start working together they need to handle the hard stuff. That way they can get used to working on hard stuff before the pressure builds. Similarly if you're working with a new language or platform, your biggest risk is that you don't know how to work in that platform. So tackling something easier can concentrate on acquiring that learning, before hitting the big stuff. However if you're in a team that's worked together before in an environment that you know, then go full tilt at the biggest, meanest monster you can see — however many teeth it has. Just push those teeth right down that throat. Or run away and choose another path. Either way figure out which you need to do as quickly as you can.

Performance

Performance is one of the seemingly technical factors that "ought" to affect the schedule. However, the trade-off between investing in performance and investing in more functionality is a business decision. A blazingly fast system that doesn't compete on features is just as dead in the marketplace as a feature-rich system that doesn't perform.

The programmer's job on performance is to make sure the trade-off is clear. "You can have that feature in two weeks, but it will only work

for up to 100 hits/hour. If you want 10000 hits/hour it will take us four more weeks." If the customer wants the slower version first, write another story for the optimization.

The same is true for other non-functional requirements, like reliability. Communicate the trade-off between various levels of service and speed of implementation. Let the customer decide.

This frightens some technical folks. What if the customer drives the team down a blind alley? That can happen. But grabbing the wheel from the customer and making business decisions for technical reasons won't solve the problem. The best you can do is make the trade-offs clear, and do your best to keep your options open by having extensive tests, keeping the system clean and orderly, and staying rested.

Negotiating between the two

You notice we have two different criteria set by two different groups of people. How do we resolve between the two of them? We want to tackle the high-risk and high-value stories first. However there are plenty of times when the conflicting priorities have to be dealt with.

Ideally it is business who should make all these choices. Business people understand uncertainty in estimates, they face this all the time with financial projections. They also understand the cost of rework in the cases where rework is an issue. If they wish to run the risk in order to get better value now, then that is really their call. It is the developers' task to make the risk apparent, not to make the decision for the customer. However while this is the ideal, sometimes you need something extra to make it work, perhaps allowing developers to choose a certain amount of stories per iteration so that they can bring the risk forwards.

We need to stress that you shouldn't worry too much about the issue of rework. If you spend a lot of time trying to figure out the consequences, or trying to figure out the best order to do them in, then you're wasting time. It would be useful if you could figure all this out correctly, but our experience is that it's nearly impossible to figure it out correctly.

Example Release Plan

Let's take a look at the release plan from the time capsule. The information was annotated on the cards, but we've transposed it all into a handy table for you.

Story	Time Estimate	Assigned Iteration	Assigned Release
Find Lowest Fare	3	2	1
Show available flights	2	1	1
Sort available flights by convenience	4		2
Purchase ticket	2	1	1
Customer profile	4		
Simple Customer Profile	2	1	1
Full Customer Profile	3		
Review itineraries	1	2	1
Cancel itinerary	2		2
Print immigration paperwork	4		
Show Hotels	3		2
Show hotel availability	2 ^a		2
Sophisticated Hotel Search	3		
Book a Hotel	1	2	1

TABLE 1. Example Release Plan

Story	Time Estimate	Assigned Iteration	Assigned Release
Hotel / Spaceline programs	3		
Airplane hire	3		

TABLE 1. Example Release Plan

a. Is 4 if Show hotels isn’t already there

The notes indicate a velocity of 6 and as you can see stories have been assigned to iterations. There’s a release after the first two iterations and a further release two iterations later. (The releases are a little short, but this is enough for an illustrative example. And in any case there’s this confidentiality agreement which stops us showing all the plan two centuries before we begin.)

There’s a few things worth noting:

- ✧ The Customer profile story has been split up into two parts with different priorities. The estimates don’t add up in this case. Often they do, but from time to time they don’t. That’s not an error, each story should be estimated independently, so it does happen. Maybe breaking it apart made us realize it was bigger than we thought, or maybe doing them separately would take longer. The reason usually isn’t important, so there’s no need to document it.
- ✧ The second iteration only has five ideal weeks of stories assigned to it, even though the velocity is six. That’s okay — it’s important for the capacity not to exceed the velocity, but the reverse isn’t true. In this case additional work may well get added closer to the iteration’s date, probably by splitting one of the later stories.
- ✧ A note on the card for Show hotel availability indicates that the estimate depends on other work. That’s not uncommon, but it isn’t worth making a fuss about. In this case it was probably only recorded as the customer was figuring out what to put in iteration 2.
- ✧ It may look odd that Book a Hotel is assigned to iteration 2 when Show Hotels isn’t assigned yet. How can you book a hotel if you can’t see the list? The point here is that the functionality won’t be available to the user yet, but will be built into the system and can

be functionally tested to satisfaction of the customer. Not all behavior has to be usable by the user to be included in the iteration, the key criteria is that it is testable. You'll often find this in cases where you need to build functionality in the core business logic but don't add the UI until a later iteration.

- ◇ You'll notice there are no details about the iteration length or size of team. This is deliberate to point out that you actually don't need that information for the basic release plan. All you need is the estimated stories and the velocity. Obviously you'll need the current date and the iteration length to tell people when an iteration will be done. Something along these lines:

TABLE 2. Iteration completion dates

Event	Date
Start	1 Apr. 29
Iteration 1 Complete	19 Apr. 29
Iteration 2 Complete	10 May 29

- ◇ It's comforting to know that even in the 24th century they use two digits for the year.

Chapter 21

Release Planning Events

Various events cause the team to do a little release planning. The customer adds and changes priorities of stories, developers estimate stories, and team notices if it has too much, or too little to do.

As you are doing release planning, you'll find there are several things you need to do. In this chapter we talk about the various events in release planning, and how you react to them.

Measuring Velocity

You can think of each iteration as a box, each of which can hold a limited number of wine bottles. The key question to any oenophile is how many bottles can you fit in a box? You could measure the box, measure the bottles and do some geometric analysis, you could form a committee of august persons, or you could just try it and see.

We like the latter approach. At the end of each iteration we measure how much stuff got done. We assume that, on average, we'll get the same amount of stuff done next time. Life being what it is the amount of stuff will change from iteration to iteration, but we should see a relatively steady number over time.

Counting stories is not quite so easy as counting wine bottles, as not all stories are of the same size. So we have to measure the size of each story. At the end of each iteration we look at all the stories that got done, and we record how many weeks of ideal time it took to do each story. We then add up all the ideal time in all the stories, and that tells us how much ideal time there is in each iteration.

Following this simple rule violates one of the dictums of project management, "Work expands to fill the available space." The project manager on one of Kent's projects recently allowed as to how this just wasn't true, since the team had delivered 37 days worth of stories after having initially committed to 23, and done it with two people sick the entire time.

Changing the Priorities of Stories

Businesses often change their priorities. By only committing to one iteration's worth of stories at a time the customer is guaranteed a chance to make priority decisions based on up-to-the-minute information about the priority and cost of the stories.

Adding a story

If you compare XP to many of the other methodologies out there, the biggest difference to customers is the fact that they don't have to commit to a detailed specification of everything they want before development begins. In other words we consider requirements creep to be perfectly reasonable and rational behavior. The requirements can creep wherever they like as long as we know where they are going and the customer is informed of the consequences. Indeed we encourage the requirements to crawl around and find whatever it is the customer needs.

The customer can add a new story at any time. They write the story as we describe in Chapter 18. Development then estimates the story in ideal effort. Now the customer can pick the story for any iteration, subject to Yesterday's Weather. Unless the plan has a lot of slack there will need to be a fair bit of moving stories. That's fine too. The customer moves stories as much as they like to make room for the new story and then plops it in.

Rebuild the Release Plan

Most of the things we talk about are small scale changes to a plan. You have to defer a few stories, move some stories about. Little changes like this keep the plan in good shape, but sometimes you need something a good bit more significant.

You need to rebuild the plan in the following circumstances

- ◇ When the pile of deferred stories has got too far. A few stories deferred till later is not worth rebuilding for, but if enough of them mount up that you are sure you aren't going to get everything done, it's time to do something about it
- ◇ If the team's velocity changes, then all the assumptions for the release plan are broken and you need a new release plan
- ◇ If the team is doing stories at a faster rate than planned and you find stories denuded from later iterations.

A rebuild begins with development re-estimating the stories. The original estimates were a good guess at the time, but since then everyone will have learned, and a rebuild is a good time to take this learning into account. During the re-estimate development can use the actual figures from recent stories to better calibrate their estimates. This re-estimating is particularly important early in development as the early release plans will be the least accurate as there is little history to go on.

Once development has re-estimated the stories the customer picks a pile of stories that will fit. Just hold your nose and do the math. If you are running at around 8 weeks worth of stories per iteration, and you have 3 iterations to go, then the customer picks 24 weeks worth of stories for this release, and will get the remainder of the stories as soon thereafter as possible.

You should expect to rebuild the plan every three or four iterations. You don't want to do it all the time, as it takes a fair amount of time, but it does need doing regularly. Think of it as changing the oil. You don't *have* to do it, but you'd be stupid not to.

Chapter 22

The First Plan

The first plan is the hardest and least accurate part of release planning. Fortunately you only have to do it once.

Making the First Plan

So now you know what a release plan looks like, and how you use it. It should be clear that the accuracy of plan depends on the accuracy of its estimates, and the accuracy of the estimates depends on the history of the project. So with the first plan you have no history, so your estimates are flaky. What does this make the plans?

Flaky. (Think Kramer with too much coffee.)

But you still need a plan to even get started. The plan gives you something to track against so you can assess progress. It gives you a framework of stories which allow you to capture what happens and build up the history to make the later plans better. The first plan is always high on expectations and usually a big let down.

The plan has two main areas of uncertainty: the velocity of the team and size of the stories. Here's a few tips on how to do the best you can.

The key to velocity is that it's a measured thing. As such you won't get a decent picture of velocity until several iterations into the project. Before you can measure, the best you can do is to guess.

One way to gauge the velocity is to measure progress during the exploration phase. During the exploration phase use tasks much along the lines of an iteration plan. Monitor progress and track the ideal time people report much as you do with iteration planning. See how much ideal time people get done and use that as a basis. Every time a programmer says, "I'll get that done in a day," notice how long it actually

takes. If "a day's worth of work" generally takes three days on the calendar, then you could guess the team's velocity as $\#people * weeks / iteration / 3$.

If there's a similar project in your organization you could use their velocity to help estimate yours. They have six people on three week iterations. That's 18 calendar weeks per iteration. Their velocity is 6. You have four people and are doing 2 week iterations. That's 8 calendar programming weeks per iteration. So a guess at your velocity is $6 * 8 / 18$ which is about 2.

Story estimates are tough at first. The approach we've had the best luck with is to get the team to start with the stories they feel the most comfortable estimating. Once they have done those do the others by comparison, much as you do with historical estimates. To estimate those first few stories, have people ask themselves, "If this story was all I had to do, and I had no distractions, how long do I think it would take?"

You may have another team that's done similar work. How can you use that experience? One thing to *not* do is to get them to estimate the stories. That breaks the rule that those who do should be those who estimate and will lead to lack of commitment and falling morale. Don't do that.

You can have the team look at the history of the other team and come up with their own gauge based on that. But you have to be careful not to let comparisons get too sticky. Once you start reasoning about how fast one team is compared with another you'll get all sorts of emotional reactions that will distort everyone's thinking. It can be helpful in flushing out all the issues that may come up. It can also be handy in comparing effort between tasks: "it took them half as much again to get the objects hooked into the relational database, so let's start with the same proportions for us".

Choosing your Iteration Length

How long should an iteration be? There are a lot of people who favor iterative development, but they don't all answer that question the same way. Some people say that an iteration can be as long as six months. We go in the other direction, iterations should be short – between one and three weeks. We favor two, and a recent survey on the XP egroup

(<http://www.egroups.com/group/extremeprogramming>) showed that two and three were the most popular choices.

Question:

How long do your iterations last?

Choices	Votes	%	37 replies
<1 week	0	0.00%	
1 week	7	18.92%	
2 weeks	14	37.84%	
3 weeks	12	32.43%	
4 weeks	3	8.11%	
1 month	1	2.70%	
>1 month	0	0.00%	

Why make them so short? One of the key things we want from a plan is an indication of progress; how far have we come, how far have we to go? Iterative development is good at providing that, because each iteration delivers working, tested code – which is hard to fudge. But these milestones only occur at the end of an iteration. So the longer the iteration, the more risk you run of sliding just a little bit out of control. So you want each iteration to be as short as possible, so that you know where you are as frequently as possible.

But there is such a thing as too short. Each iteration implies some overhead. Making sure the acceptance tests are running, planning the iteration, reporting to management and so on. We don't want the overhead to be bigger than what we are trying to do in the iteration. If it is, we will stop planning and iterating. However, iteration overhead is kept small by doing it all the time.

But don't get too hung up about it. The world isn't going to end if you pick three week iterations instead of two week iterations. As long as you are prepared to be aware of the effects of iteration length, and prepared to experiment when you spot a problem, you can start most anywhere. So start with two week iterations.

Chapter 23

Iteration Planning

Each iteration is planned by breaking down the stories for that iteration into tasks.

The Iteration Plan is the second part of the planning picture. The release plan is synchronized to the rhythms of business. It gives the business people a way of thinking about sets of stories that together tell a good story to the market. The iteration plan is synchronized to the rhythms of programming. Two weeks is long enough to:

- ✧ Develop some new functionality
- ✧ Do a substantial refactoring
- ✧ Develop some infrastructure
- ✧ Try some experiments
- ✧ and, Recover from little disappointments

Unlike the release plan, the iteration plan is very much the developers preserve. They decide how to do things and in what order. The customer is still involved. It's important to report progress mid-iteration so that the customer can see what's happening and get a sense for what will actually happen. The customer will also get very involved if the team finds there is too much to do and you need to cut scope.

The formal start of the iteration is the iteration planning meeting (Chapter 23), where the developers get together to break down the stories for that iteration into smaller tasks. We do this because we need smaller units to track than the stories. Each task is about 1-3 ideal days in length. Also while it's often worthwhile to have a programmer responsible for seeing a story through to completion, it is better for people to sign up for the smaller tasks, to give them a chance to express their specializations. Often there is overlap in the work needed between

stories, and we can handle this better by using tasks. Finally, the kind of dependencies that drive most planners crazy must be dealt with inside of iterations.

After the iteration planning meeting someone takes on the responsibility of tracking the iteration(Chapter 25). The tracker keeps an eye on which tasks are done, and how much is left on outstanding tasks. His job is to alert the team to problems that might come up: having too much to do, to little to do, people over or under committed. Every day the team has a short stand-up meeting so everyone can see what everyone else is doing. This helps keep communication flowing across the whole team.

Never slip the Date

One of the most important principles in planning for Extreme Programming is that the dates are hard dates, but scope will vary. In any project you'll often run into the situation where there is too much to do and you'll be tempted to slip the date of an iteration just a little bit.

Don't do that.

Slipping dates is one of those bad habits that is both addictive and damaging. The damage really comes from the fact that it's so easy to do. At some point, however, you'll either slip so much that you'll lose all credibility (and the project too), or you'll run into some date that's very painful to slip, like a release date. If slipping dates is the only thing people know how to do then that's what will happen, and the pain is great.

The harder but surer way to cope with too much to do is to defer functionality. Because it's harder it's more important to practice doing it. Only if the team (and especially the customer) practices this for every iteration will everyone be ready to do this when the really important date comes along. The most important stories are the ones the customer decides they can live without for the moment. Those stories can save projects.

Now it may be better to slip a release date. That's the customer's decision that will involve many issues. But only if everyone has practiced deferring function will the customer have a genuine choice between date and function. And we are constantly surprised by how

much "essential" functionality customers can do without when a big date looms.

Often you can only make those hard choices when the big dates appear, and without practice you can't do it under pressure. As Jim Highsmith says "I used to think Time-boxing was about time, but I've learned that instead it is about forcing hard trade-off decisions throughout the project".

The other reason not to slip dates is that you lose control. There's always some uncertainty within the iteration. Only at the end of an iteration do you really know where you are. So you never want to put that time off.

Chapter 24

Iteration Planning Meeting

At the beginning of an iteration the team plans out its iteration with an iteration plan. This plan breaks down the iteration into development tasks of a few days each.

The iteration planning meeting is the first thing you do in the iteration. It shouldn't take any more than a day, and most teams will do it in under a day. All of the team should attend, plus the customer. Find a comfortable room with plenty of whiteboard space.

Here's an overview of the sequence, this is just a summary, we'll describe what it all means in a moment.

- ◇ Read out the stories for this iteration
- ◇ Write on the whiteboard all the tasks that need to be done for each story
- ◇ Add to the list any technical tasks that need doing
- ◇ Developers sign up and estimate tasks up to their individual velocity.
- ◇ If any story isn't completely signed up for, the customer is asked to defer functionality
- ◇ If there is extra time the customer is asked to add functionality

The first thing to do is to remind everyone what's on the menu for this iteration. It's often a good idea for the customer to read out the stories so that everyone hears it from the customer perspective.

Listing the tasks for an iteration

Each story is a few ideal weeks of effort, so for this detail of planning we need to break it down into a few finer grained tasks. Each task

should be a few ideal days of effort. The tasks are development tasks, so they don't need to make sense to the customer.

The best way to come up with the tasks is for everyone involved to just brainstorm what they might be. Often this will involve a little thought about how you're going to build it. You don't need a major design effort here, but you do need just enough to get a good list of tasks. Take short pass at the classes and interactions you'll need to add and modify. Remember that the goal is not to design everything that is needed for the iteration, it is to help figure what the best task breakdown is. You'll do other design sessions as part of carrying out the tasks.

Often you'll find some overlap between the stories. This overlap may give you the opportunity to build software that helps satisfy several stories at once. There may be existing designs from past iterations that, with a little refactoring, could significantly help this iteration. Add tasks for this, it's okay to have tasks that satisfy more than one story.

For example, several of the stories may require that as you enter text into a field, you hunt for possible completions for that text from a given list. It would be a shame to develop this facility three or four times, once for each story. So make this into a task. It may be that other stories from past iterations need to have this new feature retrofitted onto them. You might make that retrofit a separate task.

Dependencies between tasks are likely to exist. You won't need PERT charts or similar beasts. With a communicating team over a few weeks you can rely on the team to sort these dependencies out. It's worth talking over them, for that should influence what gets done first. However, as you'll see, you don't actually plan precisely when each task gets done, or in what order the task gets done.

In the end there's no hard and fast rules for how you break something down into tasks. Use whatever approach makes sense for you. As long as the tasks are kept short and you can write tests for them, you'll be fine.

Technical Tasks

Most of your time will be spent in building stories. However there are some purely technical tasks which the customer doesn't care about, but still need to be done. These include such things as installing the

vendor's database upgrade, refactoring some messy bit of the system, trying out a design idea to learn if it will be useful, improving the configuration control process. Anything like this should be added to the list. Again it's important to keep each task small, so break things up into a few ideal days each as much as you can.

Ron Jeffries claims he can turn any technical task into a business-oriented story the customer can either schedule this iteration or not. There's something to be said for not having any technical tasks. Once you start down the slippery slope of setting priority for technical reasons, it's hard to stop. Dave Cleal suggests that a fixed portion of the budget, say 10%, be set aside for the programmers to spend as they like. We haven't tried it, but it sounds like an interesting idea.

Measuring the velocity of a developer

Each programmer is going to be able to do a certain amount of tasks during the iteration. We figure this out by asking each programmer to track their progress iteration-to-iteration. Then, just like the team as a whole, a programmer can only sign up for the same number of days worth of tasks as they completed last iteration.

Obviously there are things that can change that. If someone is spending one week of this three week iteration on vacation, then they will only have 2 / 3 of their usual velocity available.

Remember that a programmer's velocity is not a measure of her productivity, speed, or goalkeeping ability. It just says how many tasks they can sign up for. One person may have a low velocity because they are naturally optimistic when estimating. Another may have a low velocity because they spend a lot of time helping other developers. The key thing is that a programmer's estimating ability should improve so they can predict how much they can get done. The absolute value of their numbers are far less important than their accuracy.

Signing up and estimating Tasks

Once you have a list of the tasks on the whiteboard, programmers can then start signing up for them.

By signing up, each programmer accepts the responsibility for working on that task during the iteration. The first thing they need to do is

to estimate how long it will take them to do the task. Most of the time they can come up with this estimate right away. However it's also alright for a programmer to say "I'll sign up for that, but I'm not sure how long it'll take yet. Let me work on it for a day or so and I'll give you an estimate then".

Any programmer can only sign up for enough tasks that fit into his velocity. Programmers may need to break up a task and then sign up for a piece of the result.

It's up to a programmer how they come up with an estimate. As usual the best approach is compare it with previous work. The programmer can say, "this should take about the same time as the shipping GUI I did last iteration". The project records will show how long that task took, so that number can be used this time too.

Be wary of using comparable work from another programmer. We are dealing with people, not Plug Compatible Programming Units. Programmers do not work at the same speed, and their comparative speeds will be different for different tasks. It's useful to let a programmer look at the records to help come up with an estimate, but don't do things like "It only took Dinsdale three days to do this task, so why do you think it'll take five?" Once you start using the records as a club, you'll never get an honest estimate again. The most important thing here is to get estimates as accurately as you can — if you want to compare programmers' abilities don't do it during planning.

Programmers should always assume they have a partner when doing a task, since in XP all production coding is done in pairs. It may be that a particular partner can affect the estimate. "If Spiny will work with on this, we can do it in two days". That's fine and indeed it reinforces the principle that people help each other.

Programmers should also assume that they aren't done with a task until they have all the unit tests written and passing. "I'm done with the coding, but I haven't tested it yet," is a meaningless statement in XP. Code without tests simply doesn't exist for planning purposes.

Programmers can sign up for whatever they want to do. People can work things they have a desire to do, which keeps them motivated. Since motivation is the key to productivity, this is a Good Thing. Some programmers will pick one kind of task they prefer and mostly do this. Others like to broaden their interests by trying new things. This is a

Good Thing because it keeps up motivation, it allows developers to grow their skills, and it reduces project risk.

Scut Work

Some project managers, on hearing this "signing up" practice, fear there'll be some dirty work that doesn't get done. In practice we don't find this is an issue. Developers are so varied that there is almost always someone who likes doing everything. For occasional unpopular tasks some informal and often unspoken sharing goes on.

If someone always ends up on with the short end of the stick, then it's the coach's job to point this out to the team, and ask the team to come up with solution. They may choose to do a more formal rotation to take turns doing the unpopular kind of work. They may try to recruit someone to the team who likes doing this kind of work. Programmers like solving problems, so they'll find a solution, and if they come up with it they will make it work.

Too much to do

Once all the programmers have signed up for tasks and estimated them, take a look at your task list. In a perfect world all the tasks will be signed up for.

In this particular galaxy, we find often tasks are still remaining. If that's the case you have to decide what to defer.

If you have technical tasks remaining, or you can swap story based tasks for technical tasks, then that is one option. You have to consider the risks involved. Can you wait another iteration to upgrade the database? Sometimes a wait is the best option, but other times there aren't any technical tasks you think you can defer.

At this point it's time to call in the customer. The developers cannot which story based tasks to defer. Instead they have to say to the customer "we are short 5 ideal days, we need you to remove 5 ideal days worth of story".

At this point the customer has to choose. The customer can choose to defer a whole story, or they can chose to split a story and defer one of the resulting parts (see "Splitting user stories" on page 92). A customer may choose to split several stories to get the right mix for them.

A deferred story goes on the list for the next iteration and you will consider it again at that time unless the release plan has been adjusted in the meantime.

You may be tempted to adjust the release plan if you defer a story. You don't have to rush to that. Often you can go for an iteration or two before the snowplow effect becomes so great that you need to rework the release plan. Often the deferments are too small to be an issue right away and they need to mount up before it's worth trying to deal with them.

Too Little To Do

Go ahead, laugh, but it does happen from time to time.

This process is the reverse of too much to do. The developers ask the customer to add some work. The customer can add a whole story, or split a story and move a part in. However, as usual, the customer decides what gets moved, not the developers.

Example Iteration Plan

We'll dig some more stuff out of the time capsule to show the iteration plan for the second iteration. The notes look like this

Lowest Fare

- Alternative fare finder object - KB 2
- Find candidate fares by date range - MF 1
- Update planet ports to find alternatives - KB 1
- Find candidate fares for alternative ports - KB 1
- Special offers - major space lines - MF 2
- Special offers - low price space lines - RJ 3
- UI for low fares - RJ 1

Review Itinerary

- Simple UI display of itineraries - WC 2
- Display detail for one itinerary - RJ 2

Book Hotel

- Hotel Booking interfacers - MF 1
- Interface to IHAB - MF 2
- Interface to HiHat - MF 1
- Interface to Mary's Rote - MF 1
- Interface to HillTown - WC 1
- Interface to Best Southern - RJ 1
- Interface to Woodstar - WC 1
- (Show Hotels - IHAB by city)
- Query IHAB for hotels for named city - WC 2
- UI for named city display - WC 1

Other

- UI Clean up - KA 2
- Network Performance Improvement - KB 2
- Investigate using IPv84 - KB 1

Here's some points to note.

- ◇ Notice that the tasks in the stories don't necessarily add up to the estimates in the release plan. Book Hotel was marked as 1 ideal

week but comes out as 8 ideal days. Lowest Fare was 3 ideal weeks but came out as 11 ideal days. This is quite usual as you start estimating in more detail and have more information about the iteration. In this case they cancel out pretty well, but of course they often don't.

- ✧ Since there is time left over the team took a part of the Show Hotels story and built a little bit of that. Here the customer split the Show Hotels story to get a piece that was useful yet enough for the team to use up the extra time they had available.
- ✧ Most of the team have a velocity of 7, however KA has a velocity of 2. This maybe because he's part time, or new to the team, or just estimates differently.
- ✧ There are plenty of dependencies in the tasks, but the plan doesn't note any of them. From watching the videos we could tell that the alternative fare finder object was needed before you could find candidate fares by date range or find candidate fares for alternative ports. Also you need to do the update of planet ports to show alternatives before you can find candidates for alternative ports. None of this is on the plan, since this is an XP team we can safely assume that MF and KB are intelligent beings who will communicate about the order things need to be done and will sort things out between them.

Chapter 25

Tracking an Iteration

A couple of times a week, the tracker does a progress check on the iteration to see how things are going.

You have an iteration plan, so now all you have to do is sit back and watch the plan unfold — right? If you agree hit yourself on the head three times with this book (and be glad it's a thin one).

You only thing you know about a plan is that things won't go according to it. So you have to monitor the iteration at regular intervals.

Iteration Progress Check

A couple of times a week or so the tracker needs to find out where everyone is with the iteration. We don't suggest calling a meeting for this, or (heaven forbid) a written report. Instead the tracker should visit each programmer one at a time.

The tracker asks each developer two questions about each task they signed up for

- ◇ How many ideal days have you had to work on this?
- ◇ How many more ideal days do you need before you're done?

Notice we don't ask for percentage complete. We've found that such a question generates an often-meaningless answer. To monitor how much work there is left, the key question is how many days are left. (Knowing how much has been done so far is more important for building the historic record.)

At this point you can add up the figures for all the developer's tasks and assess the situation.

Most important is what is left to do. How does the amount of ideal days of effort compare with the developer's remaining calendar time? This comparison is always somewhat rough. There's no point using a ratio of the remaining calendar time as the developer won't get to work ideal days evenly through the iteration. However if there is a big difference then that's a warning sign. It's also definitely a warning sign if there are more ideal days of things to do then calendar days left to do them.

To a large extent the analysis is up to the programmer — does she think she has too much to do? The tracker's role is really to ask simple questions and point out possible problems. In the end it is the programmer's judgement whether she will be able to get in enough ideal days before the end of the iteration.

At this stage you don't do much with the "how many days so far" question. Primarily that's to help you get a list of tasks done with how many ideal days they actually took. Asking each time is better than asking at the end of the iteration when nobody can remember how much time they spent on anything last week.

One thing to watch is if a programmer reports a lot less ideal time spent on tasks than usual. This might mean nothing — she was just doing a lot of helping this week; or it may mean there's something else that's chewing up her time. If it's something else it may be worth looking into.

It's usually best to take the old figures to the programmer. That way you are asking only for the change since the last progress check.

The team at Retail Aspect discovered a simple way to get an overall progress report. Post the iteration's tasks with check boxes beside them.

Picture of task sheet

A refinement of this practice is that the customers are the only ones who can check off the boxes. This encourages them to do what they can to bring the tasks to completion.

When a Programmer finds they aren't going to make it

Tracking is passive, until you find that something is up. Up, in this case, means the programmer realizes that he can't complete all the tasks

he signed up for. There are plenty of reasons why this can happen, the more important point is what to do about it.

The first reaction is to look for someone else with time available who is willing to accept the task. They should then give their own estimate of how much time is needed to finish the task. If their answer ends up greater than the time available, then that may mean that the problem hasn't been solved yet.

If nobody has enough time to take on the task, then the team needs to figure out what to do about it. It's important to let everyone know about the problem, as someone may have a good solution to hand. This is where the stand-up meeting comes in handy. Often a bit of informal juggling of a couple of people's time can help overcome the problem.

If there aren't any great solutions available, then you have to go to the customer and let them know the score. The customer will need to know what work has been completed on what stories. They then go through much the same process as they did at the beginning of the iteration: they choose a story to defer, or a story to split and defer a part. The sooner you bring the situation to the customer's attention, the more options they have for salvaging the greatest possible business value out of the iteration.

It's annoying for the customer to do this in the middle of the iteration, so the team should work to avoid this happening. The key cure to this problem is to get more accurate at estimating, and this can only come with practice and good task records. Over time this should happen less often, if it isn't occurring less often the coach needs to figure out what is going wrong. While the story estimates are pretty vague, once you get down to tasks in the iteration plan, you shouldn't get such big swings.

What about overtime? The simple rule is that no one can work overtime two weeks in a row. If someone wants to crank for a day or two to get caught up, fine. But then don't expect full results from them for a day or two.

When a Programmer has Extra Time

Inevitably this is less frequent but also the more pleasant problem. First choice, of course, is to look at other people's task load and to see if

the programmer with time can help others. Often you'll have mix of too little and too much that will balance out pretty well.

If not then it's off to the customer again, to see what can be brought forward, either a full story or part of a story.

However if that person has had a heavy load in recent iterations, consider letting them take a break. This may mean a day off if they've had to use overtime recently. Otherwise it may mean sometime experimenting with something. Spend a day looking at some bit of technology that might be useful. Such breaks help keep people motivated and may lead to useful ideas that'll help the project.

Finding you have Too Much to Do

At some point the team realizes that you're not going to complete the stories they intended to complete in an iteration (Chapter 25 talks about how you monitor an iteration.)

Development should estimate how much more ideal effort is required to complete all the remaining stories in the iteration. Notice we "how much effort to complete" not "what percentage are you done". The two questions sound similar, but the former gets much more useful answers because it focuses on the question at hand.

Development also calculates how much ideal effort there is left in the iteration. A simple way to do this is to use the remaining calendar time as a proportion of the velocity. So if you have a velocity of 6 and are two weeks into a three week iteration, you have 2 ideal weeks left.

The customer now has to decide what won't get done in the iteration. They can take a whole story and remove it. This removes as much time as there is remaining on that story. Alternatively the customer can take a story and split it, saying, "I want this part of the story in this iteration, but I'll defer the rest to the next iteration" If the customer splits a story, development estimates the two parts separately, again concentrating on how much there is left to complete each part. The customer can then defer one part and keep the other.

We've heard from project managers that, "XP is an excuse for my programmers to tell me I can't have what I want." Customers aren't ever happy about having to cut already-committed scope. But the simple fact is that it is inevitable that occasionally the customer can't have

what they want. The only question is who is going to decide what they don't get- the customer, the programmer, or chance. We think the customer should decide what they aren't going to get.

Example Iteration Tracking

Back into the time capsule to look at some iteration tracking done two-thirds of the way through the second iteration. You can show the state of the iteration with a simple table.

TABLE 3.

Task	Who	Done	To Do
UI Clean up	KA	2	0
KA Total (out till end)		2	0
Alternative fare finder object	KB	2	0
Update planet ports to find alternatives	KB	1	1
Find candidate fares for alternative ports	KB	0	1
Network Performance Improvement	KB	2	2
Investigate using IPv84	KB	0	1
KB Total		5	5
Special offers - major space lines	MF	0	2
Find candidate fares by date range	MF	0	1
Hotel Booking interfacier	MF	1	0
Interface to IHAB	MF	3	2
Interface to HiHat	MF	1	0
Interface to Mary's Rote	MF	0	1
MF Total		4	6
Special offers - low price space lines	RJ	2	2

TABLE 3.

Task	Who	Done	To Do
UI for low fares	RJ	0	1
Display detail for one itinerary	RJ	2	0
Interface to Best Southern	RJ	0	1
RJ Total		4	3
Simple UI display of itineraries	WC	1	0
Interface to HillTown	WC	0.5	0
Interface to Woodstar	WC	0.5	0
Query IHAB for hotels for named city	WC	2	0
UI for named city display	WC	0.5	0
WC Total		4.5	0
Team Total		19	14

The table gives us a good state of the iteration. The bottom line warns us of potential problems. KA is out till the end of the iteration, so we have 4 people. We know they each can do 7 ideal days in a three week iteration. So the four people can do 28 ideal days in the whole iteration. So in the last week they can do roughly a third of that: 9 days. However we have 14 days of work available, so this indicates a problem.

Looking at the tasks and individual states we can see some more details. RJ is pretty much on track, with 3 ideal days to go. Talking to him brings out that he is comfortable that he will complete on time.

WC is very comfortable, his tasks worked out easier than he thought so he free for the last week.

However both KB and MF are in trouble with a quite a bit of outstanding time. We can do a bit of shuffling here by bringing in WC, but it won't be enough. So we need to talk to the customer to decide what

needs to be done. The customer can look at the tasks, but primarily looks at the state of the stories.

TABLE 4.

Story	To Do	Comments
Lowest Fare	7	Several pieces incomplete
Review Itinerary	0	
Book Hotel	4	IHAB interface a problem, can pick and choose with others
Show Hotels (IHAB by City)	0	
Other	3	Mainly performance work

The customer knows we have around 9 ideal days available, what should be cut? They answer that the lowest fare is the key story here, that needs to be done at all costs. The book hotel is less important and the performance work is not that critical to the customer, otherwise it would be a story. So this allows the team to look at the tasks afresh and resign up for the ones that need work. So here's the outstanding tasks.

Outstanding Task	Who	To Do
Update planet ports to find alternatives	KB	1
Find candidate fares for alternative ports	KB	1
Network Performance Improvement	KB	2
Investigate using IPv84	KB	1
KB Total		2

Outstanding Task	Who	To Do
Find candidate fares by date range	MF	1
Interface to IHAB	MF	2
Interface to Mary's Rote	MF	1
MF Total		6
Special offers - low price space lines	RJ	2
UI for low fares	RJ	1
Interface to Best Southern	RJ	1
RJ Total		3
Special offers - major space lines	WC	2
WC Total		2

The new plan drops the further performance improvement work for another iteration. In order to focus on the lowest price story WC picks up the special offers task from MF. The customer says that they would rather get the Mary's Rote interface rather than the problematic IHAB interface, so MF drops the latter completely.

Chapter 26

Stand up Meetings

You'll notice we aren't into lots of meetings. Meetings are top of most developers lists of boring time-wasters. But meetings are also essential to communicate between people. The challenge is to figure out what kind of meeting work well and how to structure them.

We've found that short daily meetings are invaluable to give everyone an idea of what other people are doing. However the emphasis is on 'short'. If the daily meetings every start dragging, you'll run into troubles.

The stand up meeting gets its name from one of the tricks we use to keep it short: everyone has to stand up for the whole meeting. That to remind everyone to be brief. If they don't remember, remind them, with a stick if necessary.

The format is simple. Everyone stands in a circle and you go round person by person (whether you go with or against the clock is up to you). Each person briefly says what they did yesterday and what they are doing today. If they have run into a problem, or have an announcement important to the team, they pass that on.

The purpose of the stand-up is to communicate problems, not to solve them. Your job is to keep the meeting brief. The usual stumbling block is when somebody says "I implemented some code the floop the foobar", someone else says "I did something like that last month", "oh I needed a triple axle", "you can do that by modifying the config file" – and suddenly it's on the way to being a long conversation. At this point you suggest: "maybe you should get together this morning and talk about this". Anything that requires anything more than a brief announcement should be shunted off to another session where only those who are interested need to be there.

You should have the stand up meeting daily, so that everyone knows roughly what everyone's doing. You should pick the same time every day, and aim at a time that everyone is there. Earlier in the day is better, as that allows time for people to get together quickly if they need to.

Chapter 27

End Game

The end of releases and the end of iterations are special times, requiring special care. We are struck by the image of Olympic speed skaters. Toward the end of long races the lesser skaters always let their technique fall apart—their left arm comes down off their back, they stand up too straight, their legs stop extending all the way. Much of the skill of the best skaters is keeping their technique intact even when they are in the most pain.

Development is like that. One project we worked on called it “third Wednesday syndrome” (they had three week iterations). Third Wednesday was a time of reckoning, a time to make final scope adjustments and ask for help so you could hit the end of the iteration cleanly.

The same effect works at the scale of releases. The last couple of iterations are always more stressful than usual.

The natural human tendency towards the end is to try to speed up. You have too much to do. How are you going to fit it all in? This doesn’t work. Don’t do that.

Your first defense against panicking at the ends of cycles is to have more of them. What? Yep. If two week iterations are tough at the end, go to one week iterations. That way there will be less chance of unsightly uncertainty build up between the beginning and the end of the iteration. If three month release cycles are tough, go to two month or one month releases.

Towards the end, productivity is not the problem, risk is the problem. Getting only three last things done instead of four won’t kill you. If you have been finishing the most valuable stuff first, all that’s left is less important stuff. What is going to kill you is trying too hard and

getting none of the four things done, and screwing up a bunch of other stuff in the process.

If you want the ends of cycles to go smoothly, you have to learn to choke up a little on the bat (Today appears to be a bargain day for sports analogies. We promise we'll move on to war or biology or something in the next chapter.) That means:

- ◇ you write more tests,
- ◇ you are more careful to defer unnecessary functionality,
- ◇ you integrate more often,
- ◇ you refactor more carefully,
- ◇ you talk more.

That's how you'll get the most possible work done at the time of the greatest stress.

Deployment

You should be deploying all along in your target environment so deployment isn't too big an issue. However, towards the ends of cycles be sure you can make the software work in all the environments where it needs to work. If you find deployment problems, write automated tests to make sure those problems never happen again (don't worry, you won't run out of problems this way—there will always be new ones).

We like using a machine for integration that precisely mirrors the set up of the production environment. That way you do a little deployment testing every day. If defects slip through to production because the configuration is different, fix the integration machine and the code so that defect could never slip through again.

Documentation

We tricked you. Documentation is not an end-game task. Since the features are going to be evolving throughout the release, the manuals will be evolving, too.

This suggests:

- ◇ Have as little printed documentation as possible. Use HTML instead. If you think you can get away with it, keep the HTML on your web site, so you can continue evolving it after deployment.
- ◇ If you must have printed documentation, drop ship it directly from the printer. This should cut a couple of weeks off the document lead time.
- ◇ Keep the system simple. If you are considering a feature but it will require 100 pages of documentation, perhaps you shouldn't be implementing the whole feature right now.
- ◇ Put the writers on the team. Good writers can spot inconsistencies in the detail of products that escape programmers and customers. If the writer is there in the room, partaking in the banter, they can make development better.

Chapter 28

Recovery

Life is what happens when you are planning for something else.

Source?

This chapter seems nearly redundant with the material in release events and iteration events

You are walking down stairs at night. You're sure you counted right, but when you come to the last step and start walking there's one more step. Your eyes come open an instant after your foot doesn't come down on anything solid. The next handful of milliseconds are a scramble to make sure you are positioned to retain your balance as quickly as possible when your foot touches down. After you're standing balanced on solid ground again you take a breath to calm down. Then off you go, none the worse for your little scare.

This isn't a happy chapter to write. Missing deliverables is a fact of life in software development. We do everything we can to avoid, we work hard to reduce the impact when it happens, but it happens. The question is not whether you are going to miss sometimes, but how you act when you do miss.

At worst, missing a deliverable can poison the relationship between business and development, set programmers at each others throats, and damage management's confidence in the team. Miss too many and your project is dead.

Perhaps it is precisely because of the severity of the consequences that folks steer right into the worst consequences of missing deliverables, like a kid on a bicycle smashing right into a post he is trying to avoid because he can't get his eyes off of it.

Missing a deliverable doesn't have to sink the team. Regaining balance after a slip can be a growing experience for the team. When you know you can count on each other to do okay even when the chips are down you'll be that much more confident in your ability to kick ass when the stars line back up.

A word about what we don't mean by recovery. At first your estimates won't be terribly accurate. Your first few iterations are likely to be rough. You'll get half as much done as you thought, or twice.

Kent has a project like this right now- the team completed 30 days worth of tasks in the first iteration out of 39 days estimated. In the second iteration planning meeting the tasks estimates added up to 59 days. The customer calmly picked the most important half of their stories. No one is calling for recovery. The future is far too uncertain for the effort to be worth it. The team doesn't know what they will be capable of in a couple more iterations. The customer isn't sure what the first release needs to contain. Everyone is comfortable (well, as comfortable as possible in the situation) letting the process ride a couple more iterations and seeing how things shake out.

Recovery is when you know you aren't going to complete everything the customer is sure they want. The goals are:

1. To complete the most valuable possible work while spending no more time on work that is less valuable or won't be completed.
2. Get back on track for next time.

Principles

Here are some principles to keep in mind when recovering.

Keep it in the family

If you are forever going to the customer and asking them to put off some stories, they will quickly lose confidence in you. Before you ask for relief, get the programmers together and see if there is some simple way to get more done. Can you put off some non-customer-related tasks? Can you shuffle some tasks around to get them done sooner, or to unblock someone who is overloaded? Can you pull in someone from outside the team to help?

A little slip sooner is better than a big slip later

As soon as you are certain that you can't finish the stories the customer has chosen, recalculate when you think you can do then ask the customer to pick the most valuable stories to retain.

Damocles could never ship software. You can't work well with that big old sword hanging over your head. The realization that you need recovery is likely to grow on you a little at a time. When you've gotten back to a realistic pile of work, you'll relax and the work will start to flow again. Better to do this sooner rather than later so you can get back to working at your best pace.

Ask the customer

The customer won't be thrilled to hear that they aren't going to get everything they picked. Get over your fear of their reaction. Take your lumps. Explain what happened. Lay out the cards for them. Step back. Take what they hand you.

That paragraph looks like you go crawling up to the customer on your belly. That's not our intent. Recovery is a fact of life in software development. It doesn't make you an bad programmer or a bad manager. It just means that you can't predict the future. We've already established that predicting the future isn't our goal. On the other hand, the customer is going to be disappointed because you are asking them to give up something they want. You will likely have to give them a chance to vent before you can get back to working together.

Change estimates only when you have more information

Okay, this isn't a principle, but it deserves a place of honor. You don't ever want to recover twice in a row. Recovering once is an adjustment. Recovering twice will feel like a betrayal. You are certain to lose if you reduce estimates because that lets you predict that you'll get more done. If you have to increase estimates or reduce your velocity based on your experience, do it. Don't reduce estimates or increase velocity out of guilt.

"Just another report shouldn't take this long." Silence. That's your best answer. It's your job to make estimates. It's the customer's job to deal with the consequences.

Don't slip the date

We were both taught, "Don't just slip the date, slip the date and promise more features." This is a really, totally stupid idea. The prob-

lem is that you have too much to do, not that you don't have enough time. You can't make more time. You can get less to do.

Stick to your guns

Kent had been working for six months on a one year contract when he was called onto the carpet. "We're not satisfied with your progress. You'll be fired if you don't get moving." Well, well.

It took him several days of soul searching to realize that in the remaining six months of the contract he could either get six months worth of work done, or less. There was no way to get more than six months of work done, no turbo boost button to get out of the situation. And the way he could get six months worth of work done was:

- ✧ writing tests
- ✧ refactoring
- ✧ communicating

Once you've felt XP working for you, you should have confidence in your process. An unhappy customer doesn't change the nature of software development. Whatever time you have left, make the most of it the best way you know how.

Don't be fooled twice

Why did it happen? Did you not spend enough time estimating? Too much time estimating and not enough time making real stuff? Were your estimates simply too low or you velocity too high?

Often the cause of recovery is that you slipped away from your practices. Did you commit to more this iteration than you completed last iteration? Did you not integrate soon enough? Did you not write enough tests? Have you gotten behind on refactoring?

What ever caused the need for recovery, make sure it never happens again. Make up a rule about it. Print it on a poster and tack the poster up in the coffee room.

Recovering an Iteration

You'd like to know half way through an iteration if you're half way done. That's why you try to complete some stories early in each iteration. What if you're half way through an iteration and you can't mark off any tasks as done, much less any stories?

First, keep it in the family. Can you shuffle tasks? Can someone outside the team grease the skids with a tricky bit of technology? Can you put off some infrastructure tasks?

Next, take stock of the iteration from the perspectives of the stories. The stories are what the customer cares about. Some tasks will be partly done. Estimate the work remaining for each tasks. Add up the task estimates for each story. Lay the stories out for the customer, give them a budget, and let them pick what you'll finish and what will have to be finished later.

Now focus your efforts on tasks related to the most important story. Nail it, then move on.

Recovering a Release

You need to initiate release recovery when you are sure that some important stories in the release plan aren't going to get done by the release date.

First, make sure that you need to recover. Go through the remaining stories and re-estimate them. Check your velocity. Do the math. Maybe you're on track and you just didn't realize it. Hey, we've seen it happen.

If the numbers don't add up, lay out the stories for the customer and give them their budget.

Be prepared to the stories to change. "What if you just did simple formatting for these reports? How long would that take?" Change the story to "Simple Report", reduce the estimate, then write a new story "Beautiful Reports" and estimate it. Simple reports will stay in this release and beautiful reports will get done later. Or not. It's up to the customer.

Regaining Your Balance

So far we've talked about the part of recovery where we're getting both feet back on solid ground. Recovery is not complete until you've got your balance back and taken a deep breath. You don't want to do this again.

So you missed an iteration, only completing three of the six weeks of stories you promised. The customer is disappointed. You're going to want nothing more than to commit to six weeks of stories in the next

iteration, or eight. Don't do it. You have concrete evidence that you can complete three weeks worth of stories in an iteration. You have no recent evidence that you can complete six weeks worth. In the next iteration planning meeting, swallow hard and ask the customer which three weeks worth of stories they would find most valuable.

Chapter 29

Visible Graphs

At any point, anyone should be able to see the state of the project by looking at a handful of graphs in the project team's working area.

Communication is one of the core values of Extreme Programming, and one of the key things to communicate is the state the project. All members of the team, any management or customer representative, or indeed any curious soul, should be able to tell at a glance how things are going. Projects get into trouble when problems are ignored or hidden. So a key part of planning is to ensure problems are made apparent to everyone as soon as possible.

Of course saying there's a problem can be a fraught experience. We've been in plenty of situations where we've cried foul on a project and got in a pissing match with whoever's running the project. Reputations, money, and pride are on the line. How's some outsider supposed to know what's going on anyway? How's an insider to prevent themselves from being like the frog in pot of water on a slow heat? When the temperature goes up slowly, you don't notice the fact that you're getting boiled.

So it's important to monitor the temperature, and to use objective measurements as much as possible.

Now we're well aware of the pitiful state of measurement in the software industry. There aren't very many measurements out there, and there are even less useful ones. However we've found that there are a few useful measurements you can keep.

Choosing Which Graphs to Show

But before we move onto some suggested graphs, we should pass on some advice as to which ones to use. These graphs are suggestions: you shouldn't be using all of them all the time. Remember that if you saturate people with graphs, you'll give them too much data and not enough information. So choose your graphs carefully. Consider what things you, your management, your programmers, and the customers are concerned about. For each worry try to think of a simple graph that'll demonstrate what's happening to everyone present.

If a graph has done its job, and it's no longer a worry: then drop the graph. Graphs are a powerful tool, but too many of them blunts the purpose. Everyone should know that the graphs count, and the chore of plotting them should be compensated by the warm feelings of useful information you are gaining.

Many people suggest putting these graphs on a web site. This is good if people on remote sites need to see what's happening. But don't let that be a substitute for putting them on the wall in the developers' area. Web sites don't get looked at if they aren't clicked on. You can't avoid what's on the wall. Many an insight comes when idly staring at a graph when you're half doing something else.

Functional Tests Defined and Passing

This is possibly our favorite: a running graph of how many functional tests are defined, and how many functional tests are passing. This graph can tell you a lot about the quality of the project.

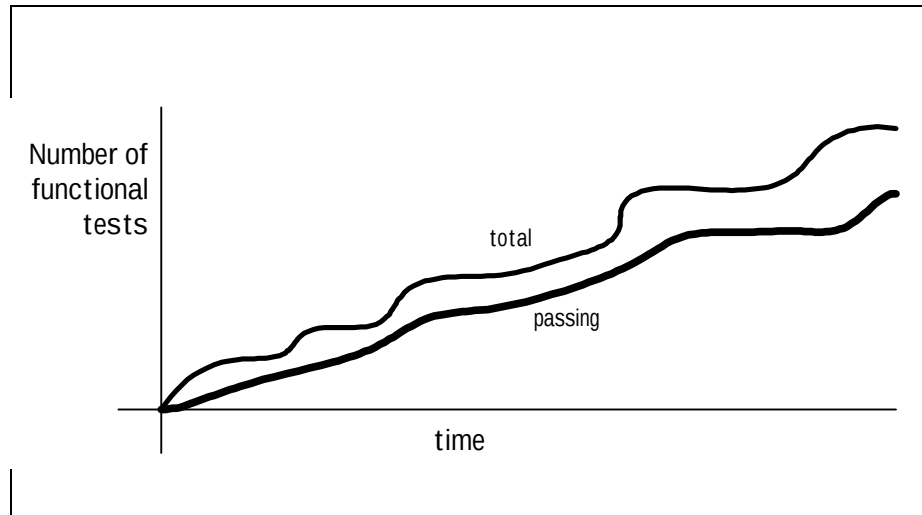


Figure 0.1 An example graph of functional tests defined and passing over time

The tests defined line tells you whether your functional tests are getting built properly. If that curve starts flattening, it's usually a sign of trouble. (The exception is when you are in a feature complete release iteration in the end game.)

The second thing to watch is the gap between the defined and working lines. If those lines start diverging, again it's a sign of problems. Different projects do different things about how they manage functional test scores. Some like to fix quickly, some manage the gap at a certain proportion. But a widening gap is never a good sign.

A closing gap, of course, is something you want to see, especially close to a release. If the gap doesn't close, that's a sign that you won't be ready to release on schedule, at least not with all functionality intact.

Production Code Bulk, vs. Test Code Bulk

We'll assume you know that it's gross stupidity to measure productivity by measuring line of code. That doesn't mean that it's wrong to

watch lines of code — you just have to properly understand what it means.

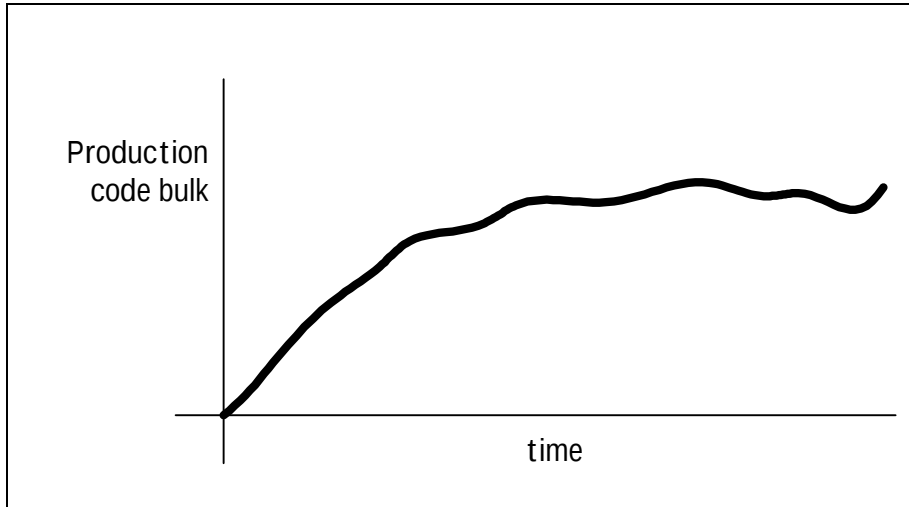


Figure 0.2 Production code bulk over time

Our experience tells us that a healthy project should show the kind of curve in Figure 0.2. There is a definite rise, but the rate of increase should drop as the project goes on. Indeed you should also see some drops. If this doesn't happen, it's a sign that the code isn't getting refactored enough and is probably getting crufty. Once you have a certain critical mass, you'll be surprised how much you can keep adding function while only slightly increasing the line of code count. This is good, because as far as software's concerned size is bad.

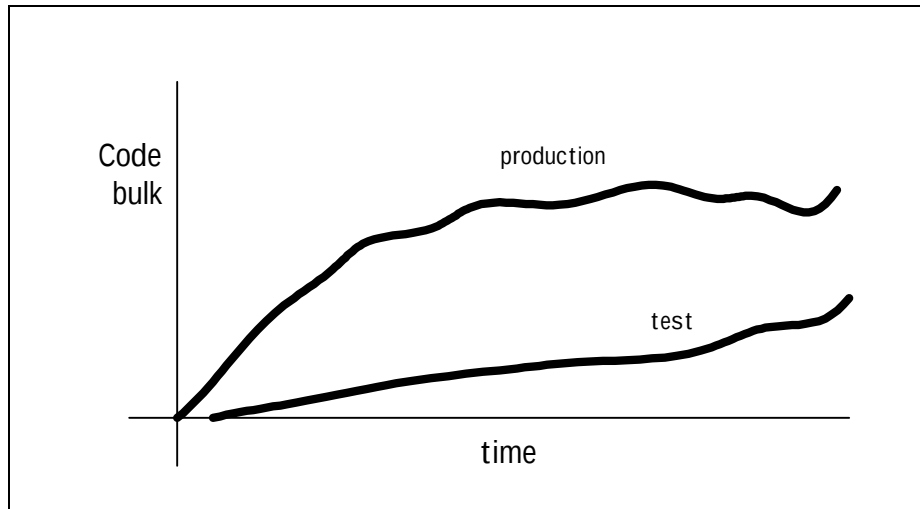


Figure 0.3 Test code over time

At least it is for production code. Test code however should increase, as Figure 0.3 suggests.

Successful Builds

If your project is having trouble getting it's build done: try this graph. Buy a year length calendar and a supply of green and red stickers just big enough to cover one day on the calendar. Every day the build succeeds, put a green sticker on that day. For these purposes a successful build is a full compile and link *and all unit tests passing 100%*. Anything else get's a red sticker.

You should see a steady parade of green stickers. Occasionally you might get a bad day and see a red sticker. But a line of red stickers indicates a problem that needs everybody's attention. Usually it means that the build process is faulty, or not being followed resulting in people merging code changes inconsistently. It may mean that refactoring is being done in chunks that are too large. Whatever the cause is, find it out and get those stickers back to green.

Relative Bug Density

Each time functional testing spots a bug, mark which part of the system it comes from. You can go to the class, but usually the nearest package or so will do the trick. You might choose to rank the bugs on how much effort they were to fix: 1 point for under half an hour, 2 points for under a couple of hours, 4 points each day or partial day after that. Then show a pie chart of points versus packages. This may help you spot areas of the system that need more attention, i.e. more unit tests and more refactoring.

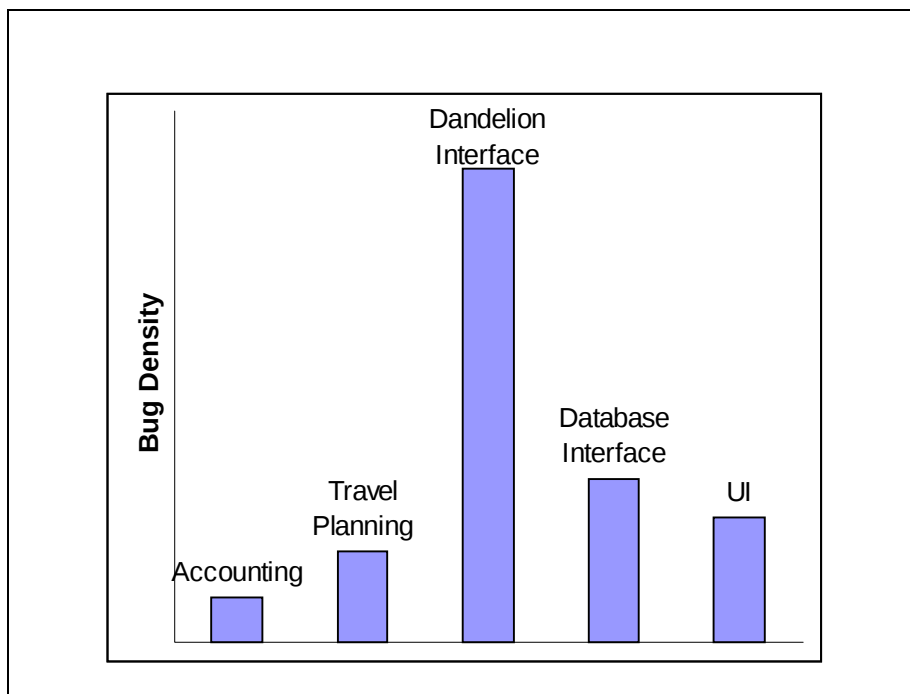


Figure 0.4 Relative Bug Density

This is more useful if you go extreme on a code base that wasn't developed extremely, as then the feedback is really useful as to where you focus your testing and refactoring effort.

Story Progress

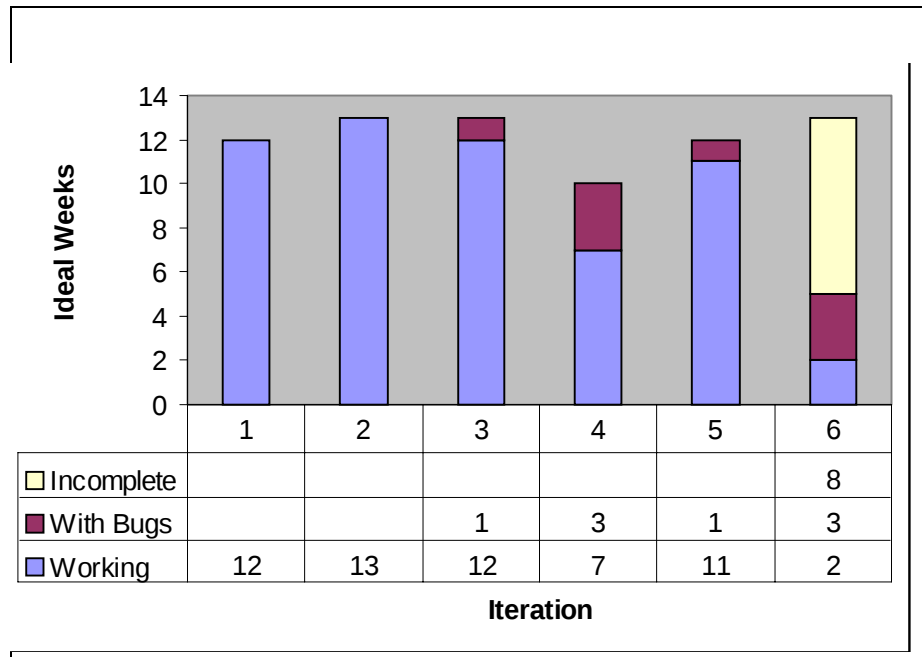


Figure 0.5 Story progress

This one helps show the velocity. For each iteration, past and future, show the amount of ideal time of stories for the iteration. Stories that aren't done are yellow. Stories that are done, but functional tests are less than 100% show red, stories with all functional tests passing 100% show green.

Notice that we are showing two things with this graph. The height of the columns shows how the velocity is changing over time. The color indicates both the state of the stories and how long problem areas are persisting. Red marks three iterations ago are a problem.

System Performance

Find some key activities that best represent the performance of the system in practice. Define automated tests that execute these activities

and capture how long it takes to do them. Plot a graph of these performance figures and how they change over the weeks and months.

Don't start plotting the graph until you actively begin performance tuning on the system.

How to use the Graphs

The most important things to do with the graphs is to plot them and make them visible. Often just that is enough to make something happen. Much as we like telling people what to do, we've noticed it usually works better to just make them aware of the problems and get out of the way.

Our favorite example for this comes from C3. At one stage the developers were concerned that they weren't getting the functional tests validated by the customer. Development would define functional tests that seemed reasonable, but the customer had so much to do that they didn't sit down and check the figures. The developers had pointed out the problem, but things were just too busy.

They were already plotting "Functional Tests Defined and Passing" on page 158. But they added a third line: functional tests defined but not validated and put that on the wall. Now it was obvious to everyone passing through that there was a problem as the defined and validated lines were diverging. Magically, nothing became more important to the customer than filling in the missing numbers. The graph succeeded where the requests hadn't. After a few more iterations the validated and defined lines started converging.

Your Graphs

The most important thing to remember is to select the graphs you need. While we'll be flattered if you pick the graphs we suggest, it's far more important that you think about your worries and choose graphs that illustrate your worries. Just trying to figure out what the graphs should be will probably do a lot to help you think through your issues. After all, we already get enough flattery.

Chapter 30

Dealing with Bugs

This is for the species boys and girls.

--Starship Troopers, the movie

*Schedule bug fixes with stories, so the customer can choose
between fixing bugs and adding further functionality.*

We've never tried farming, even though Kent now lives the middle of farms, pick-up trucks, and people who wear cowboy hats for real. One thing we imagine we have in common with farmers is a distaste for bugs. Programming bugs may not eat our source code, but they do eat at our customer relationships and productivity. And we can't get insecticide at the nearest supply shop.

Martin- parse please You may think that can claim that XP leads to software that is remarkably free of bugs, due to its strong emphasis on testing. We aren't so sure. There are plenty of software products out there with an acceptably low level of bugs (low in the sense of "high"). We're sure you can get there by a testing phase late in the project cycle. What XP does with its testing process is not something that is necessarily more efficient at finding bugs, but something that by bringing testing forward, makes the project easier to plan and increases programmer productivity.

One of the worst things about software bugs is that they come with a strong element of blame (from the customer) and guilt (from the programmer). If only we'd tested more, if only you were competent programmers, there wouldn't be these bugs. We've seen people screaming on news groups and managers banging on tables saying that no bugs are acceptable. All this emotion really screws up the process of dealing

with bugs and hurts the key human relationships that are essential if software development is to work well.

So let's get a few ground rules on the table.

We assume that the programmers are trying to do the most professional job they can. As part of this they will go to great lengths to eliminate bugs. However nobody can eliminate all of them. The customer has to trust that the programmers are working hard to reduce bugs, and can monitor the testing process to see that they are doing as much as they should.

For most software, however, we actually don't want zero bugs. (Now there's a statement that we guarantee will be used against us out of context.) Any defect, once it's in there, takes time and effort to remove. That time and effort will take away from effort spent putting in features. So you have to decide which to do. Even when you know about a bug, someone has to decide whether you want to eliminate the bug or add another feature. Who decides? In our view it must be the customer. The customer has to make a business decision based on the cost of having the bug versus the value of having another feature — or indeed the value of deploying now instead of waiting to reduce the bug count.

(We would argue that this does not hold true for bugs that could be life-threatening. In that case we think the programmers have a duty to public safety that is in fact greater than their duty to the customer.)

There are plenty of cases where the business decision is to have the feature instead. I'm most readers can think of a software product that they use regularly that they think has more bugs than it should. The company made a business decision to add features rather than fix bugs. Look at their share price over the last few years to determine if that was a good choice.

We once ran into a sharp example of this. We were involved in a project to replace an existing system. The customer decided to delay deployment because of bugs that, despite the team's best efforts, were still there. It then transpired that during one month the existing system, due to its bugs, lost the company several million dollars. The new bugs weren't anywhere near that expensive. Was the customer right to delay deployment? In hindsight we think not, although we agreed with the decision at the time.

Dealing with bug reports on deployed software

The most important thing is to remove the emotion. A bug report is a request for a change to the deployed system. As we all know many of these changes could be considered to be enhancements rather than defect fixes. We don't encourage you to try to classify them one way or the other, because doing so usually leads to unhelpful finger pointing.

First determine if the bug is critical, to do this and to deal with it see *Dealing with critical bugs* (p168) below.

If it's not critical then log it on a card. Get development to look at it and estimate the effort involved to deal with it. A lot of the time you don't know what's involved at this stage so mark it as unknown. If it's less than an ideal day mark it as small.

If it's more than an ideal day's worth of effort treat it as a story. The customer should then say which iteration should work on it, making room just as with any other story. Usually it's worth lumping several bugs together to get an ideal week's worth.

Just before the next iteration planning meeting the customer should take the small and unknown bugs and prioritize them. Then the customer should indicate how much ideal time the developers should spend dealing with them. That then becomes a story of effort that goes into the iteration planning exercise.

The point of this approach is to encourage everyone to deal with bugs in a rational way and to make sensible trade-offs between fixing defects and adding features. No two projects have the same priorities here. If fixing bugs is an absolute must, then you do them first using this process.

At this point we have to declare a health warning. We haven't managed to try this pure a process in action. Instead we've seen people use a *Production Support Team* (p167).

Production Support Team

Two or four programmers volunteer to focus on fixing bugs. Each programmer spends a couple of iterations in production support, then rotates back to development. Every iteration there is at least one developer doing their first iteration and at least one doing the second. This works well in that there is a pair that has the responsibility for dealing

with support issues and this (usually unpleasant) work is rotated around the team. Actually it's not the rotation that is key, it is the fact that the team decides themselves how to handle it.

This has worked fairly well. However, the customer didn't fully appreciate what they were trading off to get production support. The production support became a separate lump of effort that was scheduled independently to the rest of development. As such the customer wasn't forced to go through the explicit trade-offs that they had to do elsewhere.

Dealing with critical bugs

There are some bugs that can't wait until the next iteration. They really do have to be fixed now, or at least this week. The difficulty is identifying which things really are critical bugs. Again, only the customer can really make this call. Again the key is to remove the emotion if you can, and follow a similar process to the standard one. That is, developers estimate how long it will take to fix. They may well have no idea, in which case allocate two ideal days for investigation. Then the customer picks which story on the current plan takes the hit. That way the customer is explicitly making an explicit trade off of function (or even other bug fixes) versus fixing this bug. People make this trade off implicitly all the time, we prefer things to be explicit.

Chapter 31

The Customer

In XP we talk a lot about the customer. By customer we mean the person who makes the business decisions. Now of course you don't have only a single literal customer. You have users, business management, operations, all sorts of people who are customers. If you do shrink-wrap software you may have thousands of customers. However for XP to work, the customer must speak with one voice. Some people call such an animal a product manager, or a requirements champion. We use the term customer because that's who this person represents.

A lot of planning processes see the customer as some kind of disembodied entity outside of software development who provides requirements. You interpret, tease, do JAD sessions — but the customer is outside the team.

XP is not like that. XP planning assumes the customer is very much part of the team, even if the customer works for a customer company and the rest of the team works for a contractor. The customer must be part of the team because their role is far too important to leave to an outsider. There are lots of ways to sink a project. Running after technological eldoradoes, producing crappy quality, having your development team all hired away en-masse. But the single most reliable way to have a project fail is when the customer isn't able to steer.

So the customer's job is a huge responsibility. All the best talent and technology and process in the world will fail when the customer isn't up to scratch. Sadly it's also a role where we can't offer particularly sage advice, after all we're nerds, not business people. But here's what we know for sure.

Finding a Customer

Since the customer is such a critical role, it's important to find someone who will play it well. A good customer:

- ◇ Understands the domain well, by working in that domain for quite a while, and also by understanding how it works (not always the same thing).
- ◇ Can understand, with development's help, how software can provide business value in the domain.
- ◇ Is determined to deliver value regularly, and is not afraid to deliver too little rather than nothing.
- ◇ Can make decisions about what's needed now and what's needed later.
- ◇ Is willing to accept ultimate responsibility for the success or failure of the project.

This last seems to be most difficult. There is a certain comfort for the customer to maintain a distance from the team. Behind three feet of requirements documents is about right. In XP this won't work. If you get lost driving, it isn't the car's fault, it's the driver's.

The trickiest thing about XP for customer is getting used to the rhythm of regular delivery. A lot of processes ask the customer for everything they want. Instead XP asks for as little as possible that is enough to provide value. That's an unusual way of doing things. There's an argument that says that if you can't find a customer who wants to work that way you shouldn't try XP at all.

Guiding the Customer

If you're a customer, and we hope all customers read this, then here are some important things to remember.

At all times ask yourself "what is the most valuable functionality to have next?" Long term planning can be fun, but it's regular, little deliveries that keep the money coming in.

Trust the developers' estimates, yet remember they're only estimates and they will get it wrong. Estimating software development is a very difficult task, they're doing the best they can and they will get better.

Never let a date slip. Slipping dates is one of the worst habits in software development. You just slip one or two, and after a while you're addicted. It isn't completely against the rules to slip a date, it's just that the XP methodology requires you to chop one of your own fingers off each time you do it.

Provide a little valuable functionality every single release, and release as often as you can. Don't be afraid to release something that's not enough, yet. Use your creativity to look for ways that you can take a large new capability and break it up into little pieces so you can keep delivering. If you release frequently enough, you won't have long to wait before you get more of what you want.

Chapter 32

The Seed

In the beginning was the Word.

John 1:1

We have mentioned several times how starting an XP project is different than running one in a steady state. Nowhere is the difference more pronounced than getting starting programming. How are you supposed to evolve the design of a system that doesn't yet exist? How do you get five pairs of programmers working independently on the parts of a program that doesn't have any parts?

The solution, in a word, is Conquer and Divide (okay, in a phrase). You've got too much to do and not enough time- the natural response seems to be to divide the work into parts and work on them independently. This doesn't work all that well. The boundaries between the pieces aren't in the right places. There is too much communication needed between the pieces. The "independent" teams end up stepping all over each other.

Think for a moment about how a tree grows. It doesn't start by appointing a leaf team, a twig team, a branch team, a root team, and a bird experience team. A tree starts from a seed. Up comes a shoot, which sprouts two leaves. The shoot turns into a stem, the leaves feed the growth of tiny branches. The roots reach tendrils deeper and deeper into the ground, fueling growth. And so on until you have a majestic platform for tree houses and childhood accidents.

A tree grows by the opposite of divide and conquer. That first little shoot that pops out of the ground is already recognizably a tree, it's just little.

Start your software the same way. Build the system in three lines. Move those lines into an routine. Move the routine into an object. Split the routine into smaller routines. Split the object into smaller objects. Pretty soon you will see how two parts can be further evolved without interfering with each other. Now you can have two pairs working on the system, soon after that four.

We've had success starting this process in a big room with a projector connected to the computer. Bring the whole team- programmers, customers, managers. Spend a day or two implementing new test cases and evolving the design. At the end of that time you should have enough pieces that three or four pairs can work independently.

Another idea (thanks to Michael Hill) for getting started is the zero functionality iteration. There are often a bunch of little technical infrastructure tasks to get done before you can begin programming:

- ◇ Getting the testing framework working
- ◇ Getting the automated build structure working
- ◇ Getting the network up and running with all the appropriate permissions

You'd hate to commit to functionality in a first iteration, only to disappoint the customer because you couldn't get the framdoozle talking to the whatzit. If you haven't done worked with your technology before, consider spending two weeks getting everything working just right before you begin programming.

Chapter 33

Ready To Commit

Here is the question that has bedeviled people of a certain gender from time immemorial. When do you actually go public with what you intend to do in the future? As with all the big questions mentioned in this book we can't give you The Answer, but we can tell you how we approach answering.

All other things being equal, you are better off making your commitment quickly and getting on with experiencing development. The programmers will only really learn when they see real code in production. The business folks will only learn when they see real users really using the system.

Here are some of the reasons to spend more time or be more formal before committing:

- ◇ Technical risk. If the system is using new technology, or putting together technology in way new to the team, you should spend time trying out little (days or weeks) pilots before making estimates.
- ◇ Business risk. If the project is going to break business ground--reorganize people's jobs, sell an unprecedented service-- the business people should take a little time to explore their risks.
- ◇ Two parties. If the project is going to involve two or more companies, where one is legal liable to the other, it behooves (see, we start talking about lawsuits and we start using words like "behooves") you to take more care. You can easily go overboard, and spend more on protection than you could possibly lose.

Here are some of the reasons to begin getting concrete earlier rather than later:

- ◇ Experienced team. If the team has done this application five times before, you don't have to dot every "t" and cross every "i". If you get it a little wrong they will be able to fix it.
- ◇ Experienced customer. If the customer has guided development of substantially similar systems, you can go ahead more quickly.
- ◇ Trust. Those first two are increasingly rare conditions, but if the team and the customer have worked together before and established a trusting working relationship, then you can go ahead, trusting that any rockiness can be overcome.
- ◇ XP in place. Once your team is up to speed on XP it is much easier to go ahead on a handshake, because they have confidence in their own abilities, and they know where they are weak.

Here are some other notes we had:

- ◇ Metaphor
- ◇ "Architecture"
- ◇ Get to point you say "don't worry about that".
- ◇ How does it feel when you're ready
- ◇ Its okay to be realize a little late

What about research?

What if your project is going to require you to do a hundred things that no one has ever done before? All this scheduling doesn't seem to apply in that case.

True. You can't schedule innovation. What you can do, however, is be aware of when the emphasis has shifted from fear of the unknown to more ordinary fears, and begin planning.

The mistake we see is hanging onto research mode far too long. "Here are five things that might be impossible, lack of any of which will sink the project." How much do you have to find out about each of the five before you can begin planning? A little, certainly, but not near enough to produce production code.

Kent has a project that spent a year producing five binders full of potentially useful information about the product they are building, but no production code. The product managers feel better having all of those documents, but planning is difficult because they can always see twenty things they aren't doing.

Spend a month giving all your big risks, technical risks and business risks, a "once over lightly". If after that you can't tell whether the project is stupid or not, it probably is. However, there may be a part of it which clearly makes sense. If so, commit to that and do the rest later.

Chapter 34

Changes to the Team

When the team changes, how does that affect your planning?

Coming

Give new team members an iteration or two to get acclimated. They can:

- ◇ pair with more experienced folks,
- ◇ read code and test cases,
- ◇ talk to customers.

During this time, we haven't found the need to predict a reduction in the team's velocity. Time spent answering newbie questions is made up for in the new perspective they bring to old problems.

Yesterday's Weather will tell you when to increase your estimates based on the presence of new people. A project of Kent's added two new people to a team of eight and had a disastrous iteration, mostly because of deferred refactorings that caught up with them. The team declared that the "Lost Iteration" and planned for the next iteration as if it hadn't happened. The next iteration they committed to 22 days of stories and completed 37. Was it the refactorings or the new folks? Impossible to tell, but the team was on a new, higher trajectory that has been since sustained.

Going

When someone leaves the team you won't have the usual panic about "what do they know that no one else knows?" Pair programming has ensured that the knowledge is spread around.

If you have five programmers and one leaves, reduce your next iteration by 20%. Yesterday's Weather will fix the new velocity quickly enough.

Splitting the team

Ward Cunningham speculates that the way to scale XP up to 30-40 programmers is to complete the first release with a team of ten, then split into two teams. Each team gets their own stream of stories from their own customer. The teams of five then grow to ten. Do this twice and you're at 40.

For planning purposes, if we were to do this, we would begin committing each subteam to half of what the whole team accomplished before the split. Yesterday's Weather will quickly tell you how much cross-team overhead to plan for.

People growing

People aren't the same day to day. Testers become programmers, programmers become managers, managers throw off the shackles of oppression and become programmers again (oops, our bias is showing).

XP isn't executed by Plug Compatible Programming Units. It is executed by people, changing people. What if someone got bored with programming all the time? How could the team gracefully adapt to changes in lifestyle?

This is just an idea, but what if you put the management tasks on the board alongside the technical tasks during iteration planning? The project manager would typically sign up for tracking, the status report, and pizza selection. However, if someone else wanted to get a taste of project management, they could sign up for one from time to time. If the project manager needed to move on, someone could easily step in, even if only temporarily. The project manager could also take a programming task from time to time, to see how things looked from ground level.

Chapter 35

Tools

There are two problems to solve in project management:

- ✧ Keeping track of all data
- ✧ Maintaining communication and relationships between people

Our strategy towards tools for project management is biased heavily towards maintaining human communication and the relationships between people. For small to medium sized projects, keeping everybody talking honestly is a much more difficult problem to solve than calculating the schedule impact of surprises, so it deserves stronger tools. And the winner is:

Little pieces of paper

The primary physical unit of schedule information on XP projects is the index card. Index cards are:

- ✧ Portable
- ✧ Machine independent
- ✧ Approachable
- ✧ Cheap (sorry about this one—you can send us a bunch of money if you'd like and we'll send you *really* nice index cards)

Remember our purposes in planning:

- ✧ Decide whether or not to go ahead
- ✧ Coordinate within the team and with the outside world
- ✧ Gauge the impact of surprises
- ✧ Set priorities

None of these are computationally impressive tasks. We can do the arithmetic in our heads and write the answers down on the cards. More complicated calculations can easily be handled by a spreadsheet. From

this spreadsheet you can also generate all kinds of cool and impressive reports and graphs, some of which are even useful.

If you must use a spreadsheet, never forget that the spreadsheet is not as important as communication and relationships. “I discovered a couple more tasks for this iteration, but you don’t want to edit all those reports.” Time out! The tail is wagging the dog. Making sure everyone understands all the tasks required for this iteration is far more important than getting the reports exactly right this instant. If necessary, abandon the spreadsheet. If everyone understands but the spreadsheet is hopelessly out of date, that is much better than having a perfect record of incorrect information.

Chapter 36

Outsourced XP

The very best software development process in the world is helpless in the face of the wrong contract. Any contract that pits the interests of the supplier against the interests of the customer is trouble. If the opposition becomes worse as pressure increases, the trouble is well nigh insurmountable.

The typical outsourced development contract fixes three of the four variables:

- ✧ Scope
- ✧ Time
- ✧ Cost

Unfortunately, it is impossible to fix all four variables. If there is a surprise, one of the variables will change (hence the name, "variable"). Since quality is the hardest variable to measure, surprises tend to get absorbed by reducing quality-- a little less testing, a little less design, a little less communication.

The balloon story tells us that any attempt to rescue a plan by adding risk of defects is doomed. Once you have an unknown amount of rework lurking; scope, time, and cost will all explode.

The fundamental problem with fixed scope contracts is they pit the interests of supplier and customer directly against each other. The customer wants as much scope as possible for their money but the supplier wants to do as little work as possible for the money. As long as everything is going according to plan, this opposition isn't necessarily fatal. When the surprises start adding up, the contract goes sour.

Fixed scope isn't fixed

There are number of band-aids to apply to fixed scope contracts to make the opposition between the interests of supplier and customer a little less fatal:

- ◇ Late completion penalty. The cost of the contract can be reduced if the software isn't done on the date. The customer still isn't happy, because they still don't have the software when they need it, but at least they aren't unhappy and poor at the same time.
- ◇ Early completion bonus (because the supplier needs something to offset the risk of the penalty)
- ◇ Change orders. The customer can be charged a negotiable fee for each change to the scope.

Wait a minute. It looks like scope isn't so fixed after all. What would happen if we stopped predicting scope?

Negotiable Scope Contracts¹

What if the contract read, "Supplier will have eight programmers work for Customer for two months for \$320,000. Scope will be negotiated every two weeks according to the classic book 'Planning Extreme Programming'." That's it. Simple to write, simple to read (and it boosts our book sales).

But wait. The customer wants to know what they are going to get for their 300 grand. Sure, they want to know, but they can't know (planning is not about predicting the future). What you can assure the customer is they will get your best efforts for two months. After all, it's only two months and the project is likely to last for 6-8 months. It's not likely you will send in scrubs and leave 3/4ths of the money on the table.

But wait. The supplier wants to know what their employment requirements are for the next six months. Well, then, it behooves them to keep the customer happy. If this project doesn't work out for the customer, they had better do such a good job that the customer is willing to engage them for the next one.

1. Thanks to Dave Cleal for many of the refinements on the idea of negotiable scope contracts.

One of the attractive features of fixed scope contracts that the supplier gives up with a negotiated scope contract is the possibility of windfall profits. Some of our most profitable work has been when we already knew exactly how to implement what the customer wanted.

Suppliers can offset the absence of windfall profits by pricing negotiable scope contracts at a premium. If you know about pricing options, you can explain how a negotiable scope contract of four iterations is worth more than a fixed scope contract of the same duration. If you don't know options pricing, it takes us at least two beers to explain it, and so is clearly outside the scope of this book.

Chapter 37

In House Development

In house development has its own sets of advantages and disadvantages for XP. The advantages are:

- ✧ You don't have an explicit contract, so you don't have to worry about being sued if you fail. Fired, perhaps, but not sued. Okay, so this isn't such a big advantage.
- ✧ You are working inside the same culture. Business and development are likely to share some values and myths.
- ✧ You have a built in arbitrator in "the big boss". Of course you'd rather resolve your disagreements before they escalated that far, but at least there is a clear place the buck stops.

The disadvantages are:

- ✧ You business and development already have history, and it is likely to be, well, a bit strained (if business and development get along swimmingly, you don't need XP, XP needs you).
- ✧ You likely have several customers to serve, not just one. Resolving conflicting priorities can take up all of your energy.

Customer

The best customer Kent ever saw was on a customer service system project. The customer had spent seven years opening envelopes and answering phones while using the legacy system. Then she spent three years supervising other representatives. This left her with complete knowledge of the existing system, and an understanding of where the system was weak, where mistakes were likely to be made, where duplicate information was required. She was also fearless in making decisions, but conscientious verifying them after she made them.

Best if you can find a single person who can confidently speak for business. This person needs:

- ◇ experience
- ◇ contacts
- ◇ vision
- ◇ courage

If you can't find a single person to act as customer, you will have to have a committee of customers. Make this committee explicit. Insist that they get together to discuss your priorities. We watched a team go from corporate heroes to goats in one year because two customers had widely differing priorities but they were never explicit about their differences, nor did they do anything to resolve them.

If the customers can't collaborate to decide on stories for releases and iterations, try an idea from Dave Cleal: give each customer a budget. Let's say you've measured that your team can produce 8 ideal weeks worth of stories every iteration. Give the trading floor three weeks to spend and give the back office three weeks. Both customers will have to be present at every release planning meeting and each iteration planning meeting, and both will have to be available for answering questions throughout development.

Contracts

Just because you can't be personally sued is no reason not to have a clear agreement about what you intend to produce. Software suppliers are used to finding new customers. In-house, you'll still be eating lunch next to that customer for years and years.

After we did our first ever release plan, we presented it to the CIO and the customer's great-grand-boss. The programmers and customer sat together on one side of the table. The customer explained the scope cuts we'd made, and why more cuts didn't make sense. The programmers explained how they had come up with the estimates. Then we laid out the cards and tallied them up.

The big cheese weren't happy about the date, but the team spoke with such conviction that there wasn't a big problem. It also helped that we asked for a project review after the second iteration, and asked that the project be cancelled if we hadn't made good progress by then.

Chapter 38

Shrink Wrap

How do you plan when you don't have one customer, you have hundreds of customers, thousands of customers, millions and billions and trillions of customers? XP requires the customer to speak with one voice.

We believe this role is called "product manager" in most places. The product manager listens to many sources:

- ◇ Sales
- ◇ Marketing
- ◇ Customer support
- ◇ Customers

The product manager merges the stories, gets estimates from development, sets their relative priorities, and speaks to development through iteration and release planning. The key point is that responsibility for resolving impossible situations is shifted away from development. If you can only satisfy two of three customers immediately, the product manager decides which two get stroked now, and which one will be upset for a while.

Kent has a project where there isn't one product manager, there are six product managers. One acts as the overall product manager, while the other five each have responsibility for a part of the product. Through some magic process completely invisible to development, they work out their relative priority iteration by iteration. By the time they get to the planning meeting, they present their stories with one voice.

Chapter 39

Red Flags

Kent has a rolling suitcase whose wheels are too close together. When he is running between planes, the suitcase starts wobbling. Trying to hold on tighter to the handle just makes it wobble more. Eventually the stupid thing just falls over.

The solution, of course, is to buy a new suitcase with suitably spaced wheels. That wouldn't make much of a teaching story, though. Kent's solution is to slow down until the suitcase stops wobbling. It doesn't matter how late you are for your plane, if the suitcase falls over you'll be later.

Here are some common signs of trouble with daily planning and what we do if we see them. The solutions are all some form of "slow down until you are under control, then speed up again."

- ◇ Missing estimates
- ◇ Customers won't make decisions
- ◇ Defect reports
- ◇ Not going end to end
- ◇ Failing daily builds
- ◇ Customer won't finish

Missing estimates

If every time you come to the end of an iteration you have to throw away half of the stories, then something is definitely wrong. Iterations can have a moment of dramatic tension, but most iterations should finish comfortably. You might not know on the second Tuesday if you're going to make it, but when the lunch pizza comes on Friday and you

push the button to run the acceptance tests, everyone should be confident of the outcome.

Are you committing to too much? Kent had a project where the programmers' guilt level was high enough that they always rounded up. "We got 38 days done last iteration with 4 people, so call that 10 days per person and we have a new person, so that's 50." They missed a little every iteration, but more important they got behind on testing and refactoring, until finally one iteration completely pancaked. They delivered 1 (one) day to the customer. That was the wake up call to just follow Yesterday's Weather.

Have you gotten behind on testing and refactoring? If so, you'll see debugging take an increasing percentage of time, but an unpredictable percentage. The only solution we've found is to slow down, temporarily, and write the extra tests and do a little more refactoring. After a couple of iterations your speed should pick up.

Have you succumbed to the temptation to slice your estimates? It is so easy for programmers to feel guilty when the customer is disappointed, and for that guilt to turn into shorter estimates, not for any technical reason, but because they want the customer to be happy (or at least stop bugging them). This doesn't work. Don't do that. When programmers make estimates, have them find comparable work, and make the estimates comparably. "We have two new reports. They are both about the same as the framdoozle report we did last iteration, and that took a week. Call them a week each."

Customers won't make decisions

Sometimes customers simply refuse to make decisions. They won't pick the stories for an iteration. They won't specify acceptance tests. They won't answer little questions about scope.

Extreme programming cannot work without a customer making decisions. They don't have to stick with those decisions. Changing their mind is one of their rights. But the decisions must be made by someone with a business perspective.

Sam Gentile reports of a project where the customer's boss signed a contract for extreme development. When the team got there and started exploration the customer was incensed. "I hired you to do the analysis, not for me to do the analysis. Get to it. And I'm busy, so don't

ask me any questions." Fortunately they had the wisdom to terminate the contract at that point.

Find out why the customer won't make the decisions. If their priorities are elsewhere, perhaps the whole project doesn't make sense. If they don't want to be publicly wrong, the best you can do is reassure them that they will get a chance to fix any errors in the next iteration.

Defect reports

Between the unit tests and the acceptance tests, the software produced by an extreme team should be essentially defect free. There should be enhancement requests aplenty, but the software should work as specified.

If you are seeing enough defect reports to disrupt development, slow down until you aren't seeing them. Have a pair go through every defect and find where it occurred and what unit test should have been written to catch it, then report back to the team.

Sometimes a class of defects will turn into a design change--"We have six bugs when we run with Netscape. We have to create a Browser object to isolate the browser-specific behavior."

Not going end to end

It is so tempting to stop developing when you get uncomfortable. We're building a web server, and we've got all these tests, but all inside our network. Go out to Seattle Coffee and rent a station on the Internet and try to get to the server. If we can't, because our proxy server isn't configured yadda yadda yadda, then stop until it is. Every dark corner you haven't explored with your flashlight is full of bugs.

Failing daily builds

If the 8-10 integrations you do every day are going smoothly, but you have problems when you push the software to production, make the integration environment match the production environment more closely. The feedback from the little integrations won't help your planning if you don't know how much progress you've actually made.

Customer won't finish

One of the failures we've seen in XP is where the customer flits from flower to flower--a few stories from here, a few from there. The programmers blaze along, hitting their estimates, finding fabulous abstractions while refactoring, testing like mad. Then one day the project is cancelled because it didn't get anything done.

Are you making 3-4 month release plans that the customer presents to upper management? Are the results of those plans reviewed when their date rolls around? The big wheels are there for a purpose--use them. Big bosses can be good at spotting misfits at the scale of quarters, "I can see that all this web stuff is exciting, but if we don't prepare for this Federal Reserve report we're out of business."

Chapter 40

Your Own Process

We have been prescriptive here, telling you what to do to plan your software project. At the end of the day, however, you have to own your own plan and your own planning.

One way to take adapt XP planning to your own situation is to take the bits and pieces of it that make sense and mix them with what you are doing now. The problem with mix-and-match is that the parts of XP planning are intended to complement each other. If the project manager estimates tasks, then assigns them to programmers, the programmers won't have the same sense of ownership as if they did the task breakdown and estimates themselves.

Here's an alternative strategy. Adopt XP planning "by the book". Run for a couple of iterations. Then look at the problems you are having and experiment with each iteration.

- ◇ "We aren't getting done on time. Let's try one week iterations for a month."
- ◇ "Let's have the customer check off the task boxes."
- ◇ "Let's put targets for refactoring on a board."

Spend an hour or two each iteration reviewing past experiments and results and brainstorming new ones. If you have the ceremonial pushing of the acceptance test button at lunch on the last Friday of the iteration, you can spend the rest of the day talking about your process.

The shops where we've seen XP happily adopted are those where they have started simple and evolved. Some of the evolutions seem pretty radical at first. However, once the team explains the rationale for the new practices, they often make sense.

Chapter 41

Bibliography

We have a friendly competition to see who can include the strangest defensible bibliography entry. In joining forces on this book, we feel an obligation to pool our experience of weird reading for the benefit and amusement of you, our loyal reader.

The Inner Game of Golf

"Awareness is the dual of performance." Gallwey gets dramatic coaching results without ever telling students what to do. Instead he makes them aware of what they are doing now.

Critical Path, R. Buckminster Fuller

Innovator's Dilemma

