

Crystal Clear

A Human-Powered Methodology For Small Teams, including The Seven Properties of Effective Software Projects

Alistair Cockburn
Humans and Technology
copyright 1998-2004, A. Cockburn
last save date: June 17pm, 2004

Preface

Crystal Clear: A few key rules to get a small project into its safety zone.

You have barely enough resources to get the system out. You don't want the team to write long documents, but they are forgetting things they are supposed to know about. You dislike heavy software development processes, but you want your team to work better than just randomly. You particularly want the software to come out the door successfully.

You considered sitting down and writing out the basic discussions the team should have, the work products they must be careful to attend to. You asked yourself:

What have other small, successful project teams done?

What practices do they use?

This book answers those questions. It is the result of ten years of debriefing successful small teams. Most of them repeated the same message:

- Seat people close together, communicating frequently and with good will;
- Get most of the bureaucracy out of their way and let them design;
- Get a real user directly involved;
- Have a good automated regression test suite available;
- Produce shippable functionality early and often.

Do all that, and most of the process details will take care of themselves.

This book sets out one of the most efficient and habitable methodologies you might hope to find, Crystal Clear. It is a *human-powered* methodology, most simply described as follows:

The lead designer and 2-7 other developers in a large room or adjacent rooms, with information radiators such as whiteboards and flipcharts on the wall, having access to key users, distractions kept away, delivering running, tested, usable code every month or two (OK, three at the outside), periodically reflecting and adjusting on their working style.

This simple recommendation rests on both experience and theory. Software development can be characterized as an economically constrained *cooperative game*

of invention and communication¹. The way the team plays each game has everything to do with the project's outcome and the resulting software. Crystal Clear tackles the economic-cooperative game directly, addressing where to pay attention, where to simplify, and how to vary the rules. A number of teams have shared with me – and now with you, through this book – examples of their rules, work products and even office layouts.

Many so-called "best" methodologies get rejected by a team as being too constraining, too invasive, or too difficult. Crystal Clear does not aspire to be a "best" methodology; it aspires to be "sufficient," in order that your team will shape it to themselves and then actually use it.

Origin of the Material in this Book

The IBM Consulting Group asked me in 1991 to write a methodology for object-technology projects. Not knowing enough about methodologies at that time to make the crucial decisions, and at the suggestion of my boss, Kathy Ulisse², I started interviewing project teams. What they told me was very different from what I had been reading in the books. In particular, they stressed aspects not covered in the methodology texts: close communication, morale, access to end users, and so on. It was not long before these issues separated in stark contrast the successful projects I visited from the failing ones. I came to see these issues, and not the design techniques, as the key to reaching a successful project outcome.

I got to try these ideas out as lead consultant on a \$15 million, fixed-price, fixed-scope project of 45 people. The ideas worked as advertised (coupled with a lot of creativity along the way), and showed themselves as core success factors. I wrote up the lessons learned from the project interviews and that project in *Surviving Object-Oriented Projects*³.

One particular triplet showed up repeatedly: colocation of the team, frequent delivery, and access to an expert user. The differences in results between projects that did and didn't do these far exceeded any other short list of practices. This book builds from that triplet.

The projects in my career have generally been of the fixed-price, fixed scope variety. People usually underbid on these projects, which means that the only way

¹ Described at length in *Agile Software Development* (Cockburn 2002) and recapped in the first answer of *Questioned*.

² Thanks for the brilliant advice, Kathy!

³ The project debriefings ended up as the basis for my doctoral dissertation, "People and Methodologies in Software Development" (Cockburn 2003).

the team can deliver on time is by being very creative with their development process. Unlike most of the other authors of the Agile Development Manifesto, I came to the agile principles through the need for efficiency, not the need to handle rapidly changing requirements.

As a result, Crystal Clear is well suited to the fixed-price context. If you are in such a situation, use the planning, communication and reporting mechanisms I describe to meet your (probably unrealistic) deadline, and just be careful *not* to change the requirements at the start of every iteration. If you have an exploratory project in which the requirements are unknown or fluid, that is fine: then *do* allow the requirements to move at the start of each iteration.

Crystal Clear in the Crystal Family

Crystal is a family of methodologies with a common genetic code, one that emphasizes frequent delivery, close communication and reflective improvement. There is no *one* Crystal methodology. There are different Crystal methodologies for different types of projects. Each project or organization uses the genetic code to generate new family members.

The name "Crystal" comes from my characterization of projects along two dimensions, size and criticality, matching that of minerals, color and hardness (see Figure 7-1).

Larger projects, requiring more coordination and communication, map to darker colors (clear, yellow, orange, red, and so on). Projects for systems that can cause more damage, need added *hardness* in the methodology, more validation and verification rules. A quartz methodology is suited to a few developers creating an invoicing system. The same team controlling the movement of boron rods in a nuclear reactor needs a diamond methodology, one calling for repeated checks on both the design and the implementation of their algorithms.

I characterize Crystal methodologies by color, according to the number of people being coordinated: Clear is for collocated teams of eight or fewer, Yellow is for teams of 10-20 people, Orange is for 20-50 people, Red is for 50-100 people, and so on, through Maroon, Blue and Violet. Crystal Orange is described in *Surviving Object-Oriented Projects*, and its variant Crystal Orange/Web is described in *Agile Software Development*. I find that, except on life-critical projects, people can add in the verification activities through the methodology shaping and tuning workshops⁴.

⁴ If your company develops FDA- or other life-critical, "validated" systems, you may want to set up three base methodologies, one for a Clear (quartz) projects, one for Yellow or

Crystal's genetic code is made up of

- the economic-cooperative game model,
- selected *priorities*,
- selected *properties*,
- selected *principles*,
- selected sample *techniques*, and
- project *examples*.

The economic-cooperative game model says that software development is a series of "games" whose moves consist of nothing else besides inventing and communicating, which is typically resource-limited. Each game in the series has two goals that compete for resources: to deliver the software in this game, and to set up for the next game in the series. The game never repeats, so each project calls for strategies slightly different from all previous games. The economic-cooperative game model leads people on a project to think about their work in a very specific, focused and effective way.

The priorities common to the Crystal family are

- *safety* in the project outcome,
- *efficiency* in development, and
- *habitability* of the conventions (the developers can live with them).

Crystal has the project team steer toward seven safety properties, the first three properties of which are core to Crystal. The others can be added in any order to increase safety margin. The properties are

- *frequent delivery*,
- *close communication*
- *reflective improvement*,
- personal safety (the first step in trust),
- focus,
- easy access to expert users, and
- technical environment with automated testing, configuration management and frequent integration.

Crystal's principles are described in detail in *Agile Software Development* (Cockburn 2002). Among them are a few central ideas:

- The amount of detail needed in the requirements, design and planning documents varies with the project circumstances, specifically
 - the extent of damage that might be caused by undetected defects and

Orange, and a third for all of the validated systems. Those three should may provide adequate basis for shaping to any of the projects in your company.

- the frequency of personal collaboration enjoyed by the team.
- It might not be possible to eliminate *all* intermediate work products and promissory notes such as requirements, design documents and project plans, but they can be reduced to the extent that
 - short, rich, informal communication paths are available to the team;
 - working, tested software is delivered early and frequently.
- The team continually adjusts its working conventions to fit
 - the particular personalities on the team,
 - the current local working environment, and
 - the peculiarities of the specific assignment.

Among the rest are trade-off curves that highlight the cost implications of different communication mechanisms, different project situations, and different strategies for concurrent development. I used the principles to derive Crystal Clear, but don't discuss them separately in this book.

The Crystal package includes selected sample techniques, including ones for methodology shaping, planning, and reflective improvement. Crystal does not require any specific technique to be used by any of the people on the project, so these techniques are included only as a starter set.

Each member of the Crystal family is generated at the start of a project by shaping a base methodology according to the genetic code. Since the situation changes over time, the methodology is retuned during the course of the project. Both shaping and tuning are performed fast enough that the time spent gets repaid within the project timeframe.

Crystal Clear is an optimization of Crystal that can be applied when the team consists of two to eight people sitting in the same room or adjacent offices. The property of close communication is strengthened to "osmotic" communication, meaning that the people overhear each other discussing project priorities, status, requirements and design on a daily basis. This enhanced communication allows the team to work more from tacit communication and small notes than otherwise would be possible.

Because every company and project is slightly different, even Crystal Clear is not fully specified. The first step in adopting Crystal Clear is to uncover your organization's strong points and weaknesses, and to fit the recommendations of Crystal Clear around them to capitalize on the strong points and cover the weaknesses.

For some organizations, this is too much work. Crystal Clear is not for those organizations. It is for groups that want to build their own, personal, strong, and effective way to deliver software repeatedly.

Crystal Clear shares some characteristics with XP but is generally less demanding. You might think of it as a more laid-back alternative, a place to fall back to if XP isn't working for the group, or a springboard to get some agile practices in place before jumping into XP.

This Book in the Agile Development Series

This book is part of the *Agile Software Development* series edited by Jim Highsmith and myself. The series describes both theory and practice for software development.

The theoretical underpinnings are discussed in four books: *Agile Software Development* (Cockburn 2002), *Adaptive Software Development* (Highsmith 2001), *Agile Project Management* (Highsmith 2004), and *Lean Software Development* (Poppdieck 2003).

The other books in the series pick up the thread either by describing a technique for a particular individual or role, a technique set for the entire team, or a methodology sample.

- We find attention to the role of project manager in *Surviving Object-Oriented Projects* (Cockburn 1998), *Agile Project Management* (Highsmith 2004), and to the role of requirements writer in *Writing Effective Use Cases* (Cockburn 2001), *Patterns for Effective Use Cases* (Adolph 2002).
- We find attention to the entire team in *Improving Software Organizations* (Mathiassen 2001) and *Configuration Management* (Haas 2003).
- Specific agile methodologies are described in *DSDM* (Stapleton 2003), *Agile Software Development Ecosystems* (Highsmith 2003), *Iterative and Incremental Development: A Manager's Guide* (Larman 2004) and this book.

Future books will follow the theme, adding techniques for collaboration and team health.

How to Read This Book

Some people will read this book as an introduction to agile development techniques, and be relatively new to pair programming, test-driven development, osmotic communication, economic-cooperative game, continuous integration, and information radiators.

If that is you, then read this book pretty much straight through, because you are the person for whom I designed the chapter ordering.

- I wrote the first chapter, *Explained*, as an email exchange between Crystal and myself to expose how these teams look to an outsider (remember, it was once all new to me, too).
- *Applied (Properties)* is still the most important chapter. It describes what the team is aiming for, not the procedure it uses to get there.
- *In Practice (Strategies and Techniques)* should give you a handle on how some of this is done.
- *Explored (Processes)* describes the cyclical development processes core to all agile (indeed all modern) methodologies. It is something that everyone should be fluent in.
- Just scan the *Work Products* on the first pass, since it is very detailed. Use it as an encyclopedia of work product samples when you get that far.
- *Misunderstood (Common Mistakes)* and *Questioned(Frequently Asked)* should answer questions about what counts as okay and not-okay variations.
- *Tested (A Case Study)* gives you another chance to see the methodology from the outside, since that was written for people not familiar with Crystal Clear. It also contains an ISO 9001 auditor's analysis and recommendations, which shines light from a different angle.
- Read *Distilled* to see if it makes sense at that point. If it doesn't, you may have to go back to *Questioned* to see why such a simple recommendation works.

Some people are fully versed in modern agile development including test-driven design and continuous integration. If you are one such, I suggest you go directly to the chapter *Distilled*. After that (when you are done snickering), read *Properties*, probably the most important chapter in the book. My guess is that you will find some new ideas to try out in *Strategies and Techniques*. After looking through those chapters, return to *Distilled* to see if it all hangs together for you.

If you are giving this to your boss or manager, my hope is that the first chapter with the emails is the kind of thing that can be read on the airplane or in the bathtub. A manager or executive should also read the *Explored (Processes)* chapter, because learning how to fit a cyclical process into the organization is important.

Process or methodology designers are quite likely to turn straight to *Examined (Work Products)*, because this is a standard way of evaluating methodologies (although in the case of Crystal Clear, quite insufficient). To complete the evaluation, though, you need also to read *Questioned(Comparing)* to see the comparison with other methodology systems.

Finally, you will be ready to start. At this point, read the answer to the last *Question*, "How do I get started?" and work from there. That will take you back to the methodology shaping and reflection techniques and *Properties*.

I wrote each chapter in its own unique style and tone. There is a reason for this. People learning a methodology are in much the same situation as the blind men trying to guess the shape of an elephant, each feeling a different part and coming up with a different answer. Each person, coming from his or her unique background notices different things, and looks for different things. Therefore, the nine different chapters are written in quite different ways. I don't expect everyone to be happy with every chapter, but I do hope that everyone finds *some* chapter that addresses his and her individual background and interests.

Acknowledgements

I am indebted to many more people for this book than for any of my previous ones: people who told me their stories, people who tried out the ideas, people who contributed samples, people who reviewed the text, and people who supported me emotionally.

Most of the people who told me their project stories during the last ten years don't know how much value they provided as they explained – even apologized for! – their ways of working. They often said, "we don't use a methodology here, we just . . .", or "I'm sorry, we don't bother to do [xyz], or maintain our documents, but we have found . . ." It was only by comparing results across a number of projects that I discovered that in many of these instances, no apology was ever needed, that there was a strength in the way they worked.

Certain people were pioneers in trying out these ideas out on their projects. Jens Coldewey was the first, back in 1998, Robert Volker in 1999, Gery Derbier in 2002, Stephen Sykes in 2003. Dr. Christopher Jones tried out an early version on his unsuspecting senior Software Engineering students (who used every imaginable implementation technology). His students showed me where my writing was ambiguous or poorly described. Stephen got permission for me to include both his field report and the auditor's analysis in this book, a huge contribution.

Pete McBreen kept reminding me that *people* are also valuable carryovers from one project to the next (something I knew, but which kept slipping out of my descriptions until Pete reminded me again). Jeff Patton and Andy Pols were continual discussion partners on the topics.

The following people contributed office photos and work samples (I thank each again next to their samples): Lise Hvatum, Jeff Patton, Ron Jeffries, Jonathan House, Nate Jones, Kay Johanssen, Darin Cummins, Randy Stafford, Mike Cohn, . . .???

Luke Hohmann and Tom Poppendieck read everything I wrote with astonishing thoroughness and caught errors of omission, commission and even intention. Tom

came up with the idea of dating the emails instead of numbering them (Duh, why didn't I think of that?! Thanks, Tom).

The Silicon Valley Patterns Group and especially Russ Rufer and Chris Lopez read the manuscript carefully not just once, but twice, giving their usual thoughtful comments and pushing back on me when they felt I had gone off track. Russ argued me down relentlessly on some of sections until I finally got his point.

A few people read it from the web and wrote in with corrections and improvements. Thank you, Marco Cova, Todd Little, Alan Griffiths, Howard Fear, Victoria Einarsson, Paul Chisholm, Pierce McMartin, Phillip Back, Todd Jonker, Chris Matts, Gain Wong, Jeremy Brown, John Rusk, Johannes Brodwall, ???

The people in the Salt Lake Agile Group roundtable even "designed the box" for Crystal one day. The top quotes from that day were,

"Self-tuning, self-correcting, just add people!"

"Works with powdermilk biscuits!"

and the winning testimonial:

"I've used Crystal all my life, and I've never been the same!"

Thanks to Jim Highsmith, co-editor and partner in crime, for the many illuminating discussions that shaped our ideas, our book series and the wonderful Agile Development Conference in 2003.

Thanks to my new favorite coffee shop, the Salt Lake Coffee Break, which stays open until 2 a.m., has interesting clientele, power outlets that work, and baklava and Turkish coffee for those rare moments. Thanks to my family: Deanna for checking in with me at the coffee shop at 1:00 a.m. to see how the typing was going (and then letting me sleep in in the morning), and Kieran, Sean and Cameron for their positive regard of my writing habit.

Table of Contents

Chapter 1 Explained (View from the Outside) 16

I distilled Crystal Clear by asking successful small teams what they would keep or change in the ways they worked. The responded with this deceptively simple set of rules. This chapter is written as an email exchange between the fictitious Crystal and me. The emails let you encounter the rules from the outside, as I first did, and allow me to push back against Crystal's reports, asking questions.

Chapter 2 Applied (The Seven Properties) 33

*Reading how Crystal Clear works raises two particular questions: "What are these people concentrating on while they work?" and "Can we get farther into the safety zone?" This chapter describes seven properties set up by the best teams. Crystal Clear requires the first three. Better teams use the other four properties to get farther into the safety zone. All of the properties aside from **Osmotic Communication** apply to projects of all sizes.*

Property 1.	Frequent Delivery	35
Property 2.	Reflective Improvement.....	38
Property 3.	Osmotic Communication.....	38
Property 4.	Personal Safety.....	46
Property 5.	Focus.....	49
Property 6.	Easy Access to Expert Users.....	51
Property 7.	Technical Environment with Automated Tests, Configuration Management & Frequent Integration.....	54
	Evidence: Collaboration across Organizational Boundaries.....	59
	Reflection on the Properties.....	60

Chapter 3 In Practice (Strategies & Techniques) 63

Crystal Clear does not require any one strategy or technique. It is good to have a set in hand to get started, however. This chapter presents a few of the less-well documented and more significant one used by modern agile development teams.

Strategy 1.	Exploratory 360°	65
Strategy 2.	Early Victory	67
Strategy 3.	Walking Skeleton	68
Strategy 4.	Incremental Rearchitecture	70

<i>Strategy 5.</i>	Information Radiators	73
<i>Technique 1.</i>	Methodology Shaping.....	79
<i>Technique 2.</i>	Reflection Workshop.....	84
<i>Technique 3.</i>	Blitz Planning.....	87
<i>Technique 4.</i>	Delphi Estimation using Expertise Rankings	95
<i>Technique 5.</i>	Daily Stand-Up Meetings	97
<i>Technique 6.</i>	Essential Interaction Design.....	98
<i>Technique 7.</i>	Process Miniature.....	109
<i>Technique 8.</i>	Side-by-Side Programming.....	111
<i>Technique 9.</i>	Burn Charts	113
	Reflection about the strategies and techniques	127

Chapter 4 Explored (The Process)129

Crystal Clear uses nested cyclic processes of various lengths: the development episode, the iteration, the delivery period, and the full project. What people do at any moment depends on where they are in each of the cycles. This chapter linearizes the cycles to the extent possible, and points out some of their interactions.

The Project Cycle.....	135
The Delivery Cycle	141
The Iteration Cycle.....	144
The Integration Cycle.....	147
The Week and the Day.....	148
The Development Episode	149
Reflection about the Process	150

Chapter 5 Examined (The Work Products)151

This chapter describes team roles and the work products, showing examples of each work product. These particular work products are neither completely required nor completely optional. They are the ones I can vouch for both one at a time and taken all together. Equivalent substitution is allowed, as is a fair amount of tailoring and variation. Although this is where the most argument is likely to occur, it is where the argument is probably least likely to affect the project's outcome.

Roles: Sponsor, Ambassador User, Lead Designer, Designer-Programmer, Business Expert, Coordinator, Tester, Writer	155
<i>A note about the project samples.....</i>	158
Sponsor: Mission Statement with Tradeoff Priorities.....	160

<i>Team:</i> Team Structure and Conventions	163
<i>Team:</i> Reflection Workshop Results.....	166
<i>Coordinator:</i> Project Map, Release Plan, Project Status, Risk List, Iteration Plan & Status, Viewing Schedule	168
<i>Coordinator:</i> Project Map	169
<i>Coordinator:</i> Release Plan	170
<i>Coordinator:</i> Project Status.....	174
<i>Coordinator:</i> Risk List	178
<i>Coordinator:</i> Iteration Plan ↴ Iteration Status.....	179
<i>Coordinator:</i> Viewing Schedule	182
<i>Business Expert and Ambassador User:</i> Actor-goal list	183
<i>Business Expert:</i> Requirements File	185
<i>Business Expert and Ambassador User:</i> Use Cases.....	189
<i>Ambassador User:</i> User Role Model	191
<i>Designer-Programmers:</i> Screen Drafts, System Architecture, Source Code, Common Domain Model, Design Sketches and Notes.....	193
<i>Designer-Programmer:</i> Screen Drafts.....	196
<i>Lead Designer:</i> System Architecture.....	198
<i>Designer-Programmer:</i> Common Domain Model	201
<i>Designer-Programmer:</i> Source Code and Delivery Package	204
<i>Designer-Programmer:</i> Design Notes	205
<i>Designer-Programmer:</i> Tests.....	209
<i>Tester:</i> Bug Report	212
<i>Writer:</i> Help Text, User Manual, and Training Manual.....	214
Reflection about the Work Products	215
Chapter 6 Misunderstood (Common Mistakes)	217
<i>You think you are using Crystal Clear, yet your project is not working. What's wrong? Crystal Clear can fail, but let's first double check that you really are doing Crystal Clear. This chapter present sample project situations. Some of them fulfill the intention of Crystal Clear, others violate it. The purpose here is to provide you with a personal warning system that you are or are not in tune with the intention of Clear.</i>	
"We colocated and ran two-week iterations – why did we fail?"	218
"Two developers are separated by a hallway and a locked door.".....	219
"We have this big infrastructure to deliver first."	220

"Our first delivery is a demo of the data tables."	221
"No user is available, but we have a Test Engineer joining us next week."	221
"One developer refuses discuss his design or show his code to the rest."	221
"The users want all of the function delivered to their desks at one time . . ."	222
"We have some milestones less than a use case and some bigger."	222
"We wrote down a basic concept and design of the system. We all sit together, so that should be good enough."	223
"Who owns the code?"	223
"Can we let our Test Engineer write our tests? How do we regression test the GUI?"	224
"What is the optimal iteration length?"	224

Chapter 7 Questioned (Frequently Asked)227

Readers may be curious about how these ideas arose, compare to others in the industry, how far they can be stretched, and what to do when they don't seem to apply. The chapter is presented in question-and-answer form to allow for "talking about" the ideas, everything from philosophical foundations to "how do I get started?"

Question 1. What is the grounding for Crystal?	229
Question 2. What is the Crystal Family?	238
Question 3. What kind of methodology description is this?	242
Question 4. What is the summary sheet for Crystal Clear?	247
Question 5. Why the different chapter formats?	249
Question 6. Where is Crystal Clear in the pantheon of methodologies?	251
Question 7. What about the CMM(I)?	259
Question 8. What about UML and Architecture?	263
Question 9. Why aim only for the safety zone? Can't we do better?	265
Question 10. What about distributed teams?	267
Question 11. What about larger teams?	269
Question 12. What about fixed-price and fixed-scope projects?	270
Question 13. How can I rate how "agile" or how "Crystal" we are?	270
Question 14. How do I get started?	271

Chapter 8 Tested (A Case Study)275

Stephen Sykes of Thales Research and Technology in the UK experimented with an early version of this book and tried it out. Here is his report on the experience, along with the ISO 9001 auditor's recommendations. Many thanks to both Stephen and Thales.

The Field Report.....	277
The Auditor's Report.....	301
Reflection on the Field and Audit Reports.....	307
Chapter 9 Distilled (The Short Version).....	311

At the end, it is time to roll it all back up again: What is the core of Crystal Clear, and what are the add-on practices that get the team farther into the safety zone? This chapter is very short

Chapter 1

Explained (View from the Outside)

I distilled Crystal Clear by asking successful small teams what they would keep or change in the ways they worked. They responded with this deceptively simple set of rules. This chapter is written as an email exchange between the fictitious Crystal and me. The emails let you encounter the rules from the outside, as I first did, and allow me to push back against Crystal's reports, asking questions.

Preface to the emails: The following emails are written to simulate what it is like to encounter for the first time a team doing Crystal Clear. While writing, I discovered that it was better to write as though Crystal were the one doing the project interviews and teaching Alistair-the-author, even though in real life Alistair did the interviewing. In what follows, Alistair gets to be the slowpoke, with Crystal pulling him reluctantly along, even scolding him on occasion.

Here is the office layout Crystal refers to:



Figure 1-1. A work setting showing osmotic communication. (Thanks to Tomax)

June 1: Dear Alistair,

Just getting back from my trip to Cryogenic Commerce, and I have to tell you what I saw. This must be the dozenth time I have visited a successful software group, and there is great similarity between the really productive ones. I know many organizations want to work in an extremely fast, productive way, so I am going to capture this one for the others who want to do so, too. The folks at C.C. were articulate about what they did, so I think I can answer your questions.

The project consisting of three people, Kim, Pat and Chris. Kim is the team leader and lead designer.

Kim says that they run all projects with two to four people if possible, because that is how many they can coordinate easily. They run six people on occasion, but that is a stretch and they refuse to go larger than that. They say that if they need more than six people, they haven't set up their project scope correctly.

They all sit in one room, as in the photo. They do this whenever they can, but if, for some reason, they have

to use separate offices, they arrange for them to be next to each other.

Very close contact is one of the most important things to their way of working. They want communicating to be as easy as calling across the room. There is just too much happening for them to have to pick up the phone or walk down the hall. They can get away with adjacent offices, but only barely.

I asked what they use for their requirements and design medium. They pointed to the whiteboard, and said, "There is our design medium."

Using the whiteboard as the primary design medium is one of the things I have found over and over. One project manager told me, and it was on a much larger project, that he was so pressed for time that they just photographed the whiteboard as their design documentation! Another time the team leader showed me the tables, lines and scribbles on his whiteboard and said, "I defy you to tell me what published methodology supports THIS notation!"

I have to agree. I have seen many sketches on whiteboards around the world. I can hardly think of a better medium. The only thing is that you can't fold it up and take it with you! Thank goodness there are printing whiteboards and PC-attached whiteboards, the Mimeo radio device for ordinary whiteboards, and digital

cameras⁵. It could easily cost justify any of those on saved communication time.

Kim said they use very light use cases as a basis for requirements, I'll send you an example of their two-paragraph use cases later on.

On occasion they find they can live with even briefer requirements. They have very good links to their users, so the lighter stories they are mostly just a reminder of what must be developed. The continuous conversations with the users keep the requirements accurate.

Once they collect their use cases and other requirements, they don't create many other archived deliverables except the final commented code and the user help text. I know this may sound incredible, but I have seen it over and over and over again, in the highest productivity groups I have found, and I don't think it can be ignored.

They hold design reviews at the whiteboard. There is no formal writing for these, most of the information is conveyed in the talking and hand movements, with some instance diagrams and interaction diagrams drawn along the way, and some other diagrams⁶. Some of these do not match any published drawing standard - but why should they, if everyone in the room understands what they mean?

What else is there to say?

⁵ The software product *Pixid* removes glares from digital whiteboard photos.

⁶ See the Work Products chapter.

They look over each other shoulders a lot, sometimes doing *side-by-side programming*⁷, so they get a lot of code reviewing done on a continuous basis. They like this, calling it "peer code peering." I have run into some form or another this in many places.

Extreme Programming (XP) people take the idea the farthest, programming in pairs. XP'ers say they catch more mistakes this way, sooner, than they would ever have imagined, and the resulting designs are better. It also gives them some intellectual backup. There is always someone else who understands a bit of how each person's code is built, and they also learn programming tricks from each other.

I did ask Kim and company whether they used ROOM or DOOM or UML or OML or any of the other published methodology things. They were not at all shy on this one! Pat said, "Our job is to get software out, not spend our time drawing pictures!" Chris said, "Who should we draw them for and why? Our users see the results so quickly, and we all understand the insides thoroughly. You've seen our whiteboards." Kim added, "We tried them for a while, but we are so closely linked that they don't add anything."

This was a very outspoken group, but I have encountered this reaction from other teams, and not just small

⁷ Placing their screens about 18 inches apart, they each work on their own programming tasks.

ones. Most small teams are a little apologetic about not using the drawing notations and tools, as though they are doing something wrong. On the other hand, an architect on a 150-person project said he doubts that graphical techniques scale, and so on his large project they kept all their design information in text.

Well, there you have it. No doubt you have some questions, so fire away and I'll see what I can answer.

cheerily, Crystal

Dear Crystal,

Thanks for the description. I guess I do have one question to start with. Did you get enough detail to describe their process? What happened and what did they do, and what happened then and what did they do?

best regards, Alistair

*June 2: **Dear Alistair,***

Yes and no. I stuck around for several days, so I could see them in action. What I would say is this, and you can do with it what you want. . .

Their minute-to-minute process was so complicated that I couldn't possibly write it all down, and if I could, you couldn't possibly follow it, let alone install it on another team.

What I would say is that they strive to develop a community mind. They sat down at the beginning and understood the problem they were solving, worked out how they were going to go about it,

and then started working . They are in close enough communication that they know what is important and what is happening. That is enough to keep them in step with each other, and they let the dance, as it were, improvise itself around them.

Here's what I can offer:

First, they interview their users and their sponsoring executive. Pat also went and watched the users at work, so Pat became the local expert on user actions. They collected requirements as two-paragraph use cases, you recall. They prioritized these with the users, so they knew which ones delivered the greatest value and which ones were needed first.

Then they got their technology requirements straight, what they had to use and what they could use. Kim is the technical leader, so Kim pretty much decided on the technology and laid out the basic architecture.

Third, they made a delivery plan, which included a delivery every two months.

Within each delivery, they first discuss the purpose of screens and screen sequences with users, using either just words or pencil and paper, until they are pretty Clear on the user requirements. They discuss how to set up the core business objects and who gets what parts of the system.

Typically, Kim takes the infrastructure, Pat takes the user interface, and Chris takes the business

objects. They are not hung up on this. There have been times when Pat and Chris each took both business objects and user interface objects, and they shared frameworks.

There is an initial demo to the user, after a couple of days or a couple of weeks, using just the screen and some dummiied up data. A fully functional demo comes after perhaps a month. Everyone learns something from these, even though they have been in discussion quite closely.

I'll tell you what happened one time, just so you understand.

Chris had gotten all the user screens fairly well right and all, but the first prototype took seven seconds round trip for a transaction. The user looked at the screen and asked, "Uh, is that the way it is going to perform in real life?" Chris said, "Well, we have some performance tuning to do, but more or less, yes. Why?" The user replied that she gets sent a fax with hundreds of lines on it, and she simply types from the fax, 'heads down', as it were, never looking at the screen, but just touch typing the hundreds of entries. She couldn't possibly wait seven seconds between each line.

Chris was dumbfounded. "But what do you do if there is a typing error?" "Oh, well this is all recorded just for tax deduction purposes, so actually we don't much care if there is an error. It isn't worth the time to correct it," answered the user.

Chris was quite shaken up by how badly they had misunderstood the requirements, even after interviewing the user and showing drafts of the screens and all. I guess it just goes to show you the difference between a mockup and real usage.

That's most of their "process."
cheers, Crystal.

Dear Crystal,

Thanks for the field notes. I look forward to your next installment. Tell me about testing. Do they have test tools, regression tests, external testers, what do they do?

thanks, Alistair

June 3: Dear Alistair,

Yes, I meant to get to testing sooner or later. First let me finish off the process, and the testing will show up.

Imagine they have a **delivery period lasting 2 months**. They go over the requirements with the users, and start drafting their design. Here is where the process gets fuzzy.

Some of the people at C.C. are "thinkers," others are "typers."

The thinkers sit down and think about the problem, designing on paper or whiteboard for as long as they can stand, until they either have a design they believe in or else they have to start programming to work out their confusion. This thinking sometimes takes a few minutes, other times it takes hours, or even a couple of days in the

worst case. After they program some real code, they learn more about how their design is working, and they either continue with it, or think and draw some more on how it needs to be changed.

The "typers" just start typing right away. At least, that's how it looks. What they really do is think about a tiny piece of functionality, write a unit test if they're doing test-driven development, and then get that tiny section of code working. When they add the next section, they refactor the two together to improve the design. These people spend more time typing and less staring than the others, but I can't particularly tell that it makes any real difference in the long run. It seems to be more a personality thing.

Constant testing. The best groups, and C.C. is one of them, deliberately design an architecture that supports fully automated regression testing. Some groups run their tests from the workspace window, then stick them right into the class system with the rest of the production code. Other groups stick them into files and run batch jobs against their code, pulling the test cases out of the files.

Kim said that it is hard to convince even experienced people to bother setting up a system architecture that supports regression testing, to build test cases as they go. However, once they experience the pleasure of developing with automated regression tests at their fingertips, they never go

back. The regression tests and batch test architecture gives them a freedom to make changes and a feeling of security they never had before. They make some changes, then push the test button and find out if they messed up something that was working before.

Let's see, process... They have a number of small design reviews within the team, they demo the system to the users a couple of times, build regression tests, and at some point they declare they are done. They write the user manual or help text, create a production build, run the tests again, and then **deliver** the system to some friendly users

How am I doing?

Crystal

Dear Crystal,

Doing fine. Now, every methodology must announce its tolerances. You are saying that Crystal Clear has tolerances like this:

- Coding style has great tolerance, being a matter for the team to decide.
- On the other hand, regression testing, peer code peering, user links, short releases are matters with very little tolerance.
- Team size has some tolerance, the range being 2-8 people.
- Release length has some tolerance, the range being 1-3 months.

I am with you on those things. I am still stuck on the process, though. It

seems that they just "run around and do stuff." I sort of get the idea, but I can't think of how to describe it to someone else.

What do you think we should do about this lack of description?

Trusting you, but still confused,
Alistair

*June 4: **Dear Alistair,***

Get serious, Alistair, these are grown people, and have normal mental equipment. What do they do? They look around, they think, they talk to each other, that's what they do.

One note about the team lead or lead designer, though:

Kim said that they always have one experienced person on the team. They often have one novice. If they have two rank novices, the team lead must be particularly experienced and good.

I recall you told me about your first job fresh out of college, and it wasn't much different. You were on a team of three designing a hardware subsystem of a flight simulator. You were the novice. You shared the office with your boss, the team leader and lead designer. The other designer sat in the office across from you. Your boss designed the subsystem's architecture and you other two contributed design and ideas.

Your team leader had been through it before, and his boss had been through more. You lived it and learned it. And that was on a project with 26 designers total. It is much easier if the

entire project is only three to five people.

Are you willing to trust that three intelligent, trained people can use their common sense?

The main thing is that the overall process delivers releases often enough that there is little time for the project to go far off track.

So there.

You talk about techniques. Well, these people use every technique known to computer science, and a bunch of stuff never described. That is why this methodology has to have a lot of tolerance. They care about how the end result looks, not what sequence someone went through to invent it.

cheers as always, Crystal

Dear Crystal,

Uh, thanks for that, I think I needed it. Somehow it is just so easy to forget about people thinking and talking to each other when doing this process stuff. :-(

Actually, I *am* wondering how to make this ISO/CMM(I) certifiable⁸, but let's leave that for a while and figure out what else there is to say.

OK, you've talked about team setup, user connections, incremental iterative development, use cases, deliverables, testing, what else is there to cover?

regards, Alistair

⁸ See the auditor's analysis in *Tested*.

*June 5: **Dear Alistair,***

You forgot to mention something critically important in a methodology: team values. That drives the rest of the methodology.

The key phrase here is "light on work products, strong on communications," with *reflective improvement*. They work on developing a community mind, with constant linkage to the users and frequent demonstrations and deliveries. The favored milestone is delivered functionality, and with that they can correct almost any problem.

By staying light on the work products, they move much faster, and change directions faster. Since they can move faster, they can deliver something useful in less time, thus making their milestones. Since they deliver in less time, they need fewer intermediate work products. It is self-reinforcing, as you see.

They view developing software as an evolving conversation – an economic-cooperative game, actually. In any one move they go forward and create reminders for each other.

Is that enough to satisfy you?
cheerily Crystal

Dear Crystal,

That's great, thanks. I'll see if I can capture what you have said in some sort of closed format so someone can walk away with it on paper.

Shall I use list of methodology elements I use in my other books, with roles, deliverables, teams, skills, standards, and all that? I'm afraid I'll have trouble with all this tolerance you want me to stick in there.

What really strikes me about this is that when you write down how an organization works, even a small group with a light methodology and small set of deliverables, it really looks complicated!

best regards, Alistair

June 6: Dear Alistair,

I'll propose something shorter, more stable and easier to remember than your tables and tuples. I think that some of the items in your tuples aren't really critical, and there are multiple pathways that work.

How about focusing on the key *properties* they set into place, and let different teams to follow their own preferred paths, just as long as they establish the properties.

For example: **Frequent Delivery** as a property. How frequent is **Frequent**, and just exactly what counts as a **Delivery**? How do they do the delivery? I propose we leave the answers up to the team to decide, as long as **Frequent Deliveries** of some sort are happening. Some of the others would be :

Reflective Improvement
Osmotic Communication
Personal Safety

Focus

Easy Access to Expert Users
and don't forget their

Technical Environment, with automated testing, configuration management, frequent integration.

For the work products, provide *samples*. They will be much more useful than templates. Get whiteboard photos, sketches, whatever, from real teams.

Yes, even a small methodology is a complex thing, but what do you expect? It is the encoding of a culture, and of a successful culture which produces serious software. Crystal Clear is as simple and short as I can keep it, but I can't make it simpler than it is.

Just let go of the techniques.
cheerily Crystal

Dear Crystal,

Got it. We'll consider techniques a "local matter" then.

Here's the next question. Many people will think that Crystal Clear will only work with all expert developers.

What do you have to say about this?
Best regards, Alistair

June 7: Dear Alistair,

Relevant question. Long answer . . .

People learning a new skill pass through three quite different stages of behavior, known as the *following*, *detaching*, and *fluency*. We can call them levels 1, 2 and 3. These stages of

learning are well known in aikido, where they are called there *Shu, Ha,* and *Ri*, which translate roughly to *follow, break, leave*. It is probably not an accident that craft houses in the middle ages also had three levels: apprentice, journeyman and master.

This is all described in *Agile Software Development* but it is maybe a good idea to summarize here, also.

In Japan, they draw *Shu-Ha-Ri* with the characters:



Figure 1-2. Shu, Ha, Ri.

A person just starting can't learn multiple methods at once, even if there are ten that could work. So in Level-1, the *following* stage, people look for *one* procedure that works. They copy it. They learn it. They measure success by how well they can follow it. In fact, they take for granted that it works, and can get very upset if it fails on them.

The *Shu* stage manifests itself in software development methodologies as thick, detailed manuals.

This book is an example. Crystal Clear can be described in one sentence (as in *Distilled!*). To an experienced software developer, this one sentence is adequate guidance.

That sentence is not enough for a Level-1 practitioner. He wants details of what to do. When I expand a

particular technique, process or work product, I am to readers at Level-1 on that particular item.

Everyone is Level-1 on a new skill. Even advanced developers work at Level 1 the first time they try a Reflection Workshop.

Eventually, they recognize that the technique won't serve all situations.

In the *detaching*, or Level-2 stage, they look for rules about when the procedure breaks down. Let me give you two examples.

In the 1990s, organizations that had become highly tuned to Information Engineering (IE) architectures suddenly had to deliver object-oriented software. They first tried to adapt the IE methods, then they developed completely new development methodologies. Along the way, they often regressed through totally *ad hoc* techniques before discovering new structures to support the OO projects.

A second shift happened after the Agile Development Manifesto was published in 2001⁹. Many groups trying it out found a mismatch between agile and their standard organizational practices. So they experimented with alternative ideas.

Eventually, people become fluent in enough situations that they stop paying attention to which technique they are using at any moment. They invent,

⁹ <http://AgileManifesto.org>

blend, adapt everything they know faster than they can even describe.

This is the *fluency*, or Level-3 stage.

Level-3 developers don't pay huge attention to the methodology formula. They pay attention to what is happening on the project and make process adjustments on the fly. They understand the desired end effect and simply make their way to that end. Simply stating the desired *Properties* of the project suffices for them.

Discussions among Level-3 people sound distressingly Zen:

"Use a technique so long as it is doing some good."

"Do whatever works."

"When you are really doing it, you are unaware that you are doing it."

To someone at the fluency level, this is all true. To someone still detaching, it is confusing. To someone looking for a procedure to follow, it is useless.

The one-sentence description of Crystal Clear is enough for the Level-3 developer. That description has been used directly by several project leaders, one of whom reported back later, "We did what you said, and it worked!" (which, to a Level-1 audience, doesn't communicate very much).

Try being aware of these levels the next time you are in a meeting. You are quite likely to find two people stuck in an argument, and notice that they are at different levels on the current topic. What is a natural set of variations for one person seems bewilderingly vague

to the other. If you identify the level difference, the two can negotiate the level of the discussion, or at least be more patient with each other.

All of this applies to Crystal Clear.

I don't think that Crystal Clear can be run by a team of all Level-1 practitioners. I doubt that these 300 pages of description are explicit enough.

Tom Poppendieck describes the paradox neatly: Level 1 people need the detail, but can't handle its complexity. Level 3 people can handle the complexity, but have little use for the detail.

To deal with that paradox, I have in Crystal Clear that any one project needs at least one Level-3 developer or two Level-2 developers. That *Lead Designer* expands the information on the fly. These pages support him or her in doing that.

In practice, the lead designer more or less runs the project. Often there is one other fairly experienced person, but the rest are all at various stages of beginning or middling (some young, inexperienced and bright, some older and tired or just at their maximum level of capability).

So finally I can answer your question: No, it is not the case that Crystal Clear will only work with all expert developers. But at least one of them needs to be a Level 3 developer.

I have in mind that the project sponsor will be hiring according to that staffing pattern.

What is your next question?

Crystal

Dear Crystal,

Thanks, and yes, I can already tell where I can use those levels to defuse some meetings.

It strikes me, suddenly, that you have not said much about tools up to now. I sense there is more here than you have let on, and that it is actually rather important.

regards, Alistair

*June 8: **Dear Alistair,***

Yes, tooling deserves special mention.

Clear does not *mandate* any particular tools, just as it does not mandate any specific technique. That is because the tools differ in various technologies.

That having been said, there is one tool that gets mentioned by almost every team I visit, including even the smallest teams of three people: the configuration management tool.

A startling number of teams I visit don't have any configuration management system in place at all. Those are not the successful teams I am documenting, however. The absence of configuration management doesn't imply the project will fail, but successful teams almost unanimously

name their configuration management tool as the single most important tool they possess.

Crystal

Dear Crystal,

Last hard question --- What would it mean for a group to say they are "in compliance" with this methodology?

Alistair

*June 9: **Dear Alistair,***

Compliance is a funny thing. On the one hand, we want enough compliance that the project is safe, and not so much that people get uptight about the details; guidance to teams interested in setting up their projects to succeed, work efficiently, and be "habitable" at the same time.

Compliance to methodologies is a graduated incline. Imagine, if you will, a *mesa*, a flat plateau above some plains. But imagine that instead of a cliff from the plains to the mesa, there is a sloped incline.

The mesa at the top is "obviously compliant." There is a certain amount of room for movement, all safely within both the letter and intention of the methodology.

The slope is the transition zone, the gray area of compliance, "Are we compliant? It's not the letter of the methodology, but is it in the intention?"

A methodology description should say something about the transition zone, otherwise there will be incessant

bickering about the letter versus the intention of the methodology. It should say something about, "Here's what it takes to say you are safe; here are some things you can do and still say you're doing (or sort of doing) it; do these and you are definitely out."

The dividing line on the sloping part will never be perfectly clear, but there should be some way of telling the shift between the three zones. This allows people to say, "Gee, we are doing everything just as described, and it's still not working", or "Oh, I see. We forgot *that* critical part. Let's try that and see if it all works better."

Accordingly, here is what it takes to say you are using Crystal Clear.
Because these are great, I start each with a smiley face :-)

1. **:-) You have short, rich communication paths.** You are sitting very close together, preferably one big room. You might be sitting in adjacent rooms, if there are really only 4-6 of you and you visit each other a lot. Recognize that the quality of communication drops the moment you have to pick up the phone or walk out the door.
2. **:-) You deliver increments monthly or every other month, absolutely no longer than 3 months apart.** You schedule and track code execution milestones, and not by document completion milestones.

3. **:-) You have a real user on the project, even if only part time.** Your user helps you construct your screen sketches and both validates and breaks your UI designs. You get the system reviewed by real users at least once per increment prior to delivering it.

4. **:-) You have a project mission statement.** You use one of a wide variety of requirements formats. You have a description of the system design (using one of the many description techniques you can invent).
5. **:-) You have a clear ownership model for your work products.** For each class, module or use case, you know who it is that can change or, more importantly, delete parts of it. The answer may be either a single person, anyone in a given sub-team, or even anyone on the team (this is the rule in XP projects). You should never have to have the whole team sit around the screen and ask, "Who put this in here? What is it doing there? Can we delete it?" If you have to do this, you know you have a breakdown in the ownership model.

Here are some things you can vary and still say you are doing Clear.
Because these are still OK, I start each with a neutral face :-|

1. :-| **You sit in adjacent rooms**, if there are really only 4-6 of you and you visit each other a lot. Recognize that the quality of communication drop the moment you have to pick up the phone or walk out the door.
2. :-| **You use the Scrum backlog technique or XP's planning game** for prioritizing and scheduling your work (Schwaber 2001, (Beck 2000). I personally like to use the *blitz-planning* (also known as the project-planning jam session) technique. The point I wish to make here is that Crystal Clear allows borrowing from other methodologies. Using XP's *planning game* does not mean you are not doing Crystal Clear. It only means that incorporating elements of other methodologies is a natural part of Crystal Clear.
3. :-| **There is only one iteration per delivery, but you arrange several user viewings**. When the users see the system, they will comment, which will generate new work to be put into the delivery cycle. That feedback is crucial to product success. "Iterations" provide closure and velocity information, but so do short delivery cycles. Nothing replaces the user viewings, though.
4. :-| You users cannot accept operational increments as often as you produce them, or they can't accept partial functionality. So, **you deliver tested, delivery-ready code to a "staging station"**, where the increments are added together over time until it is suitable to deliver them to the end users.
5. :-| **You describe your design using hypertext documents and technical memos**, and avoid UML drawings.; or, you describe classes using UML or one of the -OOM notations, and avoid responsibility statements. **You document your system using videos** of the designers walking through the design. There are lots of variations for how to describe the design (we aren't even close to figuring out the best way). You **don't like my use case format, and come up with your own** requirements format (possibly user stories or feature lists). With most of the work products, equivalent substitution is permitted as long as the intention of the item is preserved and the project's safety properties are not in jeopardy.

Here are some things that, if you are doing, you are definitely outside of Clear. Your rule set might work for you, but it is not Crystal Clear. I start each with a frowny face :-(

1. :-| **You work in different buildings** or different cities. This means your communications paths are long and thin, that you must communicate by phone, email or occasional visits. You cannot fulfill the requirements of Crystal Clear, which requires

that team members be able to initiate and carry person-to-person conversations many times a day with little effort. Longer distances raise the barrier to conversation, the form of communication is inferior, and you cannot rely on the common shared memory of the group. (I note, as you probably do, that working on different floors of the same building falls in a gray zone in these descriptions. I doubt you can make Clear work properly across floors. I think you can't make it work properly even down a corridor. But I leave myself open to disproof-by-example.)

2. **:-(*Your increments are longer than 4 months.*** Your feedback path is too long. You can no longer avoid "promissory notes" types of documents because you are not delivering often enough to demonstrate the progress you are making. Shorter increments is a critical success factor to your project (see *Surviving Object Oriented Projects*), no matter what methodology you are using.
3. **:-(*There is no user viewing prior to delivery.*** My experience is that this means the users will be given something that doesn't meet their needs, too late to make changes. Even projects doing everything else on the Crystal Clear list, who miss this, end up in trouble when the users finally do see the system.

4. **:-(*There are no real users available*** to your team. The hazard being addressed here is that the user requirements are coming from some sort of intermediary, a marketing representative or executive who promises they are telling you what the users "really" want. Research indicates that this sort of user input is unreliable, and direct links to real users are needed.

There is one part of Clear that is between optional and mandatory: use of a fully automated regression test harness and test suites for unit and acceptance tests. It gets the ? symbol.

1. **? *There are no automated test suites.*** Many systems have successfully shipped without automatic test suites. Really successful people, on the other hand, repeatedly say that the test suites give them speed and comfort in improving and debugging the system, and is high on their priority list. These days, speed in altering code is dependent on having automated test suites available, and more teams are using test-driven development (Beck 2002) to do even better.

Teams using automated tests find they can update the design of the code with great freedom: They retest their work at the push of a button, discovering whether they introduced a new bug or not.

Avoiding introducing new bugs when removing old ones is one of the best ways to improve shorten project delivery times.

Without regression testing, making changes to the system becomes tedious and unreliable, unreliable enough that one of two bad things happen: the team makes the change and introduces new errors, or they don't make the change, and allow the code to grow unnaturally complex ("crufty" is a word often used for such code).

Why is automated regression testing optional in Crystal Clear if it is so great?

The answer is that automated regression tests are not actually required to have a project success. It makes the team's life easier, it makes the code more malleable, but I have seen and participated in many projects that succeeded without using fully automated regression test suites.

Since Crystal Clear is proposed as a high-tolerance methodology, I have to leave automated regression testing as optional. At a personal level, I can say that the best programmers I have met use them, and recommend them highly . . . but not using them does not mean the project will fail, and so is not a basis for saying a group is not using Crystal Clear.

So there you have it. What it takes to be safely inside Clear, what you can vary, and what it takes to be outside.

Whew, I'm exhausted! How much more do you want from me?

Crystal

Dear Crystal,

I promise this is the last question - Refresh for me what Crystal Clear is trying to be and why it works?

Alistair

*June 10: **Dear Alistair,***

Summarizing . . .

Crystal Clear aims to be a simple and tolerant set of rules that puts the project into the safety zone. It is, if you will, *the least methodology that could possibly work*, because it contains so much tolerance.

The entire Crystal family has been described as consisting of

- Frequent Deliveries,
- Reflective Improvement, and
- Close Communication.

Crystal Clear pushes that last one up to Osmotic Communication.

Core to all of the Crystal family are a couple of ideas. One is that intermediate work products and "promissory notes" such as detailed requirements documents, design documents and project plans might not be eliminated, but they can be reduced to the extent that

- short, rich, informal communication paths are available to the team and
- working, tested software is delivered early and frequently.

Another is that the amount of detail needed in the requirements, design and planning documents varies with the project circumstances, specifically

- the extent of damage that might be caused by undetected defects and
- the frequency of personal collaboration.

Finally, the team reflects and adjusts its working conventions on the fly to fit

- the personalities on the team,
- the local working environment,
- the current assignment.

The reason Crystal Clear it works is that the people involved are professionals led by someone who is supposed to be a Level-3 developer. With rich, low-cost communications between them, the team members will diagnose most of their own problems. If they are given access to users and deliver running code every month or two, they get feedback on what they are doing. If they are not interrupted too often, they will make progress. In other words, the project is likely to work.

Note that Crystal Clear does not aim be the most productive, rigorous, or scalable methodology in addition to its goal of being simple, tolerant and successful. People are welcome to adopt more disciplined development habits as they choose, including a full-time user, pair programming, test-

driven development, and fully automated regression tests.

They are, however, unlikely to be able to drop any of the rules of Crystal Clear and still be safe.

Crystal Clear comes with an important truth-in-advertising clause: Eight people or less, collocated, not a life-critical system. Distributed teams need other forms of communication, larger teams need more coordination, and teams working on life-critical systems need more verification procedures in place. Those teams will have to find another methodology, or tune Crystal Clear upward to fit their needs.

I think this closes off our discussion.

Best wishes,

Crystal

Chapter 2

Applied (The Seven Properties)

Reading how Crystal Clear works raises two particular questions:

"What are these people concentrating on while they work?"

"Can we get farther into the safety zone?"

*This chapter describes seven properties set up by the best teams. Crystal Clear requires the first three. Better teams use the other four properties to get farther into the safety zone. All of the properties aside from **Osmotic Communication** apply to projects of all sizes.*

I only recently awoke to the realization that top consultants trade notes about the *properties* of a project rather than on the procedures followed. They inquire after the health of the project: "Is there a mission statement and a project plan? Do they deliver frequently? Are the sponsor and various expert users in close contact with the team?"

Consequently, and in a departure from the way in which a methodology is usually described, I ask Crystal Clear teams to target key *properties* for the project. "Doing Crystal Clear" becomes achieving the properties rather than following procedures. Two motives drive this shift from procedures to properties:

- The procedures may not produce the properties. Of the two, the properties are the more important.
- Other procedures than the ones I choose may produce the properties for your particular team.

The Crystal family focuses on the three properties **Frequent Delivery**, **Close Communication**, and **Reflective Improvement**¹⁰ because they should be found on all projects. Crystal Clear takes advantage of small team size and proximity to strengthen **Close Communication** into the more powerful **Osmotic Communication**. Aside from that one shift, experienced developers will notice that all the properties I outline in this chapter apply to every project, not just small-team projects.

By describing Crystal Clear as a set of properties, I hope to reach into the *feeling* of the project. Most methodology descriptions miss the critical feeling that separates a successful team from an unsuccessful one. The Crystal Clear team measures its condition by the team's mood and the communication patterns as much as by the rate of delivery. Naming the properties also provides the team with catch phrases to measure their situation by: "We haven't done any **Reflective Improvement** for a while..." "Can we get more **Easy access to expert users**." The property names themselves help people diagnose and discuss ways to fix their current situation.

¹⁰ Thanks to Jens Coldewey of Germany for pointing this out to me!

Property 1. Frequent Delivery

The single most important property of any project, large or small, agile or not, is that of delivering running, tested code to real users every few months. The advantages are so numerous that it is astonishing that any team doesn't do it:

- The sponsors get critical feedback on the rate of progress of the team.
- Users get a chance to discover whether their original request was for what they actually need and to get their discoveries fed back into development.
- Developers keep their focus, breaking deadlocks of indecision.
- The team gets to debug their development and deployment processes, and gets a morale boost through accomplishments.

All of these advantages come from one single property, **Frequent Delivery**. In my interviews, I have not seen any period longer than four months that still offers this safety. Two months is safer. Teams deploying to the web may deliver weekly.

Have you delivered running, tested and usable code at least twice to your user community in the last six months?

* * * *

Just what does "delivery" mean?

Sometimes it means that the software is deployed to the full set of users at the end of each iteration. This may be practical with web-deployed software or when the user group is relatively small.

When the users cannot accept software updates that often, the team finds itself in a quandary. If they deliver the system frequently, the user community will get annoyed with them. If they don't deliver frequently, they may miss a real problem with integration or deployment. They will encounter that problem when it is very late - at the moment of deploying the system.

The best strategy I know of in this situation is to find a friendly user who doesn't mind trying out the software, either as a courtesy or out of curiosity. Deploy to that one workstation. This allows the team to practice deployment and get useful feedback from at least one user.

If you cannot find a friendly user to deliver to, at least perform a full integration and test as though you were going to. This leaves only deployment with a potential flaw.

* * *

The terms *integration*, *iteration*, *user viewing*, and *release* get mixed together these days. They have different effects on development and should be considered separately.

Frequent Integration should be the norm, happening every hour, every day, or at the worst, every week. The better teams these days have continuously running automated build-and-test scripts, so there is never more than 30 minutes from a check-in until the **Automated Test** results are posted.

Simply performing a system integration doesn't constitute an *iteration*, since an integration is often performed after any single person or subteam completes as fragment of a programming assignment. The term *iteration* refers to the team completing a section of work, integrating the system, reporting the outcome up the management chain, doing their periodic **Reflective Improvement** (I wish), and very importantly, getting emotional closure on having completed the work. The closure following an iteration is important because it sets up an emotional rhythm, something that is important to us as human beings.

In principle, an iteration can be anywhere from an hour to a quarter. In practice, they are usually two weeks to two months long.

The end date of an iteration is usually considered immovable, a practice called "time boxing." People encounter a natural temptation to extend an iteration when the team falls behind. This has generally shown itself to be a bad strategy, as it leads to longer and longer extensions to the iteration, jeopardizing the schedule and demotivating the team. Many well-intentioned managers damage a team by extending the iteration indefinitely, robbing the team of the joy and celebration around completion.

A better strategy is to fix the end date, and have the team deliver whatever they have completed at the end of the time box. With this strategy, the team learns what it can complete in that amount of time, useful feedback to the project plan. It also supplies the team with an **Early Victory**.

Fixed-length iterations allow the team to measure their its speed of movement—the project's *velocity*. Fixed lengths iterations give that rhythm to the project that people describe as the project's "heartbeat."

Some people lock the requirements during an iteration or time box. This gives the team peace of mind while they develop, assuring them they will not have to change directions, but can complete *something* at least. I once encountered a group trying out XP where the Customer didn't want the trial to succeed: This Customer changed the requirements priorities every few days so that after several iterations the team still had not managed to complete any one user story. In such hostile environments, both the requirements locking and the peace-of-mind are critical Requirements locking is rarely needed in well-behaved environments.

The results of an iteration may or may not get released. Just how often the software should be sent out to real users is a topic for the whole team, including the sponsor, to deliberate. They may find it practical to deliver after every iteration, they may deliver every few iterations, or they may match deliveries to specific calendar dates.

Frequent Delivery is about delivering the software to users, not merely iterating. One nervous project team I visited had been iterating monthly for almost a year, but not yet delivered any release. The people were getting pretty nervous, because *the customer hadn't seen what they had been working on for the last year!* This constitutes a violation of **Frequent Delivery**.

If the team cannot deliver the system to the full user base every few months, *user viewings* become all the more critical. The team needs to arrange for users to visit the team and see the software in action, or at least one user to install and test the software. Failure to hold these user viewings easily correlates to end failure of the project, when the users finally, and too late, identify that the software does not meet their needs.

For the best effect, exercise both packaging and deployment. Install the system in as close to a real situation as possible.

Property 2. Reflective Improvement

The discovery that took me completely by surprise was that a project can reverse its fortunes from catastrophic failure to success if the team will get together, list what both is and isn't working, discuss what might work better, *and make those changes* in the next iteration. In other words, **reflect** and **improve**. The team does not have to spend a great deal of time doing this work – an hour every few weeks or month will do. Just the fact of taking time out of the helter-skelter of daily development to think about what could work better is already effective.

Did you get together at least once within the last three months for a half hour, hour or half day to compare notes, reflect, discuss your group's working habits and discover what speeds you up, what slows you down, and what you might be able to improve?

* * * *

The project that gave me the surprise was *Project Ingrid* (described in *Surviving OO Projects*). At the end of the first iteration -- which was supposed to be four months long, but they had extended -- they were far behind schedule, demoralized, and with what they recognized as an unacceptable design. It was what they did next that surprised me: They released 23 of the 24 client-side programmers to go back to their old jobs, hired 23 new people, changed the team and management structures, paid for several weeks of programmer training, and started over, requiring the new group to redo the work of the first team and make additional progress.

At the end of the second iteration, they were behind again schedule but had a design that would hold, and the team structure and programmers were functioning. They held another reflection workshop, made additional changes, and continued.

When I interviewed them, they were in their fourth iteration, ahead of schedule and content with their design and their work practices.

Since that interview, I have noticed that most of the projects I have visited got off to a rough start, or even catastrophe. This is so common that I have come to expect, almost even welcome it: From that first catastrophe come all sorts of new and important information about the project's working environment, which would be deadly, but hidden.

On *Project Winifred* we managed at the end of the first three-month delivery cycle what I called a "bubble-gum" release (the system was just barely held together by the software equivalent of bubble-gum). However, we delivered something every three months, getting better and better each time until we finally delivered the contracted function on time.

After each delivery, a few of us got together. We identified what wasn't working and discussed ways to fix it. We kept trying new strategies until we found ones that worked. **Frequent Delivery** and **Reflective Improvement** became critical success factors to us as they are to so many projects.

* * *

The people, the technology and the assignment change over the course of a project. The conventions the team uses need to change to match.

The people on the team are the best equipped to say what is best suited to their situation, which is why Crystal Clear leaves so many details unstated, but for the team to finalize. The **Reflective Improvement** mechanism allows them to make those adjustments.

Every few weeks, once a month, or twice per delivery cycle, the people get together in a *reflection workshop* or *iteration retrospective* to discuss how things are working. They note the conventions they will keep and the ones they want to alter for the next period, *and they post those two lists prominently for the team members to see while working in the next iteration*.

Whatever the frequency, meeting format, and technique used, successful teams hold this discussion periodically and try out new ideas. Teams may try, in various forms: pair programming, unit testing, test-driven-development, single-room versus multiple room seating, various levels of customer involvement, and even differing iteration lengths. These are all proper variations within Crystal Clear.

For people to say they are using Crystal Clear, it is not necessary that they continue to use the starter conventions. In fact, it is expected that they will try new ideas. In a Crystal user group meeting, people discuss what they had experimented with, how they felt about those experiments, and how they evolved their working conventions. One team may report moving the meetings from every two weeks to every month, another moving from the format I describe in the next chapter to a straight discussion of people's values while developing.

I like to use the *Reflection Workshop* described in the Techniques chapter. Norm Kerth's book, *Project Retrospectives*, presents an extended format, along with many activities to try within the workshop. The specifics of the workshop format aren't nearly as significant as the fact that the team is holding one.

Property 3. Osmotic Communication

Osmotic Communication means that information flows into the background hearing of members of the team, so that they pick up relevant information as though by osmosis. This is normally accomplished by seating them in the same room. Then, when one person asks a question, others in the room can either tune in or tune out, contributing to the discussion or continuing with their work. Several people have related their experience of it much as this person did:

We had four people doing pair programming. The boss walked in and asked my partner a question. I started answering it, but gave the wrong name of a module. Nancy, programming with Neil, corrected me, without Neil ever noticing that she had spoken or that a question had been asked.

When **Osmotic Communication** is in place, questions and answers flow naturally and with surprisingly little disturbance among the team.

Osmotic Communication and **Frequent Delivery** facilitate such rapid and rich feedback that the project can operate with very little other structure. This is why these two properties are the first two listed.

Does it take you 30 seconds or less to get your question to the eyes or ears of the person who might have the answer? Do you overhear something relevant from a conversation among other team members at least every few days?

* * * *

Osmotic Communication is the more powerful version that small projects can attain of **Close Communication**, a property core to the entire Crystal family. **Osmotic Communication** makes the cost of communications low and the feedback rate high, so that errors are corrected extremely quickly and knowledge is disseminated fast. People learn the project priorities and who holds what information. They pick up new programming, design, testing and tool handling tricks. They catch and correct small errors before they grow into larger ones.

Although **Osmotic Communication** is valuable for larger projects, it is, of course, increasingly difficult to attain as the team size grows.

It is hard to simulate **Osmotic Communication** without having the people in the same room, adjacent rooms of two or three people each confers many of the benefits. Herring (1999) reported the use of high-speed intranet with web cameras, mikes, and chat sessions to trade questions and code, to simulate the single room to (some) extent. With good technology, teams can get achieve some approximation of **Close Communication** for some purposes, but I have yet to see **Osmotic Communication** achieved with other than physical proximity between team members.

* * *

Discussion of **Osmotic Communication** inevitably leads to discussion about office layout and office furniture.

Crystal Clear needs people be very close to each other so that they overhear useful information and get questions answered quickly. The obvious way to do this is to put everyone into a single room ("war room"), repeatedly shown as being very effective (Radical Colocation ref???).



Figure 2-1. Osmotic Communication (Thanks to Tomax)

Many people who have private offices resist moving into a group space. However, you can sometimes turn lemons into lemonade (so to speak) with this move:

Lise was informed by her management that her department would have to reduce the number of square feet they used. This meant giving up private offices. She suggested that her people work together and design their own office spaces, three to five people in a combined area. The groups put fewer square feet around each work table so they could allocate space for additional areas with chairs, sofa or, in some cases, their own meeting rooms.

The following pictures show what one group came up with. Note that although they had fewer square feet person than before, they ended up with longer sight lines and a conversation area with soft chairs.

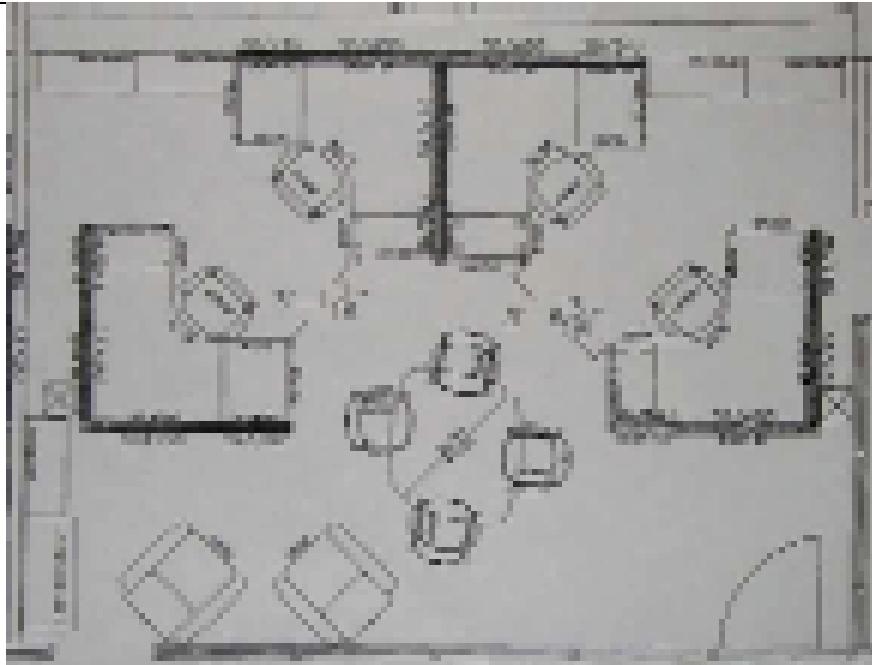


Figure 2-2. Floor plan (Thanks to Lise Hvatum)



Figure 2-3. Photo of same office (Thanks to Lise Hvatum)

Figure 2-4 shows the small meeting room one group put on the side of their shared office. They used it to talk without disturbing whomever was still programming, and also to leave their design notes and plans up on the wall.



Figure 2-4. Group work room attached to shared office (Thanks to Lise Hvatum)

Lise's group used the usual office furniture: concave, designed to have the fat CRT back into the corner. This sort of table presents a disadvantage to an agile development team, because it is hard for a second or third person to see the screen. The war room in Figure 2-1 may look less glamorous, but there is a utility in those ugly tables: People can congregate around a screen, pairs of people can work together easily. It is for this reason that agile development teams prefer straight tables, or even better, tables that bulge outward toward the typist.

If you set up a war-room work area, be sure to arrange another place for people to go to unwind and do their private email. This allows people to focus when they step into the common area, and find a bit of relief from the pressure by stepping out. Such an arrangement is referred to as a "caves and common" arrangement.



Figure 2-5. Group discussion area (Thanks to Darin Cummins)

One project team got permission to set up a common discussion area with soft chairs and sofa (Figure 2-5). On the wall in front of the chairs is the ever-present whiteboard with whiteboard capture device. This is where the team adjourned to hold their group design discussions, iteration planning meetings and reflection workshops.

Agile Software Development (Cockburn 2003) contains additional information on "convection currents" of information flow within a group, **Osmotic Communication**, the value of colocation, and examples of office layout.

* * *

Osmotic Communication generates its own hazards, most commonly noise and a flow of questions to the team's most expert developer. People usually self-regulate here, request less idle chit-chat or more respect for think time.

Attempting to "protect" the lead designer with a private office usually backfires. That person really needs to be sitting in the middle of the development team. The lead designer is often the technology expert, a domain expert, and the best programmer, and so is necessarily in high demand. When she is taken away, the younger developers miss the chance to develop good development habits, miss growing in the domain and the technology, and make mistakes that otherwise would get caught very quickly. The cost to the project ends up being greater than the benefit of quiet time to the Lead Designer. Having the lead designer in the same room as the rest of the team is a strategy called **Expert in Earshot**, described in (Cockburn URL-EiE). **Expert in Earshot** is a special use of **Osmotic Communication**.

Even the best success property is unsuitable under certain circumstances. **Osmotic Communication** is no exception. If the lead designer gets so overloaded and so frequently interrupted as to be unable make progress on anything, she needs a work place with no interruptions at all and extremely limited communications with the team, a **Cone of Silence**, I call it. Many lead designers use the hours from 6 p.m. to 2 a.m. as their **Cone of Silence**, but it is better for all involved if an acceptable **Cone of Silence** can be set up within normal working hours. The **Cone of Silence** strategy is described in detail in (Cockburn URL-CoS).

Property 4. Personal Safety

Personal Safety is being able to speak when something is bothering you, without fear of reprisal. It may involve telling the manager that the schedule is unrealistic, a colleague that her design needs improvement, or even letting a colleague know that she needs to take a shower more often. **Personal Safety** is important because with it, the team can discover and repair its weaknesses. Without it, people won't speak up, and the weaknesses will continue to damage the team.

Personal Safety is an early step toward **Trust**. Trust, which involves giving someone else power over oneself, with accompanying risk of personal damage, is the extent to which one is comfortable with handing that person the power. Some people trust others by default, and wait to be hurt before withdrawing the trust. Others are disinclined to trust others, and wait until they see evidence that they won't be hurt before they give the trust. Presence of trust is positively correlated with team performance (Costa 2002).

The different ways in which one can be hurt lead to different forms of trust and distrust (Tyler 1996). A person lacking open honesty might lie or conceal. One who lacks congruence in actions will be inconsistent. A person lacking either competence or reliability will fail to complete assignments. A person lacking concern for others may act to damage them, including giving away sensitive information.

Accepting exposure to these varied potential damages uses different forms of trust. It is neither realistic nor necessary to ask everyone on a project to trust each other in all of them. It is important that people can speak and act freely -- they need to trust each other with respect to damaging actions and betrayal.

When a person sees that others won't betray or damage her based on the information she reveals, she will reveal information more freely, which will speed the project. Therefore, **Personal Safety** is the critical property to attain.

Can you tell your boss you mis-estimated by more than 50%, or that you just received a tempting job offer? Can you disagree with her about the schedule in a team meeting? Can people end long debates about each others' designs with friendly disagreement?

* * * *

Establishing trust involves being in a situation where one of those dangers is present, and seeing that the other people do not hurt you. In other words, to build trust, there must be exposure.

Three particular exposures are relevant in software development:

- revealing one's ignorance,
- revealing a mistake, and

-
- revealing one's incapability on an assignment.

Skillful leaders expose their team members (and themselves!) to these situations early, and then demonstrate with speed and authenticity that not only will damage not accrue, but that the leader and the team as a whole will act to support the person.

One project leader¹¹ told me that when a new person joined her team, she would visit that person privately to discuss his work and progress, and wait for the inevitable moment when he had to admit he hadn't done or didn't know something.

This was the crucial moment to her, because until he revealed a weakness, she couldn't demonstrate to him that she would cover for him or get him assistance. She knew she was not going to get both reliable information and full cooperation from him until he understood properly that when he revealed a weakness or mistake, he would actually get assistance. She said that some people got the message after her first visit, while others needed several demonstrations before opening up.

Another project leader told of building cohesion and safety in the team by having the group work together to solve a difficult problem they were facing. In solving the problem together, they learned several things:

- First, they wouldn't get hurt if they admitted ignorance, even in their own area.
- Second, they learned how interpret each others' mannerisms as non-threatening, even in heavy argumentation.
- Finally, they learned that together they could solve things they couldn't solve alone.

Trust is enhanced with **Frequent Delivery**. When the software is delivered, people recognize who did their share of the work and who shirked, who told the truth, who damaged or protected whom, and who, despite their superficial manners, could be trusted along which dimensions. With **Personal Safety**, they speak from their heart during the **Reflective Improvement** sessions.

* * *

Personal Safety goes hand in hand with *amicability*, the willingness to listen with good will. The project suffers when any one person on the team stops listening with good will, or loses the inclination to pass along possibly important information. In addition to personal skill, a project's forward progress relies only on the speed of movement of information across people ("meme-meters per minute", if you will).

Usually one person on the team sets the lead in amicability. On a larger project, it is often crucially the project manager. On a Crystal Clear project, it can be anyone on the team. Unless there is a specific reason countering it, amicability spreads quickly and makes the team more comfortable in exchanging information quickly. **Personal Safety**

¹¹ Thanks to Victoria Einarsson in Sweden.

and amicability together help lead to **Collaboration across Organizational Boundaries**, the establishment of global lifelines for the project. I set amicability as significant management element on a project, partly as evidence for **Personal Safety**.

Once **Personal Safety** and amicability are established, a useful, playful dynamic may emerge. People may wage competition with each other. They may argue loudly, even to the verge of fighting, without taking it personally. In the case where someone does take it personally, they sort it out and set things straight again.

Be careful, though, not to confuse **Personal Safety** with politeness. Some teams appear to have **Personal Safety** in place, but actually are just being polite because they are unwilling to show disagreement.¹² Covering their disagreements with politeness and conciliation, they don't detect and repair mistakes that are present. This damages the project in the end, as in the case of over-amicability described in (Cockburn 2002, pp. ???).

There is a fair amount of literature on the subject of trust, some of which you may find applicable to your situation. Read more in Hohmann (1997, pp. 250??), Karmen (1996), Costa, and Adams (URL).

¹² Thanks to Kay Johanssen for this distinction.

Property 5. Focus

Focus is first knowing what to work on, and then having time and peace of mind to work on it. Knowing what to work on comes from communication about goal direction and priorities, typically from the *Executive Sponsor*. Time and peace of mind come from an environment where people are not taken away from their task to work on other, incompatible things.

Do all the people know what their top two priority items to work on are? Are they guaranteed at least two days in a row and two uninterrupted hours each day to work on them?

* * * *

Even with the best of intentions, developers will work on things that only randomly bring business value if they are not told what will provide business value. It is the job of the *Executive Sponsor*, starting from the project *chartering* activity and running continuously throughout the project, to make it clear to everyone where the organization's priorities lie.

The Vice President of a 50-person company sat down one night, prioritized the 70 pending company initiatives, and announced the results to her managers the following day. She went around to each developer and made sure they each knew the top two items for them, individually.

One *Lead Designer* I met kept the project's mission statement and priorities posted on the wall and referred to them regularly.

Just knowing what is important isn't enough. Developers regularly report that meetings, requests to give demos, and demands to fix run-time bugs keep them from completing their work. It quite typically takes a person about 20 minutes and considerable mental energy to regain their train of thought after one of these interruptions. When the interruptions happen three or four times a day, it is not uncommon for the person to simply idle between interruptions, feeling that it is not worth the energy to get deeply into a train of thought when the next distraction will just show up in the middle of it.

People asked to work on two or three projects at the same time regularly report that they are unable to make progress on any one project. It seems to take an hour and a half for a person to regain her train of thought after working on a different project.

Among the experienced project managers that I interview, the consensus is that about one and a half projects is the most that a person can be on and stay effective. By the time a third project is added, the developer becomes ineffective on all three. Contrast this with the inexperienced managers who, underestimating the cost of switching between projects, assign developers to work on three to five projects at the

same time. I encountered one developer assigned to 17 projects simultaneously! You can imagine that he barely had time to report at the various meetings his ongoing lack of progress on all fronts.

The repair is simple, though uncomfortable. The *Sponsor* makes it clear which projects and work items are top priority for each person, and arranges for the top two items to be distinctly higher in priority than all the rest.

The team should then adopt conventions that provide **Focus** time for the team members. One such is the convention that once a person starts working on a project, she is guaranteed at least two full days before having to switch to a second project. This allows for some project switching, while guaranteeing the person enough time to make actual progress instead of using all the time just to get back up to speed on each project before leaving it again.

The next convention to adopt may be to localize distracting interruptions. My experience is that it is generally impractical to bottle up interruptions to something so neat and tidy as "mornings only" or "between 1 and 3 in the afternoon." It is in the nature of interruptions to come sporadically and with high priority. What the team can do is to create a two-hour time window during which interruptions are blocked. There are very few interruptions that can't wait for two hours. Some teams use from 10:00 to noon as a time when meetings, phone calls and demos are not allowed.

With two hours of guaranteed focus time each day, and two days in a row on the same project, a developer who otherwise is being driven to distraction may get four full hours of work done in a week. One developer who adopted these reported after a few weeks that he had gotten more done in those few weeks than in the several months before that.

Property 6. Easy Access to Expert Users

Continued access to expert user(s) provides the team with

- a place to deploy and test the **Frequent Deliveries**,
- rapid feedback on the quality of their finished product,
- rapid feedback on their design decisions, and
- up-to-date requirements.

Researchers Keil and Carmel published results showing how critical it is to have direct links to expert users (Keil 95). Surveying managers who had worked both with and without easy access to real users, they write:

" . . . in 11 of the 14 paired cases, the more successful project involved a greater number of links than the less successful project. . . . This difference was found to be statistically significant in a paired t-test ($p<0.01$)."

Their research led them to a specific recommendation: "Reduce Reliance on Indirect Links." In other words, get real access to real users.

Does it take less than three days, on the average, from when you come up with a question about system usage to when an expert user answers the question? Can you get the answer in a few hours?

* * * *

All very nice, but how many users, and how much time?

Even one hour a week of access to a real and expert user is immensely valuable. The more hours each week that an expert user is available to a team, the more advantage they can take of that proximity. The first hour, however, is the most crucial.

The other thing that is important is the length of time until a question gets answered. If a question won't be answered for another three days, the programmers are likely to put into the code their best current guess, and may forget to recheck their decision when they are with the users again. Therefore, they should have telephone access to the expert user during the week.

Here are the three user access methods I hear about most often:

- *Weekly or semi-weekly user meetings with additional phone calls.* You may find that the user loads the team with information in the first weeks. Over time, the developers need less time from the user(s), as they develop code. Eventually, a natural rhythm forms, as the user provides new requirements information and also reviews draft software. This natural rhythm might involve one, two or three hours a week per expert user. If you add a few phone calls during the week, then questions get answered quickly enough to keep the development team from going off in a false direction.

- *One or more experienced users directly on the development team.* This is only rarely possible, but don't discount it. I periodically find a team located inside the user community, or is in some way collocated with an expert user.
- *Send the developers to become trainee users for a period.* Odd though this may sound, some development teams send the developers to either shadow users or become apprentice users themselves. While I don't have a very large base of stories to draw from, I have not yet heard a negative story related to using this strategy. The developers return with an appreciation and respect for the users, and an appreciation for the way their new software can change the working lives of the users.

Keil and Carmel name additional user links, including facilitated teams, user-interface prototyping, interviews, tests, bulletin boards, usability labs, observational study, and focus groups. In a quick search on the internet, I turned up a number of companies that specialize in finding subjects and testing software with real users.

I distinguish between the *expert user* and the *business expert*, because they are often different people. The business expert knows the business policies, including which are fixed, which are likely to change and the dependencies between them. Users generally don't know this information. On the other hand, the expert user knows which operations are common and which are rare, what shortcuts are needed, what information doesn't really have to be entered, and what information needs to be visible at the same time. The business expert won't know this information, since it comes only from continuous daily operation.

The development team will contain a *Business Expert* (see Roles, in Chapter 5). That person may be the sponsor, or the expert user, or it may be the Lead Designer. Such a person is almost always available to a project, and so I don't fuss about it so much. The *Expert User*, on the other hand, is usually missing, to the detriment of the project, which is why I fuss about it so much here. **Easy Access to Expert Users** provides a safety net for the team as well as being a competitive advantage. It is likely to be a critical success factor for a small team.

* * * *

"OK, we've got the users – now what do we do with them?"

You need to know what they want, what their sponsors are willing to pay for, where their fast and rare-but-significant usage patterns lie, whether you have overlooked something critical. You need the users before, during, and after design.

Before you get too far into designing the system, you need to identify the user roles that the sponsors consider the most important people to fit the application. These are the *focal roles*. The system will present different "personalities" (fast and efficient, for example, or warm and friendly) to each different role. The designers will accentuate

one personality over others, and you want to make sure they accentuate the most important one(s).

The technique described in the next chapter, *Essential Interaction Design*, is one way to identify the focal roles and personalities to develop. The attraction of this workshop technique is that you can gather the information in just a few days.

During design, you will need answers to many small questions. For this you need **Easy Access to Expert Users** on an ongoing basis as described in this section.

After design, when you think you are done, you need users again, to evaluate your results. If the system will go to a few, local users, simply invite them in for a test drive. If, on the other hand, you have a large number of geographically dispersed users, then the cost of evaluation is greater. I don't know of any special efficiencies for this situation. Techniques for usability evaluation have been described for decades (customer focus groups and usability samples being the prime examples). I recently discovered there is an entire industry for usability testing¹³.

Before I leave this property, I ask you to read again the last paragraphs of the **Frequent Delivery**, in which I describe the troubles arising from not arranging for *real* user feedback. Even teams that do every other practice in agile development find themselves facing catastrophic bad news at the end of the project if they neglect such feedback during the project.

¹³ On Google, for examples, see Computers > Human-Computer Interaction > Companies and Consultants > Usability Testing.

Property 7. Technical Environment with Automated Tests, Configuration Management & Frequent Integration

The elements I highlight in this property are such well-established core elements that it is embarrassing to have to mention them at all. Let us consider them one at a time and all together.

Automated Testing. Teams do deliver successfully using manual tests, so this can't be considered a *critical* success factor. However, every programmer I've interviewed who once moved to automated tests swore *never to work without them again*. I find this nothing short of astonishing.

Their reason has to do with improved quality-of-life. During the week, they revise sections of code knowing they can quickly check that they hadn't inadvertently broken something along the way. When they get code working on Friday, they go home knowing that they will be able on Monday to detect whether anyone had broken it over the weekend – they simply rerun the tests on Monday morning. The tests give them freedom of movement during the day and peace of mind at night.

Configuration Management. The configuration management system allows people to check in their work asynchronously, back changes out, wrap up a particular configuration for release, and roll back to that configuration later on when trouble arises. It lets the developers develop their code both *separately* and *together*. It is steadily cited by teams as their most critical non-compiler tool.

Frequent Integration. Many teams integrate the system multiple times a day. If they can't manage that, they do it daily, or in the worst case, every other day. The more frequently they integrate, the more quickly they detect mistakes, the fewer additional errors that pile up, the fresher their thoughts, and the smaller the region of code that has to be searched for the miscommunication.

The best teams combine all three into **Continuous Integration-with-Test**. They catch integration-level errors within minutes.

Can you run the system tests to completion without having to be physically present?

Do all your developers check their code into the configuration management system?

Do they put in a useful note about it as they check it in?

Is the system integrated at least twice a week?

* * * *

How frequent should **Frequent Integration** be? There is no fixed answer to this any more than to the question of how long a development iteration should be.

One lead designer reported to me that he was unable to convince anyone on his team to run the build more than three times a week. While he did not find this comfortable, it worked for that project. The team used one-month long iterations, had **Osmotic Communications**, **Reflective Improvement**, **Configuration Management** and some **Automated Testing** in place. Having those properties in place made the frequency of their **Frequent Integration** less critical.

The most advanced teams use a build-and-test machine such as Cruise Control¹⁴ to integrate and test nonstop (note: having this machine running is not yet sufficient . . . the developers have to actually check in their code to the main line code base multiple times a day!). The machine posts the test results to a web page that team members leave open on their screens at all times. One internationally distributed development team (obviously not using Crystal Clear!) reports that this use of Cruise Control allows the developers keep abreast of the changing code base, which to some extent mitigates their being in different time zones.

Experiment with different integration frequency, and find the pace that works for your team. Include this topic as part of your **Reflective Improvement**. For more on configuration management, I refer you to *Configuration Management Principles and Practice* (Hass 2002) *Configuration Management Patterns* (Appleton and Berczuk 2002) and *Pragmatic Version Control using CVS* by the "Pragmatic Programmers" (Thomas 2003). You may need to hire a consultant to come in for a few days, help set up the configuration management system and tutor the team on how to use it.

* * *

Automated Testing means that the person can start the tests running, go away, not having to intervene in or look at the screens, and then come back to find the test results waiting. No human eyes and no fingers are needed in the process. Each person's test suites can be combined into a very large one that can, if needed, be run over the weekend (still needing no human eyes or fingers).

Three questions immediately arise about **Automated Testing**:

- At what level should they be written?
- How automated do they have to be?
- How quickly should they run?

Besides *usability tests*, which are best performed by people outside the project¹⁵, I find three levels of tests hotly discussed:

¹⁴ www.CruiseControl.??

¹⁵ Google even has a category for it: Computers > Human-Computer Interaction > Companies and Consultants > Usability Testing.

-
- *Customer-oriented acceptance tests running in front of the GUI and relying on mouse and keyboard movements;*
 - *Customer-oriented acceptance tests running just behind the GUI, testing the actions of the system without needing a mouse or keyboard simulator; and*
 - *Programmer-oriented function, class and module tests (commonly called unit tests).*

The **Automated Tests** that my interviewees are so enthusiastic over are from the latter two of those categories. Automating unit tests allow the programmers to check that their code hasn't accidentally broken out from under them while they are adding new code or improving old code (*refactoring*). The GUI-less acceptance tests do the same for the integrated system, and are stable over many changes in the system's internal design. Although GUI-less acceptance tests are highly recommended, I rarely find teams using them, for the reason that they require the system architecture to carefully separate the GUI from the function. This is a separation that has been recommended for decades, but few teams manage.

Automated GUI-driven system tests are not in the highly-recommended short list because they are costly to automate and must be rebuilt with every change of the GUI. This difficulty makes it all the more important that the development team creates an architecture that supports GUI-less acceptance tests.

A programmer's unit tests need to execute in seconds, not minutes. Running that fast, she will not lose her concentration while they run, which means that she will actually bother to run the tests as she works. If the tests take several minutes to run, she is unlikely to rerun the tests after typing in just a few lines of new code or moving two lines of code to a new function or class.

Tests may take longer when she checks her code into the **Configuration Management** system. At this point, she has completed a sequence of design actions, and can afford to walk away for a few minutes while the tests run.

The acceptance tests can take a long time to run, if needed. I write this sentence advisedly: the reason the tests run a long time should be because there are so many tests or there is a complicated timing sequence involved, not because the test harness is sloppy. Once again, if the tests run quickly, they will get run more often. For some systems, though, the acceptance tests do need to run over the weekend.

Crystal Clear does not mandate when the tests get written. Traditionally, programmers and testers write the tests after the code is written. Also traditionally, they don't have much energy to write tests after they write code. Partially for this reason, more and more developers are adopting test-driven development (Beck 200???).

The best way I know to get started with **Automated Testing** is to download a language-specific copy of the X-unit test framework (where X is replaced by the language name), invented by Kent Beck. There is *JUnit* for Java programmers, *CppUnit* for C++ programmers, and so on for Visual Basic, Scheme, C, and even PHP. Then get

Test-Driven Design (Beck 2002???) or Astel??((Astel 2003?f?) and work through the examples. A web search will turn up more resources on *X-unit*.

Both *httpUnit* and Ward Cunningham's FIT (Framework for Integrated Tests) help with GUI-less acceptance tests. The former is for testing HTML streams of web-based systems, the latter to allow the business expert to create her own test suites without needing to know about programming. Robert Martin integrated FIT with Ward's wiki technology to create FITnesse¹⁶. Many teams use spreadsheets to allow the business experts to easily type in scenario data for these system-function tests.

There are, sadly, no good books on designing the system for easy GUI-less acceptance testing (Plumtree 2004???). The Mac made the idea of scriptable interfaces mainstream for a short while¹⁷, and scripting is standard with Microsoft Office. In general, however, the practice has submerged and is used by a relatively small number of outstanding developers. The few people I know who could write these books are too busy programming.

* * *

I end this section with a small testimonial to test-driven development that I hope will sway one or two readers. Thanks to David Brady for this note:

Yesterday I wrote a function that takes a variable argument, like printf(). That function decomposes the list arguments, and drops the whole mess onto a function pointer. The pointer points to a function on either the console message sink object or a kernel-side memory buffer message sink object. (This is just basic inheritance, but it's all gooky because I'm writing it in C.)

Anyway, in the past I would expect a problem of that complexity to stall me for an indefinite amount of time while I tried to debug all the bizarre and fascinating things that can go wrong with a setup like that.

It took me less than an hour to write the test and the code using test-first.

My test was pretty simple, but coming up with it was probably the hardest part of the whole process. I finally decided that if my function returned the correct number of characters written (same as printf), that I would infer that the function was working.

With the test in place, I had an incredible amount of focus. I knew what I had to make the code do, and there was no need to wander around aimlessly in the code trying to support every possible case. No, it was just "get this test to run". When I had the test running, I was surprised to realize that I was indeed finished. There wasn't anything extra to add; I was actually done!

¹⁶ [www.fit.org???](http://www.fit.org/) and [www.fitnesse.org???](http://www.fitnesse.org/) respectively.

¹⁷ See http://www.mactech.com/articles/develop/issue_24/acording.html for an article by Cal Simone.

I usually cut 350-400 lines of production-grade code on a good day. Yesterday I didn't feel like I had a particularly good day, but I cut 184 test LOC and 529 production LOC, pLOC that I *know* works, because the tests tell me so, pLOC that includes one of the top-10 trickiest things I've ever done in C (that went from "no idea" to "fully functional" in under 60 minutes).

Wow. I'm sold.

Test infection. Give it a warm, damp place to start, and it'll do the rest....

David Brady

Evidence: Collaboration across Organizational Boundaries

There is a side-effect from attending to **Personal Safety**, *amicability* within the team, and **Easy Access to Expert Users**: it becomes natural to include other stakeholders into the project, as well.

Géry Derbier, working with the French postal service (La Poste) to build software to run a new facility to handle all the mail going into and out of northern France, reported on his use of Crystal. With 25 people, his was a project in the Crystal Yellow category. However, he knew the principles of the Crystal methodologies family, particularly the "stretch to fit" principle, and therefore chose to extend Crystal Clear to his larger setting wherever possible.

We discussed his project, and at one point covered their project's linkage to the integration testing team located 30 km away and to the business and usage expert working for La Poste. I asked questions of the sort: "How often did that person visit the team? How did he feel about that? How did his manager feel about his coming over so often?" Géry's answers were, for both external groups: "One day a week; comfortable; happy to be involved so early."

After our discussion, I realized that Géry had built the additional safety into his project of **Collaboration Across Organizational Boundaries**. His project was happily linked into both the customer and integration environments with a colleague on each end. La Poste's contract measured and paid according to integrated test results every few months (the **Frequent Delivery**). The La Post executives got software delivered in growing increments and paid accordingly. Géry's bosses, who had no previous experience with incremental delivery, were happy about this also, since they saw regular delivery turn into regular payments. Géry had a support structure on all sides.

Collaboration Across Organizational Boundaries is not a given result on any project. It results from working with honesty amicability and integrity within and outside the team. It is hard to achieve if the team does not itself have **Personal Safety** and to a lesser extent, **Frequent Delivery**. I consider the presence of good collaboration across organizational boundaries as partial evidence that some of the top-seven safety properties are being achieved.

Reflection on the Properties

I don't believe that any prescribed procedures exists that can assure that projects land in the safety zone every time. Nor, with the exception of incremental development, do I show up on a project with any particular set of rules in hand, even though I have my favorites. This is why Crystal Clear is built around critical properties instead of specification of procedures.

A Crystal team works to set the seven properties into place, using whatever group conventions, techniques and standards fit their situation. The conventions may vary by project and by month. New techniques get invented with each new technology (and usually go out of style again a few years later). These seven properties, on the other hand, have been applied on good projects for decades.

My intention with Crystal is to not invade the natural workings of individuals on the project where possible, and to allow the most possible variation across different teams, while still getting those diverse projects into the safety zone. To allow variation, I must remove constraints. Removing constraints means finding broader mechanisms that provide a safety net. The ones I choose to rely on are these:

- People are by nature good at looking around and communicating.
- They take initiative when provided with information.
- They do better in an environment that is safe with respect to personal emotional safety, and particularly freedom from personal attacks.
- They do their best work if they can satisfy their need for contribution, accomplishment, and pride-in-work.

The Crystal Clear safety net is built on those things. **Personal Safety** gives people the personal courage to share whatever they discover. **Osmotic Communication** gives them the greatest chance to discover important information from each other, and does so with very low communication cost. **Reflective Improvement** gives them a channel to apply feedback to their working process. **Easy Access to Expert Users** gives them the opportunity to quickly discover relevant information from the user(s). **Frequent Delivery** creates feedback to the system's requirements and the development process. The technical development environment including **Automated Tests, Configuration Management & Frequent Integration** allows people to safely make changes to the system. synchronize the multiple minds that are in motion at the same time, and get feedback on the system's intermediate stages quickly. **Focus** allows the team to spend their energy well on the most important things.

Ron Jeffries once characterized Crystal Clear as, "Bring a few developers together in peace, love and harmony, shipping code every other month, and good software will emerge." He is close.

* * *

You should be asking at this point, "But what is special in all this about small projects? Shouldn't *all* project teams set these properties in place?"

The answer -- with two side notes -- is, "Of course." The properties that make up a small-team project successful *should* be very similar to making any project successful, but optimized for the small-project situation.

The first note is that the properties are easier to reach on a small project: **Personal Safety** is easier, since the people interact with each other more often and come to know each other sooner. The feedback loops are much smaller, and the rest of the properties follow accordingly.

The second note is that **Osmotic Communication**, which lives from background hearing and communication along lines-of-sight, really only works with small teams. Larger team will set up **Osmotic Communication** within subteams and **Close Communication** across subteams.

Chapter 3

In Practice (Strategies & Techniques)

Crystal Clear does not require any one strategy or technique. It is good to have a set in hand to get started, however. This chapter presents a few of the less-well documented and more significant one used by modern agile development teams.

Many useful strategies and techniques have been named in the last decade, from XP's "planning game" to Dave Thomas and Andy Hunt's "tracer bullets." The books by Dave Thomas and Andy Hunt (*Pragmatic Programmer*), Kent Beck (*Extreme Programming Explained*), and Martin Fowler (*Refactoring*) are particularly rich in ideas.

There are some strategies and techniques not pointed out in those books that are useful to the Crystal Clear project team, particularly in guiding their way through the early months of the project and getting to the first of their **Frequent Deliveries**. I include a few here that make a good starter set, are recommended by experienced developers, few people seem to know, are simple, and most of all, are useful.

The Strategies

The strategies I have selected to outline are:

1. *Exploratory 360°*, part of project chartering,
2. *Early Victory*, a project management strategy,
3. *Walking Skeleton* and
4. *Incremental Rearchitecture*, related strategies for prioritizing work in the early iterations,
5. *Information Radiators*, a strategy for communication.

Strategy 1. Exploratory 360°

At the start of a new project, usually during the *chartering* activity, the team needs to establish that the project is both meaningful and they can deliver it using the intended technology. They look around them in all directions, sampling the project's

- business value,
- requirements,
- domain model,
- technology plans,
- project plan,
- team makeup,
- process or methodology (or working conventions).

(They may check other aspects of the project, but these are the usual.)

The entire *Exploratory 360°* for a Crystal Clear project takes a few days up to a week or two if some new and peculiar technology is to be used. Based on what they learn, they decide whether it makes sense to proceed or not.

* * *

Business-value sampling consists of capturing, with key stakeholders, what the system should do for its users and their organization(s). This should result in the names of the key use cases for the system, along with the focal roles the system should serve, the personalities and functions it should present to the world.

Requirements sampling consists of low-precision use cases that show what the system must do, and with what other people and systems it will have to interact. Often that drafting exercise turns up interfaces between organizations or technology systems that had not formerly been identified.

Concurrently or from the use case drafts, the developers sample the *domain-model*. This sample serves to highlight the key concepts they will be working with, the core of the business, the programming and discursive vocabulary. It also helps the team to estimate the size and difficulty of the problem at hand.

The developers create a *technology* sampling, running a few experiments with the technology. Ward Cunningham and Kent Beck call these *Spikes* (see <http://c2.com/cgi/wiki?SpikeSolution>). The assignment is to ask: Can we really connect these technologies? Will they withstand the intended loads? Can our developers master the technologies in time? The programmers perform the smallest programming assignment needed to make it clear the project is not running down a blind alley. The experiments establish the technical plausibility of the project.

The team creates a coarse-grained *project plan*, possibly from the project map or a set of stories and releases. It might be done using the blitz-planning technique described later in this chapter, or XP's planning game. This plan is reviewed by the *Lead Designer*,

the *Executive Sponsor* and the *Ambassador User* to make sure the project is delivering suitable business value for suitable expense in a suitable time period.

Finally, the developers discuss their process, either in a *Reflection Workshop* or a *Process Miniature*.

* * *

Here is the story of one project that failed three elements of the *Exploratory 360°*:

During the technology sample, the developers found, much to their surprise, that they could not connect the organization's email system to their intranet and browser system. It was not clear how they could actually deliver the software services they had envisioned.

The project planning sample showed the cost to be about three times higher than the *Executive Sponsor* was interested in spending.

The business value sample showed that the organization should not allocate very many developer resources to this problem; it would be better outsourced (or simply bought), and the development team allocated to a more significant business problem.

You would think that canceling or outsourcing such a project would be obvious to all involved. However, the developers were keen to experiment with the new technology, and therefore kept the project alive, under the guise that this project would serve as a good learning vehicle. Fortunately, the executive sponsors paid attention to the *Exploratory 360°* results, stopped the programmers, outsourced the project, and put these key developers on a project of much greater value to the organization (which, I am happy to add, they enjoyed much more).

Strategy 2. Early Victory

Winning is a force that binds a team and contributes to the self-confidence of its members. Sociologist Karl Weick (1977???) found that *small wins* helps a group develop strength and confidence. Happily, these can be arranged relatively early in the project, when the team badly needs them. This is the *Early Victory* strategy.

On software projects, the *Early Victory* to seek is the first piece of visibly running, tested code. This is usually the *Walking Skeleton* (a tiny piece of usable system function, very often not much more than the ability to add an item to the system database and then look at it). Although this may not sound like much, team members learn from this small win each others' working styles, users get an early view of the system, and the sponsors see the team delivering.

* * *

Project teams often argue over the sequence in which to attack their problem. An oft-mentioned strategy is *Worst Thing First*. The reasoning is that once that worst thing is out of the way, everything will be easier.

The problem with worst-thing-first is that if the team fails to deliver, the sponsor has no idea where the failure lies: Is the team not good enough to pull off this project, is the technology wrong, or is the process wrong? In addition, the team members may get depressed or start to argue with each other.

Since I often join teams that haven't worked together before and are tackling a new problem with new technology, I prefer *Easiest Thing First, Hardest Second*. The team members get to debug their communication and their process on a relatively simple assignment. They and the sponsors get the confidence of an *Early Victory*. If the most difficult problem is still outside the team's capabilities, I look for the *hardest thing the team can succeed with* as the second task.

Once the risk of team and technical failure abates, a good strategy is *Highest Business Value First*. This strategy not only generates maximum financial returns for the project (SbN 2004??), but maps well to earned-value charts¹⁸. You can *show* everyone you are delivering business value and not just work hours. If the project should run out of time, it will be abundantly clear to the sponsors that they got the best value for money in the time spent, useful in both friendly and hostile circumstances¹⁹.

¹⁸ See p. 118.

¹⁹ See badly formed fixed-price project bids, on p. 309, and also (Patton 2003).

Strategy 3. Walking Skeleton

A *Walking Skeleton* is a tiny implementation of the system that performs a small end-to-end function. It need not use the final architecture, but it should link together the main architectural components. The architecture and the functionality can then evolve in parallel.

* * *

I first learned of the *Walking Skeleton* idea when a project's *Lead Designer* approached me for a conversation and described a project of his. He said (approximately):

We had a large project to do, consisting of systems passing messages around a ring to each other. The other technical lead and I decided that we should, within the first week, connect the systems together so they could pass a single null message around the ring. This way we at least had the ring working.

We then required that at the end of every week, no matter what new messages and message processing was developed during the week, the ring had to be intact, passing all of the previous weeks' messages without failure. This way we could grow the system in a controlled manner and keep the different teams in sync.

It worked wonderfully, and we would do it again.

I had the opportunity to apply this idea on a client-server system on my first large project.

We were moving frighteningly slowly. Our first three-month delivery was scheduled to deliver only a small set of functionality that might be considered interesting to the end users. We had to cut scope to meet even this mild ambition, and in the end delivered what I refer to as "read a number, write a number."

Much to my surprise, the system went live on schedule, the sponsors threw a big party, and the users started putting numbers into the system.

The system's design and code was not pretty, but we had connected the server-side software to the database system and could write functions on top of that architecture. Having the architectural elements connected and a sample piece of function running on it, we were able to develop more functionality in parallel with revising the architecture to be more robust (the *Incremental Rearchitecture* strategy).

Our sponsors were happy with this *Early Victory*. We tested the team, the process, the technologies, and the architecture at a very early point in the project.

Other authors have other names for similar sorts of ideas. The Poppendiecks talk about a "spanning application" (Poppendieck 2003), Dave Thomas and Andy Hunt use what they call "tracer bullets" (Hunt 2001??).

What constitutes a *Walking Skeleton* varies with the system being designed. For a client-server system, it would be a single screen-to-database-and-back capability. For a multi-tier or multi-platform system, it is a working connection between the tiers or platforms. For a compiler, it consists of compilation of the simplest element of the language, possibly just a single token. For a business process, it is walking through a single and simple business transaction (as Jeff Patton describes in the technique *Essential Interaction Design*).

Note that each subsystem is incomplete, but they are hooked together, and will stay hooked together from this point on.

The *Walking Skeleton* is not complete or robust (it only *walks*, pardon the phrase), and it is missing the flesh of the application functionality. Incrementally, over time, the infrastructure will be completed and full functionality will be added.

A *Walking Skeleton* is different from a *spike*²⁰. A spike is "the smallest implementation that demonstrates plausible technical success." The spike typically takes between a few hours and a few days to complete, and is thrown away afterwards, since it was built with non-production coding habits. A spike serves to answer the question, "Are we headed in the *wrong* direction?"

A *Walking Skeleton*, on the other hand, is permanent code, built with production coding habits, regression tests, and is intended to grow with the system. Once the system is up and running, it will stay up and running for the rest of the project, despite the *Incremental Rearchitecture* that is quite likely to occur.

²⁰ <http://c2.com/cgi/wiki?SpikeSolution>

Strategy 4. Incremental Rearchitecture

The system architecture will need to evolve, from the *Walking Skeleton*, and also to handle technology and business requirements changes over time. It is rarely effective to shut down development to perform an architectural revision, so the team evolves the architecture in stages, keeping the system running as they do so.

The team applies the idea of incremental development to revising the infrastructure or architecture as well as the system's end functionality.

* * *

The question naturally arises: How completely designed should the system architecture and infrastructure be during the early stages of the project?

Any one person has his personal "thought horizon," how much complexity he can keep in his head for several days in a row, how much he can foresee from his experience and knowledge of the situation. An architect who has done similar systems in the same domain using similar technologies can think through to a further horizon than one just getting started.

Some people keep the thought horizon down to a few days. They get started immediately with an initial design, and learn from that early version in which direction they should push the design. Others like to think longer and consider more contingencies before committing to an initial architecture.

The thought horizon on a Crystal Clear project is almost certainly reached within a week or two. At that point, the designers are probably speculating beyond their thought horizon, and would be better setting up the *Walking Skeleton*. They should use that to get to the first of their **Frequent Deliveries**, and use the feedback to improve the architecture. "Don't overdrive your headlights," is some people phrase it.

Here is a story about the use of *Incremental Rearchitecture* from my first large project.

The infrastructure team found after the second delivery that the object-to-relational database mapping they had planned to use would require an ever-increasing amount of programming as new functionality was added; in other words, it wouldn't scale. Being under heavy time pressure on a fixed-price contract, pressure was on them to simply keep plowing forward and just work harder and faster to keep up with the increasing work load.

The team lead decided, however, that an architectural redesign was needed. His team continued to support the old design for one more increment, while the system's delivered functionality was still small, and at the same time started their redesign. They slipped the new architecture into the third delivery cycle, coaching the function-development teams on how to write to the new interfaces, and supporting them with some number of automatic code-generation mechanisms. in the fourth delivery cycle, they ripped out all of the uses of the initial architecture.

On the fourth delivery cycle, the infrastructure architect breathed a sigh of relief: they could finally keep up with the now rapidly system functionality.

The converse also applies. Here is the unhappy story of a team that did not apply *Incremental Rearchitecture* when they should have.

The architects promised their executives that their radically new architecture would allow direct translation from use cases to running Java code, making the product incredibly responsive to changing business needs. This approach was, of course, a risky proposition, since no one had accomplished this before.

Recognizing this risk, I suggested to the project manager that their project use the exact strategy we had used in the story just above: create a simple, straightforward architecture that could be delivered on time, and swap in the new architecture if and when it proved itself.

The project manager said she had great confidence in the lead architects and didn't want the extra rework, so she chose to hang all hopes on the new architecture. Unhappily, the problem of going from use cases to running code turned out to be insoluble to those people. In the end, the project was left with no running architecture at all, and no product was ever shipped.

Developers in the last decade have shown that tidy, simple architectures are reasonably straightforward to upgrade to their next stage of complexity and performance. The business consequence of this is that, very often, a company can create an early version to demonstrate function and possibly even generate revenue, and then use the *Incremental Rearchitecture* strategy to revamp the infrastructure under the running system.

Starting from a simple working architecture and applying *Incremental Rearchitecture* is a winning strategy for most, though not all systems²¹. It provides a number of advantages when it can be used:

- The architecture is easier to modify when modifications are needed.
- The function- and infrastructure-teams get to work in parallel, advancing the moment at which usable function becomes available, or at least visible.
- The end users can view the system's proposed functionality early, and correct its fundamental fitness for business use.
- The running system might reveal shortcomings in the architecture that the early thought experiments didn't catch.
- The system can sometimes be deployed to a limited market, in which case the business will start earning revenue to help pay for ongoing development.

In deciding when to apply the strategy, ask

- whether the system can, in fact, be developed in two streams, with the infrastructure or architecture evolving in parallel with the system functionality (the answer is usually Yes);

²¹ It would, for example, probably be a bad idea to ask all cell phone users to take their phones back to the supplier so that a new internal software architecture could be downloaded!

- whether, by first creating a fallback, simple architecture and testing the system in live use, the business might catch an unpredicted architectural mistake (the answer is often Yes);
- whether the business can earn early revenue on deployment of a version with limited functionality to a limited market (the answer is surprisingly often Yes).

Incremental Rearchitecture is further discussed at length in the article "Extending an Architecture as it Earns Business Value" (Cockburn 2004).

Strategy 5. Information Radiators

An *Information Radiator* is a display posted in a place where people can see it as they work or walk by. It shows readers information they care about without having to ask anyone a question. This means more communication with fewer interruptions.

A good information radiator

- is large and easily visible to the casual, interested observer,
- is understood at a glance,
- changes periodically, so that it is worth visiting,
- is easily kept up to date.

Typically, it is on paper, posted on the team room or in the hallway. In a few exceptional circumstances, it is on a web page that people refer to frequently. Unusual examples of information radiators include a (real!) traffic light, a colored orb, and a computer monitor hung outside a cubicle's partition into the hallway.

Todd Little of Landmark Graphics made the interesting observation that information radiators generally serve to inform people *outside* the project team. The people on the project team generally know the information posted, because of their close communications with each other. It is the people outside the team who want or need to know that information in order to make their own decisions, and otherwise would interrupt the team to get that information, or would simply guess at it (often incorrectly).

* * *

Information Radiators can be used on any project, large or small. A small team can use them very conveniently to maintain information that they otherwise would have to maintain on the computer (which is both slower and less visible).

Information radiators are typically used to show status information, such as

- the current iteration's work set (use cases or stories),
- the current work assignments,
- the number of tests written (or passed),
- the number of use cases (or stories) delivered,
- the status of key servers (up, down, in maintenance),
- the core of the domain model,
- the results of the last *Reflection Workshop*.

Online files and web page generally do not make good information radiators, because an information radiator needs to be visible without significant effort on the part of the viewer. I have, of course, run into exceptions with this rule. One was the team performing *Continuous Integration* with CruiseControl, a dedicated server running a build-and-test script and posting the test results to a web page. The programmers tended to leave that web page visible at all times on their screens, so they

could respond to failing integration tests immediately. The other exception was the monitor hung over a cubicle wall into the hallway (see Figure 3-5), displaying current run-time measurements of the system in use.

Freeman-Benson and Borning wrote an experience report on a methodology they call YP (Freeman-Benson 03). They report on the effect of using a real traffic light:

Actually there are several: one in the hallway of our laboratory, two more in developer offices, and one virtual traffic light on the web. We use four color combinations:

- green when the build and all the tests have succeeded,
- yellow when the build and tests are in progress,
- yellow and green together when the build has passed the point where it is likely to fail (in practice, this means that all the tests have passed and that the final installer and distribution are being produced), and
- red if any part of the build or tests has failed.

The traffic light is a powerful symbol of the current state of the software. The web version at www.urbansim.org/fireman makes the status visible to anyone else who might be interested (including you, Gentle Reader, if you wish), and in particular for our customers, although in a less compelling way than the physical device.

The first author has used the traffic light as part of applying YP in four other development projects as well. Interestingly, only one team in addition to ours is still using the light — the other teams disconnected their lights because they didn't like them being red all the time. Rather than fixing the underlying problem (that their system was sufficiently unstable that their regression tests would not pass reliably), they chose to "eliminate the messenger." Only one of the teams acknowledged this decision explicitly.

In conversations with other team leads and software managers, we learned that a number of them had tried "failure indicators" such as sirens, flashing lights, red lights, and so forth, and that each had cancelled the experiment after a few days. Apparently the use of negative reinforcement (a red light) without the corresponding positive reinforcement (a green light) was too damaging to morale. Our experience with the traffic light is quite the opposite: everyone who joins a YP team immediately reports a sense of comfort at the large (8-12") green light glowing its message of "all is well with the build." Or on the rare occasions when the software has failed the nightly build tests, as the staff arrives in the morning, in the winter's gloom, with the lab hallway illuminated by the red glow of the traffic light, it's clear that a) something has gone wrong, and b) it should get fixed as soon as possible.

Below are photographs of different information radiators.

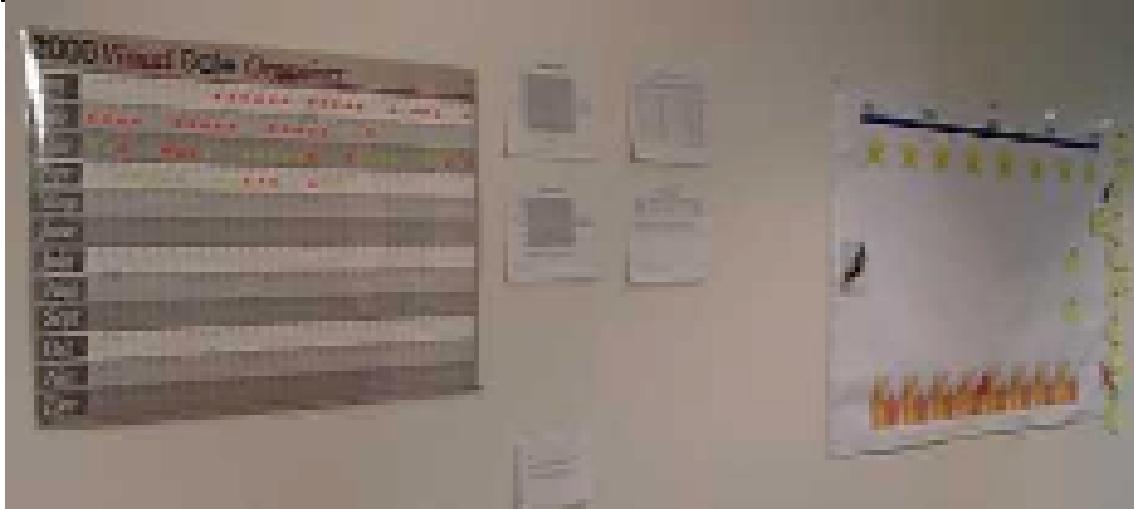


Figure 3-1. Development status of the user stories. (Thanks to ThoughtWorks)

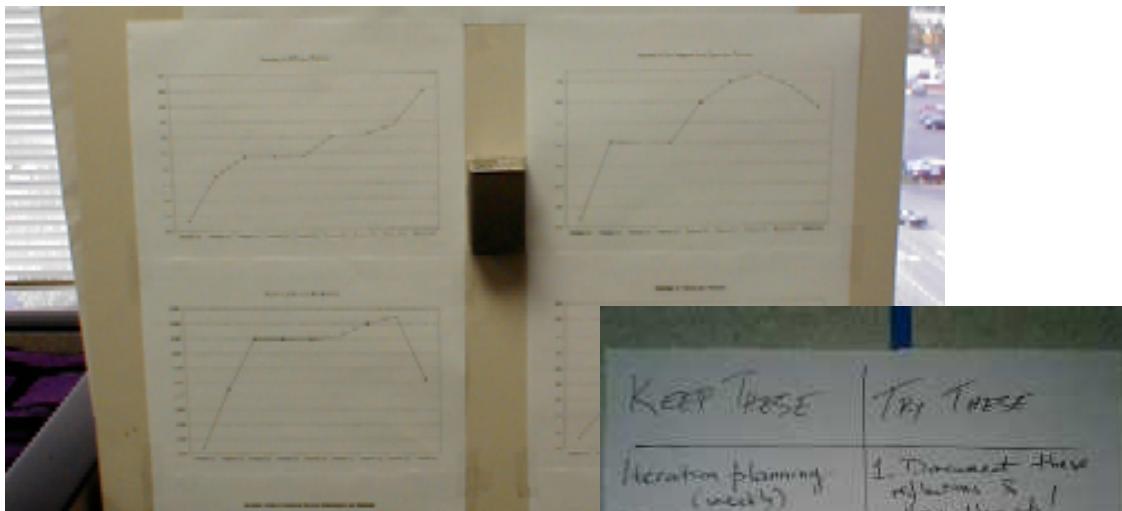


Figure 3-2. Charts of the code base kept near the programmers. Notice the code size going down in the last two deliveries! (Thanks to Randy Stafford)

Figure 3-3. The results of a reflection workshop. (Thanks to Jeff Patton)

Figure 3-4. Keeping track of the CVS branches. I like the creative use of hangers, pads and cubicle partitions. (Thanks to John Bullock)

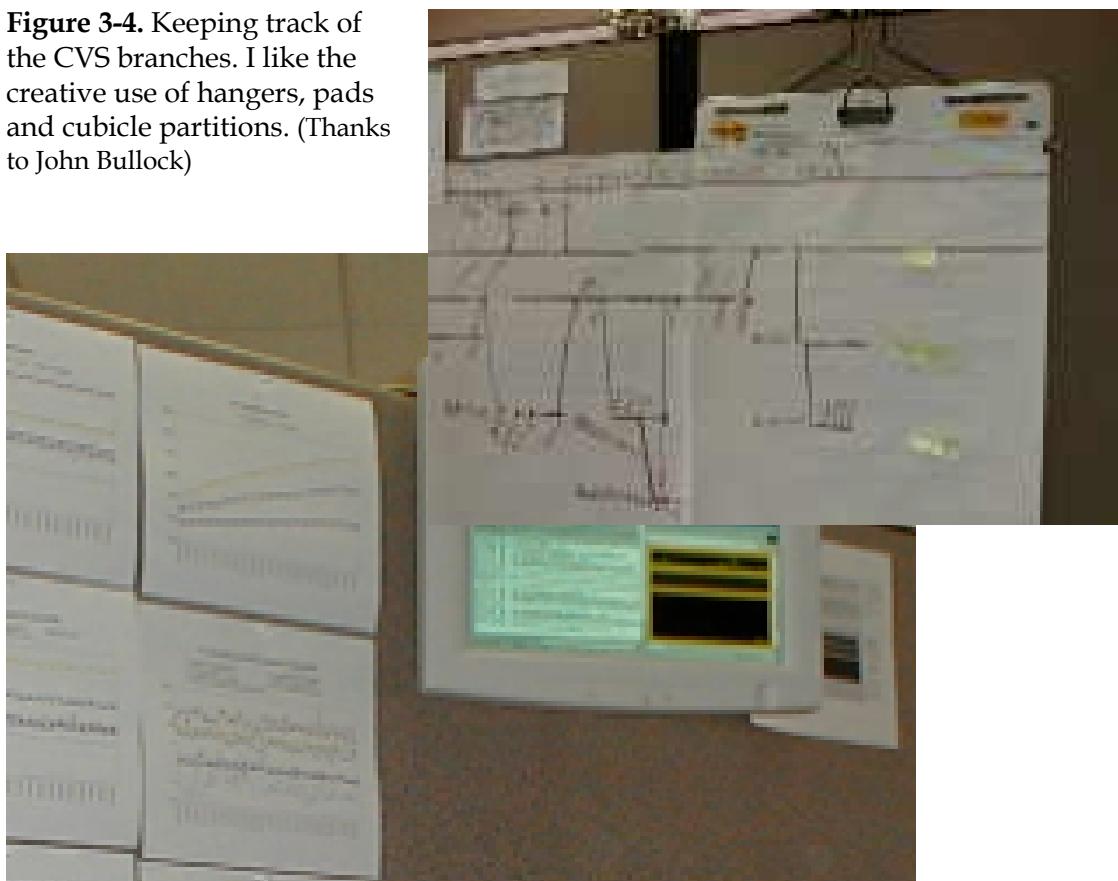


Figure 3-5. A monitor hung over the cubicle wall to show real-time measurements of the system in operation. (Thanks to Randy Stafford)

The Techniques

The techniques I have selected are

1. *Methodology Shaping*, gathering information about prior experiences and using it to come up with the starter conventions,
2. *Reflection Workshop*, a particular workshop format for **Reflective Improvement**,
3. *Blitz Planning*, which I also sometimes refer to as a project planning "jam session" (as in jazz) to emphasize its collaborative nature, a quick and collaborating project planning technique,
4. *Delphi Estimation*, a way to come up with a starter estimate for the total project,
5. *Daily Stand-ups*, a quick and efficient way to pass information around the team on a daily basis,
6. *Agile Interaction Design*, a fast version of usage-centered design,
7. *Process Miniature*, a learning technique,
8. *Side-by-Side Programming*, a less intense alternative to pair programming,
9. *Burn Charts*, an efficient way to planning and reporting progress, particularly suited for use on *Information Radiators*.

As described in the email series between Crystal and Alistair, no single specific, technique is mandated by Crystal. These particular strategies provide a good starting point, particularly the methodology shaping and reflection workshops, since those are most likely to be new to the group. Burn charts have become an interesting topic in their own right, since they have a natural match to the earned-value charts used in systems engineering project management.

Technique 1. Methodology Shaping

Establish the starter methodology in two steps:

1. Project interviews
2. Methodology shaping workshop

The information from the first feeds into the second.

Project Interviews

With this little technique, you will build a small library of experiences in your organization that show the strengths, weaknesses and themes of your organization. When you go into the methodology shaping workshop, you will examine them and discuss how to take advantage of the strengths and how to compensate for, or avoid, the weaknesses.

We found in one organization a theme that projects that had good internal and external communication went well, but when they didn't have that communication, the teams had a bad time and the outcome was negative.

Seeing this in front of us, I made extra effort to bring our project experiences to the directors of development and of the requesting division. We wrote a one-page summary after every iteration, summarized our costs, accomplishments, frustrations and lessons learned. We arranged for these two people to meet for an hour and review the project using the report as a base for their discussion. They actually talked about many other things during that hour, but they both commented on the positive effects of allocating time for discussion, and beginning by going through the report.

At the end of that year, the director of development commented, somewhat quizzically, that we had had a fairly similar project assignment as two other groups, but those other two projects failed, while ours succeeded. Looking at the projects, we noticed that ours was the only project that took time to attend to communication paths both within and external to the project. The theme we had found in our project interviews had played out once more, true to form.

Technique Variant 1

Start with yourselves, but also include other people from a few other projects in your organization. Have several people on your team interview people on other projects. Start your collection with four to ten interview reports. It is useful (but not critical) that each of your people interview more than one person on one project. You might talk to any two of the following: the project manager, the team lead, a user interface designer, and a programmer. Their different perspectives on the same project will prove informative. Even more informative, however, will be the common responses across multiple projects.

Keep in mind is that whatever the interviewee says is relevant. During an interview, don't offer your own opinions about any matter but use your judgement to select a next question to ask.

The following steps summarize how I have worked over the years in doing my own project interviews (Cockburn 2003).

Step 1. I ask to see one sample of each work product produced. Looking at these, I detect how much ceremony was likely to be on the project and think about what questions I should ask about the work products. I look for duplicated work, places where they might have been difficult to keep up to date. I ask whether iterative development was in use, and if so, how the documents were updated in following iterations. I look, in particular, for ways in which informal communication was used to patch over inconsistencies in the paperwork.

On one project, the team lead showed me 23 work products. I noticed a fair degree of overlap among them, and so I asked if the later ones were generated by tools from the earlier ones. The team lead said no, the people had to reenter them from scratch. So I followed up by asking how the people felt about this. He said they really hated it, but he made them do it anyway.

You can guess that that last piece of information is very useful in the methodology shaping workshop.

Step 2. I ask for a short history of the project. This history includes date started, staff changes (growing and shrinking), team structure, the emotionally high and low points of the project life. I do this to calibrate the size and type of the project and to detect where there may be interesting other questions to ask.

Step 3. I ask, "*What were the key things you did wrong that you wouldn't want to repeat on your next project?*" I write down whatever they say, and I fish around for related issues to investigate.

Step 4. I ask, "*What were the key things you did right that you would certainly like to preserve in your next project?*" In response to this question, people have named everything from where they sit, to having food in the refrigerator, to social activities, communication channels, software tools, software architecture, and domain modeling. Whatever you hear, write it down.

Step 5. I revisit the issues, "*What are your priorities with respect to the things you liked on the project? What is most critical to keep, and what is most negotiable?*" This is redundant, technically speaking, but my experience is that people come up with slightly different answers. I write those down. It is useful to ask at this point, "*Was there anything that surprised you about the project?*"

Step 6. Finally, I ask whether there is anything else I should hear about. I see where question goes.

You may find it useful to construct an interview template on which to write the results, so you can exchange them easily. The time that we did this, our template was 2

pages long (to control the amount of writing the interviewer does) and contained the following sections.

- 1. Project name, job of person interviewed**
- 2. Project data (start / end dates, staff size, domain, technology).**
- 3. Project history**
- 4. Did wrong / would not repeat**
- 5. Did right / would preserve**
- 6. Priorities**
- 7. Other**

Technique Variant 2

Jens Coldewey, in Germany, used a different technique to get the information he sought for his methodology shaping workshop. He sent to each member of his upcoming group a short questionnaire, asking what they had liked about their previous projects and wanted to retain on the new project, and what they had not liked about their previous projects and would want to set up differently on the new project. He took those answers straight into the methodology shaping workshop.

Technique Variant 3

Following Jens' lead, I have also run a facilitated workshop to come up with the same information. You can do this workshop in groups of any size, from a Crystal Clear project of just four people, to an organization of several dozen people. Allow an hour for this workshop if you have a small group, and two to three hours for a large group.

Break into work groups of three to five people each. Set up several flipcharts with pens of differing colors at each workgroup.

Have each work group brainstorm and list all the things they have personally experienced on projects in the past and would like not to repeat on the current or next project. They write all those on the one flipchart (or more likely, several pages of that flipchart). Time box the activity so they don't go all afternoon!

On the other flipchart and with the other color of pen, have them brainstorm and list all the things they have personally experienced on their own projects in the past (this is important, because otherwise they are likely to write down ideas they have heard about but not experienced), and would like to see repeated.

Spend some time combining and merging the common ideas in each list to make a single list of disliked / don't repeat and a separate list of liked / repeat items.

Giving each person a number, say seven, votes for each list, ask them to mark on each list the items they feel most important. They can stack all seven votes on one item or spread them around however they wish.

Count the votes for each item and sort by voting results.

At this point, you have a table of contents for project situations to "avoid" and ones to "keep," prioritized by personal significance to the people in the room.

During this workshop, you will not solve any of the problems mentioned. That is the work to do during the methodology shaping workshop. What you have is the information needed in that next workshop, which might take place on the same or a different day.

Methodology Shaping Workshop

The methodology shaping workshop is nothing more or less than a larger version of the periodic reflection workshop described next. In the periodic reflection workshop, the team already has a list of rules and conventions they are using, and reflect on which to keep or drop. That workshop should make relatively small changes to the set of conventions.

The methodology shaping workshop, on the other hand, starts with a group of people who have not done the exercise before and don't yet know what their operating conventions will be. Their output will therefore be a single large list of proposed ideas, rules and conventions. The book *Agile Software Development* contains a sample 36-point list that was the result of one such investigation.

The workshop starts with a review of the organization's fixed rules about software development. These are taken as given rules, unless someone on the team has an idea on how to shift any of them.

The people go through the list of "liked/keep" items that came from the project interviews and see how many of those they can easily arrange on the upcoming project. They table the difficult ones for separate study.

They go through the list of "disliked/avoid" items and brainstorm ways to avoid them on the upcoming project. The answers get written down on the list of ideas, rules and conventions.

They locate the themes of the organization (as shown in the story at the start of this technique), and discuss how to handle, compensate for, or take advantage of those themes. They write their answers on the list of ideas, rules and conventions.

They go through the difficult items saved up to now, discuss and brainstorm any ways to deal with them, given all the ideas they already have written down. They write down the ideas they want to try on the main list, and write down on a separate list the items they are worried about but don't have ideas for. (They look at this list later in the project, either to see if they have new ideas, or to discuss with the project sponsor and their managers, in case they run into exactly those problems.)

At this point, the list of ideas, rules and conventions is probably quite long.

The people go through the list again, and mark the ones that are most important to pay attention to, and put into parentheses the ones that are interesting but possibly marginal. These latter ones are the ones that are nice to have but the first to go in case the team gets overloaded with following the list.

The team will update these conventions a month later during their first *Reflection Workshop* (I write the time "a month later" as a reminder that it should not go too long before review. The exact time chosen is up to the team to decide.)

Believe it or not, these are now the "starter methodology" details for the project!

Technique 2. Reflection Workshop

The team should pause for an hour periodically – certainly after each delivery – to reflect on its working conventions. In the reflection workshop, the team members discuss what is working well, what needs improvement, and what they are going to do differently during the next period.

I like to do one of these half-way through the first iteration as a sanity check: "Working this way, are we on track to complete our assignment in this iteration, or do we need to compensate now, before it is too late?" Since my experience is that most teams schedule too much work to do in the first iteration, I pay less attention to bad estimates of how much can be done in the first iteration, and look for disastrously bad working conventions. Repairing bad working conventions is what I focus on in the first workshop.

People who hold reflection workshops on a regular basis tell me of a pattern that shows up: the group finds a lot to discuss in the first few workshops, and then, after a while, they find that there is little more to say from the previous time. In some cases, they lengthen the time between workshops, holding them only after each delivery. In other cases they use the time for a closer examination of their team values, personal working styles and the like.

There are various formats for a workshop handling these themes. The one that follows is my favorite because it is both simple and brief (as are all of my favorite techniques). Norm Kerth wrote a book called *Project Retrospectives* that discusses many other exercises you could perform during a reflection workshop.

The Keep/Try Reflection Workshop

Very simply, capture the following three things on a flipchart and then post the flipchart prominently as an *Information Radiator* for the group to see as they work.

1. *What we should keep.* These are the conventions we don't want to lose in the next time period. The sorts of things that get mentioned often include: sitting close together, non-interruptable focus hours, daily stand-up meetings, and automated regression tests.

Keep these test lock-down quiet time daily meetings	Try these pair testing fines for interruptions programmers help testers
Problems too many interruptions shipping buggy code	

Figure 3-6. Poster format for the reflection workshop.

2. *Where we are having ongoing problems.* It is not generally healthy to focus too much on problems, but people get stuck in the workshop if they can't mention a problem that is bothering them and see it get written on the poster. Create a half-column or less on the flipchart to capture problems that we are having trouble getting around. Topics that often get mentioned are: too many interruptions, requirements changing too often, people changing code without notifying anyone else, unable to buy new tool suites.
3. *What we want to try in the next time period.* In a sense, these are the most important of the three categories. These are what the team agrees to try for the next few weeks. Some of the suggestions I hear include: two hours of uninterrupted work 10 a.m. to noon each day; use of coding standards; more unit or acceptance tests; occasional pair programming; daily stand-up meeting.

You might run your first reflection workshop in no more than 15 minutes. This will constrain the time of the discussion so that people do not run too far afield, and focus the team on coming up with concrete suggestions for what to keep and what to change. This provides the team with a *Process Miniature* for the technique.

After the first one, you can decide how long to allocate for each workshop. One group chose to allocate two days in an off-site location every three months for the first year of their project. They used the first day for team building and reflection workshop, and spent the second day planning the next three-month delivery cycle. They eventually cut it down to just one day in their offices. I have used an hour in the cafeteria every couple of months, down to 15 -30 minutes every two weeks.

Here are some photographs of reflection workshop results using the above technique. You will notice in the second one that we created an additional section just above the *Problems* to capture ideas that people had but didn't want to try right away. This section served to feed the discussion at the following workshop.

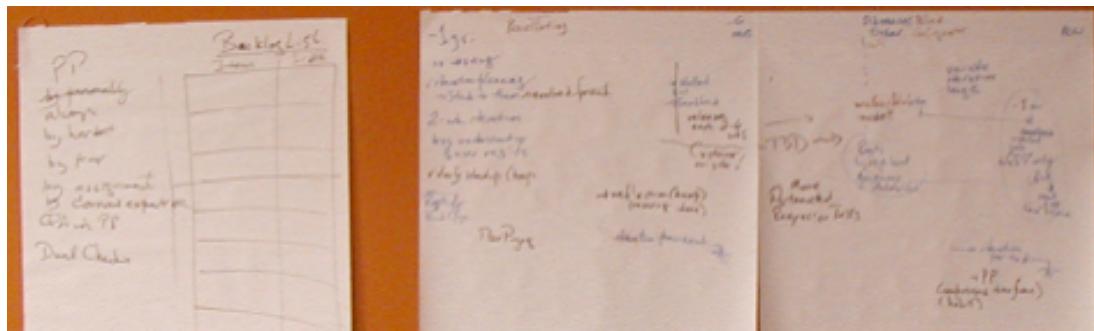
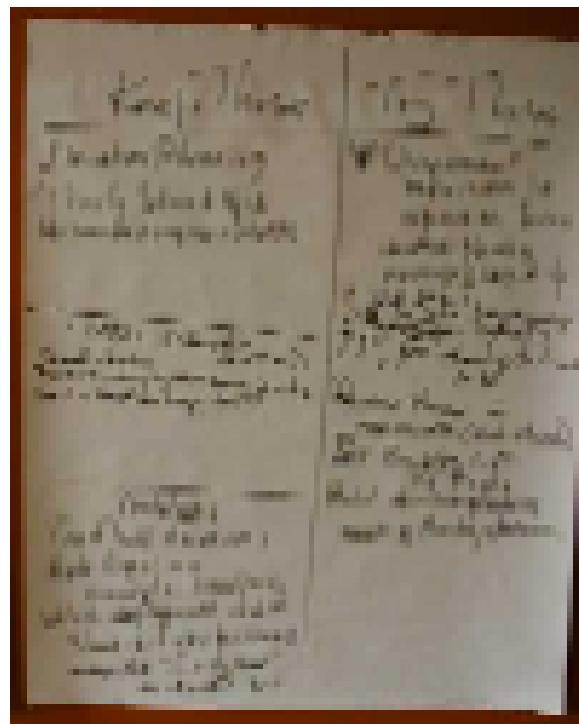


Figure 3-7. Reflection workshop flipcharts showing project history and ideas tried. (Thanks to Jeff Patton and Tomax)

Figure 3-8. Reflection workshop output.(thanks to Tomax and Jeff Patton)



Technique 3. Blitz Planning

The planning session is an opportunity for the executive sponsor, ambassador user and developers to contribute together to build the project map and timeline. As always, there are several ways to work.

One technique is the "planning game" from XP. In the planning game, people place index cards on the table, one user story per card. The group pretends there are no dependencies between cards, and simply lines them up in the preferred sequence of development. The developers write on each card an estimate of how long it will take to produce the function; the sponsor or ambassador user puts them into development priority sequence, taking into account the development time and business value of each function. The cards are clustered into iterations, and those iterations are clustered into releases, usually not longer than a few months each. The planning game is described in detail in (Beck 2001)

The following describes the variation I like to use. I call it *Blitz Planning* to emphasize that it goes fast. However, I also like to refer to it on occasion as a *Project Planning "Jam Session"* to emphasize that everyone is supposed to play together and off each other in a collaborative way, as in a jazz jam session. If you lose the amicable collaboration, you can do the technique, but you lose many opportunities to creatively improve the project plan.

I find this technique works well for a planning horizon up to about three months. After that, the amount of detail is overwhelming. For time horizons longer than three months, I use the *Project Map* described among the work products²².

Here is a brief overview of the technique. I describe the steps in more detail below.

Gather into one room representatives of each stakeholder category, the *Executive Sponsor*, end users, and developers, in particular. Brainstorm onto index cards all the development tasks to be done. Collate and merge the tasks to avoid duplication. Sequence the cards according to priority and dependency order. At this point the developers put down the work time estimates. If a particular person is required for the task, add that person's name. If there is a dependency on an external group, write that somewhere else on the card.

Having the cards on the table, work through the plan looking for key milestones and optimizations. Identify the *Walking Skeleton*, the first delivery, and the first delivery that produces a revenue stream. Look for excessive dependencies on any one person, and off-load that person. Look for early tasks that unnecessarily block early revenue stream, and tasks listed for late that for risk reduction reasons should be done

²² This is an example of "two-tiered" project planning. The long-term plan is the coarse-grained *Project Map*, the short-term plan is the result of the *Blitz Planning* session. Two-tiered project plans are common in agile projects.

early (e.g., load testing). Move those cards around to better places. Work with the sponsor to make sure the plan is delivering functionality in accordance with the project's true priorities.

The result is your project plan.

Figure xx.xx shows a sample of such a card, and Figure xx.xx shows a sample of how the cards look when laid out on the table.

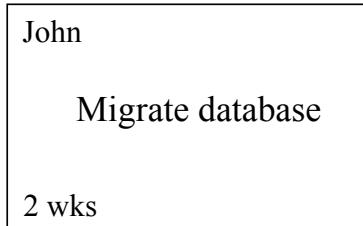


Figure 3-9. Sample blitz planning card.

Here are the steps in detail. Note that this technique is spelled out for Level-1 practitioners²³. When you reach Level 2 with this technique, try some variations to handle situations where the requirements are not yet known, where the time horizon is longer than three months, using sticky notes instead of cards, and so on.

1. Gather the attendees

I subtitle this technique a project planning "jam session" because the technique allows the different project stakeholders to gather together and share ideas on how to make the optimal plan. Traditionally, the plan is made by the project manager or team lead without buy-in from the other stakeholders. That leads to two dysfunctions: First, the plan is of course wrong, since the poor person assigned to construct it can't possibly know all the tasks and times; second, when the errors in the project plan become evident, people find it easy to point accusing fingers at the person who made the plan. To counter these dysfunctions, make sure the *Executive Sponsor*, one or two key users, any business analysts, and the entire development team, including anyone involved in testing and deployment, are in the room. This group will name the tasks more completely, the time estimates will be more reasonable, and just as importantly, everyone will see all the trades-offs made in constructing the plan. Everyone is jointly responsible for the result, and they all know it. As one person said, "We all made it, we all discussed the optimizations made."

2. Brainstorm the tasks

Everyone grabs cards and writes tasks on them as fast as they can. Usually the developers do most of the writing, but the executive sponsor and ambassador

²³ See "shu-ha-ri" in the early emails, page 25.

user often have a few tasks to contribute. List every task that will take half a day to several weeks, including interviewing users, programming specific functions, writing user help text, migrating databases, and installing the system (I often group tiny tasks together). The idea is to be complete. This step may last 5, 10, or 15 minutes.

3. Lay out the tasks

Everyone lays the cards out on the table in dependency order, the first ones at the head of the table, with successive tasks down the table. Tasks that can run in parallel are placed side by side across the table; tasks that run sequentially are placed above and below each other; duplicate task cards are removed.

It usually requires a big table. I have used a large conference table and also several six-foot cafeteria tables laid end to end.

4. Review the tasks

Everyone walks up and down the table, looking for tasks that didn't get captured in the brainstorming. One person's card often triggers a thought in another person. Add cards as needed.

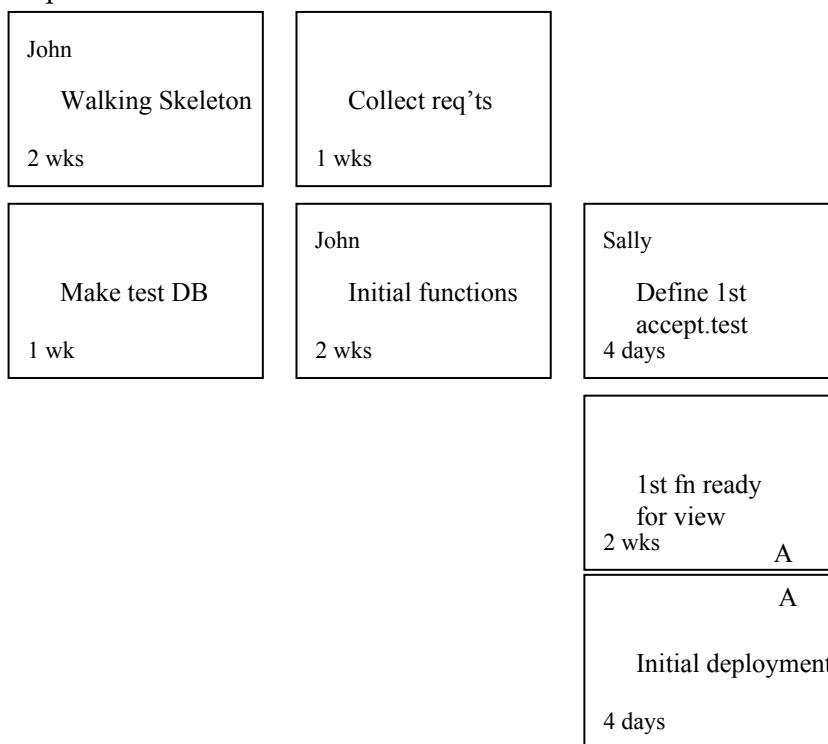


Figure 3-10. Cards as they might be laid out on the table.

5. Estimate and tag the tasks.

Whoever is going to do each task writes down their estimate for how long it will take (I write this on the lower left corner). If multiple people come up with different estimates, they have a brief discussion and put down their best second estimate. They also write down the names of any specific people required for any particular task (I write this on the top of the card). It often happens that the team lead's name is on a disproportionate number of cards. Seeing that person's name on so many cards should lead the team into a discussion of how to off-load that person or otherwise reduce the dependency on him. If a task is dependent on an external event or task, mark that on the card (I write this on the center of the right side of the card.)

6. Sort the tasks

In this step, the people work to identify more closely the presence or absence of dependencies, and to question the placement of cards on the table.

Tasks that are placed sequentially often can be started in parallel. Identifying possible parallelism loosens the constraints on the plan. Also, it often turns out that there are parallel streams of activity, one for the *Ambassador User*, one for the developers, or perhaps three separate streams for infrastructure, functionality and user interface development.

These parallel streams start to look like separate tracks of cards running down the table, occasionally coming together.

Certain cards have a strictly sequential relationship, such that the second simply cannot be started before the first has finished (example: "Evaluate vendors" must fully precede "Set up contract with vendor"). Mark these pairs in some special way so that information doesn't get lost when the cards get moved around (I put the same capital letter, A, B, C, at an adjoining edge of the card, the bottom edge of the first card and the corresponding point on the top edge of the second card). These strict dependencies are surprisingly rare; I rarely get beyond the letter E.

7. Mark the *Walking Skeleton*, the earliest release, and the earliest revenue

Finding the first and smallest collection of functionality that can conceivably be of use to some users is critical. It is critical because that release represents the first time the users will see the system, their first idea of the look, feel, and shape of the system. Setting that point as early as possibly gives the entire organization the most time to adapt to any new ideas or mismatches that show up from the initial release. On some occasions, this first release represents the earliest moment at which revenue can be gained; in this situation, the group should focus on getting all other cards out of its way²⁴. All in all, three events of

²⁴ *Software by Numbers* (Who 2004???) is an excellent tutorial on this subject, using language of financial analysts. They plan in terms of "minimal marketable features" (MMF).

interest are located: the *Walking Skeleton*, the earliest usable release, and the point of earliest revenue.

I mark these points by putting a bit of distance between the cards above and below those points. I have also used string, yardsticks, pencils - anything to demark those points.

It requires cooperation between the *Ambassador User*, *Executive Sponsor*, and *Lead Designer* to discover and optimally set these points, because some tasks will have to be moved up and others deferred. Whoever is moving the tasks around needs to be aware of the consequences of these moves, and that means each stakeholder group has to be present, alert and cooperative.

A word is in order about the tasks that precede the initial release:

Some of the early tasks are likely to be strictly technical, such as ordering software, setting up a contract, or loading a database. It is useful to have those tasks visible to the *Executive Sponsor* and *Ambassador User*, because they need to explain the state of the project to other people. The technical task cards show what work needs to be done before business software will become visible.

Historically in our business, the user group gets told only, "The programming team says they have a bunch of technical work to do first." This often translates to months going by without deliveries or even visible progress. By making the tasks visible and public, the *Executive Sponsor* or *Ambassador User* can report, "They have five technical things to do before we can see some software," "They still have two technical things to do before we can see some software," "They're on the last technical thing, and then they'll show us some running software."

This sort of visibility goes a long way to reducing the tensions between the groups.

8. Identify other releases

The group works out where other natural release points happen. Quite often, there is a particular cluster of function that really should be bundled together for the users to make good use of the system. At some point, the functions to be included simply become a long list, and at that point it may be more valuable to base the releases on regular time periods (monthly, bi-monthly or quarterly), rather than on collections of functionality.

Neither Crystal Clear nor the Blitz Planning technique mandate what algorithm you use for choosing iteration lengths and release periods.

Mark the key releases using a space between the cards, string, yardsticks, or whatever.

9. Optimize the plan to fit the project priorities

At this point, you have a plan. Typical the plan is not yet a *good* plan. It is my experience that when the numbers on the cards are added up, the team is in for "sticker shock" (this is the shock a car buyer experiences when stepping around to read the sticker posted on the car window announcing the price after all the car's options are added together).

The entire group, and particularly the *Executive Sponsor*, *Ambassador User* and *Lead Designer* now have some creative problem solving to do to come up with an acceptable plan. What they shift depends on the priorities set for the project: time-to-market, cost, or feature set. This is where they really start "jamming." Typically, following the sticker shock, one of three things happen:

- The *Executive Sponsor* reconsiders the business need for the project (in one case, the executive team decided simply to buy a commercial package instead);
- The team removes tasks from the project;
- They shrug their shoulders and simply move forward.

Pressuring the developers to change their estimates is not advisable. That not only makes a lie of the plan, but it convinces the developers that the *Executive Sponsor* is not being realistic, open and honest.

In the worst possible case, the *Executive Sponsor* has the right to say, "OK, I see the tasks and the time estimates. However, I can't change the deadline, and I can't see anything else to remove, reorder or outsource. I'm going to multiply all estimates by 80% so that we can meet our target date. In the meantime, let's all keep our eyes open for new alternatives so we can reestablish these original estimates" (see the discussion on Crystal Clear and "Reallocation of Power," page 309)

This is a drastic measure and should not be used lightly. The point of doing it is to make public both how the priorities affect the schedule, and the fact that the team's estimates are being overridden. Everyone has seen the cards and the original estimates. The team should track both plans, report against both in their status charts, and keep visibility of progress high.

Take it seriously if you have to resort to this measure even twice. That indicates there is another problem in the organization needing to be sorted out, having to do with the viability of the organization's business model, the trust between developers and sponsors, or the historic accuracy of the development team's estimates. Straightening those out will be crucial to the ongoing viability of the development team.

When optimizing the card layout on the table, the people look for three improvements in particular.

- They spot that a particular cluster of cards *almost* creates a coherent release having business value. By moving a card up higher, they can get that coherent business value much earlier. In one case, I heard the *Ambassador User* say, "If

we only had this card up here, we could start charging revenue with this release." You can guess that we immediately moved that card up, *and then moved down all cards not strictly needed to produce that release!*

- They spot a risk from having a card late in the project. This is typically done by someone with technical background playing the challenger.

I saw a card near the very end of the table labeled "Load Testing." Not thinking too much about it, I asked the *Lead Designer*: "How long will it take to perform?" He answered, "A few days." Still not worrying, I asked, "And suppose it fails, how long is it likely to take to revise the architecture?" He answered, "Oh, three or four weeks." At this point the *Ambassador User* started looking alarmed. I asked, "Would it be possible to do it earlier?" "Sure," he answered.

We looked for the earliest moment at which he could do it without damaging the other critical business items on the table. I tentatively placed it just after the initial release. "Could you do it already here? (and then you'd have lots of time to revise the architecture if it fails)" I noticed the *Ambassador User* vehemently nodding her head on the other side of the table, and noticed her large sigh of relief when he said, "Sure."

- They spot that one particular person has too many cards allocated to him or her. That person is going to be overloaded and will probably get into trouble. They brainstorm how to assign those tasks, or the bulk of some of them, to other people to reduce the risk to the project and the strain on that person.

10. Capture the output.

You have a plan, but it is only cards on a table. You need to preserve the information.

You can photograph the table, tape the cards together and mount them on the wall as an information radiator, or type them into another tool of your choice.

I like to number the cards so that their placement on the table can be reconstructed. I number them 1, 2, 3, and so on down the table for the sequential relationships, 3a, 3b, 3c and so on across any particular parallel set (and on occasion 3a1, 3a2, 3a3 and so on if there are nested subsequences).

* * *

There is one note I am obliged to put in here and repeat several times in the book. A common interpretation of both XP's planning game and *Blitz Planning* is that the *Executive Sponsor* is at the mercy of the developers in constructing the project plan. In actuality, there is a three-way division of responsibilities: the *Executive Sponsor*'s, the developers', and their joint responsibility.

The *Executive Sponsor* is responsible for choosing the priorities that drive which system features stay on the table, which are removed if scope is to be cut, and what functions should go into which release.

The developers are responsible for assessing the time needed to do their work. The *Executive Sponsor* needs to recognize that these are the people she has hired to the work; they are professionals; these are their best estimates.

They are jointly responsible for being creative in creating strategies to maximize their effectiveness. I usually find the initial layout of the cards on the table depressing, and unsatisfactory for business reasons. The developers can't change the priorities and the *Executive Sponsor* can't change the estimates. What are they to do?

They work together. The initial layout is usually inefficient. Working together, they come up with creative reorderings to get a better result. If the resulting plan is still unacceptable, the *Executive Sponsor* must decide what the top priority items are, what gets dropped, or perhaps whether the deadline or team be changed.

When they work together, they jointly see the constraints, jointly generate options, and jointly construct a plan that produces the best result for the resources expended. It is for this reason that I sometimes refer to this technique as project planning *jam session*. (P.s. Done with good will, this technique also is a lot of fun, another characteristic of a good *jam session*.)

Technique 4. Delphi Estimation using Expertise Rankings

People always get around to asking, "But how should we estimate the length of time it will take to develop the software?" The true answer is, "Best guess." While technically correct, and practiced on every project, this answer is rarely comforting.

A group of us created an estimation technique one night, while trying to work out the bid on a \$15 million fixed-price, fixed-scope project. I have since found the line of reasoning we took to have an interesting separation of questions. I am told that this corresponds to what is known as the "Delphi" technique (forecasting the future, but presumably without incense, vestal virgins, or answering in riddles).

This technique was first described in *Surviving Object-Oriented Projects*. I present it here, in story form. The story relates to a larger, Crystal Orange project, but you can still use it in adjusted form on a smaller project.

Before beginning the project proper, but after spending two weeks gathering 140 rough use cases, we made our first project estimate. Our four best OO designers, the project manager, and a few other experienced, non-OO people made up the planning team. We split the session into four phases:

- estimating the size of the system to be built
- estimating work time according to the type of person we would need
- suggesting releases, by technical and business dependency
- balancing the releases into approximately similar sizes.

We held an open auction to arrive at a size estimate. We constructed a large table on the whiteboard. Each senior designer created row-labels for the factors he thought would determine the project effort. One wrote, "technical frameworks, use cases, UI screens." Another added, "business classes"; another added, "database generation tool".

Each person wrote in their column of the table his guess of how many of each factor was present. After everyone had had a turn, we ran a second round. The first person added a new column with his revised estimate based on what he had learned during the first round.

We did three rounds this way. At the end, there were about 20-25 factors in all. Some people used multiplication factors from the estimate of business classes, some from use cases, some from UI screens. It turned out that the key drivers for the estimates were the number of:

- business classes
- screens
- frameworks
- technical classes (infrastructure, utility, etc.)

We discussed whether we had achieved convergence, and what the differing factors were. In the end, we agreed on some numbers and understood where we differed.

In the second phase, we decided what type of person we would need for each section of code. Frameworks are hard, and there are only a select number of people who can write them reasonably, so we made that sensitive to the specific person we could hire. We decided that the business classes would be relatively easy to write, but would require business knowledge. Technical classes would be harder, but not require business knowledge. In the end we settled on:

- expert developers for the frameworks
- intermediate level developers for technical and UI classes
- relatively novice developers who knew the domain for those classes

We split the table of classes to be developed into those three categories, summed the classes per category, and decided on how many classes per month that level of developer could develop. We gave the framework developers 10 weeks per framework, the technical and UI developers 3 weeks per class, and the business class developers 2 weeks per class.

The sum of all those weeks gave us our first project time estimate. We hemmed and hawed over that for a long time, comparing it against other reasonability measures, such as the rate we could hire and train people. In the end, we kept the estimate.

Deciding the releases was straightforward. We already knew we wanted a release each three months. Nothing could be delivered until most of the infrastructure was done, so that went into release 1 [note: it was later completely replaced, using the *Incremental Rearchitecture* strategy]. We created a *Project Map*, a dependency graph of the business functions, by technical and business dependence. From the size estimates, we circled areas of similar size. That gave us our release plan.

I tracked the project against our original estimate periodically during the project. Our progress tracked our original plan over the long run, but going slower at first and then faster at the end.

I present this technique here because I notice that most people who use a Delphi technique forget the crucial second phase, when the group assesses their ability to hire specific people with specific talents and skills. Without this information the result is not a plan, but a wish.

Technique 5. Daily Stand-Up Meetings

The daily stand-up meeting is a short meeting to trade notes on status, progress and problems. The key word is short. The meeting is not used to *discuss* problems, but to *identify* problems. If you find you are discussing how to solve problems in your daily stand-up, raise your hand and ask that problem resolution be dealt with right after the stand-up meeting, with only those people who have to be there.

The term "daily stand-up" comes from the Scrum methodology (Schwaber 2002). The idea is to get rid of long so-called status meetings where programmers ramble on for five or ten minutes about a piece of code they are working on, or management people ramble about various project initiatives. To keep people from rambling, the convention is that the meeting takes place standing up, so people can't fall asleep, type on their laptops, or doodle on paper. We want them sensitive to the passage of time.

The authors of Scrum suggest that each person simply answers three questions:

- What did I work on yesterday?
- What do I plan on working on today?
- What is getting in my way?

The daily stand-up meeting is amazingly effective for disseminating information, highlighting when someone is stuck, revealing when the item someone is working on is too low a priority, is off-topic or adding unrequested features, and generally keeping the group focused and on track. It is simple, and it has been added into projects using every imaginable methodology with good effect.

Technique 6. Essential Interaction Design

Most authors on agile development don't say much about user-interface design²⁵, giving the impression we think it is unimportant. Speaking at least for myself, I don't write about it because I have so little personal experience with this specialty, even though I recognize its importance.

For that reason, I include here four adaptations of usage-centered design (Constantine 1999???) that Jeff Patton created for the agile context. Jeff highlights and simplifies the activities needed to define, design and test the system's *interaction contexts* and its *personalities*²⁶. These techniques are well suited to projects where people are collocated and have to get a lot accomplished in a limited time. (Just because you have **Easy Access to Expert Users** doesn't mean you get to waste their time!)

About personalities. A software system presents a certain amount of help, information and speed to its various end users. It might be designed to be fast and efficient, for example, or warm, friendly and informative. These are the *personalities* it presents to the outside world. Most systems present different personalities to different users, depending on those users' backgrounds and needs. Except . . . most designers don't develop the personalities in any deliberate fashion. They simply throw together whatever they have on hand, and the personalities of the system are just whatever happens to come out.

There are probably one or two *focal* user groups that the sponsors really want to see satisfied with the new system. The team needs to find out who those are, what they need to accomplish using the software, what personality is best suited for each, and be sure they are properly served by the system. They need to detect, write down, and periodically recheck who those are.

Jeff illustrates with the example of a chain of retail stores. The cashier uses the system daily, gets familiar with it, and prioritizes for speed in registering sales. The store consultant, on the other hand, won't use the system very often, won't be fluent with its interface, but knowing all aspects of the store's operations, will want to do more functions than the cashier. The cashier wants a fast-and-efficient personality to work with; the store consultant wants an informative-and-helpful one.

About interaction contexts. When working through their tasks, the users will want to see information in clusters. Part of interaction design is detecting the clustering of information suited to each users' tasks, and looking for commonality across those. The developers will convert those to user interface settings, often with a set of information matching an interaction context.

²⁵ (Cohn 2004) breaks the silence with a chapter on usage-centered design.

²⁶ My term. It seems to me the interaction design community is missing this higher-level concept of their work.

Essential Interaction Design produces

- shared understanding among the sponsors, users and developers who were present in the room as to the roles and tasks to be addressed by the system and the relative priorities of each, and also
- paper-and-pen collages ("reminding markers" in cooperative-game language) showing roles, tasks, and interaction contexts, marked with notes so that the team can deliver them in business value order.

Jeff articulates four techniques, in all.

1. Essential Interaction Design (the Workshop)
2. Deriving the UI
3. Usability Inspection (during Design)
4. QA Testing the System Personalities

The first is essential interaction design itself, done either near the beginning of the project or the start of an iteration. After that, he presents short techniques for designing the screens themselves, for reviewing the UI with the users, and for testing internally that the finished software is appropriate for the user roles that will be using it.

Here are the techniques as Jeff describes them.

* * *

Essential Interaction Design (the Workshop)

The approach is based on the usage-centric design approach (Constantine 1999???) except that this initial requirements elicitation and design process can be completed in a couple of hours to a couple of days. When an experienced facilitator is used, no advanced training of the business people is required. The techniques can be applied during initial requirements elicitation, when building incremental release plans, while defining the general form of the software, deriving user interface from uses cases, acceptance testing and even during end-user usability evaluation, if need be.

What follows is a step by step outline of the initial requirements elicitation and design process. The overall goal of the technique suite is to *hasten the discovery of requirements by bringing together people from each critical aspect of the project and allowing them to explain what they know to each other.*

Step 1. Get the right people into the room.

We're replacing weeks of interviewing and field research with conversation during this meeting, so it's critical the meeting involve members of each constituency the system will be serving, including those developing the system. Remember, if you're the facilitator, you aren't interested in what participants say to you, but rather what they say to each other.

You will need in the workshop

- selected key project stakeholders,

-
- selected key users,
 - domain experts, and
 - members of the development team, including test/QA people.

Eight to twelve people are good. It's sometimes hard to limit attendees to that number, but do it.

Set participants in a comfortable workspace with a big worktable, lots of wall space for hanging posters. Include markers, tape, 3"x5" index cards, and lots of snacks to keep the less busy people occupied.

You're hosting a party.

Step 2. Capture and annotate user roles.

The term "actor" refers to any of job title, user role, or a mixture of both. We are after "roles," phrases that indicates what particular users' goals are. I always look for "thing doer" phrases to describe a role, or even, "*adjective* thing doer," to make it more specific (only in English can a noun describing an adjective be used as a verb in the past tense!).

For example: we might find not just an "order-taker" in the store situation, but a "special-order taker," or even better, a "hurried special-order taker." To handle customers angry because the special order is late, we find a "late-order researcher" or a "complaint handler." (People with titles like Sales Associate and Manager perform all the roles above.) The extended role name makes it easier to keep the role clear enough so that anyone, with or without special domain knowledge, can understand what the role might be doing. It lets us know the person performing in this role is in a hurry and needs functionality, and a user-interface, that can support that.

Brainstorm user roles onto index cards, "card-storming," as Constantine & Lockwood call it. After card-storming, refine your role list by removing duplicates and verifying the names are clear. Good names are important. We won't be relying on pages of documentation to describe each role, so the name is all you've got. Make sure it's expressive.

You are likely to name roles that you might consider stakeholders of the system, not necessarily users. Don't discard these. It's important to consider the goals of these people. It's important to ask how the software could support those goals. I often find that roles that might have been set aside as stakeholders actually need functionality in the system that supports proving to them that their interests were protected.

On the index card, write the role's primary goal or goals under the role name. What constitutes success for this user in using the system? What constitutes failure? We will keep checking that the requirements we capture really help user roles meet their goals.



Figure 3-11. The role modeling session. Be sure to listen for the conversations that occur during this process. (Thanks to Jeff Patton)

Write only what constitutes success from the user role's perspective, not his boss's or some other stakeholder's. For example a call center employee in a credit card service center, a "credit-card application taker" might have a goal to take applications quickly enough and accurately enough to avoid

a manager's attention. The manager, on the other hand, might want to improve speed and accuracy and reduce customer complaints. Other stakeholders may be concerned with up-selling credit insurance policies. The manager's and stakeholder goals aren't necessarily those of the "credit-card application taker." Make sure the appropriate goal finds its way onto the appropriate role card. If you want to capture the other stakeholders' needs, consider adding some roles, "efficiency watcher," "upselling watcher," and create role cards for them.

Write on each card a subjective estimate of that role's business value or importance, high, medium, or low. I mark "H", "M", or "L" on the lower left corner of the card.

Write on each card an estimate of how often the role will use the system. This can be high, medium, low, or hourly, daily, monthly, quarterly, yearly, or some other frequency you find suitable. I write these on the lower right hand corner of the card.

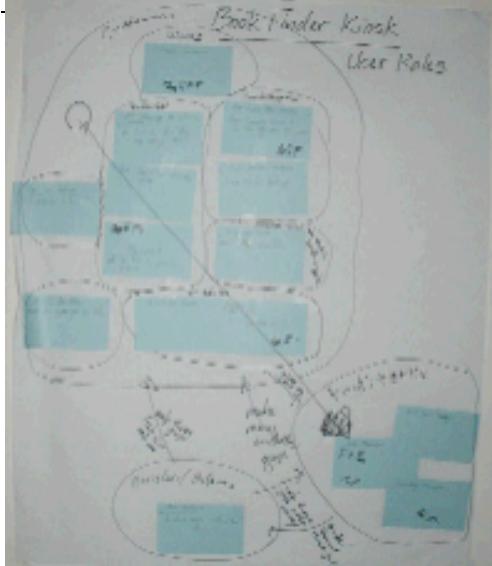
Step 3. Construct the User Role Model

In this step you understand and mark the relationships, dependencies, and collaborations between roles. The result is the agile version of a *role model*.

Using a sheet of poster paper, the participants will cluster the user role cards on the table. The only important instruction to give is, "Try to cluster. Keep roles similar to each other close together, dissimilar roles farther apart." Once the model "feels" right, fix the cards to the poster paper with tape.

The roles probably got clustered together because they have similar goals or participate in the business process at similar times. Circle each cluster of roles. Label each with some indication of why those roles got clustered.

Draw a line from each cluster to other clusters where a relationship exists. I might, for example, draw a line from a cluster marked "order takers" to one marked "order fulfillers" and label it "sends orders to."



Make any other notes of interest on the model, possibly including a notable business rule, user role characteristic, or relationship across user roles.

Figure 3-12. A sample role-model. To those present during its creation, the marked-up sheet brings back a flood of conversation and information. (Thanks to Jeff Patton)

Step 4. Identify the *Focal Roles*.

Participants now vote for the roles (not role clusters!) they consider being most critical to the success of the software (including also the roles where dire consequences follow when they fail in their goals). I usually only allow folks with either domain expertise or strong and educated opinions to vote.

To get the focal roles, I may give five voters 3 votes each. Since there's usually candy (e.g., Hershey's kisses) on the table during our sessions, I like to use those as voting tokens. Have people place the tokens directly on the role cards they find most significant. Doing this with physical markers as opposed to just pen marks is helpful to people making choices. I often find people who've used up their votes will begin to lobby others to vote as they did, or vote for the role they didn't because they ran out of votes. These are interesting conversations. I often see perhaps three or five or so roles picked out as focal from 15 to 25 roles on the table.

After everyone is done, replace the physical tokens with marks such as an 'F' for each vote. Write the 'F' using a bright colored pen. This makes it easy to find the focal roles in a busy role model. Roles with the most 'F's are easy to spot. I remind folks to pay close attention to these ones.

You are likely to make interesting discoveries at this point, for example, that the most-voted on, or focal, roles may not be the ones with the highest business value or have the highest usage frequency. Discuss your results. Make any interesting notes directly on the model.

Step 5. Capture the tasks required to accomplish the goals.

Using the role model as reference, think of a person going to the system to perform a task to meet their goal in that particular role. Write that task name onto an index card. I like task names that start with a verb, such as "add items to order" or "choose

customer from known customers." Just as with role names, the task names are important. The more clear and concise the tasks names, the less supporting documentation you'll need for them.

"Card-storm" the user tasks using this thinking activity, covering all roles. When there's likely to be a large number of tasks, I like to work on one cluster of roles at a time. Remove duplicates and clarify task names.

For each task card, note directly on the card three things:

- The goal of this task. What is a successful outcome for the task?
- Its frequency. You can use high, medium, or low, or time references like hourly, daily, weekly, monthly, or yearly.
- Its business value. This is a subjective best guess, high, medium, or low.

Step 6. Build the Task Model

In this step, you will identify relationships and dependencies tasks have with each other, just as you did with the role model.

Arrange the task cards on a fresh sheet of poster paper. Without having to match the role clustering on the role model, cluster similar tasks and let dissimilar tasks fall far apart. When the model feels right, fix the cards in place with tape.

As with the role model, circle and label clusters of cards, then look for relationships between clusters. Draw and label relationship lines between the clusters.



Figure 3-13.
Marking up the
task model.
Participants often
invent notation
meaningful to the
domain and to
themselves.
(Thanks to Jeff
Patton and Tomax)

See if you can spot a 'timeline.' Look for the task likely to be performed earliest. Start there and draw a line connecting each subsequent task until you arrive at the task likely to be performed last. I usually find the line runs from the upper left side of the model down to the

lower right – but not always. The timeline helps to point out the overall workflow from task to task.

The task model should be a concise picture of the work the system needs to perform.

Step 7. Identify the *Focal Tasks*

You need to understand and mark the tasks that are most critical to the success of the software. Just as with the role model, everyone gets three votes, or perhaps more if the task model has a large number of cards.

Have people vote for the tasks most critical to the software. What tasks deliver the most value? What tasks are done the most frequently? Alternatively, think of the tasks that if not done, or done incorrectly can cause the most trouble. You can use the candy again if it has not all been eaten by now.

Write an 'F' for each vote directly on the cards. As with the role model, use a brightly colored pen to write the 'F's on the cards. Those are your *focal* tasks.

Look for interesting discoveries. Are there high frequency tasks with high business value that weren't voted as focal? Why? Are there low frequency low value task that were focal? Focal roles often perform focal tasks. Is this true of your models? Interesting details arise out of discussing these points.

Step 8. Extract Interaction Contexts

An interaction context is a place in the software where a user might go to perform some of their tasks. It's the designer's job to find those places, organize them appropriately, and then put the right tools there so the task execution goes smoothly.

For example, if I were designing for a project called "my house", I might have some tasks called "put away groceries", "make dinner", and "listen to radio." I might decide those three tasks belong together in an interaction context I'll call "kitchen." In my kitchen I'll put tools I need to perform the tasks: a refrigerator, a stove, pots and pans, and a radio. You'll find interaction contexts in the software you use every day.

You'll find that the clusters in your task model are likely to be interaction contexts. The tasks clustered together because they were similar for some reason; often because they're performed at similar times by similar user roles.

Name each interaction context and write the name on a 3x5 card. Arrange these cards on yet another sheet of poster paper. Fix them with tape when the arrangement feels right. Draw lines between each interaction context card to indicate how a user might navigate from one context to another. This is a *navigation map*.

At this point you should start to be able to visualize the software. A well named interaction context becomes a suitable module name and a good way to refer to the tasks performed there (just as when I say "kitchen" you can quickly imagine the kinds of tasks I might do there). You may find it useful to index sections of the project plan

by interaction context. It's easier to explain that you are working on the "order processing" functions than it is to itemize all the tasks individually.

Step 9. Identify an Incremental Release Plan

You have in front of you a set of poster sheets that show you the roles that will use the system and the functions the system needs to provide them. What you still need to discover is the order in which to deliver them to the user community. You may be making multiple deliveries, in which case you want to deliver the functions in a *useful* order. If you are only going to make a single delivery, then you still want to develop and integrate the functions in the same order, for protection against the worst case situation, that you run out of time before getting to everything.

My [Jeff writing here] preferred order is "highest composite business value." I use that phrase because sometimes, to deliver a complete task thread, end-to-end, you actually have to develop and integrate one or more tasks that have lower individual value. If you don't include those tasks, then the users won't actually get full value from the most valuable (focal) tasks. We want to deliver full business value, the composite value of the entire thread of tasks along a work flow.

The shortest, most useful path is what I call a "system span²⁷." Develop this first, as an *Early Victory* to the users and what they will consider the *Walking Skeleton* of the system functionality. Subsequent releases will incrementally add features to that system span.

To build the incremental release plan, make a copy of the task cards. You can hand copy all the task cards. I find that keying them into a spreadsheet then printing them onto cards using a word processor works well. I do this during a break or at lunch that same day.

Place each task card on a (fresh) poster sheet according to its time and criticality.

- Time runs from left to right, referring to when a user role might perform that particular task.
- Business criticality runs top (most important) to bottom (least important). Criticality refers to how necessary the task is to perform (optional tasks will never be on the top row.)

You'll find the task cards start to arrange themselves in rows. The top row will contain critical tasks arranged in time order. Each subsequent row will contain other, less critical tasks, also in time order. Draw a line directly under the top row. This is the system span, the smallest set of features that can be released to perform an end-to-end business process flow.

²⁷ Borrowed from Lean Software Development (Poppendieck 2003).

Finally, it's the developers' turn. Have the developers present give rough development time estimates on each task card²⁸. Course-grain estimates are fine at this point. I often restrict the estimates to one week at a minimum, 3 or 4 at a maximum. Draw lines left to right across the model where you'd like to group tasks for future releases. What you have here isn't the project schedule, but a way of identifying similarly sized clusters of function to develop in each iteration, useful information for building the *Project Map*.

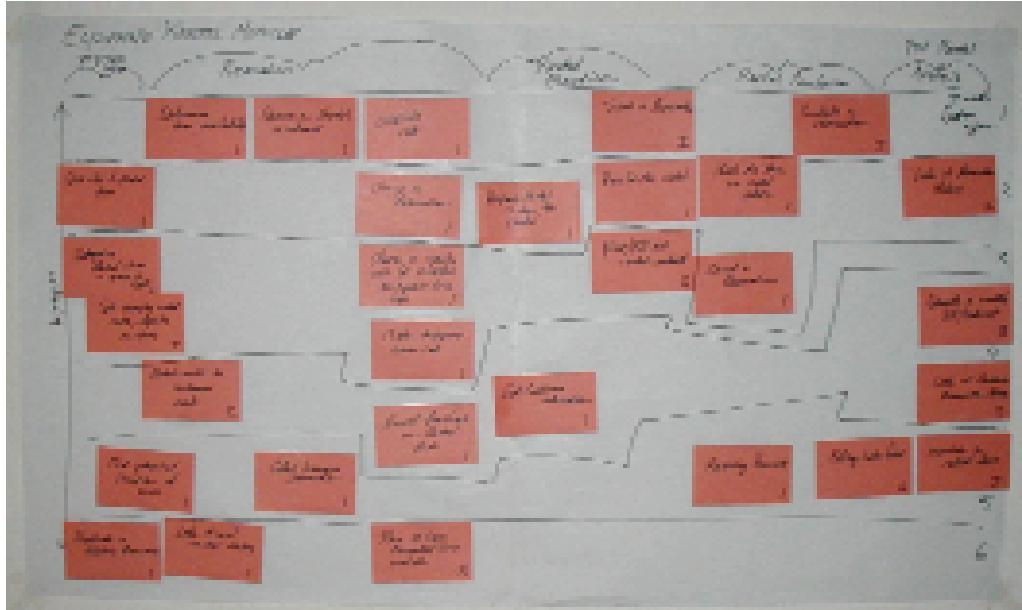


Figure 3-14. A span plan, which lets you see both the workflow and possible releases at a glance. (Thanks to Jeff Patton)

This concludes the session, exhausting but fun work that strengthens relationships between the users, sponsors and developers.

Be sure to place the flipcharts prominently in the development area as information radiators, constantly reminding the developers of the product features, the focal roles and focal tasks, and the conversations around how they came to be. This allows the developers to see the software through the eyes of those user roles rather than through the eyes of a project manager, developer, or tester.

* * *

Deriving the UI

²⁸ Notes from Alistair: First, these should be seen as relative sizing estimates, not elapsed time estimates. It can be damaging to commit to a development schedule so quickly. I would recheck these numbers later using *Blitz Planning* or a similar technique. You are likely to find additional tasks to be done that weren't noticed during the rush of this workshop.

When it comes time to develop the UI prototype, consider a technique described by Constantine, Windl, Noble, and Lockwood²⁹, which I [Jeff] also practice and have slightly modified to suit my personal style.

Do the following for each user task, on poster sheets:

- Write in two columns a synopsis of the dialog between the user and the system. Put the *User Intentions* in the left column and the corresponding *System Responsibilities* in the right. This is what Constantine calls an “essential use-case” (Constantine 1999???).
- Place sticky notes next to the user intentions and system responsibilities that will be UI elements. Put those for input areas and user actions on the left side and those for to information containers on the right side.

For example, a user intention such as “user identifies self” suggests two input areas and one action (name, password, some sort of “GO” button). Write “username” and “password” on two sticky notes. Write “action” on a third, or just use one with a different size or color to indicate its presence. Stick all three next to the phrase “user identifies self.”

For a phrase such as “system displays open orders,” you want a component showing a list of orders. Write “list of orders” on a sticky note and put it next to the phrase “system displays open orders.”

When you’re done, the use case should be spotted with sticky notes.

- Transfer the sticky notes to another flipchart, representing screens the user will see, placing them in areas that seem appropriate for the user-interface of the system. When you’re comfortable with the placement, replace each sticky note with a simple line drawing of the component drawn approximately the size you’d expect the component to appear on the user interface. Do this with each component. You now have what is called a “wire-frame” drawing of the user interface.

Test the wire-frame user interface by referring back to the use case. One participant plays a user role performing the task, the other plays the role of the system. Make sure the user interface you’ve designed supports easy performance of the task.

* * *

Usability Inspection (during Design)

It is time to inspect your (running) design.

Project the running software onto a wall big enough for a group of participants to easily see it.

Choose a participant to perform the user role that will be using the part of the software being inspected. I prefer someone not familiar with the functionality. Give that person a task or list of tasks to perform, chosen from the task model you drew at the start of the project or from the essential use cases.

²⁹ <http://www.foruse.com/articles/canonical.htm>

The person performing the task explains out loud what she is doing. As the system displays information explain you can explain what they're seeing. Everyone in the room notes defects or suggested changes on index cards.

After the tasks have been performed on the system, collect the cards. Eliminate duplicates, clarify issues, and prioritize the issues to be addressed by the development team. (Don't be depressed if you find dozens of issues on your first pass, that's normal. Just make sure you address the issues.)

Repeat this process as necessary until the user interactions when executing the tasks move smoothly.

* * *

QA Testing the System Personalities

Finally, you will need to confirm that the finished software is appropriate for the user role that will be using it. We often have a QA person pretend to be a user in the part of a role indicated by the role model.

For each role in the role model, assume the knowledge and goals of that role. Use the role description and any other information you can gather about the people who might step into that role. Method acting skills come in handy here.

Perform all the tasks assigned to that role using the running system, and write down, as though you were a person in that role, any problems you feel the person will have.

You may find you perform the same task many times. Each time, though, you'll be doing it from a different user role's perspective, and evaluating it differently.

* * *

Jeff summarizes:

User-centric design leverages information about users' traits and goals throughout the development process and ultimately validates the system against that information. I've found this generally results in easier to user software that the actual end-users are happier with. I've found we generally have far less rework on finished features and we discover fewer unanticipated features late in development.

Technique 7. Process Miniature

Any new process is unfamiliar and perplexing. The longer the process duration, the longer before new team members understand how the various parts of the process fit with each other. You can speed this understanding by shrinking the time taken by the process, using the *Process Miniature*. Here are three example of its use:

A large organization, using a methodology much larger than Clear, put each new employee through a one-week project. By the end of that project, the new employee had exercised every part of the process. He or she could then work on a real project knowing the connection between all the process activities and work products.

"Extreme hour" was invented by Peter Merel to introduce people to XP in 60 minutes³⁰. The group runs two half-hour iterations: 10 minutes for the planning game, 15 minutes to develop a solution and acceptance tests, five minutes for acceptance testing. The group runs through the process twice so they can experience reducing project scope partway through an iteration, pushing features from one iteration to the next iteration, and adding and shifting requirements within and across iterations. To make development time work in 15 minutes, the team designs something arbitrary, such as a mousetrap or a fishing device, and only draws their design on overhead transparencies.

For a small, 50-person company with 16 programmers, I once ran a 90-minute *Process Miniature* that required programming, test creation, checkin-checkout and integration across three architectural tiers. We used two 45-minute iterations and a very, very simple programming problem: an up-down counter run through a web interface. It took us two tries to get through the problem in the 90 minutes. The first try we used as a practice session to understand what was being requested. The second try was done live in front of the entire company as a demo of the working method.

You may wish to introduce Crystal Clear by using a *Process Miniature* somewhere between 90 minutes and one day. This might be done before the initial methodology shaping workshop, or just after, so the team can "taste" their new methodology. The methodology shaping workshop itself can be sampled by running it in a very digested form lasting only half hour, just so the team can learn how it works on a "safer" topic than their project.

³⁰ <http://c2.com/cgi/wiki?ExtremeHour>

Many techniques can be introduced using a *Process Miniature* to reduce the problem that people have with starting to use an unfamiliar process. You might run your first *Reflection Workshop* in just 15 minutes as a *Process Miniature*. I run a *Process Miniature* for writing use cases in my use case course so that people can see the overall picture before we get into details.

Technique 8. Side-by-Side Programming

"Programming in pairs" involves two people working on one programming assignment, at a single workstation³¹. Some people find this to be too much togetherness. Side-by-side programming allows them to get some of the effects of pair programming without giving up their individual (programming or other) assignments.

In side-by-side programming, two people sit close enough to see each others' screens easily, but work on their own assignments. This is an amplification of **Osmotic Communication** for the programming context.

In the photo below, Justin and Andrew have set the workstations so that the monitors are just two feet apart. The idea is that Justin should only have to turn his head or lean over a bit to see Andrew's screen. Then, they can work on their own respective assignments, but each can ask the other at any moment to look at a piece of code, run a test, or similar.



Figure 3-15. Side-by-side programming. (Thanks to Jeff, Justin, Andrew and Tomax)

Studies have shown that design- and code-reviews are cost-effective ways to reduce defects in the software (McConnell ref???). People find it hard to find the energy and interest to either call a code-review meeting or attend one. Being right next to each other, however, these two people can examine

and comment on small amounts of code in short amounts of time, with little ceremony or disruption to their work. It provides the "peer code peering" described by Crystal in the first of the emails at the start of the book.

The developers who first told me of this method said it allowed them to work in parallel on their programming, so they were not both occupied by the same task. At the same time, they could review each other's code as called for by the complexity of the code. They were programming a client-server system at the time, one working on the client, the other on the server. One could say to the other, "I've got the stub compiled. Can you test it?" The other would run the system, and then return to his programming while the first person fixed whatever bug showed up in the test.

³¹ See *Pair Programming Illuminated* (Williams 2002).

Another senior developer commented that he spent most of the day writing reports and managing conversations, rather than programming. Having his sit just two feet away from another programmer meant that he could be useful as a coach and second pair of eyes without having to give up his non-programming work.

Jim Coplien has speculated that full-time pair programming is not the optimal amount of time for people to spend together³². Side-by-side programming allows people to mix and match the time they spend on the same task and on separate tasks.

As with many of the ideas in Crystal Clear, you may find side-by-side programming to be a useful stepping stone to doing Extreme Programming, or as a fall-back in case people do not take to the full XP practice.

³² Personal communication.

Technique 9. Burn Charts

Burn charts have become a favorite way to give visibility into a project's progress. They are extremely simple and astonishingly powerful. They reveal the strategy being used, show the progress made against predictions, and open the door to discussions about how best to proceed, including the difficult discussions about whether to cut scope or extend the schedule. They have a natural mapping to the earned value charts used in military/government projects. They should part of your standard bag of tricks for project planning and reporting.

It turns out that packing a house has striking similarities to software development as far as planning and tracking go. I use the following exercise to illustrate the use of burn charts:

Imagine you have 30 days to pack up a house you are living in (with a couple of kids, just to make it more painful). Assume there are 14 rooms (or room equivalents) on 3 floors. Chart your plan so that you always know what is done, what is left, and how well you are doing.

Most people plan to pack the house as shown in Figure 3-16. They plan to first throw out all the junk, then pack the non-critical items, and pack the essential living items in the last few days. This is, of course, the technique I used the first two times I was in this situation, and I can speak with first-hand pain in saying that it is terrible.

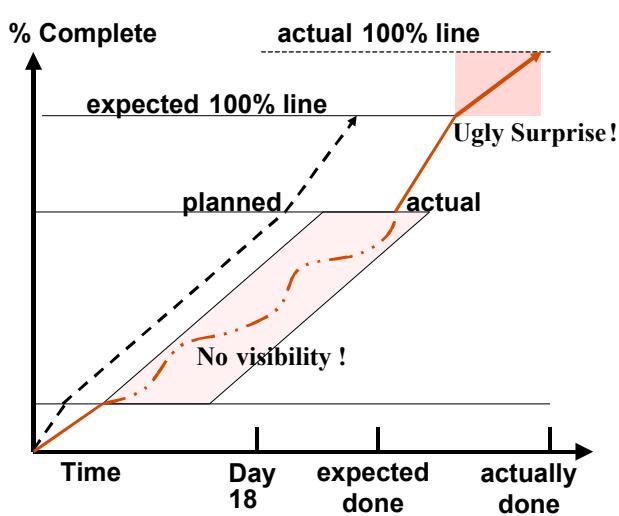


Figure 3-16. Scheduling with no definitive milestones.

The big flaw in it is the absence of clear intermediate milestones.

It is practically impossible to tell whether one is 60% or 70% done (speaking to both software and room-packing). For most of the project, one sees neither the true size of the task at hand, nor the (slower than expected) rate of progress. Missing both of those, it is impossible to tell when one will get

done. In both cases, an ugly surprise awaits at the end, when a myriad little unexpected items suddenly become visible (think about when you loaded all the boxes into a moving truck and walked back into the house, only to discover all sorts of things that suddenly "appeared" when everything else was taken out). The classical result on

software projects is that programmers keep reporting that things are 80% complete, or "We'll be done when we're done."

The repair is to find "interesting" and clear intermediate milestones.

In packing, those milestones are rooms completely packed and empty, containing "not even a sock" (and possibly with police tape across the doorway to keep people from dumping stuff back into them!). The thinking is that if there is so much as a sock left, then there is possibly something else besides the sock, and the reporting is flawed. We will see an application of this "sock" idea in just a few pages.

In software, those milestones are the delivery of running, tested code, as described in **Frequent Delivery**.

This new strategy has implications that make life a little awkward (again, both for software and house packing). In house packing, people have to move out of their rooms, and eventually the kitchen becomes unusable. In software, tests have to be re-run and designs and documentation must be altered on each delivery. These are not free – cost and discomfort both come with the strategy.

The payback for the cost and the discomfort is improved visibility, improved tracking, and less likelihood of an unexpected project overrun. In most cases, this benefit is well worth the cost.

Let's look at this strategy on a software project.

- Make a list of all the items to be delivered. You might work from the *Project Map*, *Blitz Planning* or XP's planning game. Associate with each delivery item a *relative cost*. Try to associate with each item also a relative *business benefit*, so that you can discuss with the *Executive Sponsor* the value being delivered to the business over time (Figure 3-17 shows a fragment of such a list). The reason for using relative estimates instead of absolute ones is that if you are, for example, 10% over on your first set of items, all the remaining items will scale accordingly. The burn charts will show this automatically.
- Sort and sequence the work items by development dependency, cost and value, and cluster them into a sequence, as shown in Figure 3-18 (you may recognize the similarity of what we are doing here and what happens in a *Blitz Planning* session).
- Estimate how much can be accomplished in each iteration or delivery period. Draw a line under the last item that fits in each. Number the releases.

I have come to call this the *Iceberg List*. The "above water" part lists all the items that can be delivered in the current delivery cycle. The "below water" part lists every that will be delivered in later delivery cycles. The name reflects that when you add a new item to the above water part, it pushes everything else down, and something that was above water falls below water (will not be delivered in the current delivery cycle). The *Iceberg List* is particularly useful on projects where the requirements change frequently.

In Scrum terminology, the below water part is referred to the *Product Backlog* and the above water part is the *Sprint Backlog*³³.

The primary difference between this list and the standard systems engineering work-breakdown structure is that *you get no credits for any item that does not result in running, tested code*. OK, you also get credit for *final deliverables* such as training materials and delivery documentation.

³³ More about Scrum's backlogs is online at <http://wiki.scrums.org/index.cgi>

	Feature	Feature Description	Value.	Est ideal days	Est. Elapsed Days
1	User Interface Configuration generates order line item comments	Allow the configuration to populate order line item comments with detailed configuration information along with configuration comments. Stop change of line item notes other than from within the appropriate configurator.	M	8 2	5
1.2	Configrator UI Rework: Verbose wizard style	Rework user-interface to match the prototye discussed in collaborative review session. Fast, simple data entry. No unnecessary frills.	H	6	15
1.3	Configuration generates customer specific pricing	Configuration is passed current order customer for use in determining customer specific pricing.	H	5	12.5
1.4	Allow adding a configuration and continue	Currently a configured blind may be added to an order. To add a subsequent similar blind you'd need to go back into configurator and reenter all the information. This feature would allow the user to accept a finished configuration and immediately allow adding an additional configuration using the data from the previous configuration.	L	2	5
2	Special Order PO Generation			5	
2.1	PO generated correctly for configuration	Currently POs are generated for regular order line items. Change to allow configuration components to generate 1 or more purchase orders at a cost furnished by the component.	H	5	12.5
2.2	Create/Confirm vendor items exist for skus	confirm that a vendor item exists for each sku used for base window treatments and options. Where no vendor item exists for an assigned vendor, dynamically create the vendor item.	H	1.5	3.75
2.3	PO generation is automatic	Currently Pos can be generated after a special order is saved and a deposit is made. Allow automatic PO generation immideately after deposit is tendered, or immideately if no deposit is required.	L	4	10
3	Advanced Order			9	
3.1	Advanced Order form shows more details	Alter Advanced Order form to show an "overflow" are for each line item where additional information such as comments, Po number, PO order date etc.. Can be shown.	M	4	10

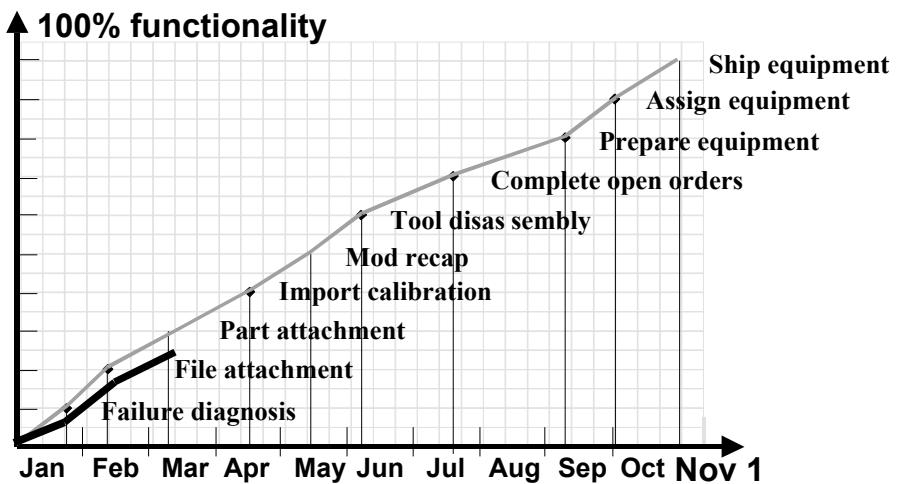
Figure 3-17. Extract of feature list with estimates of size and business value (Thanks to Jeff Patton)

Module	Feature Name	Value	Raw Dev. Time (ideal days)	Estimated Elapsed Days	release #
1.1	Configuration generates order line item comments	M	2	5	1
1.2	Configurator UI Rework: Verbose wizard style	H	6	15	1
2.1	PO generated correctly for configuration	H	5	12.5	1
2.2	Create/Confirm vendor items exist for skus	H	1.5	3.75	1
3.1	Advanced Order form shows more details	M	4	10	1
3.2	Order fulfilled at PO cost	H	2	5	1
3.3	Repeat orders works with blind configurations	M	3	7.5	1
3.4	Configuration comments are viewable, not editable	L	1	2.5	1
4.1	Base hierarchy change	H	1.5	3.75	1
4.2	Style can locate price charts based on color selection	H	2	5	1
4.3	Assign specific color group colors to price charts	H	2	5	1
4.4	Separate messages from options	H	2	5	1
4.5	Separate questions from options	H	2	5	1
1.3	Configuration generates customer specific pricing	H	5	12.5	2
1.4	Allow adding a configuration and continue	L	2	5	2
2.3	PO generation is automatic	L	4	10	2
3.5	Order shows sq. footage & running length	L	2	5	2
3.6	Vendor orderable items show additional detail	L	2	5	2
3.7	Printed window treatments show disclaimers	M	6	15	2
3.8	Allow order line item copying	H	2	5	2
4.1	Associate style with Retail.net DCL	L	1	2.5	2
4.11	Grid validation	L	1	2.5	2
4.6	Assign customer specific selling discounts	H	3	7.5	2
4.7	Customer lookup	H	1	2.5	2
4.8	Customer account type lookup	M	1	2.5	2
4.9	Options may have required questions	L	3	7.5	2

Figure 3-18. The *Iceberg List* with Product Backlog. (thanks to Jeff Patton)

- At this point you can make a *Burn-up* chart (Figure 3-19). Mark either relative work units or % complete on the vertical axis, and calendar time on the horizontal. Draw a line from the origin to the completion point, either by estimating the rate at which work can be accomplished, or as often happens, the "drop dead" delivery deadline. That shows the "planned" progress.

Figure 3-19. The *Burn-up* chart quickly shows progress and earned value. The long thin line shows the planned functionality and schedule, the short thick line shows progress to date.



After each iteration, mark how much relative work got completed. You will find that with just two marks on the chart you can see how far slower you are moving than you expected! Yes, that may be bad news, but at least it is *bad news detected early*, as opposed to *bad news detected too late to do anything about it!*

The burn-up chart resembles the classical earned-value chart (Lett 98, Fleming 88). The essential difference is only that people traditionally include on the earned-value chart *tasks completed*, whether or not they resulted in code getting integrated. On agile projects, credit is only given when code is integrated and tested (the "no leftover sock" rule). The agile strategy yields more reliable information about the project's actual state than the general one.

The *burn-up* chart has many nice features.

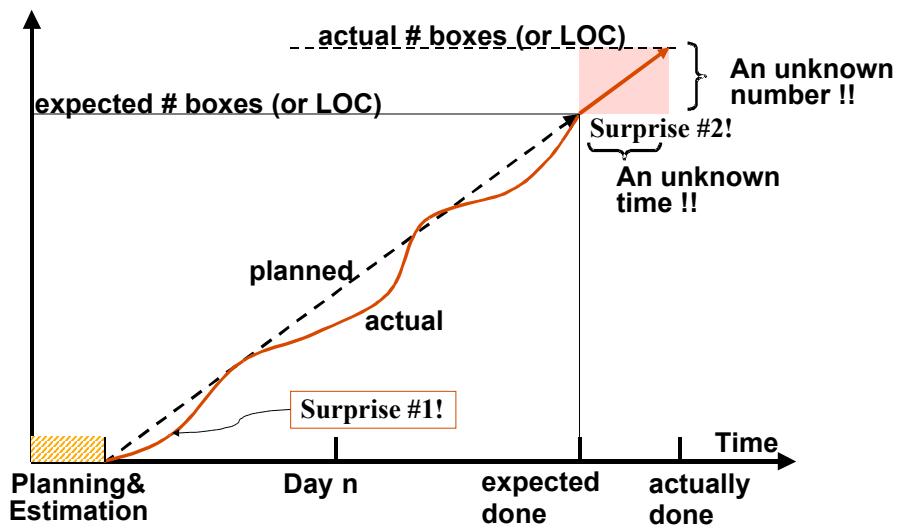
- It shows both the status and rate of progress ("velocity") in a way that is both clear and easy to discuss.
- If you plot business value on the vertical axis instead of development effort, you get a chart that properly shows how much value you have delivered over time. Plotting both on the same chart may help remind the team about where the real business value lies in their work. If you sequence the *Iceberg List* to float the greatest business value items to the top, and track business value delivered, then no matter when the project gets terminated, it will be clear to everyone that they got the most business value in the time spent.
- Finally, the burn-up chart is a key part of a good one-page project status report (examples of status reports are shown in the Word Products chapter). Such one-page status sheets simplify the work of executives looking at reports from many projects.

Even though the burn-up chart is very nice, there is a nasty surprise if you are not careful in choosing the units you use for the vertical axis.

In house packing, the obvious unit of estimation and tracking is the packing box. In programming, the obvious unit is the line of code (LOC). These are convenient and give wonderful detail.

The problem is that you won't know the actual number of boxes or lines of code needed until the very end (shades of "We'll be done when we're done."). The graph tracks *rate* wonderfully, but *progress* poorly. Not only will you not discover bad news early, but you will not know how many boxes or LOC you need until the last item is packed or programmed. That is too late.

Figure 3-20. A burn-up chart when the wrong units are measured.



The repair is to choose a unit of measure that can't possibly expand. In house packing, the replacement measure is easy: use rooms instead of boxes (it would be quite a surprise to discover a new room after you have packed all the rest). The answer is not generally obvious on a software project. You may be lucky and have a relatively stable number of use cases, or a fixed number of modules to replace, but there is no generally applicable answer.

I met two teams that had found an answer for themselves. They were doing system replacement projects where the subsystems communicated in a work-flow pattern. Their non-expanding unit was the work-flow component. These teams were able to color each subsystem yellow as they started work on it, and then green as it was completed. They were able to use "components replaced" as their non-expanding unit (see the sample status sheet in the Work Products chapter).

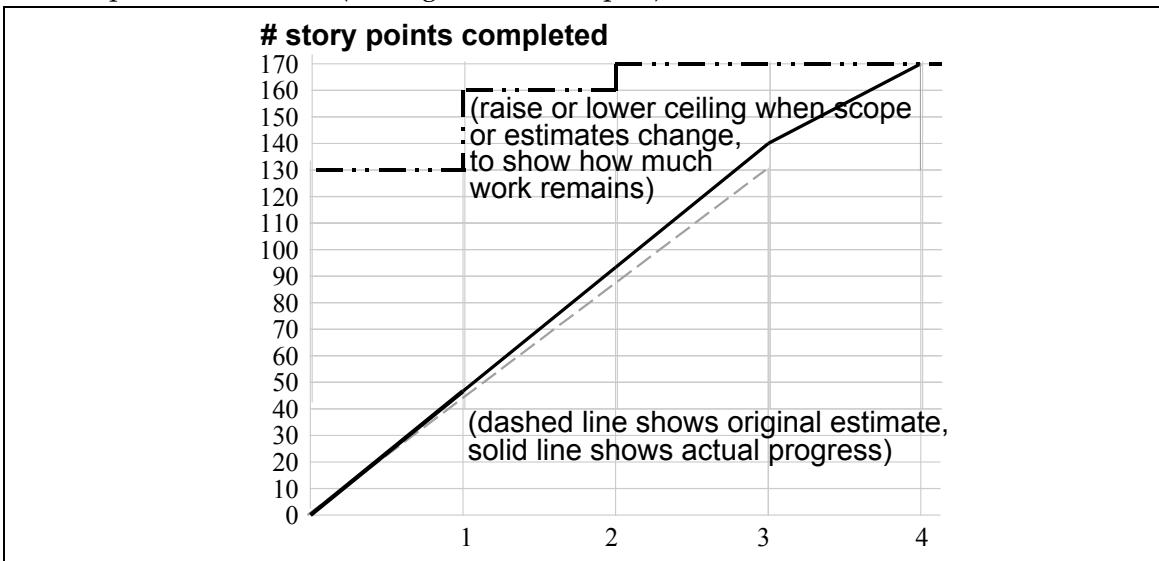


Figure 3-21. A *burn-up* chart showing predicted and actual accomplishments, and changing 100% line, using the Goodwin-Rufer style.

In the more usual situation, project scope does in fact change over time. Phil Goodwin and Russ Rufer introduced the idea of marking the raising and lowering of the ceiling, or 100% mark, as the project progresses. This charting allows the team to show the original plan and actual accomplishments even in the face of scope changes. The figure below show the Goodwin-Rufer style burn-up chart in a situation where the team actually performed better than their estimates but had to deliver increasing scope. (The project actually used the standard Scrum burn-down chart shown in Figure 3-24. I think you will agree that their conversations with the sponsors would have been easier had they used either Figure 3-21 or 3-26.)

If you are lucky enough to have a good non-expanding unit of measure, then you can use the *Burn-down* chart, which some people find emotionally more powerful.

The *burn-down* chart is emotionally powerful because there is a special feeling about hitting the number zero that helps people get excited about completing their work and pressing forward. I can speak first hand in saying that as soon as we shifted our house packing chart to using rooms as the unit of measure and put the rooms on a *burn-down* chart, our confidence that we would actually meet our schedule increased dramatically. I have seen the same apply to a software project team.

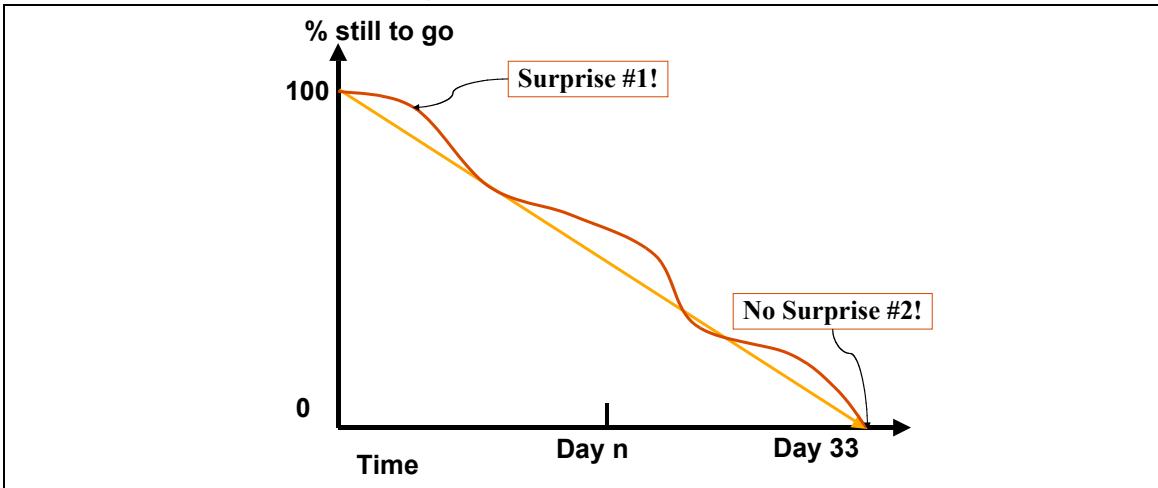


Figure 3-22. The Burn-down chart.

It is easy to say, "We can't use the burn-down chart, because nothing is fixed on a software project." Oddly, the first people to say this were those working on the system replacement project just mentioned. After a bit of examination, we discovered that they had exactly the house-packing problem in front of them. They were able to convert immediately to the burn-down chart, using replaced subsystems as their non-expanding unit of measure.

* * *

There are three wrinkles left to discuss.

The first is the little white lie I just told in the last paragraph about non-expanding unit of work. The team discovered after a short while that their *Executive Sponsor* kept adding demands for new functionality in both the old system and the replacement. In other words, although they were promised that the scope would be fixed (since the time and budget was), that wasn't really true. As if that were not bad enough, he was actually opposed to the project itself, and was obliged to support it only because of pressure from the user community (see how powerful user community support can be!).

This put the team in a damned-if-you-do-and-damned-if-you-don't position. They couldn't report that a subsystem was complete (not even a sock left in it) if he was going to keep throwing more socks back into the room, and they couldn't stop him from throwing more socks in.

Their eventual strategy has some bearing on burn charts, but also some bearing on the positive role of intermediate managers. The intermediate manager decided that they would take their project mission as originally stated, to replace the existing system. They would deliver that successfully, and track all increased scope as growing "post-replacement" activity.

They charted this as shown in Figure 3-23. With this strategy and chart, they were able to keep their **Focus** on the critical issue of replacing the existing system, show they were tracking the new functions being added, and keep the *Executive Sponsor* from derailing their project.

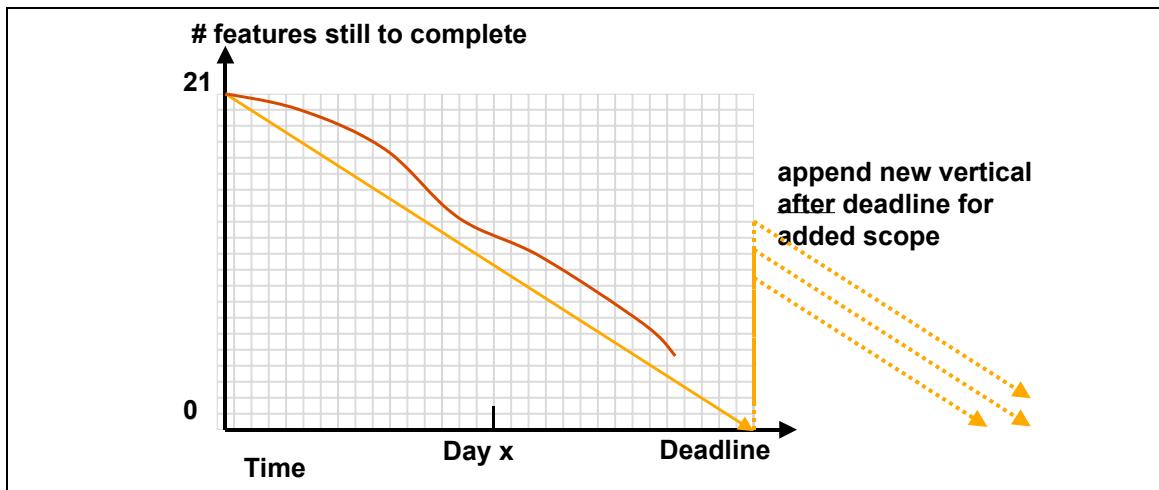


Figure 3-23. The *Burn-down* chart for a hard-deadline project with scope increase added at the end.

Mike Cohn reports a similar situation in his book *User Stories*, although in friendly circumstances. The team was given 130 story points (a relative unit of size for XP user stories) to deliver in three iterations. After each iteration, the sponsors added new stories and the team revised the relative sizes of the remaining stories. Both increased the estimate of the amount of work remaining. Mike and I discussed how they should chart the team's progress against changing scope, assuming they like the emotional power of the burn-down chart.

Figure 3-24 shows what the team showed the sponsors at their meetings (Cohn 2003). Mike and I felt that this chart conceals very important information. Figure 3-25 shows how I thought to modify the chart to illustrate the actual accomplishments against the changing scope, and Figure 3-26 shows how Mike proposed to modify the chart to show the changing situation. You may notice that 3-26 is the downward version of the Goodwin-Rufer burn-up chart.

We offer these examples of charts to show different ways in which you can show predicted and actual accomplishments with simplicity and expressive power. There are probably more variations to discover, but these should get you started.

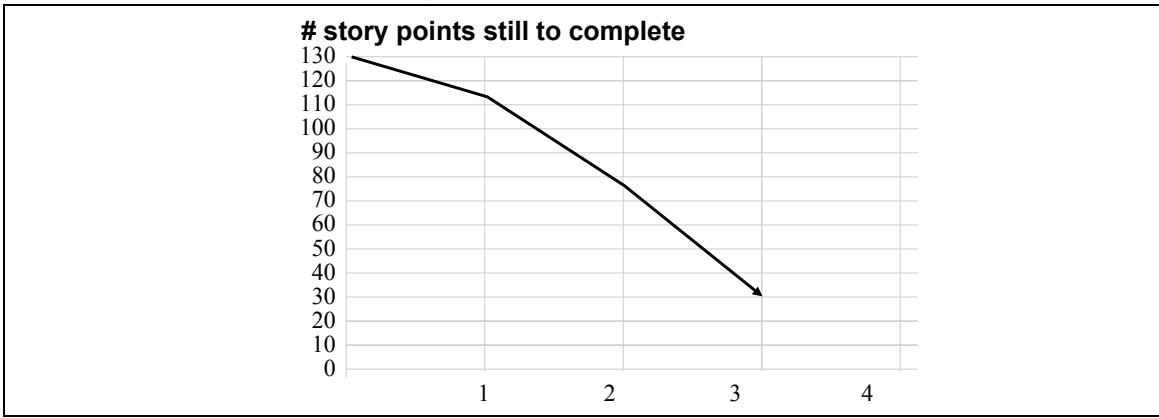


Figure 3-24. Scrum-style *burn-down* chart as shown to the project sponsors (after Cohn 2004).

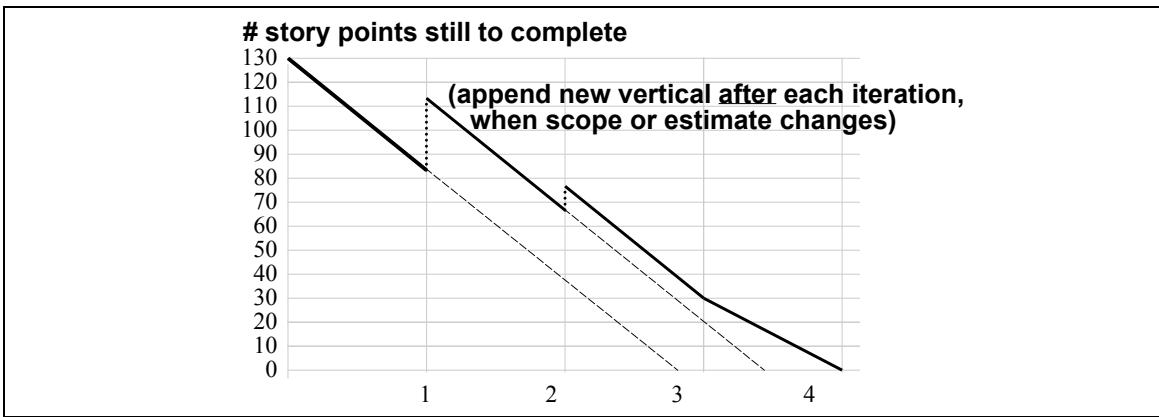


Figure 3-25. Burn-down chart for the same project showing scope increase after each iteration.

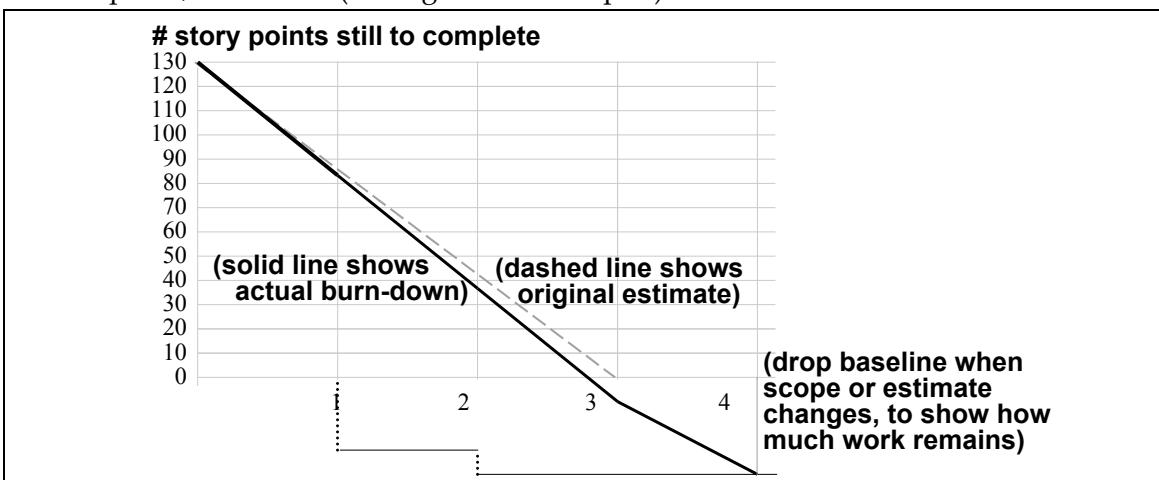


Figure 3-26. Burn-down chart for the same project using the 'down' version of the Goodwin-Rufer chart.

The second wrinkle to be discussed is the size of the selected unit of measure. This is very relevant at the start of a project because the team will almost always move more slowly at the beginning than expected, and just *how* much more slowly is a valuable piece of information.

Let us return to the house packing example to see this clearly. The graphs in Figure 3-27 show that the coarser the unit of measure, the longer the blackout period during which no information is available. Having already rejected "boxes packed" as a good unit of measure, we can choose "rooms packed" or "floors packed" as non-expanding units.

Figure 3-27 shows *burn-down* charts for both of those choices. In both cases, the people fall behind at the same rate. However, with floors as units, they don't discover how bad the news is until a third of the way through the project, which is a relatively long time. Using rooms as units, they discover how bad the news is after only a few days. The moral of the story is to choose the smaller non-expandable unit of measure.

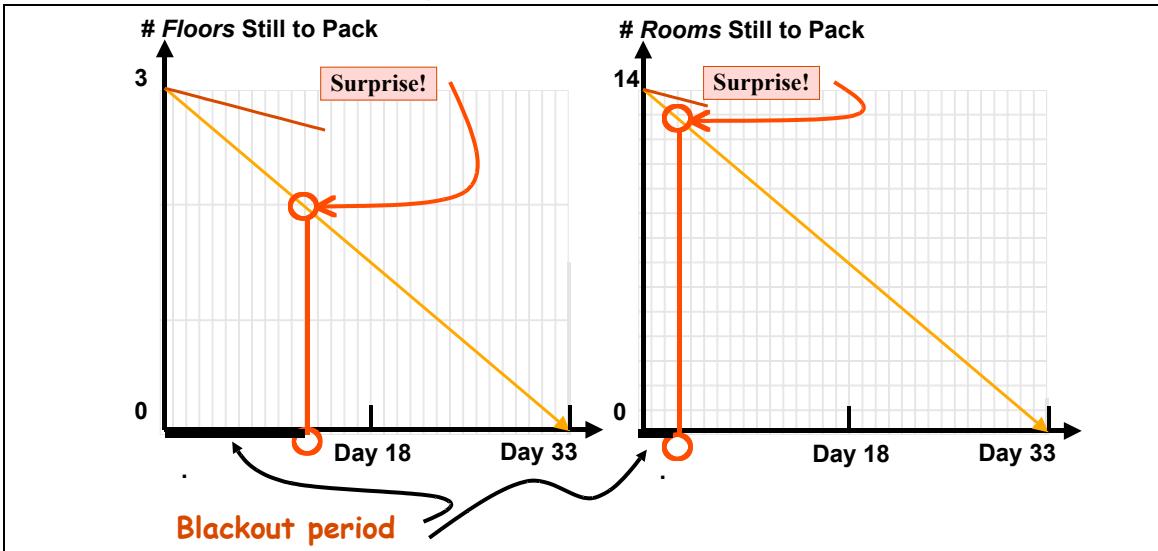


Figure 3-27. Burn-down charts for two different base units. The smaller the unit, the shorter the initial black-out period.

* * *

The final item to discuss is handling items that change priorities. This gets us back to the *Iceberg List*, which has shown great utility on projects in which the priorities change frequently and violently.

A large project was delivering value to a customer on a regular basis. The customer would change work assignments and priorities so often and so late that no project plan was meaningful. The customer was, however, happy with the software being delivered, so they kept requesting more work done.

In desperation, the project manager just put all requested features into a single prioritized list with estimated sizes. He announced that whatever was above the "water line," as calculated from the size estimates and team's velocity, would get delivered in the next release, and everything else would be delivered "later." The customer could insert, remove, modify or reprioritize features at any time, even affecting the current release. Without resorting to argument, all parties could calculate what would reach delivery during this cycle and what was pushed back.

The name *Iceberg List* comes from the rising and sinking effect the list shows: adding high-priority items to the top of the list causes some above-water items to sink below the water line; removing above-water items may cause a new item rise above the water line). Just as with an iceberg, the above-water portion is only a fraction of the total size.

The *Iceberg List* is useful in hostile environments, where the sponsors change the requirements and priorities on a daily basis and without warning. In such an

environment, post the current *Iceberg List* in a highly visible place, and make sure the contents of the next releases are *derived* rather than argued.

Module	Feature Name	Value	Raw Dev. Time (ideal days)	Revised ideal days estimate	release #
1.1	Configuration generates order line item comments	M	2	2.6	1
1.2	Configrator UI Rework: Verbose wizard style	H	6	7.8	1
2.1	PO generated correctly for configuration	H	5	6.5	1
2.2	Create/Confirm vendor items exist for skus	H	1.5	2	1
2.3	PO generation is automatic	H	4	5.2	2
3.1	Advanced Order form shows more details	M	4	5.2	1
3.2	Order fulfilled at PO cost	H	2	2.6	1
3.3	Repeat orders works with blind configurations	M	3	3.9	1
3.4	Configuration comments are viewable, not editable	L	1	1.3	1
4.1	Base hierarchy change	H	1.5	2	1
4.2	Style can locate price charts based on color selection	H	2	2.6	1
4.3	Assign specific color group colors to price charts	H	2	2.6	1
4.4	Separate messages from options	H	2	2.6	1
4.5	Separate questions from options	H	2	2.6	1
1.3	Configuration generates customer specific pricing	H	5	6.5	2
1.4	Allow adding a configuration and continue	L	2	2.6	2
3.5	Order shows sq. footage & running length	L	2	2.6	2
3.6	Vendor orderable items show additional detail	L	2	2.6	2
3.7	Printed window treatments show disclaimers	M	6	7.8	2
3.8	Allow order line item copying	H	2	2.6	2
4.1	Associate style with Retail.net DCL	L	1	1.3	2
4.11	Grid validation	L	1	1.3	2
4.6	Assign customer specific selling discounts	H	3	3.9	2
4.7	Customer lookup	H	1	1.3	2
4.8	Customer account type lookup	M	1	1.3	2
4.9	Options may have required questions	L	3	3.9	2

Figure 3-28. The *Iceberg List* from Figure 3-18 revised on the second iteration.

Figure 3-28 shows how this works, starting from the example in Figure 3-18. The size estimates have changed, and "automated PO generation" got moved above the water line. As a result, "separate messages/questions from options" fell below the water line and got deferred to a later release.

Reflection about the strategies and techniques

Crystal Clear states explicitly that no strategy or technique is mandatory. Why do I describe these ones and not others?

Both shaping the methodology and **Reflective Improvement** are required elements of Crystal Clear, even though no specific technique is mandated for them. Most people have never these things, and need to see a sample to get started.

The particular reflection workshop technique I describe is short, efficient and lets people both voice their thoughts and then get back to work. If you are doing any sort of monthly retrospective that allows free discussion, then you should be able to try different styles of workshops in successive months, and generate your own personal style within a few months.

I describe *Blitz Planning* as an alternative to XP's "planning game." The two differ in three ways:

- The planning game cards list *user stories*, and the *Blitz Planning* cards list *tasks*;
- The planning game has the people assume there are no dependencies between stories, while *Blitz Planning* has people analyze the dependencies between tasks;
- The planning game assumes fixed-length iterations, while *Blitz Planning* does not assume anything about iteration length.

There are times when these differences are of interest. *Blitz Planning* lets the team

- see the shape of the work ahead of them (producing the *Project Map*),
- work out where the *Walking Skeleton* should show up and what it might consist of,
- work out the smallest set of tasks before revenue can start flowing, and
- report to the sponsors the number of internal tasks that need to be performed before useful functionality becomes visible.

These differences are of particular importance at the start of the project. Later in the project, the team may be able to simply list the features to be developed and their relative development costs.

The *Process Miniature* lets people build a short internal movie of how a new process works. Since moving to Crystal Clear is likely to involve quite a lot of change in working habits, it is useful to reduce tension and uncertainty.

The *Exploratory 360°* gives people a chance to catch certain kinds of fatal mistakes, mistakes that quite possibly everyone (incorrectly) assumes someone else has already checked. Since the *Exploratory 360°* does not take very long, it has a high payback value for the time taken. It also aligns the team on the project's mission and approach.

Early Victory, Walking Skeleton and *Incremental Rearchitecture* fit together. Even quite experienced developers disagree about how much architecture to develop early in the project. These three strategies describe a way to take advantage of incremental

development to establish safety and value in the early stages of the project, and still deal with the mistakes and learning that form a natural part of architectural development.

Information Radiators are useful at every point in the project. Most agile teams make great use of them, but most teams outside the agile community do not take nearly enough advantage of these devices. It should be a pronounced strategy to use all the wall available for capturing information of use to the project team and their visitors (there is a danger of overloading the wall's visual space, but I have only encountered one team who had reached that point). Once you start using *Information Radiators* you may find you want to rearrange the office to make more walls available.

The *Daily Stand-up* meeting was introduced in the Scrum methodology. I have heard of people adding this single technique to every sort of project and methodology to good effect, and so I am happy to spread its use.

I include *Essential Interaction Design* because there is so little literature on how to work user-interface design and interaction design into an agile project. This adaptation of usage-centered design fits into both XP and Crystal Clear projects. You may want to study more of Jeff Patton's writings and the book by Lucy Lockwood and Larry Constantine to adapt and refine your use of it.

Burn charts are extremely useful in the planning stages of the project, because they show the workload so clearly and people can respond with comments about the plan's feasibility. They are also well suited as status reports, for the same reason: immediate clarity.

Crystal Clear lets a team use the techniques that work best for its members. I rather expect that simply as the normal course of professional activity, Crystal Clear practitioners will learn each of the techniques and strategies listed in this chapter, and uncover many more as they go.

Chapter 4

Explored (The Process)

Crystal Clear uses nested cyclic processes of various lengths: the development episode, the iteration, the delivery period, and the full project. What people do at any moment depends on where they are in each of the cycles. This chapter linearizes the cycles to the extent possible, and points out some of their interactions.

Most of the development processes described from 1970 through 2000 were described as sequences of steps, even when the process recommended iterations and increments. This always caused confusion among readers. The text appeared to dictate a "waterfall" process, while the authors kept asserting it was not.

These linear-looking process descriptions suffer from two problems. The first is that they really describe the dependency graph between work products, *work flow*. A dependency graph should be read in back-to-front order: The team cannot deliver the system until it is integrated and runs. They cannot integrate and test the code until it is written and running; They can't design and write the code until they learn what the requirements are (and so on). The work-flow graph does not mandate a waterfall process but simply reflects dependencies.

When a process is described by work-flow dependency graph, people mentally treat the dependency graph picture as encouragement to complete each mentioned work product before starting on the next work product in the graph. This is rarely a good strategy in software development. When looking at a work-flow dependency graph, the relevant questions to ask are these:

- *How many requirements, at what level of completion, are needed before design can usefully get started?* (The answers are "relatively few" and "relatively low", respectively.)
- *How much of the system must be designed and programmed before useful integration and testing can be done?* (The answer is "relatively little.")
- *How much of the system, at what level of correctness is needed before being useful to the users or suitable for user review?* (The answers are "relatively little" and "relatively low," respectively.)

The low levels of each actually needed comes as a surprise to many people.

Concurrent development is the name given when work proceeds in parallel on dependent work products. In concurrent development, many or all activities are in play at the same time. People trade notes continually to keep abreast of the changing state of each part. Concurrent development is not new, either to software development or project management in general³⁴. It is done by almost every project team I visit, although few have noticed they do it.

Concurrent development is described at length in *Agile Software Development*, in the *Gold Rush* strategy in *Surviving Object-Oriented Projects*, and is generally presupposed in most of my writings. Therefore, I won't try to describe concurrent development here, nor the dependency graph in Crystal Clear (that should be quite obvious). Removing these two topics allows me to describe other interesting aspects of the process.

³⁴ *Simultaneous Development* (??1997) contains an excellent description of overlapped phases in civil engineering projects.

The second problem plaguing linear descriptions of process is that the processes they are trying to describe are cyclic, using multiple cycles of differing lengths³⁵.

I notice seven cycles in play on most projects:

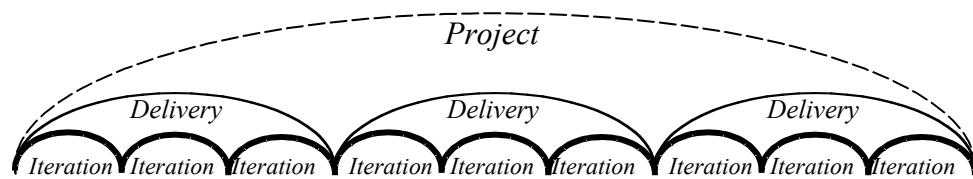
- The project (a unit of funding, which could be any duration)
- The delivery cycle (a unit of delivery, one week to three months)
- The iteration (a unit of estimation, development and celebration, one week to three months)
- The work week
- The integration period (a unit of development, integration and system testing, 30 minutes to three days)
- The work day
- The development *episode* (developing and checking in a section of code, taking a few minutes to a few hours)

Crystal Clear requires multiple deliveries per project, but not multiple iterations per delivery. I will repeat this point several times, but I wish to draw your attention to it right away.

Teams using short iteration periods inside a long delivery cycle sometimes add a "super-cycle" of perhaps four or six iterations, to provide an intermediate rhythm. They use this super-cycle to reflect on their process, to celebrate, to unwind. I will not elaborate much on the super-cycle in this chapter. Teams can derive what elements of the iteration and delivery cycles to move to the super-cycle.

Each cycle has its own sequencing, its own rhythm. In any given day, different activities will be in play from the various cycles. The activities change from hour to hour, day to day and week to week. This makes a complete linear description of the process virtually impossible.

The following figures show several ways of unrolling these cycles. These simple sketches have to omit some of the interactions between cycles, a few of which I'll point out shortly. Notice that episodes, days and integration periods can nest inside an iteration in various ways. Figures 4-2 and 4-3 show two possible ways. There are other valid combinations even more difficult to draw.



³⁵ I am indebted to Don Wells' description of XP as nested cycles. See <http://www.extremeprogramming.org/map/loops.html>

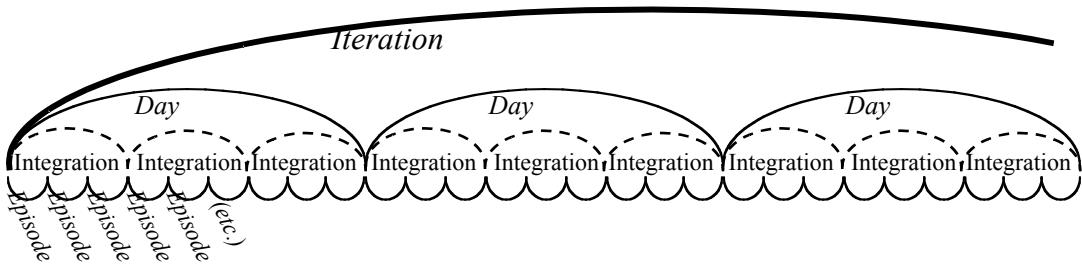
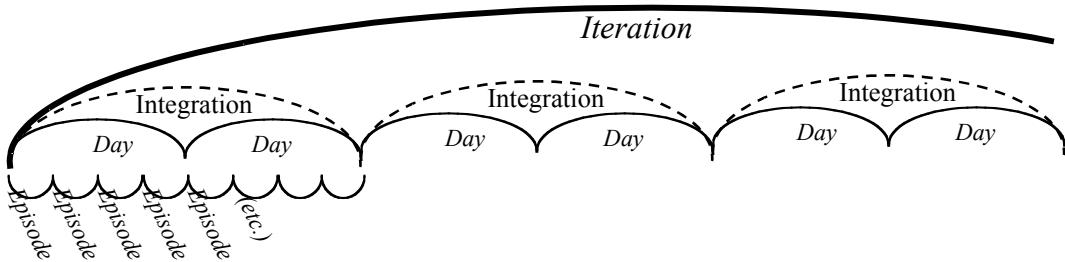
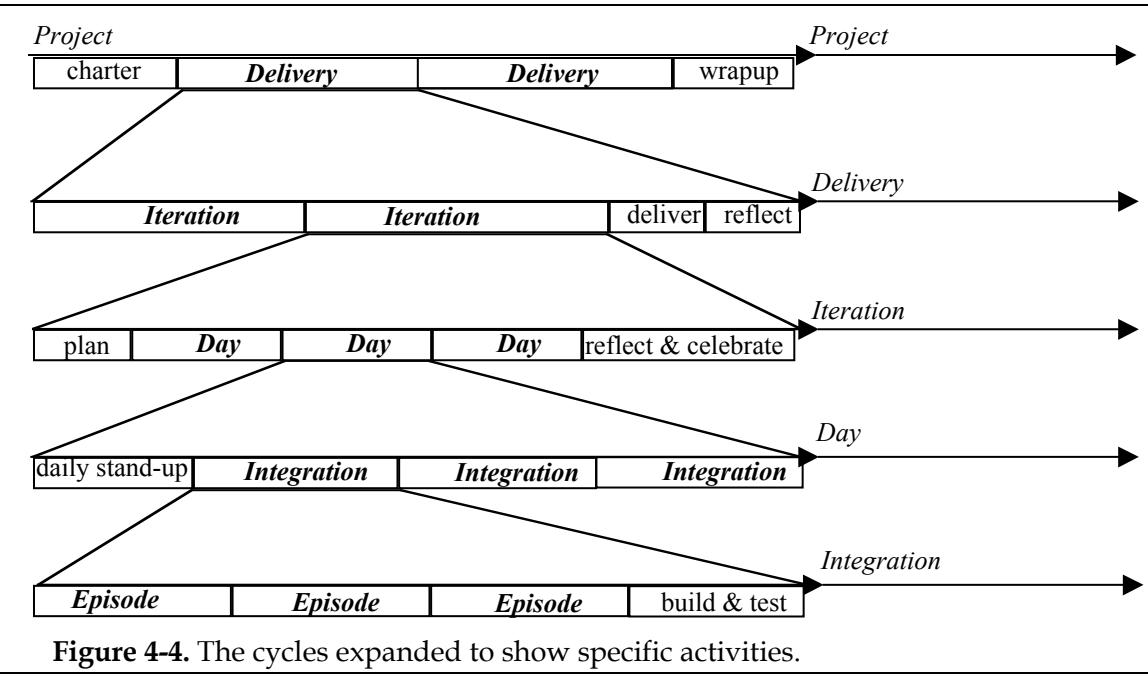
Figure 4-1. Iteration and delivery cycles within a project.**Figure 4-2.** An iteration cycle, integrating multiple times per day.**Figure 4-3.** An iteration cycle with multiple days per integration.

Figure 4-4 shows the expansion of each cycle separately, listing specific activities that occur specifically for that cycle type. Figure 4-5 shows these activities in a plausible time sequence (vertically down the page), placing each activity in a column for its cycle type.



Project	Iteration	Day	Integration	Episode
Charter				
	Plan	Daily standup	Design & Check-in Design & Check-in	
		Build and test	Design & Check-in Design & Check-in	
		Build and test	Design & Check-in Design & Check-in	
		Daily standup	Design & Check-in Design & Check-in	
		Build and test	Design & Check-in Design & Check-in	
	Deliver	Build and test	Design & Check-in Design & Check-in	
	Reflect and celebrate			
	Plan (etc.)			
Wrapup				

Figure 4-5. The activities in a plausible sequence vertically.

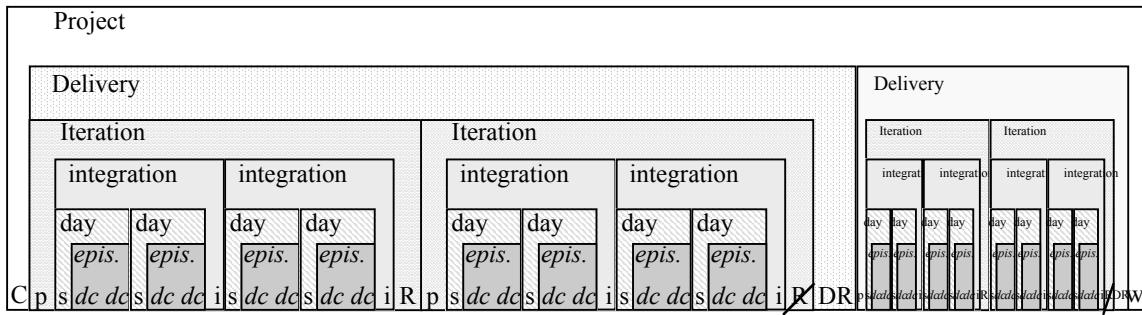


Figure 4-6. The cycles unrolled all the way to show daily activities.

Finally, in Figure 4-6, I have unrolled all the cycles and shaded the cycles in different ways so the activity can be connected to its cycle. The capital and small letters stand for project Chartering, iteration planning, daily standup, development, check-in, integration, Reflection workshop, Delivery, and project Wrapup. Of course, I had to choose one particular relationship between the daily cycle and the integration cycle for this particular unrolling.

People interested in detail will notice that there is a line drawn through the R at the end of the last iteration of a delivery cycle. That indicates that team does not "reflect, deliver, reflect again," but rather, more naturally waits until after the actual delivery, then reflects just once.

Another detail is that the planning session in the first iteration of a delivery cycle is likely to have two parts: adjusting the coarse-grained project plan and creating a fine-grained iteration plan³⁶.

These are the sorts of minor process adjustments that most teams make naturally, but make cyclic process so hard to describe explicitly.

The following sections expand on all but the weekly cycle.

³⁶ Thanks, Jeff Patton, for making this explicit to me.

The Project Cycle

A project as a cycle? Although each project is funded as a once-and-for-all activity, it is typically followed by another project, in a cycle that repeats. It is useful for the organization and the development teams to get used to that cycle.

A project cycle in Crystal Clear has three parts:

- a *chartering* activity,
- a series of **two or more** delivery cycles,
- a completion ritual: the project wrapup.

Chartering

The *chartering* activity takes a few days to a few weeks. It consists of four steps:

1. Build the core of the team,
2. Perform the *Exploratory 360°* (which may result in canceling the project),
3. Shape and fine-tune the methodology conventions,
4. Build the initial project plan.

Note that having only one delivery cycle is a violation of Crystal Clear (for some definition of the word "delivery"). A project large enough to need a methodology description will benefit from multiple delivery cycles. I have seen projects that the sponsor thought was small enough not to need multiple deliveries that did quite poorly for the simple lack of feedback that comes at the end of a delivery. In the rare other cases, the project was so small as to be a case of "just do it and go home." If it is not a matter of "just do it and go home," please arrange for at least two delivery cycles. Check the book sections on **Frequent Delivery** and **Delivery Cycle** to learn what constitutes an adequate delivery.

Chartering: Build the Team

A project of the type that will use Crystal Clear often starts with an *Executive Sponsor* and a *Lead Designer*, and eventually a key user.

Normally, two to five other people are added to the project, with a various mix of skills, experience and ability. The more novices on the team, the harder it is for the lead designer to both train and develop software. Therefore, if there are more than four people involved, the team should include at least one other experienced and competent developer, to back up and offload the lead. There should be an experienced and competent person for every two or three junior or novice people.

The Executive Sponsor

The *Executive Sponsor* is the person who ultimately cares that the project gets done. This person provides the money – or at least arranges for it to show up – and provides essential direction to the group. This person highlights the priorities the group needs to be aware of, and gets the project not only monetary support, but also logistical and emotional support. Often, the *Executive Sponsor* is also the domain expert.

During the project, the *Executive Sponsor* needs to remind the development team of their true goal and keep them apprised of any changes in business direction that affects the project's direction. In return, the team needs to keep the *Executive Sponsor* apprised of the project's progress (burn charts are excellent for this).

A hazard lies in wait for the project with a highly technical *Executive Sponsor*: oversteering. It often happens, particularly for small projects, that the project's *Executive Sponsor* was once a superb technical designer. This sort of person can't resist spelling out for the design team just how they *might* implement the system, *not that they have to*, of course. In some cases, this person is correct, but still gets in the way of the team. The former top designer who is sponsoring a project will have to accept that the team may not create quite as superb a design as he or she could, and live with that.

One of the ways to keep the former top designer from interfering too much in the design is to use properly written use cases for the system specifications (see *Writing Effective Use Cases*). Properly written use cases accurately specify the desired system behavior without specifying the design.

I have to mention that sometimes the *Executive Sponsor* does, in fact, know what is possible with the technology, or what is required, even if that appears difficult. In these cases, the best situation is for the sponsor and design team to develop a form of discussion that permits the sponsor to express what he or she knows without running over the team. After all, one of the primary contributions of the *Executive Sponsor* is team motivation.

The Lead Designer

The person given the assignment of *Lead Designer* has a complicated job description, which is not normally spelled out in advance. This person is often the best designer on the team, a person who could, in principle, design the entire system. This person is also supposed to train the younger members of the team, talk with the *Executive Sponsor* and end users, build a project plan, and manage the development of the project while also designing the most difficult part of the system. It is an impossible job description, except that it is carried out successfully by many (overworked) people around the world.

The best thing the *Executive Sponsor* can do for the *Lead Designer* is to provide a second expert developer. This second expert developer should be the one who handles the most difficult design tasks, because this second expert developer is not so likely to get caught up in meetings about users, plans, status, interfaces and so on.

A key hazard surrounding the *Lead Designer* is that this person can get burned out before the project is over. In one case, I met a *Lead Designer* who was already working 60 hours a week at the start of the project. His own project plan had him teaching Java to the new programmers, as well as doing the most difficult design tasks, as well as . . . well, you get the idea. Although common, this situation is unhealthy. By the time the project passes its halfway mark this person is likely to be working 90 hours a week and sleeping under the desk for a few hours each night. Sometime after that, a calamity of some sort is likely to hit the project, and still more time is needed from this person.

It was for just this reason that Jens Coldewey devised the project management risk reduction strategy I refer to as **Spare Leader Capacity**³⁷. That strategy roughly says, "Ensure the technical lead has some spare capacity to assist in emergencies." We know that emergency situations do arise on most projects; if the technical lead has no spare capacity, he or she can't work through the emergency. Starting off a project at 60 hours a week does not leave the technical leader with much spare capacity, and at 90 hours a week, that spare capacity is all gone.

In the case of the *Lead Designer* I just mentioned, we worked together to identify every task that someone else could possibly do, even if he was best suited for the task himself. By taking these loads off him (he would still have to set up or monitor them), we were able to reduce his work load to about 45-50 hours a week at the start of the project, so that he might be able to keep within about 60 hours during the high-load period of the project.

The Ambassador User³⁸

The system will be better suited to what the users need if the developers have access to an expert on system usage. The value the system brings to the organization is very sensitive to the information the developers get about the usage context. That means it is important that the team has easy access to proper usage experts.

How much of that user's time does the team need?

It would be nice to have the expert user available to the team the whole time, of course. That is rarely practical. Some methodologies suggest or require the user expert be present 50% of the time. That would also be wonderful, of course, but is usually also impractical.

My experience is that if the team can phone the *Ambassador User* periodically during the week to ask questions, and gets a few hours (even as low as two hours) of dedicated time each week, they can work through their questions, get good requirements information, show demos and get usage and design feedback of a quite adequate sort.

³⁷ XP uses the term "sustainable pace." While that is sound, I find Jens' strategy to be usefully more specific.

³⁸ I borrow the term from the DSDM methodology (DSDM Consortium 2003??).

It may be that the *Ambassador User* shows up each Tuesday afternoon for a 90-minute meeting, and can be phoned half a dozen times each week. This is usually enough to put the project into the safety zone. If the user cannot even commit to one hour a week, then the project is not in the safety zone, but actually is in some danger with respect to appropriateness of requirements and adequacy of user feedback.

The Rest of the Team

Crystal Clear only names the above three roles specifically. There will be other people on the project. How they divide up the rest of the project roles is up to them. In particular, someone, or several people, will have to capture the requirements in use case or user story form. Also, someone will have to act as coordinator. Perhaps the Lead Designer can off-load this onto someone else who has good communications skill.

Chartering: Perform the Exploratory 360°

At an early point in the project, the team – including the executive sponsor – does a once-around check of the key issues. This is the *Exploratory 360°* described earlier, or an equivalent, such as RUP's "inception" phase (Kruchten 2002??). In this check, they make sure the project won't founder on

- the business value,
- the requirements,
- the domain model,
- the technology to be used,
- the project plan,
- the team makeup, or
- the methodology or conventions to be used.

Each of these is reviewed in a coarse-grained fashion, to detect whether the intended team and projected methodology, using the intended technology around the projected domain model, can deliver the intended business value with the projected set of requirements according to the draft project plan. The Exploratory 360° results in a set of adjustments to the project setup, or in the most drastic case, a decision by the executive sponsor to cancel the project (better now than later!).

Chartering: Shape the methodology

Shaping the methodology can often be done in two days, and should not take longer than a week for a Crystal Clear team. It may be done using the *Methodology Shaping* technique, the RUP's development cases (Kruchten 2003??) or similar.

Recall that a methodology is nothing more than the conventions the team agrees to adopt. Since the set of conventions will be revisited twice per iteration or delivery cycle, it would, in principle be possible to start from any methodology and simply tune it over time to suit the team. There is probably not enough time on a typical Crystal

Clear project to follow that strategy. It will be fastest to start with the team's best thoughtful list, so that the subsequent tunings won't have to be so drastic.

As explained at length in *Agile Software Development*, there is a fairly high cost in using an overly heavy methodology: quite possibly that the team will fail to produce software for the first release at all. For this reason, start with a lighter initial set of conventions than you might suspect correct. With fewer rules, the team usually is more likely to deliver working software, and from reconsidering the conventions in the light of experience, they can add the missing conventions in the next reflection workshop.

Chartering: Build the initial project plan

There are various ways within the tolerance limits of Crystal Clear of constructing a base project plan.

- My own favorite is to use the *Blitz Planning* technique described earlier.
- The most rigorous and careful method I know is DSDM's (DSDM Consortium 2003). It involves a set of checks of the project objectives with the executive sponsor, followed by the construction of a release plan and a series of time-boxes, each one to three months long.
- Scrum's planning method fixes the time-boxes at one month long (Schwaber 2001).
- In XP's planning game (Beck 1999, 2002), the team, including their "customer," writes a set of story cards for the functional requirements. and sorts those cards into three-week time-boxes³⁹ (iterations). The team discusses with the *Executive Sponsor* the appropriate frequency of release to real users, and then groups the iterations into release periods, often about three months long.

Each of these techniques is within the methodology tolerances of Crystal Clear. Each provides an initial project plan, and is fast enough that it can be used on a regular basis to track changes in the project's situation and make an updated project plan. Choose one that fits you.

Reflection on Chartering

For the average project, the team is being assembled during the chartering period. The people doing the chartering have to deal with a number of variables. The people may or may not know each other, or may show up at some point in the middle of the project, needing time to learn to communicate with each other.

If the project emerges from *Exploratory 360°* with a positive result, the team has a sense of where the project is headed, what problem it is tackling, the technology to be used, the project plan, and its business value. This creates an early alignment of goals and actions that strengthens the team.

³⁹ In practice, most XP teams seem to vary it anywhere from one to three weeks.

The methodology shaping and initial project plan should take a couple of days. If the results have to be communicated to a wider audience, it may take a few more days to write up.

All of the chartering should take a few days to a week to complete, unless the project involves a new technology that is difficult to evaluate.

How important is the order of the steps I outline above? I wrote them in the order that makes sense to me, but I can't see that the order is critical. Choose the order that makes sense to you.

The outcomes of the chartering activity are:

- a *go* or *no-go* vote on the project, and
- a group of people with a draft charter and an idea of what they are doing.

The Delivery Cycle

The delivery cycle has three or four parts:

- A recalibration of the release plan
- A series of one or more iterations, each resulting in integrated, tested code
- Delivery to real users
- A completion ritual, including reflection on both the product being created and the conventions being used.

Recalibrating the Release Plan

On the first delivery cycle, it should, of course, not be necessary to recalibrate the plan just created. After the first delivery, however, the team members will find themselves with new and valuable information to feed into the project plan. They will have learned both how fast they really work and how mistaken their initial size estimates were. Furthermore, they and their users will have learned more about what is really needed in the system.

They have a choice: they can stick with the original set of requirements and merely update the plan, or they can revisit both the requirements and the plan. Crystal Clear does not mandate either strategy – that decision is up to the project sponsor. They would, however, be remiss in their professional duty if they didn't update their work estimates and reexamine possible strategies going forward.

I am obliged to repeat here the note from the *Blitz Planning* technique. There is an interpretation of agile development that the *Executive Sponsor* is at the mercy of the developers in constructing the project plan. This interpretation is faulty. What is true is that there is a careful allocation of responsibilities in the planning activity. The *Executive Sponsor* controls the project priorities, the development team controls the estimate of how long each task will take, they *jointly* share responsibility for coming up with a plan to deliver the best results fitting the *Executive Sponsor's* priorities with the time and people available. If the *Executive Sponsor* still feels that the project is taking too long, she has exactly three options:

- replace the team,
- adjust the project's scope or time boundaries,
- come up with a more creative strategy to get the work done with the time and people available.

My experience is that a combination of the last two works well to deliver maximum business value for resources expended.

Iterations

Iteration cycles are described in the next section.

Delivery to Real Users

As described in **Frequent Delivery**, delivery in Crystal Clear means "to a real user." In differing circumstances, that might mean full deployment with training classes, or it may be to only one person, a friendly user willing to give the budding system a walk through, either out of curiosity or just as a courtesy to the team.

The cost of deployment in the former case is quite expensive, and so "delivery" probably cannot be done every month or even two, but may have to be done quarterly. The second case is inexpensive. My experience is that you can usually find a friendly user if you try; the value obtained from this person is very high.

Many of my colleagues, used to deploying over the web or to friendly, internal users not needing training materials, assail me regularly for suggesting that a team can go for three months before deployment. There are three reasons that I accept delivery in Crystal Clear to be as long as, but not longer than three months:

- People seem to be able to retain their work focus and detailed memory for about three months, but not noticeably longer. In all my project interviews, I found a few teams able to make quarterly delivery cycles work reliably, but only one at four months, and none longer than that.
- The cost of delivering a system ready for full-scale use, complete with user training, is much higher than delivering the same system to just one or a few friendly users. The time and money cost of testing the system, writing the manuals, and creating the training is sufficient that it is often impractical to do it more than quarterly.
- Quarterly delivery of value to the business is quite often sufficient for the executives (the contrasting view, of course, is that their business might itself become more agile if delivery could be done monthly).

Completion Ritual: Reflect on the Delivery

Pressure is typically quite intense toward the end of the delivery period.

Unwinding is part of a general human need for rhythm. I was surprised to hear three XP masters talk about burnout after a year or two of successful XP practice. They said that their work was operating at a constant pressure, day after day, week after week, month after month. It was not that the pressure was high, but that it was unrelenting. The same was reported by a pair of programmers using Crystal Clear with two-week iterations. After a while, the monotony of work life was getting to them. In other words, people need a chance to unwind and "shake it out," so they can ramp up again for the next period of intensity.

Therefore, after delivering the software, unwind. Use the reflection workshop as part of "shaking it out." Instead of holding a reflection workshop at the end of the last iteration before delivery, save it up until just after the delivery, then reflect.

Include a celebration after a delivery. I have heard of people holding a party, going for a walk in the mountains, going sailing as a group, going home early, or allocating some days for people to work on a new technology, something other than the system under development.

One manager arranged for a two-day offsite meeting after each of the first few (quarterly) deliveries. They spent the first day on team building, socializing and reflecting. They spent the second day planning the next delivery cycle. With the exception of "team building and socializing" (which alert readers will recognize as essential activities in agile development), the same time was spent on project activities as they would have been in the office. The difference was that both the place and the pace were different, so that the team returned to their offices in a refreshed state. After several deliveries, they decided they didn't need two days any longer, and held the combined activity on-site within a single day. However, they still got the completion ritual and change of pace they needed.

After a delivery, the team has two additional issues to reflect on:

- How did the deployment go? What different actions should be taken early in the delivery cycle to reduce the pain of deployment and training?
- What do the users think of the system? What are its strong points and weak points? Most importantly, can the team learn anything about what is *really needed* by the users, compared to what was originally requested.

Reflecting on the delivery process is the same as for any other reflection workshop: the group asks, What do we like and want to keep the same? and What would we like to do differently? The only difference is that the changes affect either work habits throughout the delivery cycle or activities that should take place early on, perhaps in the first iteration.

To evaluate the system, the team may sit and watch the users, they may videotape some users using it, they may hold customer focus groups, or they may even hire an outside firm to evaluate the system in use. The point to be made is that it is the *product*, not the process, that they review here.

The Iteration Cycle

Iteration lengths and formats vary with different teams. I shall describe two possible iteration cycles, a one-week iteration and a two-month iteration. Yours is likely to fall in between these two.

An iteration has three parts:

- Iteration planning
- Daily and integration cycle activities
- Completion ritual (reflection workshop and celebration).

Within the iteration period , the team will find itself adding to its requirements set, trying out user interface designs, extending the system's infrastructure, adding functionality, showing the system to the user(s), adding tests, and adding to the automation capabilities of their working environment.

The one-week iteration

In the case of a one-week iteration, the planning is likely to occur on Monday morning and the reflection and celebration late on Friday afternoon.

The planning session on Monday morning is used to set the team's priorities for the week. Since a week is not a long time, the team must divide their work into fairly small pieces in order to guarantee that they can develop, test and integrate the pieces by Friday afternoon. The good news is that since a week is a short time, they probably do not need to spend much time estimating and analyzing dependencies. I would hope that the planning session for the week would not take more than about an hour on Monday morning.

Each morning, noon, or late afternoon, the team is likely to have its daily *check-in*⁴⁰ or *stand-up* meeting. Many teams report that meeting for five to ten minutes every day increases the rate of flow of information within the team, and improves the team members' awareness of both political and technical situations that may become relevant to them. The essence is to keep the meeting short (ten minutes) and to let each team member know what is happening to the other team members.

During the week, aside from the above meetings, the team simply operates in its normal development activities. Their development episodes consist of nothing more than picking up a work assignment, developing it and checking it in to the configuration management system, plus performing an integration build and system test, if that is the convention in use on the team. They will discuss and possibly show

⁴⁰ Jim McCarthy has written a complete protocol for people to register their moods and intentions during the project (McCarthy 2001??). The *check-in* protocol is the one used to announce readiness to work.

their work to their executive sponsor and ambassador user according to the rhythms they have set up.

The hazard for teams that use short (one- and two-week) iterations is that they forget to ever bring in real users. If this is your situation, consider adding a super-cycle that includes user viewings, or at least create an explicit *User Viewing* schedule (see *Work Products*).

Completion ritual for the one-week iteration

For an iteration only a week long, the iteration's closing ceremony is more about providing emotional closure than anything else.

The burn-out syndrome described in the *Delivery Cycle* is most likely to show up when the iteration cycle is very short. As Ron Jeffries wrote:

Another thing that I like is the varying rhythm of an iteration. It starts with a plan, it builds up over a couple of weeks, then spins down. (If there's a frenzy at the end, we're not doing it right yet.) The iteration - free mode, it seems to me, would be just a relentless march march march. We did that on C3 for a while and it drove us mad, do you hear me, mad.

Ron Jeffries 9/24/03 xp-egroup

With longer iterations, the final integration and the longer reflection workshop and long succeeding planning session provide a form of decompression. That is less the case with weekly iterations, and so a series of computer games, ping-pong games, a light discussion on any professional topic, a bike ride, or a visit to the pub could be in order.

With a one-week-long iteration, do not expect too much from the post-iteration workshops. After a few weeks, the team is unlikely to be able to think of much to change, and they are likely to become very short. Once this happens, consider holding the full reflection workshop once each month or two, and block out an entire hour for it, so that people can reflect over what has happened over a longer period of time and notice repeating patterns that need altering. (Note: this creates the "super-cycle" mentioned at the start of this chapter.)

Some teams capture suggestions during their daily stand-up meetings. This is excellent, but it shouldn't replace the end-of-cycle reflection workshop. During the daily stand-up a suggestion for improvement allows the team to respond immediately. The end-of-cycle workshop gives people a special time to reflect on what they find positive and negative about their working habits. It is for this reason that a monthly or post-delivery reflection workshop may be an hour to half a day long.

The two-month iteration

The planning for a two month iteration may take half a day or a full day, depending on the technique used and the practice the team has had doing it. It is common to plan the next iteration immediately following the post-iteration reflection workshop. For those without an effective planning technique, I recommend starting with either the *Blitz Planning* technique in this book or XPs planning game (Beck 2002, Cohn 2004).

During the iteration, the same daily activities occur as for the one week iteration: daily check-in or stand-up, development episodes, integration-build-and-test, visits with the user and the sponsor, and so on.

When the iteration is two months long, it is useful to hold a mid-iteration reflection workshop. This is shorter and simpler than the end-of-iteration workshop. The primary purpose of this reflection session is to discover whether anyone has detected anything that will completely derail this iteration's success. If not, then perhaps the people have found something they want to improve; or else the meeting is just very short. If they have discovered a major danger, then obviously they must work out what to do about it. The purpose of these workshops is to catch mistakes while there is still time to correct them.

Completion ritual for the two-month iteration

The team might use an hour to half a day for the reflection workshop if they hold it every four- to six weeks. They should take the time to review every aspect of their work, from their relation with their sponsor and users, to their communication patterns, hostility and amicability, the way they gather requirements, their coding conventions, the training they get or don't get, new techniques they might want to try out, and so on.

After the first iteration or delivery cycle, very often teams tighten their standards, get more training, streamline their work flow, increase testing, find a "friendly user" and set up configuration management conventions. At the end of subsequent cycles, their changes tend to be much smaller, unless they are experimenting with a radically new process.

Most people get caught up in their work and do not have the time or inclination to reflect on their work habits. The point of this periodic reflection workshop is to catch mistakes in time to repair them within the project and to give people a chance to notice ineffective patterns.

In Japan, they call these sessions *Kaizen* workshops⁴¹. See an example output from a (non-software) *Kaizen* workshop in the Work Products chapter.

⁴¹ Strictly speaking, *Kaizen* is more frequent and typically has a specific form and output. However, even in Japan I have not found many groups doing *Kaizen* on their software process.

The Integration Cycle

An integration cycle can run from half an hour to multiple days, depending on the team's practices. Some teams have a stand-alone machine run a build-and-test script continuously. Others integrate every few design episodes, staying in close touch with each others' activities. Still others integrate once a day or three times a week. Although shorter is generally better, Crystal Clear does not legislate the length of time to use.

For more on integration, review the property, **Technical Environment with Automated Tests, Configuration Management and Frequent Integration**.

Whether using my reflection workshop or the standard *Kaizen* form, the idea is to examine and improve the working conventions frequently.

The Week and the Day

Of the various cycles, only the daily and weekly cycles are calendar rhythms.

Many group activities occur on a weekly basis. These may include such things as a Monday-morning all-hands or department meeting, team-leaders' report meeting, a regular bring-your-own-lunch technical discussion ("brown bag") seminar, or a Friday afternoon wine-and-cheese party or beer bust (depending on your culture!).

A work day also has its own rhythm. It is likely to start with a daily stand-up meeting, and then consist of one or more design episodes, with a lunch break thrown in at some point.

Some teams integrate their code multiple times a day, in which case the integration cycle doesn't interact much with the daily and weekly cycles. Other teams do a nightly, twice-weekly or weekly build.

The Development Episode

Ward Cunningham coined the term *episode* to describe the basic unit of programmer work in agile development⁴². During an episode, a person picks up some small design assignment, programs it to completion (ideally with unit tests), and checks it in to the configuration management system. This might take between fifteen minutes and several days, depending on the programmer and the project conventions. Keeping the episode less than one day in length generally works best.

(Important note: To me, as to the many people in the world who work in the Crystal style, *designing* and *programming* are such closely linked activities that they are not worth prying apart. Thus, there is only the role "designer-programmer" in Crystal Clear and Crystal Orange, not the two roles designer and programmer.

Consequently, I write "to program" meaning "to design and program" or "to design" meaning "to design and program". I write "programmer" meaning "designer-programmer". Thus, in the last paragraph, the phrase "picks up a small design assignment, programs it to completion," means picking up a small assignment that requires designing and programming, then designing, programming, debugging and testing it, to completion.

I need to clarify this because many readers naturally think of designing and programming as actions of two separate job categories carried out by separate people. That would be a serious misinterpretation of Crystal Clear.)

⁴² <http://c2.com/cgi/wiki?DevelopmentEpisode??>

Reflection about the Process

Due to the history of our literature, most practitioners are used to seeing a linear version of their process. The mental shift to accommodate a cyclic process is hard. It is not that the *practice* of a cyclic process is hard – you probably already work in cycles on your current project. However, I find that even the most experienced project managers and methodologists can't *explain* cyclic processes, and particularly their boundary interactions (such as the absence of replanning at the start of the first delivery cycle and the absence of the reflection workshop at the end of the last iteration in a delivery cycle). To be honest, I've been trying to explain them for over ten years and only now feel I have a proper handle on the matter.

The difficulty is that some activities occur on daily and weekly rhythms that repeat without creating any particular sense of "forward motion." These *operations* types of activities include daily and weekly status meetings, checking in code, running unit tests, going for lunch, convening around the coffee or soda machine. Operations activities contrast with activities that show "forward progress," such as design reviews, getting requirements sign-off, moving into test or alpha release phases. People tend to catalog the progress activities in the process and neglect the operations activities.

The nested-cycle view allows the discussion of both progress and operations activities. Both are important to the life of the team member, both are part of the "process."

Two cycles need a little more clarification: the iteration and the delivery.

In the recent literature of our field, "iteration" is used without much distinction. It could refer to strictly internal iterations, as I describe it in this book, or for iteration-with-delivery. Many people forget about fulfilling the delivery portion of their iterative process. See the discussion under "*We colocated and ran two-week iterations – why did we fail?*" in the chapter **Misunderstood (Common Mistakes)**.

In Crystal Clear, you are allowed to have just one iteration per delivery cycle, but if you do this, you *must* have some intermediate viewings by real users. If you have multiple iterations per delivery, some of those *must* include viewings by real users. If you don't do this, you are building an effective software-producing team that makes the wrong software very efficiently.

Chapter 5

Examined (The Work Products)

This chapter describes team roles and the work products, showing examples of each work product. These particular work products are neither completely required nor completely optional. They are the ones I can vouch for both one at a time and taken all together. Equivalent substitution is allowed, as is a fair amount of tailoring and variation.

Although this is where the most argument is likely to occur, it is where the argument is probably least likely to affect the project's outcome.

Just as describing a methodology through its process introduces problems of interpretation, so does describing it with its work products. In a small methodology such as Clear, the number and formality of intermediate work products is reduced quite significantly. The team lives from their personal communication, notes on the whiteboards or posters around the room, and demos or deliveries to the user base.

Nonetheless, a description of the work products is necessary. People just starting with Crystal Clear need to see what counts as an "acceptable" set of work products. Executives and sponsors need to see what they are entitled to ask for. Teachers need a set of work products to have students practice on. Teams that are doing too little in the way of planning and documentation need to see what is worth preparing in even a light agile methodology. People working to understand a methodology will want to examine the work products as part of the overall methodology package.

In this section, I describe roles and the work products. These work products should be seen as the default set for a typical Crystal Clear project, because they have demonstrated their value to many projects. It is, of course, up to the project team to add, subtract or modify the list based on their situation.

Each item serves a communication purpose in the economic-cooperative game. Sometimes the economically appropriate artifact is *low-precision* (not very detailed) and *large-scale* (summarizing a large topic), examples being the project's mission statement and the release plan. At other times, a *medium-precision* artifact is needed, such as the actor-goal list, which outlines functional requirements at a glance. Sometimes, *high-precision* (detailed) descriptions are needed, the final code, user manual, and test cases being examples.

There are almost two dozen work products, depending on how you count them. People coming from a traditional project background may be shocked by how few there are; dyed-in-the-wool agile developers are shocked by how many there are (they are shocked because unless they are doing XP, they just haven't counted how many similar work products get produced on their current projects). The work products here are light and distributed across the roles, so that the team should not find them burdensome in practice. As an exercise, I suggest counting the total number of work products generated on your current project. My experience is that the number is seldom less than 60. Compare them to the ones listed here.

The question arises over work products more than over any other aspect of a methodology: "Do we have to do *all* of these?" The answer is difficult to give, as it lies somewhere between the answer I was able to give for Properties ("For Crystal Clear, absolutely the first three, and as much of the next four as possible.") and the one I was able to give for Techniques ("Completely at the discretion of the team; here are an interesting and useful starter set to consider.").

If you skip too many of these work products, you lose alignment and visibility. Your direction and support can suffer accordingly. On the other hand, it would be just

silly for me to insist you do all of them as given – the specific work products needed by teams varies according to changing techniques, technologies, communication habits and even fashions.

I have been asked to name "the core" work products, as I was able to name "the core" properties, but this is not possible. If I was going to choose one work product to drop, it would be the Project Map (however useful it might be). If I was going to choose another, it would be the viewing schedule (that could be done verbally and on the fly). If I had to drop another couple, I might suggest merging the coarse-grained and fine-grained iteration plan and status and dropping the fine-grained iteration plan. It could happen that another person would drop them in another order, or I might drop them in a different order on a different project.

Misalignment within the team and miscommunication with the outside world grow with each omission. The risks accumulate, slowly at first, until the project is not in the safety zone, and it is never clear when it moved from being safe to being unsafe.

It is very easy to overburden a project with work that is useful individually, but which, when taken all together, slows the group to the point of losing more safety than it adds. This set of work products consists of items I can defend individually and all together. Even if your team does every single one of them, it should not be overburdensome. You may still be able to find a way to substitute for or omit some of them. Discuss it in your methodology shaping workshop, and discuss it again in your monthly or quarterly reflection workshops. That's what those are for.

In the pages that follow, I describe each work product, indicating which role is ultimately responsible for it. I include several examples of each work product to show variations in style and format different teams have adopted. Where possible, I show informal and formal renderings, graphical and textual. Seeing this range, you can choose a form that works for you and the external groups you need to keep informed.

The Roles and their Work Products

The **Sponsor** is responsible for producing just one item:

the *Mission Statement with Tradeoff Priorities*.

The **Team as a Group** is responsible for producing two things:

the *Team Structure and Conventions*, and

the *Reflection Workshop Results*.

The **Coordinator**, with the help of the team, is responsible for producing

the *Project Map*,

the *Release Plan*,

the *Project Status*,

the *Risk List*,

the *Iteration Plan & Status*,

the *Viewing Schedule*.

The **Business Expert & Ambassador User** together are responsible for producing

the *Actor-Goal List*,

the *Use Cases & Requirements File*, and

the *User Role Model*.

The **Lead Designer** is responsible for producing the

the *Architecture Description*.

The **Designer-Programmers** (including the Lead Designer) are responsible for

the *Screen Drafts*,

the *Common Domain Model*

the *Design Sketches & Notes*,

the *Source Code*,

the *Migration Code*,

the *Tests*,

the *Packaged System*.

The **Tester** (whoever is occupying that role at the moment) is responsible for

producing

the *Bug Reports* at that time.

The **Writer** is responsible for producing

the *User Help Text*.

Roles: Sponsor, Ambassador User, Lead Designer, Designer-Programmer, Business Expert, Coordinator, Tester, Writer

There are eight named roles for Crystal Clear, four of which probably have to be distinct people. The other four can be additional roles assigned to people on the project. The first four people are:

- Executive Sponsor
- Ambassador User
- Lead Designer
- Designer-Programmer

Executive Sponsor. This is the person who is either allocating or defending the allocation of the money for the project. The *Executive Sponsor* is supposed to keep the long-term view in mind, balancing the short-term priorities with those of subsequent releases and subsequent teams evolving or maintaining the system. This is the person who will create outside visibility for the project, and provide the team with crucial business-level decisions. If there is a question of balancing the money being spent for the value being delivered, it is up to the *Executive Sponsor* to make the decision of when and whether to continue or stop, and if continuing, how to trim the remaining system functions to recover the business value. Part of the methodology involves giving the *Executive Sponsor* good information to make those decisions.

Sometimes it is the user community who is actually paying for, or *sponsoring*, the project, and they, as a department, work through an *executive*, who directs the development team. It is easy in this situation for the *Executive Sponsor* role to get awkwardly split, where the executive does not have the same long term interests, priorities or vision as the user community. There is nothing that a methodology like Crystal Clear can do to remove the potential conflict here. You, as a combined group of people, have to recognize the hazards of this situation and work extra hard on communication to keep priorities and goals aligned, visibility and amicability high.

Ambassador User. This is the person who is supposed to be familiar with the operational procedures and the system in use (if there already is one), knowing which are frequently – and infrequently – used modes of operation, what short-cuts are needed, and what information needs to be visible together on the screen at the same time. This is a different knowledge base than the *Business Expert* is expected to have. The person holding the role of *Business Expert* is expected to know the business rules needed within the system, what business policies are stable versus which are likely to change. The *Business Expert* is not expected to have intimate familiarity with the minute-to-minute activities of the user population, whereas the *Ambassador User* is.

Lead Designer. This is the lead technical person, the person supposed to have experience with software development, able to do the major system design, tell when

the project team is on-track or off-track, and if off-track, how to get back onto track. In terms of levels of competency, the *Lead Designer* is expected to be a level-3 designer⁴³.

There are arguments about whether agile development requires top-notch programmers to succeed. I have never had the luxury of only using top programmers, nor were the teams I interviewed made up of them. They were made up of the usual mix of people one finds in a company. Importantly, though, at least *one* person on the team was competent and experienced, that is, level 3.

The ratio of level-3 people to level-1 people is a factor that I consider significant enough that I offer it to researchers to examine as a primary correlation factor to project success. My experience (and I suspect the research will show) that this applies to both small and large projects, both low- and high-ceremony projects. High-ceremony need much more boilerplate work done, and so can live with a higher mix of less talented people to the talented ones. Barry Boehm and Rich Turner capture this difference on their starfish diagram (*Balancing Agility and Discipline*, p. xxx??), where one of the axes is ratio of level-3 to level-1 people.

Since there are only three to eight people on a Crystal Clear project, at least one of them has to be competent and experienced, i.e., level 3. Usually, there are a couple of trainees or junior people on the project. For the purpose of project safety, then, I designate the *Lead Designer* role separately from the other roles.

The *Lead Designer* is likely to have major influence in the methodology shaping workshop. Very often, the *Lead Designer* is the only experienced designer on the team. If the entire team is experienced, then of course the methodology shaping workshop can work in a peer-to-peer fashion.

I use the word "designer" for this role to shorten the role name from the overly long "*Lead Designer-Programmer*." The *Lead Designer* is expected to both design and program, just as are the other *Designer-Programmers*.

Usually, but not always, the *Lead Designer* is the most experienced person on the team and also handles the most difficult programming assignment. Although this is common, it actually presents a risk for the project. The *Lead Designer* tends to get overloaded, having the role as *Coordinator* as well as architect, mentor and most experienced programmer. Anything that can be done to offload the *Lead Designer* is likely to improve the risk profile of the project. The obvious choice is to have someone else play part or all of the *Coordinator* role.

Designer-Programmer. I merge the words "designer" and "programmer" in the *Designer-Programmer* role to highlight that each person both designs and programs. Designing without programming is full of flaws due to lack of feedback in projects of

⁴³ Remember from Chapter 1, Level 1 skill is the "following procedures" stage of learning; level 2 is "breaking away from specific procedures" and level 3 is "fluency," or mixing and inventing on the fly.

all sizes. It very particularly has no place on a project of the Crystal Clear type.

Programming by its very nature involves designing. Neither "designer" nor "programmer" stands alone as a role name, hence *Designer-Programmer*.

The other roles. The *Coordinator* is probably a partial occupation for someone on the team; projects of only four to eight people seldom have a dedicated project manager. There may be a person managing several projects, and playing the *Coordinator* role for each of them. Alternatively, the *Executive Sponsor* or *Lead Designer* may get this role, or even that people take turns holding this position.

The person occupying *Coordinator* must, at the minimum, take notes at the project planning and status sessions, and combing the information for posting and presenting. The *Coordinator* is responsible for giving the project sponsors visibility into the structure and status of the project. With luck, the *Coordinator* also is someone with a good human touch, and can facilitate discussions and reduce strife.

The *Business Expert* is the expert on how the business runs, what strategies or policies are fixed, what is likely to vary soon, often, or seldom. This person will answer all the varied questions the developers will have about the heart of the system. It is different information than the *Ambassador User* typically provides, although in some cases it may happen that the *Ambassador User* is also the *Business Expert*. In various situations, I have seen the *Executive Sponsor*, the *Lead Designer* or the *Ambassador User* be the *Business Expert*. Sometimes an external person is brought in for that expertise.

Tester and *Writer* are likely to be rotating or temporary assignments. I give these separate role names, but in many Crystal Clear projects there are only the four to eight people, all programmers, sitting in the room. In this case they obviously have to take turns occupying these roles. Some teams may get use of a *Writer* for periods of time, or have a dedicated *Tester* working and even sitting with them.

A note about the project samples

In the following, I draw from a set of projects, some but not all of which were Crystal Clear projects.

The *PRTS* project was the project to build a system to track all purchase requests issued by employees of the Central Bank of Norway, whether they were fulfilled from the internal warehouse or external vendors. This project, for its short life, consisted of four people who worked on it exactly one day each week. It met the **Osmotic Communication** requirement because nobody worked on it at all for four days, and then we worked together on the fifth day each week.

I was the *Coordinator* for this project. We proceeded from mission statement, through *Blitz Planning*, and through the *Exploratory 360°*. During the *Exploratory 360°*, the proposed project failed both the technology spike and the business evaluation tests. The programmers wanted to keep going on it, but the *Executive Sponsor* decided to purchase a package for the system in order to free up the programmers for work more important to the business. Not only was this the correct business decision⁴⁴, but the programmers found their next project much more exciting, since it involved both critical business value and new technology.

Consequently there are work products for PRTS up through that point.

The *NICS-NBO* project. Due to a bank failure in the early 1990s, all Norwegian banks have to keep a "checking account" with the Central Bank, and all the bank-to-bank transactions have to get funneled through these "checking accounts." The bank union, representatives from key banks, and the manager of the Banking division of the Central Bank comprised the steering committee for a series of projects. The NBO project here was the third in the series. I was *Coordinator* for this project.

At the beginning it was staffed with the same three programmers who had done the first two projects. It looked like an easy candidate for Crystal Clear, even though the technology was mainframe with COBOL, assembler, CICS and LU6.2. Over time, we lost the *Lead Designer* (paternity leave), half of the second senior person (Y2K projects) and were left only with the junior developer and a new hire. The staff then grew in number and lost **Osmotic Communication**. At this point, the project was out of Crystal Clear range. The work product sample I am using here is the coarse-grained project-plan and the single-sheet combination of status and risk list that we used to report to the steering committee each month.

The *BSA* project. This was a project performed as part of a senior year course at Weber State University. The small team met once a week outside class and communicated by phone and email otherwise. They used database technology for the

⁴⁴ Niel Nicklaison (2004) has written a simple business evaluation procedure to decide which projects to develop in-house and which to outsource, see <http://??>

project, so the domain model consists of a database schema. I would like to thank Dr. Christopher Jones and this team, Eric Schultz, Keith Deppe, Tony Hess, and Bethany Stimpson for this project.

Project *Winifred*. Project Winifred is the 50-person, \$15 million, 18-month fixed-price, fixed-time project I described at length in *Surviving Object-Oriented Projects*, and for which we constructed the original Crystal Orange. I use the project map and coarse-grained delivery plan from that project, because it seems interesting to me that our 18-month project had a coarse-grained plan consisting of about eight bubbles (backed by about 240 two-paragraph use cases).

The *CamCal Camera Calibration* prototype. This was a trial run of Crystal Clear in Thales, an ISO 9001 certified organization. This project involved four people for four months and fit the Crystal Clear guidelines. They were kind enough to let me include their experience report and auditor recommendations in this book (see the chapter, *Tested*).

Additional projects are represented in the work product samples. These, however, are sample descriptions of the sorts of projects used.

Sponsor: Mission Statement with Tradeoff Priorities

The mission statement is a brief description, typically a paragraph to a page, of what is to be built, its purpose in a larger context, and the project's priorities in sequence, from the most critical to those that can be sacrificed.

It is produced by the *Sponsor*, before the project starts or during *project chartering*. It is reviewed by the *Business Expert* and *Lead Designer*, and referenced by everyone on the team. Any major change in the mission statement triggers a team meeting, so that everyone on the team understands the new mission.

A good mission statement retains its clarity and relevance over time, keeping the work efforts focussed on the most important features of the system.

Because people can generally only protect one and possibly two project priorities (see *Peopleware*, pp. ??), it has no more than two top priority items. The priorities might be chosen from: hitting a particular delivery date, cost, ease of use, ease of learning, speed in use, correctness, performance, ease of maintenance, design flexibility, legal liability protection (you may come up with other choices for your project). The mission statement makes clear to the team what can be sacrificed if necessary to preserve the top priorities. I note here that it is unlikely to succeed at both quick delivery and correctness as the top two priorities. Those are two that need to be traded against each other.

I provide three examples here, from the Camera Calibration project, the Boy Scouts tracking program and the PRTS project, respectively. See also *Agile Project Management* (Highsmith 2004) for examples.

CamCal Camera Calibration Mission Statement

The mission of the camera calibration project is to:

- Develop a maintainable, simple to use, portable and extendable software tool that will enable intelligent video surveillance system installers and repairers to rapidly, accurately and efficiently calibrate their system cameras with respect to a pre-defined three-dimensional scene model.
- Establish a basis for developing a future tool set that can manage all aspects of intelligent video surveillance system installation and repairers activities. Such a tool set will include the camera calibration tool, and will add scene model creation capabilities. It will be able to grow with the technology growth of the video surveillance market.
- Develop and trial the use of new software development methodologies and processes, with a view to improving the software development process used for small to medium sized software projects.

The development priorities are:

Sacrifice others for this: Complete by end of January,

Ensure accuracy and quality

Retain if possible:

Usability, potential to grow into a more extensive tool

Sacrifice these first:

Execution speed, portability, additional interfaces,
internal format storage.

Figure 5-1. CamCal mission statement. (Thanks to Stephen Sykes at Thales)

The “BSA Progress Tracker” is designed for scoutmasters to track the progress of scouts through their advancement in the Boy Scouts of America program toward Eagle Scout.

Development Priorities

Overarching: Delivery Date, Functionality

Foreground Constraints: Ease of Use, Formal Correctness

Sacrificial Lamb: Performance

Figure 5.2. BSA mission statement.

The "Purchase Request Tracking System" has two goals. The first and most essential is to provide a basic system for the official Buyers of the company to track what they have ordered from Vendors against what has been delivered. The second is to simplify the lives of people who wish to order things, who must sign purchase requests, and who are to track the purchases against budgets.

Project Priorities:

	Sacrifice others for this	Try to keep	Sacrifice these for others
Simple to use	X		
Low cost to develop	X		
Defect freedom		X	
Deliver soon		X	
Ease of learning			X
Performance			X
Design flexibility			X

Figure 5-3. PRTS mission statement.

Team: Team Structure and Conventions

Team structure is an allocation of people to roles. Team conventions is a collection of rules, practices and conventions the team agrees to adopt.

Team conventions shift constantly. This is appropriate, because the situation in which the team finds itself is constantly changing. The team should not only be conscious about these changes, but should deliberately search for the optimal set of conventions for their project.

The published part of Crystal Clear contains a large number of agreements and conventions for the team to adopt. Being written for many projects, it cannot be complete. The team needs to settle additional conventions, at least the following:

- allocation of people to roles,
- programming conventions such as naming, formatting, commenting,
- code ownership conventions,
- design and code-reviewing conventions, whether code reviews or pair programming or none,
- iteration length and forms of reporting status, frequency of user viewing,
- configuration management conventions.

Depending on the experience of the team, the conventions may be built through an open methodology-shaping workshop, or drafted initially by the *Lead Designer* and allowed to evolve by suggestion and consensus of the entire team in reflection workshops ("consensus" means each person is willing to accept and defend it, even if it is not that person's personal preference). Some conventions require consensus of the *Sponsor* as well.

The conventions should be consciously reviewed at the post-iteration reflection workshop, or at least every 3 months. Reread the *Iteration Cycle* to review this. At any time, of course, someone on the team may discover that something is really not working, and may request a change to the way the team is working.

Team conventions are often spoken rather than written. Some may get written on the information radiators. Coding conventions are often shown via sample code.

I categorize a methodology as "successful" if

5. the software gets delivered and
6. the team is content to work the same way again.

This may seem to be a very low threshold for success, but recent project reports show it to be surprisingly difficult.

If your methodology fails on either count, change it using the methodology shaping workshop technique.

The team structure in Crystal Clear should be quite straightforward to establish. Four roles are crucial: *Executive Sponsor*, *Lead Designer*, *Designer-Programmers*, and *Ambassador User*. The other roles, *Business Expert*, *Coordinator*, *Testers*, and *Writers* can

be combined roles, assigned either to additional people or shared by people having other roles. Quite often, the *Executive Sponsor*, *Lead Designer* or *Ambassador User* is the *Business Expert*.

Here is an example of assignment of people to roles.

Team Name:	BSA-Team
Sponsor:	Eric Schultz
Lead Designer:	Keith Deppe
User:	Eric Schultz
Designer-Programmer:	Tony Hess Bethany Stimpson
Business Expert:	Keith Deppe
Coordinator:	Bethany Stimpson
Tester:	Tony Hess
Writer:	Bethany Stimpson
Communication Paths:	Weekly meetings outside class. Phone and email communication.

Figure 5-4. BSA role assignments

A simple example of a convention adopted by a testing department team follows. They were trying to move from manual to automated testing, but ran into the problem that they spent most of their time fighting fires (using manual tests), and had trouble getting themselves to sit down and write automated tests. They therefore established for themselves the convention:

"No manual testing before 3 p.m."

This simple rule gave them the **Focus** (both priority and time) they needed to get over the initial hurdles and build a starter set of automated regression tests.

Here is an excerpt of some team conventions we created for a 50-person company (50 people puts it in the Crystal Orange category). There were several dozen conventions, which we divided into five categories: Regular Heartbeat with Learning; Basic Process; Maximum Progress & Minimum Distractions; Maximally Defect Free; A Community Aligned in Conversation. I list the entire set of conventions in *Agile Software Development*. For space reasons, I include here only the third section as illustrative.

Maximum Progress, Minimum Distractions

The purpose of this category is to ensure that people are working on what is of greatest value to the company and have time to focus and make progress on that work.

1. The top corporate key initiatives are prioritized and visibly posted for each two-week production cycle.

- | |
|---|
| 2. They are allocated to individual people so that each person knows his or her top two or three personal priority items for the cycle. |
| 3. Work is broken into what can be completed and tested in the two-week cycles and is further broken down into things that can be accomplished in one to three workdays. |
| 4. <i>Each person who is working on more than one initiative is guaranteed at least two consecutive days to work on any one initiative without being pulled onto another assignment.</i> |
| 5. The developers post on the whiteboards outside their office the current status of the work they plan to complete during a given week. |
| 6. Every morning, the developers meet with the business owner of the current work initiative and conduct a short meeting to determine the current state of the work and the top work priorities and to discuss any questions. |
| 7. <i>The business owner is not permitted to ask for status again the rest of the day.</i> |
| 8. The period 10:00 - 12:00 each day is declared "focus time," in which no meetings take place, and everyone in the company is encouraged to turn off the phone. |

Figure 5-5. Project conventions. (Thanks to eBucks.com)

Team: Reflection Workshop Results

The Reflection Workshop Results is an information radiator that shows what the team has concluded after its reflection workshop. It is often a flipchart containing "Things we should keep", "Things to try", and "Ongoing problems" (see *Reflection Workshop*, in chapter 3). The chart is hung in a highly visible place so the team can notice what conventions and practices are the theme of the iteration.

The output is produced by the entire team during or at the conclusion of the reflection workshop. Feedback from teams indicates that it is not a good idea to give one person the job of "cleaning up" the flipchart used in the workshop: For the first part, that person usually changes the words on the chart, so it is no longer the words that received the team's approval. For the second part, the "cleaned up" version simply looks different than the people recall, and so they don't identify with it as well. Generally speaking, it doesn't much matter if the chart is a bit messy as long as it is readable. If you are going to clean it up, do it as part of the workshop.

A good reflection workshop results chart is big, visibly placed, and shows the team what they should pay special attention to in the coming weeks. Thus, it is better to use the phrase "Try This" (which states concretely what everyone should try) as opposed to "Needs Improvement" (which reminds everyone of what is bad without suggesting a remedy).

Here are some examples.

KEEP THESE	TRY THESE
<p>Iteration planning (weekly)</p> <p>Daily Standups</p> <p>Automated Regression Tests</p> <p><u>Ideas for next time</u></p> <p>Tool Check-in</p> <p>Review from my system engineer</p> <p>Test coverage tool</p> <p><u>Designing PROBLEMS</u></p> <p>Too fluid iterations</p> <p>Agile project is a reasonable timeline</p> <p>Web-X deployment didn't show real user problems</p> <p>→ No "friendly user" to review them</p>	<ol style="list-style-type: none"> 1. Document these reflections & / Having them up 2. Review these ideas in 1 month. 3. Post backlog list of tasks 4. Do iteration planning meeting on Monday PM 5. "Side-by-side" programming Jeff & Jason, Stephen Ray, Jason & Andrew 6. Pair Program - 7:30 am after standup meeting

Figure 5-6 (Thanks to Tomax)

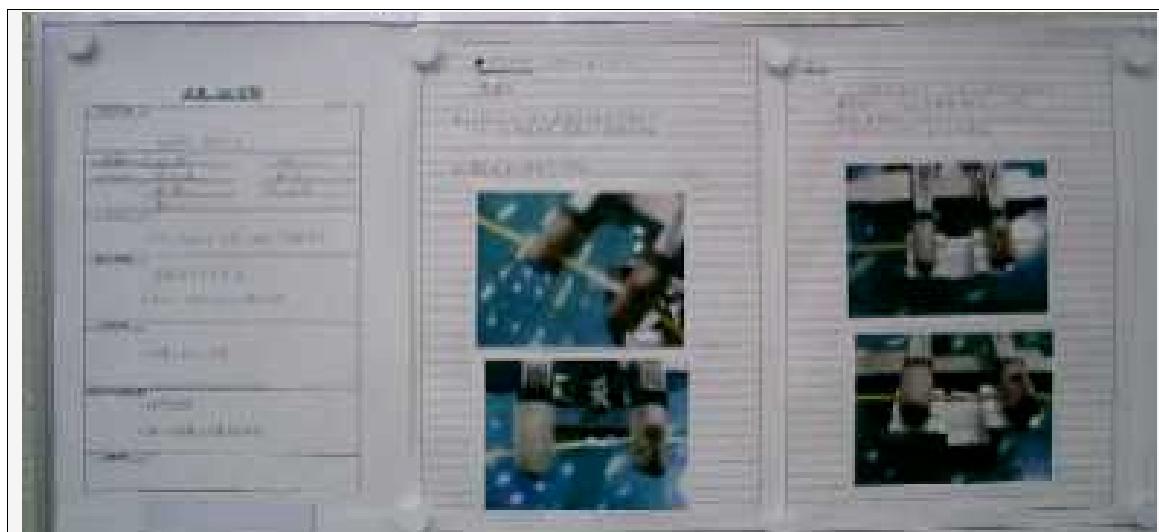


Figure 5-7. Kaizen output chart from Japan (Thanks to Basaki Satoshi)

Coordinator: Project Map, Release Plan, Project Status, Risk List, Iteration Plan & Status, Viewing Schedule

The sponsor, the development team and the users all need to be able to plan their activities around when the system functions are supposed to become visible and available. The team needs to know the order in which to create them and the target dates, the users need to know when they must allocate time to evaluate them or start using them.

Crystal Clear projects operate with two distinct time horizons, long term and short term. The long-term horizon typically refers to anything over three months, although it can be used for anything longer than a single iteration. Planning and tracking for the long-term horizon is *coarse-grained*. This allows the team to provide the executives with the resource- and expense- estimates needed for overall enterprise planning, while taking into account that the team won't know what the actual costs will be until they get farther along.

The short-term horizon typically refers to a single iteration. *Fine-grained* planning and tracking is used for the short-term horizon. This is so everyone can see the team's progress and detect problems quickly. The short-term plan and status are usually captured on information radiators in the team room.

These work products are produced at different times and maintained in different ways. These are the

- project map
- risk list
- viewing & release schedule (the coarse-grained plan)
- project status (coarse-grained)
- iteration plan (the fine-grained plan)
- iteration status (fine-grained)

As mentioned earlier, there may be a dedicated person acting as *Coordinator* on the project, but more often it is the *Sponsor* or *Lead Designer* who takes that role. Some groups find it helpful to have someone else take on the *Coordinator's* role, to reduce the load on the typically overburdened *Lead Designer*.

Coordinator: Project Map

The project map is a dependency diagram identifying the major work to be done and which ones depend on others (see the *Blitz Planning* technique). Its purpose is to show the structure of the problem and the sequence of attack. It contains no dates. That is partly because it is often drawn up before the team members and their capabilities are known, but more generally, to permit different staffing and timing strategies to be considered. It is intended to be viewed at one time, either on a conference room table or on a wall, using index cards as task markers. This limits the number of items that get put onto the map.

The project map is produced by the *Business Expert*, *Lead Designer*, and the *Sponsor* in the *Coordinator role*. It is approved by the *Sponsor*, referenced by everyone on the project, and updated by the *Coordinator*. It is produced very early in the project, before the release coarse grained project plan. It is likely that the project map morphs into the project plan as the dates are added, and changes are made directly into the project plan as the project shape changes.

A good project map answers at a glance: " In what order are we delivering what? What do we build next? What are the dependencies between our releases?"

Some project teams discard the project map as soon as the *Release Plan* is produced, others find it useful to retain. Personally

Here is an example:

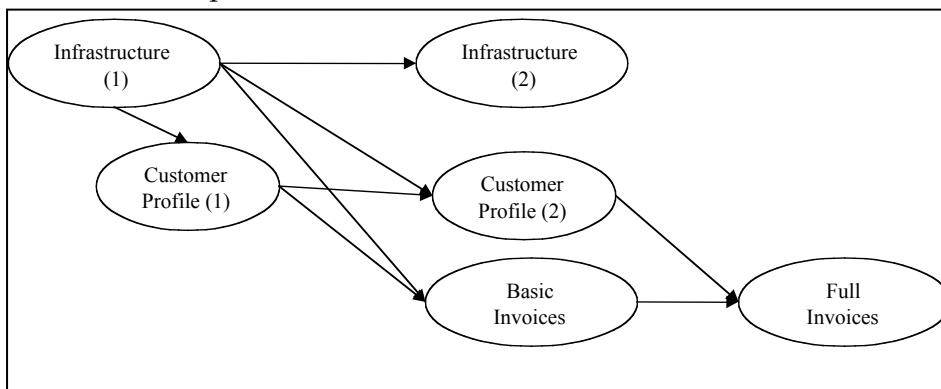


Figure 5-8

Coordinator: Release Plan

The *Release Plan* is an association of dates with major milestones, which are usually iteration ends and deliveries. The *Project Map* may become the *Release Plan* as dates are added. The *Release Plan* may itself directly become the *Project Status* (the NICS-NBO release plan, below, evolved into that project's *Project Status*).

Coming up with dates requires everyone's input. Usually, the first *Release Plan* is produced by the *Coordinator* and *Lead Designer*. The *Business Expert* and *Sponsor* check the priorities of the plan. The plan is rebuilt at the start of every *Delivery Cycle*, possibly even each iteration (recall, there are multiple deliveries per project, and one or more iterations per *Delivery Cycle*). Once the team is familiar with the project assignment, the entire team can contribute to updating the project plan.

Here are some examples from real projects, in graphical and text form. *Comment on the formats used:* Many team leads getting started with Crystal Clear and other agile development approaches feel obliged to put their plan onto a Gantt chart. Gantt charts quickly get out of date, are tedious to update, and are hard to read. Project leads and sponsors who have moved from Gantt charts to dependency diagrams and lists report that these latter are easier to read and maintain.

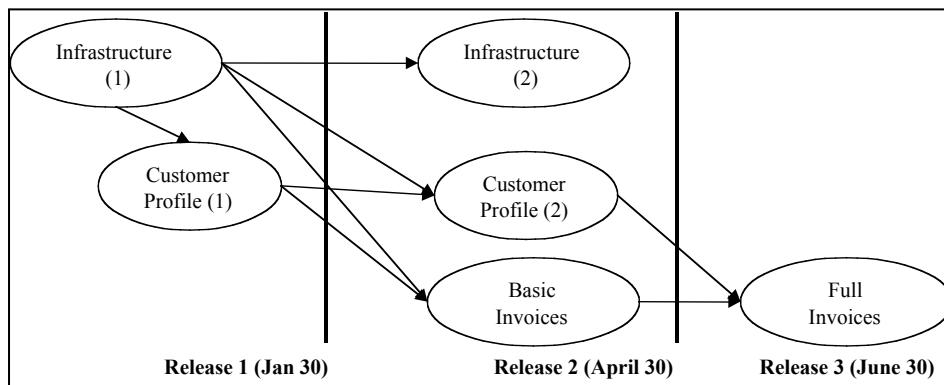


Figure 5-9

Release 1, Jan.30: Single function. Buyer can enter a request that was completed and signed on paper, and generate POs that can be printed. Receiver can mark delivery against PO and request. Supports split and partial deliveries.

Release 2, April 30: First reports (Searches, collects and prints selected reports), and first security functions. Allows Requestor to create request, collects approval from Approver, notifies Buyer.

Release 3, June 30: Full function, with full database functions.

Actor	Goal	Release
Requestor	<i>Initiate an request</i>	1
	<i>Change a request</i>	1
Buyer	<i>Complete request for ordering</i>	1
	<i>Initiate PO with vendor</i>	1
Receiver	<i>Register delivery</i>	1
Any	<i>Check on requests</i>	1
Authorizer	<i>Change authorizations</i>	2
Approver	<i>Complete request for submission</i>	2
Buyer	<i>Change vendor contacts</i>	3
Authorizer	<i>Validate approver's signature</i>	3
Requestor	<i>Cancel a request</i>	4
	<i>Mark request delivered</i>	4
	<i>Refuse delivered goods</i>	4
Buyer	<i>Alert of non-delivery</i>	4

Figure 5-10

	Milestone	Planned
I 1	A single, simple transaction makes a round trip between the computers.	Feb 1
I 2	Simple purchase transaction successfully posted from GUI through database.	Mar 31
I 3	Reports of active requests & work-flow through Buyer.	Jun 30
I 4	All work flow sampled. All reports run.	Sep 1
I 5	All functionality works	Sep 30
I 6	User acceptance	Oct 10
I 7	Deployment	Oct 20

Figure 5-11

Module	Feature Name	Value	release #
1.1	Configuration generates order line item comments	M	1
1.2	Configurator UI Rework: Verbose wizard style	H	1
2.1	PO generated correctly for configuration	H	1
2.2	Create/Confirm vendor items exist for skus	H	1
3.1	Advanced Order form shows more details	M	1
3.2	Order fulfilled at PO cost	H	1
3.3	Repeat orders works with blind configurations	M	1
3.4	Configuration comments are viewable, not editable	L	1
4.1	Base hierarchy change	H	1
4.2	Style can locate price charts based on color selection	H	1
4.3	Assign specific color group colors to price charts	H	1
1.3	Configuration generates customer specific pricing	H	2
1.4	Allow adding a configuration and continue	L	2
2.3	PO generation is automatic	L	2
3.5	Order shows sq. footage & running length	L	2
3.6	Vendor orderable items show additional detail	L	2
3.7	Printed window treatments show disclaimers	M	2
3.8	Allow order line item copying	H	2
4.1	Associate style with Retail.net DCL	L	2
4.11	Grid validation	L	2
4.6	Assign customer specific selling discounts	H	2

Figure 5-12. (Thanks to Jeff Patton and Tomax)

The following is adapted from a reengineering project to replace a workflow system. Each name is the name of a workflow station. They didn't use a burn-down chart for their project plan. I include it as an example of how one might do so.

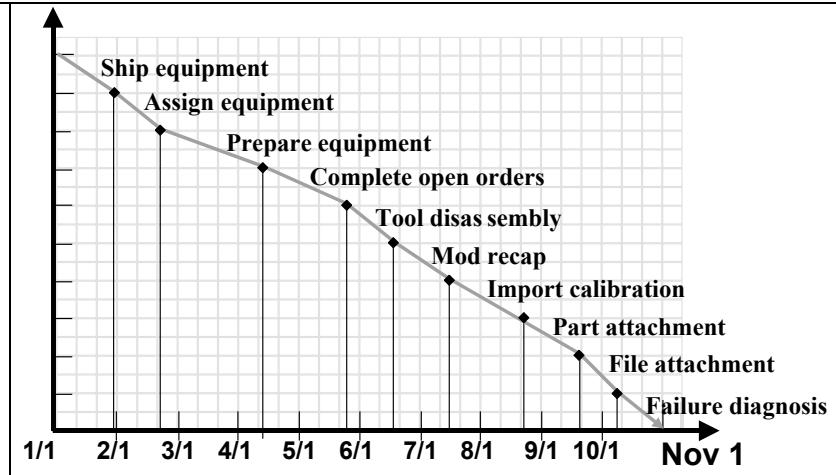


Figure 5-13

Coordinator: Project Status

The *Project Status* is a listing of the state of the project with respect to the *Release Plan*. It is often just that with the status marked. The status information might include the currently predicted due date, or the originally scheduled completion date, foreseen completion date, and foreseen risks. On one project, we were asked to also compare original and projected cost figures on this chart. The project status should generally take less than half a page; you can use the other half of the page for the risk list, making a complete project report on a single sheet of paper.

The project status is maintained by the *Coordinator* in conversation with the development team. It is used by the *Sponsor* and the manager of the user community to plan their separate efforts. It is updated every few weeks or every month.

A good project status is brief, easy to read, and shows the information the readers need in order to assess the cost, date, and risk trajectory of the project.

Here are some examples. *Comments on the strategy and formats used:* People are increasingly using simple tables to report status. The second example shows one of the differences in agile development techniques: something of business significance gets delivered approximately every week, with a full cycle every month. By completing a full conversion in the first month, the team has a chance to learn what is involved, where they are running into problems, and have plenty of time to learn how to do better in subsequent cycles.

The project plan for the NICS-NBO project became its status chart:

	Milestone	Planned	Delivered	Notes
I 1	A single, simple transaction makes a round trip between the computers.	Feb 1	Feb 3	Complete
I 2	Simple purchase transaction successfully posted from GUI through database.	Mar 31		Underway. Uncertainty over the DCOM components, still in learning curve. Experiments underway.
I 3	Reports of active requests & work-flow through Buyer.	Jun 30		Started. Initial design work.
I 4	All work flow sampled. All reports run.	Sep 1		not started
I 5	All functionality works	Sep 30		not started
I 6	User acceptance	Oct 10		not started
I 7	Deployment	Oct 30		not started

Figure 5-14

Here is a status sheet for a project involving conversion of 200 applications and their databases. In preparing this example for publication, I just numbered the systems, where the company had system names. The dates in parentheses are the predicted completion dates, the ones without parentheses are the actual completion dates.

System	New tables operating	Migration code works	Old tables converted	Old tables removed
1	(Feb 1) Feb 3	(Feb 7) Feb 10	(Feb 14) Feb 18	(Mar 1) Mar 1
2	(Mar 1) Mar 1	(Mar 7)	(Mar 14)	(Apr 1)
3	(Apr 1)	(Apr 7)	(Apr 14)	(May 1)
4	(May 1)	(May 7)	(May 14)	(June 1)
5	(June 1)	(June 7)	(June 14)	(July 1)
(etc)				

Figure 5-15

Here is status reported on a *burn-down* chart. See the technique, *Burn Charts*, for more details on this method.

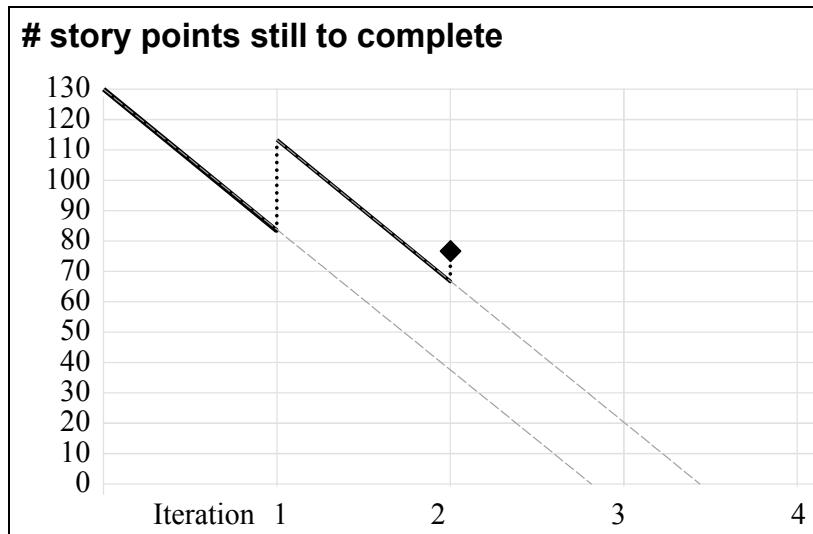


Figure 5-16

The following is a chart showing changing status over a two-month period. White means not started, medium-gray means completed.

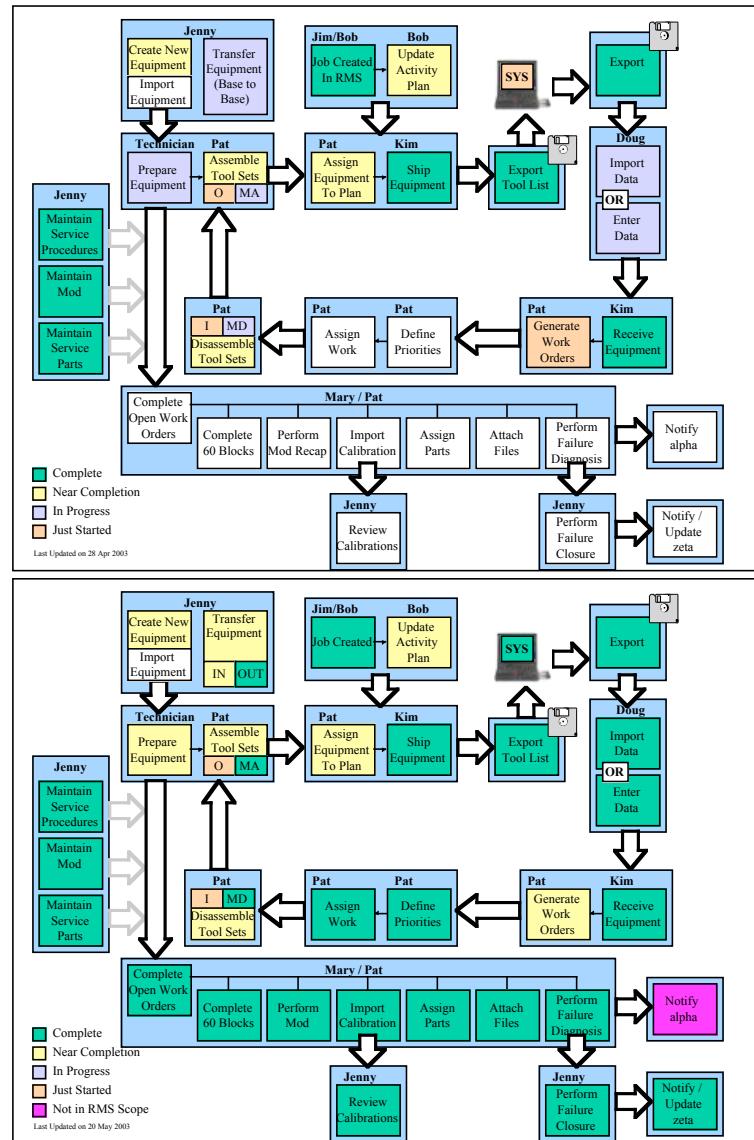


Figure 5-17. (Thanks to Omar Alam)

Coordinator: Risk List

The risk list consists of the top risks facing the project, how significant and likely each is, the damage it can cause, and what prevention or response is possible. I like to track whether they have grown or shrunk recently.

The risk list is produced and maintained by the *Coordinator* in conversation with the team, updated weekly or monthly. It is used in discussions with the sponsor.

Creating the risk list is trivial. The hard part is deciding what to do in case the risk materializes. As with the mission statement, the brevity of the work product belies the thought put into constructing it. There is a very satisfying feeling from watching a risk materialize and realizing that one has a response ready for it.

The risk list can live on an information radiator or in a spreadsheet.

Here is an example from the NICS-NBO project. This plus the project status resulted in a one-page project summary sheet examined at every steering committee meeting.

Priority	Risk	Possible result	Likelihood	Response	Change	Milestones affected
1	Very high performance demands in a new, untested technology	unexpected failures in system operation	quite high	test performance early	as before	2-5
2	User community changes their idea of requirements late on	delays in delivery	medium	show users as soon as possible	as before	5
3	Need to test on factory floor as well as in development	misunderstandings, delays in delivery	low	clear test routines	less	2-6
4	Year 2000 compatibility	bad data in system causes faulty operation	medium	special Y2K test	less	6
5	Data conversion problems	delay in delivery	medium	create conversion strategy	as before	6-7

Figure 5-18

Coordinator: Iteration Plan □ Iteration Status

The fine-grained iteration plan lists what is being developed in this iteration. Each item may be something that could take from a day to two weeks to develop, depending on how long the iteration is. The iteration plan may show dependencies between work items (if, for example, it was produced *Blitz Planning*), it may treat all work items as being independent (if, for example, it was produced by XP's planning game or is simply a list of product features), or it could be a marking of selected sentences in the use cases.

The iteration plan is produced by the *Coordinator* working with the development team. It is kept up to date by the *Coordinator* in the same way, usually becoming the iteration status directly.

A good iteration plan lists all the work items the team has to complete so that they can be checked off when completed, and so that the sponsor and the developers can see all significant work items. The granularity varies with the iteration length and the team's knowledge of their assignment.

Here is an iteration plan, courtesy of Thoughtworks corporation. In this case, one XP user story is written on each flipchart. Tasks for each story are written on sticky notes and attached to the flipchart (the tasks below the flipcharts have been taken out of scope for the iteration).



Figure 5-19 (Thanks to Thoughtworks)

* * *

The iteration status lists the state of the iteration with respect to the plan. It is often written on an information radiator. It is usually just the iteration plan items with their current work state, usually marked either *started* or *done*, with nothing in between. I have seen some require a third mark, *integrated*, for when the item passes integration testing. Although the tasks are often so short that it is not really useful to mark % complete, the first example below does show one creative way to do this.

The iteration status is created by the *Coordinator* in conversation with the development team. Updating is in principle done by each developer, but typically that requires directed effort by the *Coordinator*. The status chart serves to broadcast to all the rate of movement through the iteration's work list.

A good iteration status is easy for all to see and shows at a glance what has been done versus what needs doing.

Here is an iteration status for the user stories on flipcharts, above. Each was copied to a sticky note, started at the bottom left of the graph, and moved to the right as it progressed, up as its quality improved. Viewers could tell the state of each user story by how far to the right and how high each story's sticky was. This picture was taken toward the end of a one-month iteration.

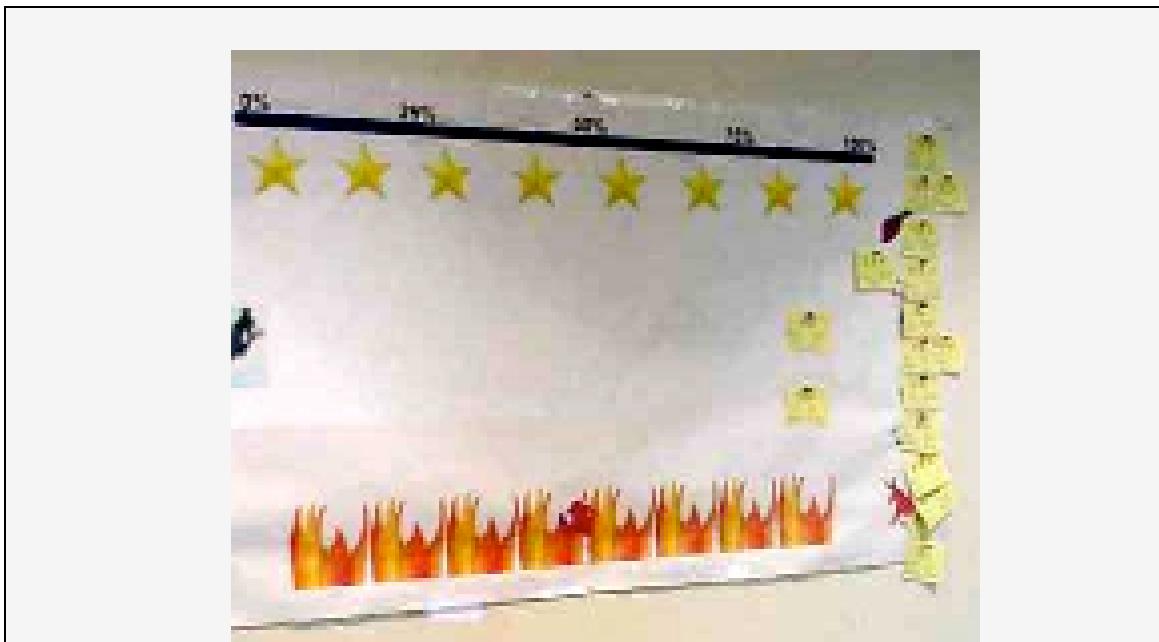


Figure 5-20. (Thanks to Thoughtworks)



This is an iteration plan marked up for iteration status. The two boxes on the left indicate "started" and "done" (there is, of course, no interesting in-between state). The numbers on the right are relative size estimates.

In this chart, the red index cards are the task cards from the planning session. They copied the text onto the whiteboard to track iteration status. Here we see the team using one-week iterations, with 15 work days available between them. They are estimating they can get 9.5 relative units of work completed this week.

Figure 5-21 (Thanks to Jeff Patton and Tomax)



Increasingly often we see people move the sticky notes across columns to show changing status. The columns are: *not started*, *started*, and *done*.

This picture was taken at their thrice-weekly status meeting. Géry is moving an item's marker from *started* to *done*.

Figure 5-22 (Thanks to Géry Derbier.)

Coordinator: Viewing Schedule

The viewing schedule is a listing of when, during the iteration or delivery cycle the *Ambassador User* or other users should plan to come and see the new growth of the system. This work product is needed when the users' time is not so easy to get, they have to travel far, or there are many of them. It is not needed if the *Ambassador User* visits the project every day or every week.

The viewing schedule is produced by the *Coordinator* in cooperation with the development team at the start of an iteration. The development team uses it as a reminder of when to target completion of their work (minor milestones). It is a living document, the *Coordinator* changing it as necessary to balance two opposing forces: giving viewers testers as much advance notice as possible and giving the developers the most time to react to the changes in their design. It may live on an information radiator or in email.

As noted in the section *Reflection on the Process*, and also discussed in "We colocated and ran two-week iterations – why did we fail?," the viewing schedule is more important to the outcome of the project than its modest size would indicate.

This example is from the Boy Scouts tracking system:

Release 1: March 1, 2002. Input system to allow scoutmaster to enter personal information about scouts and personnel including name, address, phone, email, rank (where applicable), etc.

Viewing 1: February 15, 2002

Screen shots or screen mock-up sketches

Viewing 2: February 22, 2002

Completed databases and GUI screens

Viewing 3: March 1, 2002

Completion of Release 1

Figure 5-23

Business Expert and Ambassador User: Actor-goal list

The actor-goal list is a two-column table or use case diagram. In two-column form, the left column names the person, organization or computer system that will drive the system, calling upon its service promises. The right column names that service promise, which is the actor's goal with respect to the system, what he/she/it wants to accomplish using the system. The goal, a short verb phrase, becomes the name of the use case describing the details of the function.

The actor-goal list is created primarily by the *Business Expert* and *Ambassador User*, and is reviewed for completeness and prioritization by the *Sponsor*. It gets referenced by everyone. It gets created either within or immediately after the *project chartering* activity, and updated as needed over the course of the project.

A good actor-goal list captures every primary actor (one that will drive the system) that the system must recognize (must show specific behavior for), and for every primary actor, every major goal of that actor with respect to the system. Goals are stated at "user task" level, i.e., complete transactions of value to the actor. (Goal levels are described in detail in *Writing Effective Use Cases*). A well-constructed actor-goal list contains entries that the *Sponsor* and *Business Expert* feel are at a reasonable and interesting level of activity (not too detailed, not too high-level), and they can review to declare that it is complete, omits a goal, or has unnecessary goals.

The project team may decide to start work on the project even when the actor-goal is known to be incomplete.

The actor-goal list is an index into the total functionality needed from the system. It serves both as a low-precision requirements statement and also as a scaffolding for the project status. It is used to help construct both the coarse-grained project plan (see the final example) and the fine-grained iteration plan. Some people use it directly in reporting project plan and status.

When used for reporting the project status, the use case names (goals) are useful for the first few releases. Later in the life of a system, the functions being requested tend to get so small as to not be worth writing use cases for. They tend to be *features* such as, "hyphenation," "export to RSS format," and the like. Once that moment is reached, the team often changes to tracking the fine-grained progress by features rather than use cases. Why don't we just start with features, then? The first reason is that there are usually too many features. The team needs a shorter list to work with, to see at a glance what they will get. The second reason is that at the start of the project, the users need to see what the system will do for them in their work context. That is difficult to get from feature lists, but is exactly what use cases provide.

I include the actor-goal list as a separate work product from the requirements file (which follows) because it gets created earlier than the rest of the requirements file, and often it is maintained separately.

Here is an example:

Actor	Goal
Requestor	<i>Buy something</i>
	<i>Initiate an request</i>
	<i>Change a request</i>
	<i>Cancel a request</i>
	<i>Mark request delivered</i>
	<i>Refuse delivered goods</i>
Authorizer	<i>Change authorizations</i>
Buyer	<i>Change vendor contacts</i>
Approver	<i>Complete request for submission</i>
Buyer	<i>Complete request for ordering</i>
	<i>Initiate PO with vendor</i>
	<i>Alert of non-delivery</i>
Authorizer	<i>Validate approver's signature</i>
Receiver	<i>Register delivery</i>
Any	<i>Check on requests</i>

Figure 5-24

Business Expert: Requirements File

In a conversation about requirements back in 2000, Kent Beck made one of his provocative pronouncements, that "there are no such things as *requirements*, there are only *wishes*." As so often, I have found his pronouncement to be right on the mark. Try rereading this and related sections with the thought in mind that when someone says, "The requirement(s) . . ." they are really expressing something closer to a wish, which is probably adjustable under suitable circumstances.

Particularly with incremental *delivery* with product and process feedback, there are plenty of opportunities for the sponsor and user community to learn that they didn't really *require* what they originally asked for, but perhaps something else would be more cost-effective or more useful, or simply adequate in the circumstances.

Crystal Clear is already difficult enough to adopt without me breaking standard convention yet again, and calling this section "The Wishes File." Therefore, I stay with Requirements. However, plant the seed that they are closer to wishes, and see if that improves the communication between the sponsors, users and developers about just what should be produced.

Crystal Clear requirements are not intended to be so complete and comprehensive as to be clear specifications for an external contract programmer not familiar with the domain (writing requirements specifications that explicit would be a waste of money in the Crystal Clear context). The requirements file is intended to hold in place requirements information that might otherwise be forgotten, as records of decisions that the team needs to remember, giving a context to those decisions.

A good requirements file for a Crystal Clear project communicates to the people on the project with their particular knowledge base and communications channels. The more they know and the more they can learn by talking to someone close by, the briefer the requirements can be.

The requirements file is a collection of information indicating

- what is to be built,
- who is intended to use it,
- how it provides value, and
- what major constraints affect the design.

The requirements file may be a written document, or it may simply be files scattered across the disk that, taken together, comprise *the requirements*. In many organizations the concept of requirements as a *document* is both foreign and unnecessary. What is important is that there is a place where these things are recorded.

There are many sensible formats that can be used for the requirements file, and many timing strategies on *when* to fill in the details.

- In some cases, such as for a fixed-price, fixe-scope bid, the requirements need to be locked down early in the project.
- In other cases, such as for shrink-wrapped products, in-house development, or time-and-materials contracts, the requirements may be evolve over time or be created in a just-in-time fashion.

Crystal Clear does not legislate any format or timing strategy. Typically, though, requirements do evolve over the course of the project, and the team needs to both update and adjust to the changes at the start of each iteration. The team needs to discuss periodically in their reflection workshops just how and how often the requirements should be updated.

Quality in a requirements document is relative to the team. One test for *clarity* and *completeness* is if a company executive or other business expert from the same company sits down with the business expert who wrote the requirements, they can be satisfied that no areas are forgotten, and within each area, they understand what is written. *Comprehensiveness* means that that all issues have been thought through (this is of course harder to test for).

In general, a full sample requirements file is too long to insert here. One is available online at <http://Alistair.Cockburn.us/crystal/articles/prts/purchaserequesttrackingsystem.htm>. It outlines typical sections in a complete requirements file, such as

- the mission statement,
- the actor-goal list, followed by use cases with annotations, particularly performance requirements,
- particular business rules for the use cases that may be needed; references to other sources of business rules, legal requirements, etc.,
- data descriptions, formats and validation rules,
- technology requirements,
- I/O protocols and formats for external communications
- (a Glossary of business terms is often recommended).

It is unlikely that a Crystal Clear type of project will need all of these written down, given that they have close communication with the users and the sponsor. As an example, Kay Johansen contributes the following *complete* requirements example with the introductory note:

Alistair,

I dug around and came up with two examples of "requirements" from a project I was on around 1999. It would probably count as a Crystal Clear project: two programmers in a room (one of the programmers is also a user expert) a couple of other user experts in nearby office, plus access to the "paying customers." Some automated tests. Release about every 2 months to customers.

We drew up the requirements jointly with the user experts for the two new features that mostly defined the 2.61 update of the product. The product was a small business accounting/inventory/service contract system. The requirements got sent to the customers requesting these features, for their review. Customers were satisfied, and the programmers worked from this document.

Contracts

User Story

- ? Create a service contract with a pre-paid balance that is debited based on services rendered to the customer.
- ? The contract maintains an unearned balance, minimum balance and amount to be billed.
- ? The contract should include discounted percentages for labor and parts for the services rendered under contract.
- ? As service calls are cleared, the labor and material costs (less discounts) are deducted from the unearned balance of the contract.
- ? The customer is billed the billing amount when the contract unearned balance is less than the minimum balance.

Code Changes

- ? Contracts
 - ? Add a *Bill by Services Rendered* check box to indicate that the contract balance is debited based on services rendered.
 - ? Add a *Minimum Balance* to the contract so that the contract can be billed when the current balance falls below the minimum balance.
 - ? Add a *Renewal Billing Amount* to allow the billing system to renew the contract and replenish the contract balance.
 - ? Add a *Labor Discount Rate* for labor calculations.
 - ? Add a *Materials Discount Rate* for parts and supplies calculations.
 - ? Store this additional information in the database.
- ? Contract Billing
 - ? The contract billing system needs to check for contracts that are Our Contracts (Services Rendered) that have gone below their Minimum Balance. It will then create contract billings based on the Renewal Billing Amount.
 - ? Billings that are negative will sum the absolute value of the negative amount with the Renewal Billing Amount to bring the account up to the positive minimum.
- ? Dispatch Console
 - ? When a call is cleared for Our Contracts (Services Rendered), all labor and material costs will be discounted by the *Labor Discount Rate* and the *Materials Discount Rate* for the contract, respectively.
 - ? The cleared call will debit the unearned balance of the contract and the call will show a zero balance due. If the services exceed the current balance on the contract, the contract balance will go negative and will bill for the excess cost on the next renewal.

- ? The cleared call will debit the unearned balance in the GL Account and credit the revenue account. It will use the same accounts specified for the accrual unearned balance and revenue transfers.

Block Time Billing

User Story

- ? Create a Sales Order for 10 hours of training at \$100/hr (overhead of \$50/hr) and 5 sets of documentation at \$50/ea (cost of \$25/ea). Bill this as fulfilled.
- ? Invoice immediately for 3 sets of documentation and 5 training hours.
- ? The customer pays \$650 for this invoice.
- ? Fulfill 1 set of documentation and 4 training hours.
- ? Examine the status of the sales order – view amounts ordered, fulfilled, and billed in both dollar amount and item quantities.
- ? Fulfill the remaining 4 sets of documentation and 6 training hours.
- ? Invoice for the 2 sets of documentation and 5 training hours that have not yet been invoiced.
- ? Customer pays \$600 for this invoice.
- ? The Sales Order is now complete.

Code Changes

- ? To create a Sales Order you would use the same method that exists now. Training Hours would need to be set up as a non-stocked ICItem. The Sales Code on the item will determine where the unearned revenue will go.
- ? You can open the Sales Order and view the current status of all line items sold. This information is available in both Dollar Amount and Quantity. The “Bill” command button will allow you to generate an invoice for this, or a portion of this, Sales Order. The invoicing action will debit the A/R account and credit Unearned Revenue.
- ? When the customer pays on the Invoice the existing functionality will be used with appropriate adjustments to the Cash and A/R accounts.
- ? The Sales Order may be incrementally fulfilled by opening the Sales Order and using the “Fulfill Items” and “Fulfill Services” buttons. Fulfillment will make appropriate adjustments to the Unearned Revenue, Revenue, Cost of Goods Sold, Inventory, and Cost Applied accounts.
- ? A new report will be created which allows you to see the current status of the Sales Order including all line items with amounts ordered, fulfilled and billed. This information would be available in both Dollar Amounts and Quantities.
- ? If the total Sales Order has not yet been invoiced at the time of final fulfillment, an invoice can be generated at that time. The invoice process will debit the A/R account and credit the Unearned Revenue account.
- ? The customer can pay this final invoice using the existing functionality.

Figure 5-25 (Thanks to Kay Johanssen)

Business Expert and Ambassador User: Use Cases

A use case is a text collection of scenarios describing how an external actor achieves something of value using the system. It starts with a success scenario in which the external (*primary*) actor requests something of the system. That scenario describes the actions and interactions between the system and various other actors in satisfying the primary actor's request. After that, it describes how the system behaves when things go wrong, possibly failing to satisfy the primary actor's request. Use cases describe what the system should *do*; they don't capture UI design, performance requirements, interface definitions, or data definitions.

The use cases are created jointly by a *Designer-Programmer*, a *Business Expert*, and an *Ambassador User*. Responsibility can move between those three roles. I write that the *Business Expert* is responsible for them, but in my experience, it doesn't matter so much which one of them is primary, as long as the others are proof-readers and knowledge providers. It does cause trouble if one of them is absent. Non-programmers tend to write loosely or ambiguously and not fully consider consequences and alternatives. Programmers are usually good at those, but tend to get too detailed, start describing their intended design, and get business rules wrong. Working and reviewing together, they can cover each others' mistakes.

A use case is often written in two phases, the success scenario in a first stage, the failures scenarios later, as the team needs delves deeper into the requirements, either to estimate the complexity of the job, or to work on the design. At some point, the writers or the tam may declare that the use cases are "complete" for the iteration, meaning that no more use cases will be considered for that iteration, and that no more details or extensions added to them

A good use case is easy to read, identifies the system's responsibilities, and considers the most interesting failure cases. The complete set of use cases covers all the goals the primary actors have with respect to the system, including covering every external event that triggers the system. A good Crystal Clear use case is different than a good use case on a larger, distributed or life-critical project. Because the Crystal Clear team sits together or very close and can get clarification on the content easily, they can write in a briefer, more casual style.

Most people get too fancy about the use case format, making them very detailed rather than communicative. For Crystal Clear projects, I suggest starting with the *two-paragraph* form. This consists of a title, a description of success in the first paragraph, and a description of failures and recovery in the second. Once the team has practiced with the two-paragraph form, they can investigate other use case formats (see *Writing Effective Use Cases*). For example, even though I understand the two-paragraph form, I usually find it just as easy and more legible to write with numbered steps. I always write down the use case's goal level, the name of the system being designed, and the system's stakeholders and their interests in use case. This additional information

communicates a lot to the readers without adding a lot of time to the writing. Try the two-paragraph form, first, though, before adding anything else.

Here is an example of a user-goal-level use case in two-paragraph from the BSA system:

View a Scouts' Progress (user-goal level)

The *scoutmaster* searches for and selects an individual scout. The system shows the information about the rank the scout is currently working on.

If the scout is not in the system, the scoutmaster can Add a New Scout.

Figure 5-26

Here is an example of a summary, two-paragraph use from the PRTS system. (**Note:** An underlined phrase represents a hyperlink to another use cases.)

Buy something (summary level)

The *Requestor* initiates a request and sends it to her or his *Approver*. The *Approver* checks that there is money in the budget, check the price of the goods, completes the request for submission, and sends it to the *Buyer*. The *Buyer* checks the contents of storage, finding best vendor for goods. The *Authorizer* validates Approver's signature. The *Buyer* then completes the request for ordering, initiates a PO with the *Vendor*. The *Vendor* delivers goods to *Receiving*, gets receipt for delivery (out of scope of system under design). The *Receiver* registers delivery, send goods to *Requestor*, who marks request as delivered.

At any time prior to receiving goods, *Requestor* can change or cancel the request. Canceling it removes it from any active processing. (delete from system?) Reducing the price leaves it intact in process. Raising the price sends it back to *Approver*.

Figure 5-27

Ambassador User: User Role Model

A role model is a two-dimensional map showing user roles and their relationships. It highlights the *focal* roles, the ones the system should satisfy the most (see *Essential Interaction Design* on page 98). A user role is a name representing a high level goal that the user of the system might step into; similar roles cluster closely together and dissimilar roles appear farther apart. Relationship lines connect roles or clusters.

The role model is created jointly by a *Designer-Programmer*, a *Business Expert*, and an *Ambassador User*. Although responsibility can move between those three roles, I attach it to the *Ambassador User* to indicate the importance of this person in the process.

The role model is used

- to prioritize work when roles, business value, focal status and frequency of use are issues that affect priority of feature development,
- when writing use cases: the collaborations between roles furnish use cases with additional actors or dependencies,
- to help make specific decisions about user-interface and interaction design,
- in testing: the tester assumes the goals and attributes of a user role.

A good role model names all the roles that will use the system, reveals quickly the focal roles and key relationships, and serves as a reminder to the team as to who their software will be serving.



Designer-Programmers: Screen Drafts, System Architecture, Source Code, Common Domain Model, Design Sketches and Notes

The question always arises as to just what documentation is needed. In Crystal Clear the answer is:

Whatever the sponsor and the team decide.

There are four reasons for this answer:

- The system will always be changing, both in function and in design. The time, effort and resources put into documenting the design subtract from those making progress toward delivering the system. The documentation will always be somewhat out of date and therefore incorrect.
- At the same time, the designers need to leave a trail behind, to jog their own memories some months down the line, and to lead other designers to understand what they have done.
- The above are in conflict, and the correct resolution of them depends on the game of software development being played on this particular project. Only the sponsors and the team can guess what trail is the optimal one to leave behind, balancing the need for progress and the need for clues. What is certain is that I can't, in this book, make that decision for all small teams everywhere.
- Finally, technologies change. The correct form of documentation changes somewhat with the technologies being used. Database systems, mainframe systems, scientific systems, object-oriented systems, real-time systems, all have different appropriate documentation formats. These things change often enough that, again, it is not for me in this book to legislate what is correct for the team; only the team knows.

Crystal Clear projects operate with two time horizons with respect to documentation: getting the software written (the less time spent documenting the better), and handing the software over to a different group (the richer the documentation the better). One might think of the first set as greedy actions (meeting the first goal in the cooperative game), and the second as investment actions (setting up for the next game). The two are in conflict with each other, so the team inevitably plays a game of brinkmanship with the documentation.

Ideally, the team produces the archival documentation as late as possible, as fast and cheaply as possible. This is allowed in Crystal Clear.

However, "none" is not a valid answer. That would only be a valid answer if it is known that no one will maintain or extend the system later on. Such a project is unlikely to need even Crystal Clear for its rule set.

Thus, teams using the Extreme Programming rules within a Crystal Clear project must discuss with their sponsors what forms of documentation to leave behind. They can use either XP's planning game (Beck 2002) or the *Blitz Planning* technique to integrate this need for documentation into the ongoing project work. This is the only addition to XP needed to be a valid implementation of Crystal Clear.

In addition to discussing *what* to document, the team needs to discuss *how* to document it, and *when*. The *when* part is easy to understand although not so easy to choose - the later they leave it, the more up-to-date it will be and the less it will need to be changed. However, the later they leave it, the more likely they are not to do it at all. There is inevitably brinkmanship involved in making the optimal choice.

The team should be creative about *how* to document their designs. Typed-in, paper-based documentation is one of the most expensive, time-consuming and least communicative forms available (never mind that it is traditionally the most frequently requested).

- Effective small teams supplement their typed-in documents with photographs of their whiteboards, flipcharts and other information radiators.
- There is no reason why a hand-drawn diagram can't be scanned in and stored online as an image, included in HTML or other documents.
- Paper napkins happen to be my favorite documentation medium. They can be posted on the wall, photographed or scanned.
- The team can videotape one of their designers explaining and discussing a section of the system design with a designer from *outside* the team. This discussion should be held to 10-15 minutes per topic, as described in *Agile Software Development*.

Smart teams are likely to come up with their own alternate cheap and effective documentation forms.

The points to note are that

- *some* documentation is needed,
- the team is in the best position to decide what, how and when,
- the team in conjunction with the sponsor are in the best position to decide how much is appropriate for this particular project.

Finally, software development is not just a game of *communication*, but also a game of *invention*. Some work products facilitate the creation of new ideas, help the people make new moves in the game. Paper-based prototyping (Snyder 2003) is an example, as are CRC cards (CunninghamURLcrc, CockburnURLcrc) and the instance diagrams in UML (FowlerUMLD).

Invention-enhancing work products are not archived, and as a consequence we tend to overlook their importance in a project. To help correct this imbalance, I call out *screen drafts* as a specific work product. These help the team work out the user interface

design collaboratively with the users. They are specific, useful work products with short lifetimes.

In the following sections, I describe three work products that almost certainly need to be produced – system architecture, common domain model, and screen drafts – inside of the larger category, "design sketches and notes, as needed."

Designer-Programmer: Screen Drafts

Screen drafts are low-cost renditions of the way screens will look, used to explore the ways that users might interact with the system, and invent better ways for them to get their goals accomplished. Very often they are just drawings on paper (Constantine???, Snyder???). The team may prefer screen-prototyping software.

Screen drafts are created primarily by whichever *Designer-Programmer* is designing the user interface, in tandem with the *Ambassador User*. They may hang on the wall during the iteration, or they may get converted to software prototypes or real code and discarded. I mention them here because, although they are transient work products, they are immensely useful and often overlooked on projects.

A good set of screen drafts is inexpensive to construct (hence made from paper and markers or from special storyboarding software) and easy for the *Ambassador User* and other users to walk through with respect to the use cases and usage scenarios (see the technique for walk-throughs in *Essential Interaction Design*).

Here are some examples of screen drafts.

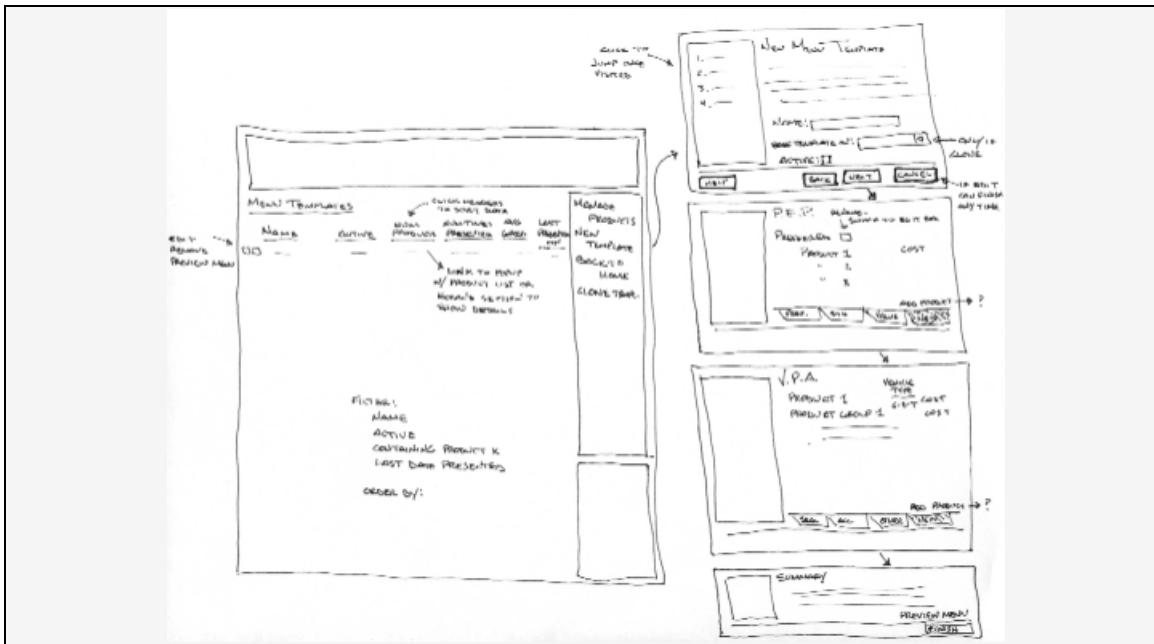


Figure 5-29 (thanks to Nate Jones)

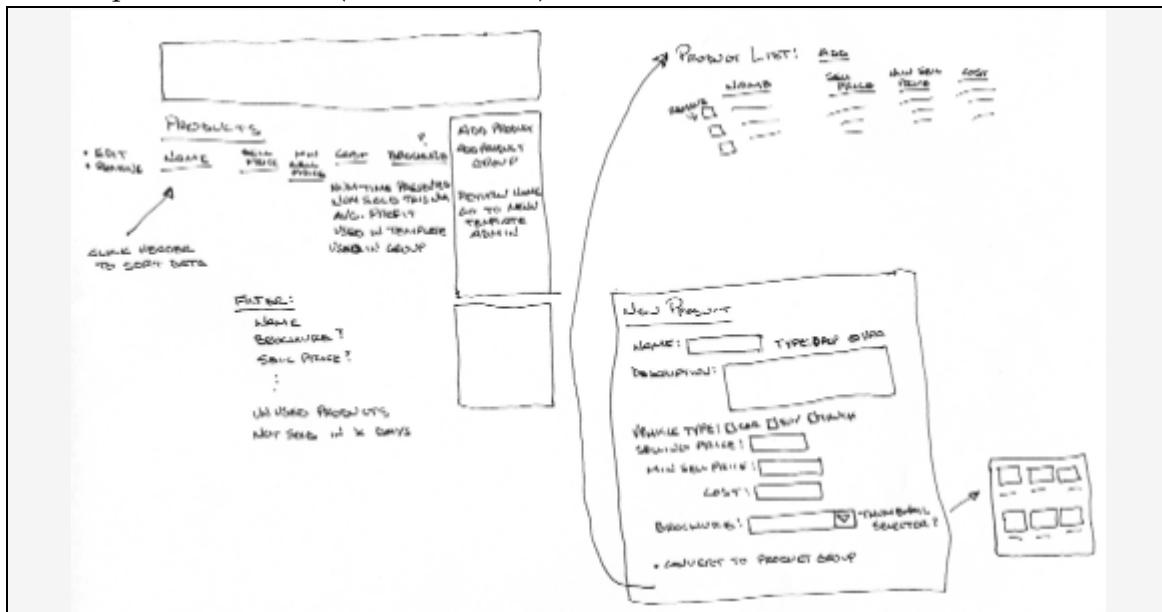


Figure 5-30 (thanks to Nate Jones)

Lead Designer: System Architecture

The system architecture description is text and/or drawings showing the major components and interfaces of the system. There is a lot of controversy about just what constitutes an *architecture*, when it should be made and how it should documented, all of which are out of scope of this book to decide. There does need to be some description of the system's main design to unify the team members work and to educate follow-on system designers. On larger projects, two architectures may occur: the technical (infrastructure) architecture and the domain architecture. *Domain Driven Design* (Evans 2003) has examples of domain architectures.

The system architecture description is created by the *Lead Designer*, usually fairly early in the first iteration. The architecture will probably evolve, particularly if the *Walking Skeleton* and *Incremental Rearchitecture* strategies are used. This means that the architecture document will need to be updated during each iteration.

A good system architecture speaks in the language of the technology being used and the *Designer-Programmers*. It may consist of a technical memo with descriptive text and drawings, or just text, or just drawings. It shows or describes the major interfaces and the interaction paths.

Here are samples of very different styles of architecture documentation:

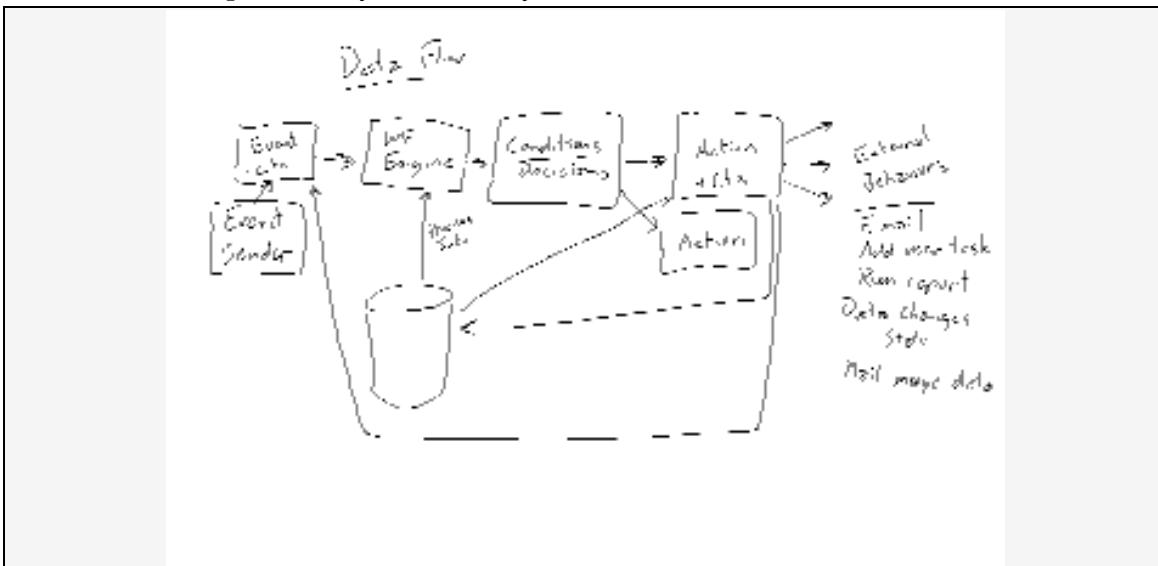


Figure 5-31. Design captured by a Mimeo whiteboard device (Thanks to Darin Cummins)

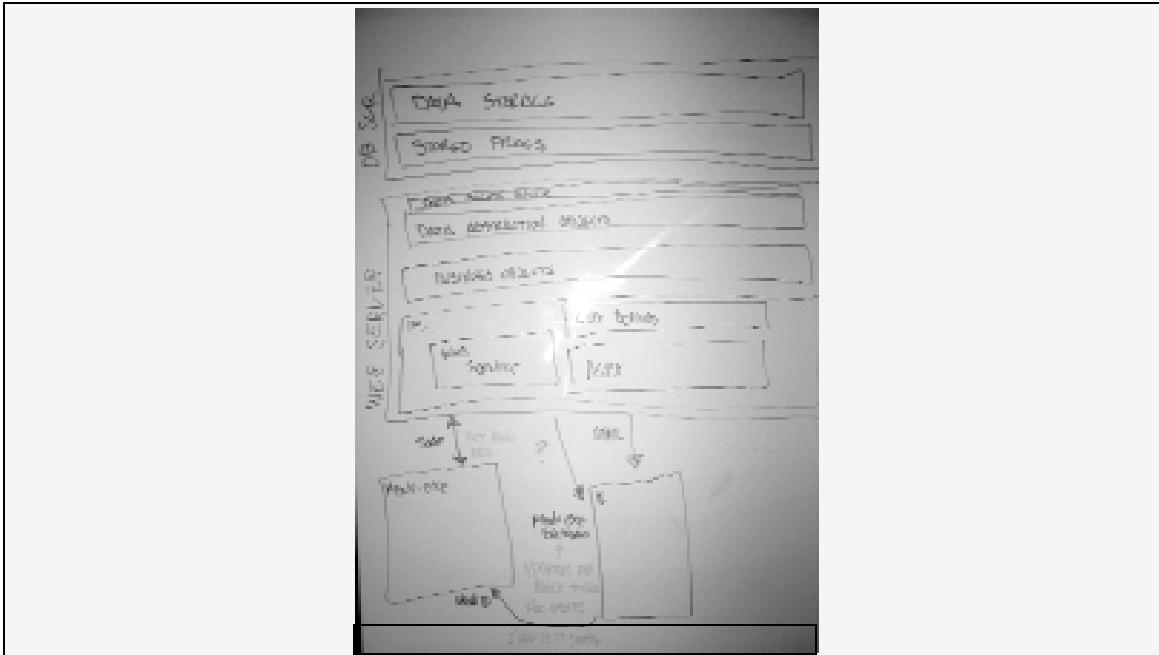


Figure 5-32. Architecture kept on a whiteboard (Thanks to Nate Jones)

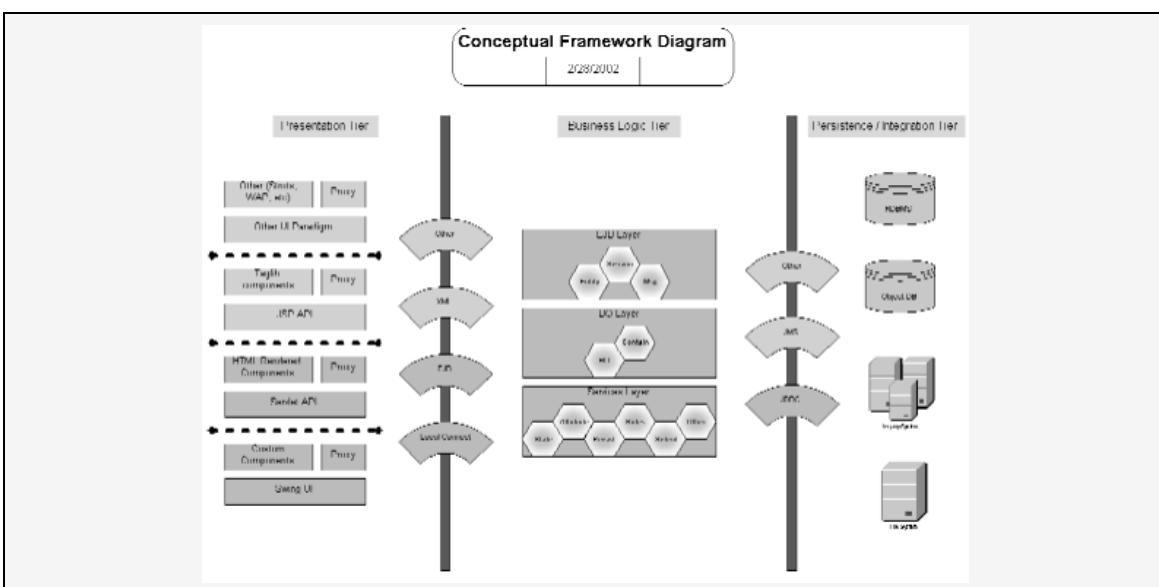


Figure 5-33 Architecture drawn into a drawing tool. (thanks to Jonathan House)

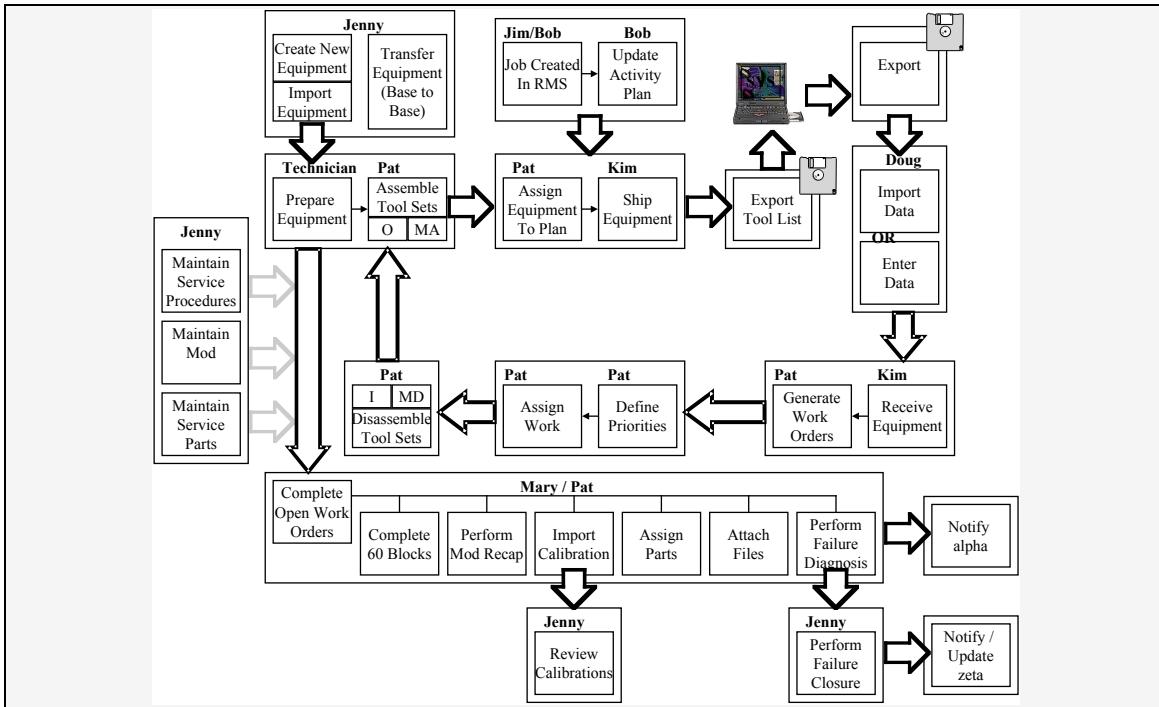


Figure 5-34 Work-flow architecture. (Thanks to Omar Alam)

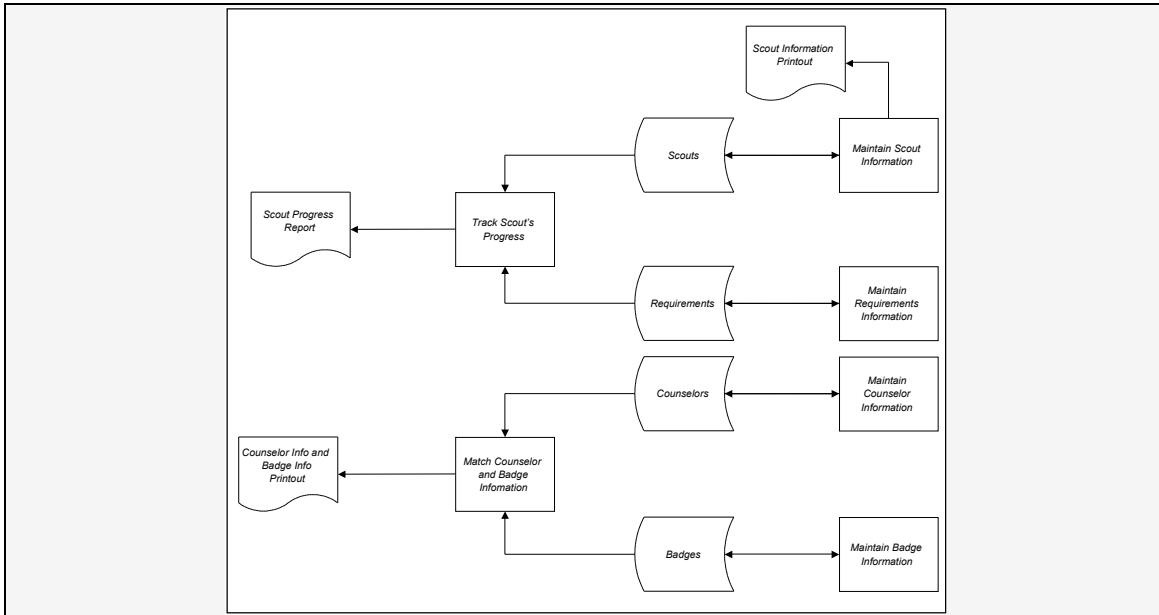


Figure 5-35. Architecture from a non-OO system

Designer-Programmer: Common Domain Model

The common domain model is typically a drawing, either a database schema or a class diagram, showing the principal entities or classes used in the system. Some teams still prefer straight text to describe their domain model.

The common domain model is created by the *Designer-Programmers* as they understand the domain and incorporate increasing amounts of it into their design. It is reviewed by the *Business Expert*, who may help to construct it in the first place.

There are several timing strategies for constructing the domain model. In some cases, there is an intense domain-modeling activity at the start of the project, which results in draft #1 of the common domain model. The model gets revisited and adjusted continuously throughout the project after that. This is the mode of work described in *Surviving Object-Oriented Projects* (pp. ??). The other strategy is a just-in-time strategy, in which the model is built incrementally and minimally, with little or no looking ahead at future requirements. Crystal Clear does not legislate which of these strategies is preferred. That is up to the team to discuss and decide.

There is a reason I call this work product the "common" domain model, as opposed to the "analysis" model, the "design" model or just "domain" model. Having multiple models creates two hazards to the project. The first is double maintenance: you have to resynchronize each model as soon as either changes. Typically, that doesn't get done, so the models simply get out of sync. The other is that there is a tendency to think that the analysis model is magically "true" in some way, and the design model needs justification for each difference. In actuality, there are many "true" models of the domain. The final implementation should contain one that is both correct and a good design with respect to maintenance, performance and system resources. The initial, analysis model may be correct, but is quite likely to be weak with respect to those design criteria. As a result, it is a better strategy to keep only one model of the domain, and evolve it so that it is both a valid domain model and a strong design. This topic is discussed at length in *Surviving Object-Oriented Projects* (Cockburn 1998). Eric Evans refers to the evolving common domain model with his pattern, *Ubiquitous Language* (Evans 2003).

We decided on project Winifred that the set of classes to put into the common domain model should be the "public elements of persistent classes." I found this to be a useful guideline. The model that results is very much what an analyst would call an "analysis model," which means that it contains only items that should be in the *Business Expert's* (and probably the *Ambassador User's*) vocabulary and sphere of caring. At the same time, it names classes and relationships that actually exist in the implementation - it is an "implementation model." Having an analysis model that is an extract from the implementation model keeps the team from maintaining two models, and raises the likelihood that the entities and classes put into the implementation actually are meaningful business terms.

A good common domain model is both understandable to the *Business Expert* and current with the implementation.

Here are some examples.

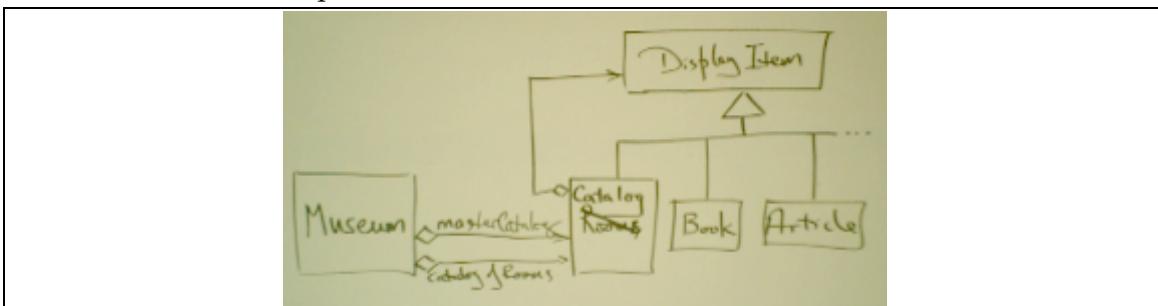


Figure 5-36. On paper. (Thanks to Alistair Cockburn)

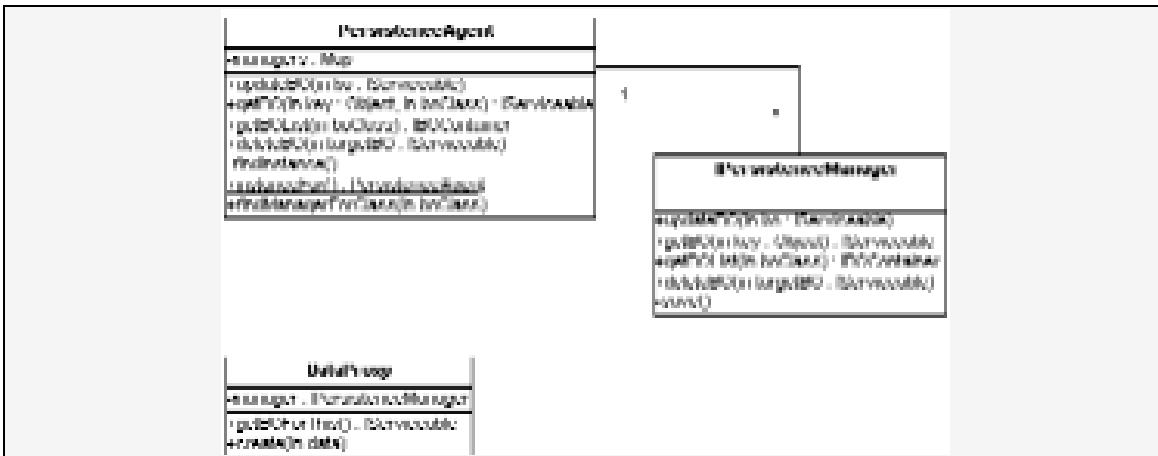


Figure 5-37. In a CASE tool. (thanks to Jonathan House)

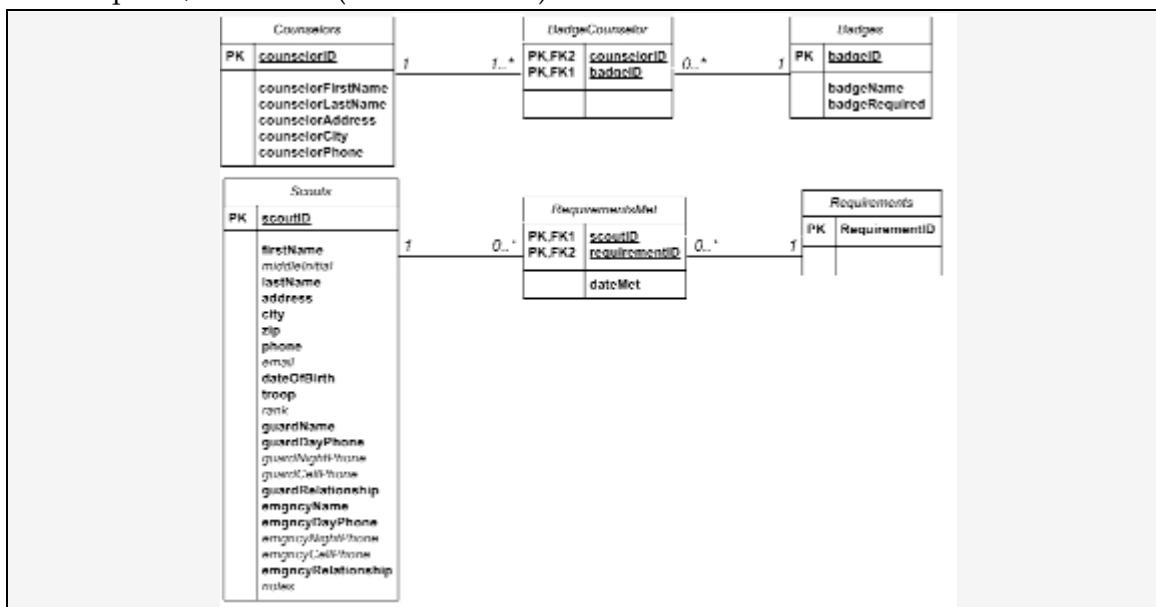


Figure 5-38. A non-OO domain model (The BSA domain model)

Designer-Programmer: Source Code and Delivery Package

I have to include this page just so people don't think I ever forget that the designer-programmers actually produce source code and delivery packages besides all the various items used to plan, track, test and describe them.

Crystal Clear does not regulate the language, naming, formatting and commenting conventions you use. Those are for your team to decide.

I hope you already know what source code looks like. If you really want to look at some, turn to the JUnit test example :-)

Designer-Programmer: Design Notes

Crystal Clear requires "design sketches and notes, as needed." This is an open-ended phrase, allowing teams to write technical memos, create UML diagrams, create wikiwiki webs describing their design (wikiURL), take photographs of their whiteboards, or videotape mini-lectures about the design.

The design notes are produced continuously throughout the project by the *Designer-Programmers*.

In terms of the theory of software development (see *Questioned*), design notes serve one of two purposes and have different paybacks in the economic-cooperative game.

- They can serve to *remind*, as part of the *greedy* action set, aimed at the primary, near-term goal of delivering this system soon.
- They can be among the *investment* actions, aimed at the secondary goal of setting up for the next game.

Even within the greedy action set, *reminder* markers serve a valuable service in reminding people what they previously decided. Being only for the people who were present (including yourself a month or two later), they are naturally drawn on napkins, flipcharts and the like. These markers rarely need to be redrawn (and may even lose information when redrawn).

Informing markers are created as part of the investment activity, to help those who come later to catch up to the group. They are more longer and more laborious to produce. The economic payback is that the same information doesn't have to be repeated over and over again by the busy senior developers on the project.

It is useful to keep several things in mind when deciding what and how to produce these markers. First, it will never be possible to convey the entire theory of the design through these documents. Second, even the people who have been on the project from beginning don't have a shared or complete understanding of what is in the code and what theory best describes it. That means the design notes can never be complete, consistent and fully instructive. Therefore, don't think it is your goal to make them such. Your goal is to get them *close enough* so they can ask good questions, or develop their own theory far enough that they can get into the code and find out more on their own (and hopefully, so that the theory they come up with is not too far distant from your own).

Good design sketches and notes leave a trail for another *Designer-Programmer* to follow. The best ones describe not how the system currently looks, but why certain choices were made and others rejected.

Part of both reminding and informing is retaining and conveying the history of the choices that were made. Recently, a few people have reported that using a Yahoo eGroup type of chained-email system, or a newsgroup-type system like Starteam with threaded notes is useful in this fashion. These systems allow people to follow the chain

of discussion about various particular topics. I add the note that the chain of history is more informative in both reminding and informing than is the summing up at the end of the chain. This method is clearly suited to larger and distributed teams. When the team consists only of 3-5 people located all in one room, it may be less natural to have people go online to add these notes.

There is an ongoing cost to creating, changing and maintaining these documents as the design itself evolves, and a competing cost to not creating them. Just how that tension is resolved is up to the team, *including the Sponsor*, who must consider the balance between the short-term and the long-term health of the system. It is unlikely that the *Sponsor* will be happy with no documentation at all for following teams, and so Crystal Clear requires *some* design notes to be produced.

Here are some examples of design sketches and notes.

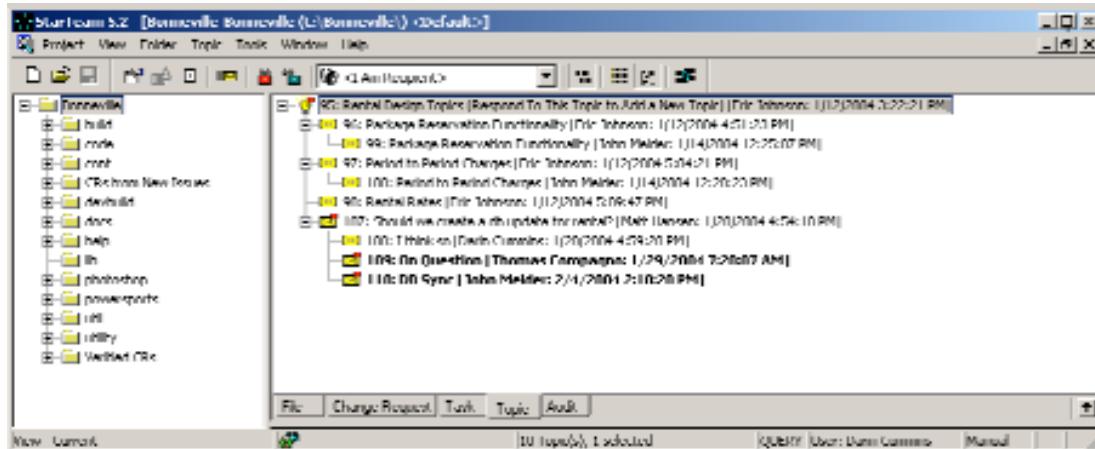


Figure 5-39. Screen shot from the Starteam newsgroup-style system, showing the threaded discussion trail. (Thanks to Darin Cummins)

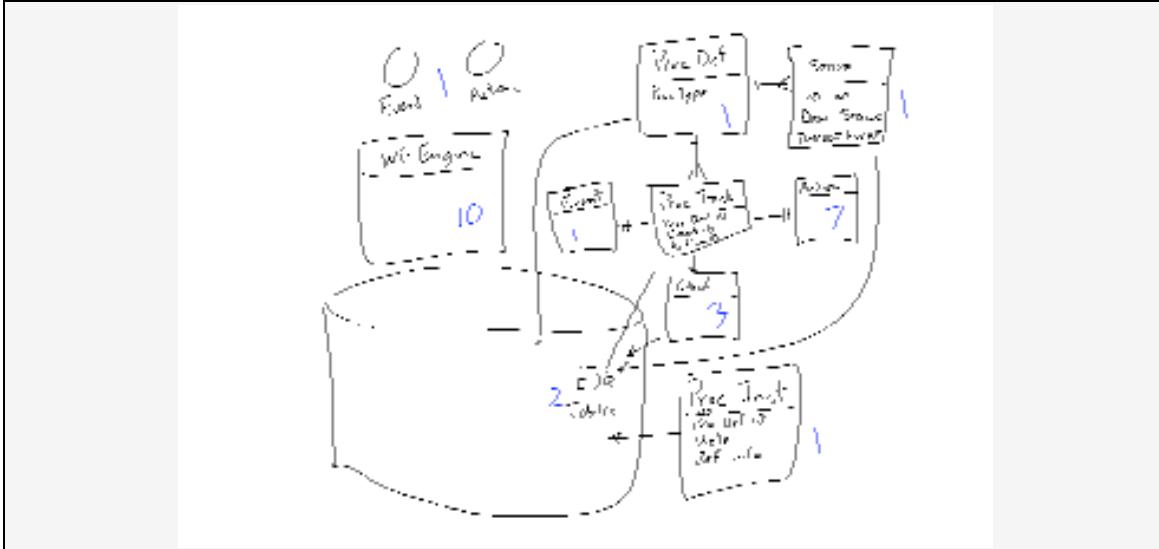


Figure 5-40. A *reminder* design note captured with the Mimeo device hooked to a whiteboard and a laptop. (Thanks to Darin Cummins, who says he periodically pulls up this and similar Mimeo whiteboard captures for re-examination)

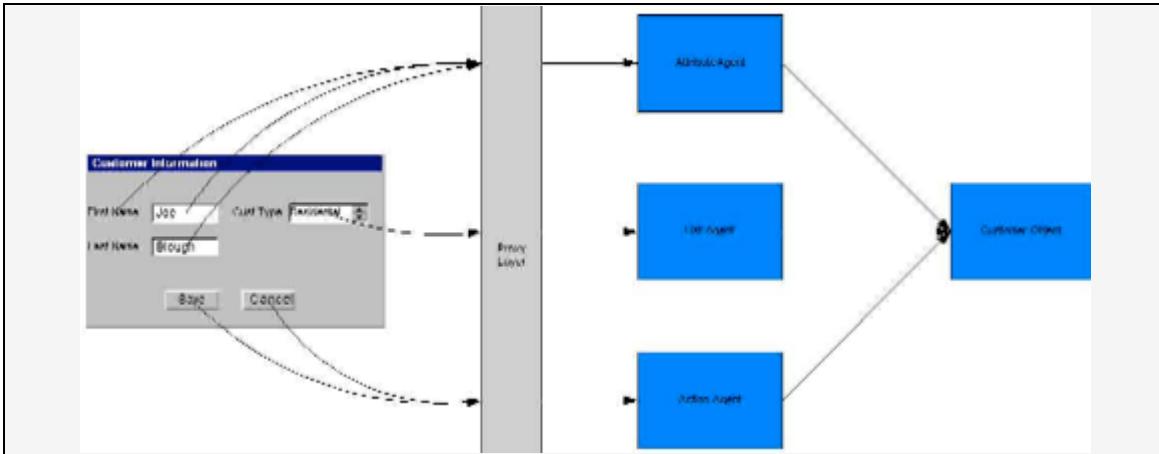


Figure 5-41. An *informing* note about binding visual components. (thanks to Jonathan House)

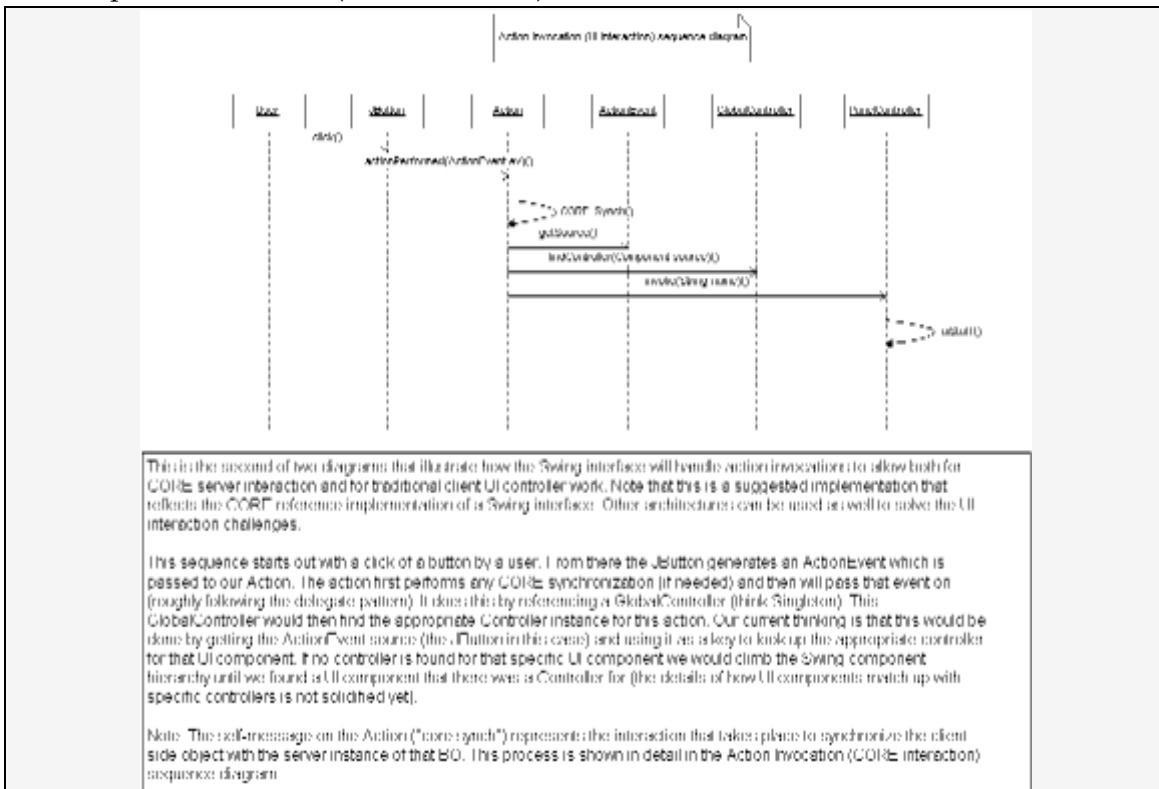


Figure 5-42. Sequence diagram annotated with a textual description, part of a series domain model fragments intermixed with sequence diagram and explanatory text, each section illuminating just one aspect of the design. (thanks to Jonathan House)

Designer-Programmer: Tests

A test is preferably executable, *code* that tests an aspect of the system. Non-executable tests involve a person following a written document that describes what the person must do to test the system. There are times when these are necessary, also.

Tests are produced continuously during each iteration. Although there may be an external test group, I wish to highlight that it is not acceptable for the *Designer-Programmers* just to write code and hand it over to a test group to repair. Creating and running tests is an intrinsic part of the *Designer-Programmer* job description.

These days, with the availability of the JUnit unit test harness and its variants, cppUnit, VBUnit, StUnit and so on, there is little excuse for not writing automated unit tests. GUI-driving acceptance tests are harder to automate. Good designers find ways to automate at least data-manipulation tests, if not mouse-, button- and keystroke tests. The test harness httpUnit and its relatives help automate testing of web interfaces. The frameworks FIT and FITnesse⁴⁵ help business people, not just programmers, write data-driven acceptance tests. There is a fairly strong literature on automated testing, including books on test-driven design (Beck 02??, Astels 03, Thomas 04)

A good unit test not only tests what it is supposed to, but has a name that shows what "concern" is being tested (the test name reads as "test *that* something is true"). Attending to readability in naming the concern means that the collection of test case names illustrates a set of concerns that went into designing the system. That list serves as a portion of the design documentation for the next programmer along (see the second example, below)

Good acceptance tests are readable to the domain expert, as well as testing the normal and boundary cases of the system. This readability serves to inform the next domain expert along.

Here are two examples of JUnit tests, courtesy of Andy Pols and Jeff Patton. Note the use of concern-based test names in both. In the second example, I include the names of five test cases so you can see how the list of names serve as *informing* markers of design concerns. (In the first example, the assertEquals message will tell the test report reader what went wrong; in the second example, the assertEquals message is telling the programmer what concern is being tested here. Both styles work.)

⁴⁵ <http://fitnesse.org>

```
public void testInitialFormDefaultsToUseTheAntBuilder() throws Exception {
    // execute
    String result = myAction.doDefault();

    // verify
    assertEquals("Should be in input mode, once the form has been initialised",
                Action.INPUT, result);
    assertEquals("Should have ant as the default builder on the input form",
                "Ant", myAction.getBuilder());
}

public void testPreventsAddingDuplicateProjects() {
    // setup
    String duplicateProjectName = BeetleJuiceStub.DUPLICATE_PROJECT_NAME;

    // execute
    try {
        myAction.setName(duplicateProjectName);
        fail("Should have thrown an exception");
    } catch (IllegalArgumentException iae) {
        // verify
        assertEquals("Should have returned the correct form error message",
                    "This Project already exists.", iae.getMessage());
    }
}
```

Figure 5-43. Test cases with "concern-based" naming. (Thanks to Andy Pols)

```
public class DatabaseModelTest extends FrameworkLocalTestCase {  
    public void testDataModelCanBeCreatedFromClientDirectives() {  
    public void testDataModelCanBeCreatedFromDirectivesFoundInXMLFile(){  
    public void testDataModelCorrectsExistingSchema() {  
    public void testTableCanDropItself()  
    public void testDataModelCanDropItselfFromDB(){  
  
    // Just the one test expanded here...  
    public void testDataModelCanBeCreatedFromClientDirectives() {  
        DatabaseModel dm = new DatabaseModel("test");  
        assertEquals("databaseModel was created and name is correct",  
                    "test", dm.getName());  
  
        Table tb = new Table("parent");  
        assertEquals("table was created", "PARENT", tb.getName());  
  
        assertEquals("database model has no tables",  
                    0, dm.getAllKnownTables().size());  
        dm.addTable(tb);  
        assertEquals("database model has a table",  
                    1, dm.getAllKnownTables().size());  
        assertEquals("table knows its parent model",  
                    dm, tb.getDatabaseModel());  
  
        Column col = new Column("parentId", java.sql.Types.VARCHAR, 20);  
        assertEquals("table has no columns", 0, tb.getAllColumns().size());  
        tb.addColumn(col);  
        assertEquals("table has one column", 1, tb.getAllColumns().size());  
        tb.addColumn(col);  
        assertEquals("table still has one column after double add",  
                    1, tb.getAllColumns().size());  
    }  
}
```

Figure 5-44 Test cases with "concern-based" naming. (Thanks to Jeff Patton)

Tester: Bug Report

A bug report is a note logging an error, saying what circumstances caused the error.

Automated test harnesses such as JUnit automatically produce such reports.

Automated build-and-test systems such as Cruise Control (`cruiseControlURL`) automatically produce and email them to the designated test owner. Manually run acceptance tests require that the tester type up test reports during acceptance testing so the team can go back and change the requirements, design and code as required.

Many Crystal Clear projects have no dedicated external test team. The users become the system testers. Even in these cases, they report bugs. A popular open source bug report system is Bugzilla and there are other open-source ones being developed (for example, <http://projects.edgewall.com/trac> provides Trac, which uses WikiWiki technology for bug reporting and tracking).

A good bug report provides enough information to make the test easy to recreate.

Here is an example of a manually typed bug report from the BSA project.

Test Incident Report	Scout 01
Test Case Identifier	AddNewScout
Incident Description	
Date:	14-Apr-03
Tester(s):	All
Actual System State:	<i>Code has a counter for the scoutID, but the database had an autocounter and could not be manually inputted.</i>
Actual Output:	<i>When clicking of New Scout, an error message would pop up stating that scoutID field could not be changed.</i>
Defects Detected:	<i>The scoutID had an error and system crashed when trying to save a new scout.</i>
Test Environment:	<i>Regular Regression Testing</i>
Attempts to Repeat:	<i>Ran the test twice with same error and crash happening</i>
Anticipated Impact	
<i>No scouts can be added into system, therefore making the system useless.</i>	
Resolution	<i>Problem was found, and the autocounter in the database was taken out so the counter in the code could work correctly.</i>

Figure 5-45 (Thanks to the BSA project)

Module	Bug Description	Status	Orig
Client	Home link produces a 404 error from the widget.html page. Linked to home.htm and should be linked to index.html.	Open	NJ
Client	The refresh rate entry box accepts non-numeric values. fresh every <input type="text" value="0.r4a"/> min. <input type="button" value="OK"/>	Open	AC
Business Tier	Receive the following error when saving a purchase order. busObj.PO divide by 0.	Open	NJ
Web Service	.disco file not found.	Open	NJ
Data Tier	Unable to run procedure app_getapo. Invalid parameter list.	Closed	AC
Database	When searching for vendors the response times out after 30 seconds.	Closed	AC

Figure 5-46. Build report. (Thanks to Nate Jones)

Writer: Help Text, User Manual, and Training Manual

I assume that you have seen help text, user manuals and training manuals before in your life, so I won't belabor the point by describing what one is. I include these in the work products list for completeness. The team will have to decide not only who writes each, but also how much of it should get written for each iteration and delivery.

Paper user manuals are increasingly being replaced by on-line help text, so it may be that from a work-products point of view, the two are equivalent. Remedyng the general uselessness of most on-line help text ("To turn on hyphenation, click the box marked "Turn on hyphenation") is outside the scope of Crystal Clear.

I assign these work products to the *Writer* role, because some teams may hire or contract an external person to write them. Others may assign them to team members.

For obvious reasons of size, I don't include an example here.

Reflection about the Work Products

That is a set of work products that I feel I can defend as a group, is quite light and still safe, and that you should be able to adapt to your situation. Some developers break into a sweat when they see a listing of all the work products that get produced on a project, so I hope that didn't hurt too much.

I want to thank the people who contributed the work product samples. They have provided a wide range of different formats for your consideration. You have seen that some people use standard, formal documentation, some use whiteboards, flipcharts, index cards and sticky notes, and some use custom notations created with various sorts of tools. Each group found a way to convey the needed information with both clarity and economy.

It is worth revisiting two ideas.

- There is no single, required, core or minimal set of work products in Crystal Clear, the way there is with the seven properties. Taking the intersection from all the successful projects I have visited yields too small a set to be generally safe, taking the union yields too large a set (unsafe due to its bulk). The set in this chapter should be close to what you need.
- The economic-cooperative game involves *two* goals: delivering this system, and setting up for the next game. Crystal Clear is for small teams communicating closely, with *both* goals in mind.

Attending to the first goal means that the intermediate work products should be quick and cheap to produce, with a primary emphasis on *reminding* markers. They should attend to *informing* markers if new people will be added to the team within the project, because they may come out ahead even within a six-month time horizon by spending time creating a learning path for new teammates to follow, so they don't ask quite so many questions for their first weeks.

The second goal is setting up for the next game. There is again a dual emphasis: improving the abilities of the team members so they will be better on the next round, and setting into place *informing* markers for the next wave of developers. Don't forget that the people on the team are themselves highly effective "informing markers" and "information radiators." Build a strategy for training the next wave of developers using the existing people (having the new people do pair programming in rotations with existing team members is one such strategy).

Finally, there are several stakeholders in the question of documentation. The current developers are interested in a light set of markers, primarily reminding ones. The future developers are interested in informative (and not-so-boring) markers for catching up to the group. And the *Sponsors* are interested in the long-term stability of the knowledge base.

None of these stakeholders is to be forgotten. I believe that with some inventiveness, you may come up with even more expressive, less expensive, and less painful ways to satisfy their combined interests.

Chapter 6

Misunderstood (Common Mistakes)

You think you are using Crystal Clear, yet your project is not working. What's wrong? Crystal Clear can fail, but let's first double check that you really are doing Crystal Clear. This chapter present sample project situations. Some of them fulfill the intention of Crystal Clear, others violate it. The purpose here is to provide you with a personal warning system that you are or are not in tune with the intention of Clear.

Misinterpreting Crystal Clear is inevitable, no matter how many words I write. One way to help reduce the misinterpretations is to discuss specific situations and questions that have arisen.

Here are some situations I have run into. The first series indicate clear violation of Crystal Clear, the following series discuss violations of intention and borderline situations.

"We colocated and ran two-week iterations – why did we fail?"

The software development community currently overuses the word *iteration*⁴⁶ and underappreciates the *user*. The literature accentuates *iterations* instead of *deliveries to real users*. XP calls for an *on-site customer*, which, while laudable, is often so difficult to arrange that organizations simply throw up their hands and don't involve a real user at all.

Recall that Property #1 of Crystal Clear is **Frequent Delivery**, not *frequent iteration*, and Property #6 is **Easy Access to Expert Users**.

The issue was driven home to me the other day when I visited a group whose habits derived from XP. They knew about pair programming, test-driven design, automated regression tests, short iterations, and all the rest. They had evolved over the year, as is quite often the case, to something similar to Crystal Clear. We were doing a *Reflection Workshop* at the time of this story. Half of the people in the room were winding up a five-month project. They kept referring to a section of the system that was "just plain wrong," despite the fact that it worked and had a good set of supporting tests.

We drew a timeline of their practices over the previous year. They had gathered their requirements from what was supposedly a well-informed set of stakeholders. After three months, they had deployed an early version of the system to the customer site using Web-X, and had held an online discussion of the system. All went well in this practice deployment.

After the fourth month, a week before our little workshop, five future users of the system came to the development site and sat down with the developers to practice using the system. That's when they told the developers it was "wrong." You can imagine this shocked the development team no end. The Web-X deployment had somehow not caused the same detailed examination of the software as the visit.

⁴⁶ In recent discussions with user groups around the world, we are finding the phrase "iterative development" itself is problematic, because "iterative" implies rework, which many sponsors find threatening right from the start. The alternative, "incremental development" is much more reassuring, as it implies "adding onto." To keep from constantly fighting standard terms, I have adopted the industry-standard word *iteration* in this book to refer to an internal development period. I still refer to the overall strategy as *incremental development* with *incremental delivery*, to reduce stress in the sponsors' minds.

We discussed what they could have done differently and what they might do differently in the future. Much to our mutual surprise, the team discovered that they never really had had access to a "friendly user" who could take tentative deliveries and examine the growing system, and worse, they still couldn't see that they could get one in the near future.

In other words, looking back at their timeline, they couldn't see how they could have avoided this major surprise.

The story has a happy ending, I'm glad to say. This was a productive group, they still had a month left, so they gutted the "wrong" piece and rewrote it to meet the needs of the users.

This team had listened to their sponsors and users, colocated, integrated frequently, and reflected after each of their iterations, and they hadn't seen this coming. They still had no way to fix it. That responsibility lies with the *Executive Sponsor*, who is the person who has to arrange for more visits by real users. Fortunately, this particular *Executive Sponsor* is aware of the importance of the issue and is actively engaged in improving it for future projects.

Iterations provide feedback to the team about their process, early wins for their morale, and a steering mechanism to see they stay on track. But they do not provide end-to-end feedback about the fitness of the software for business use. That requires involvement of real users.

If I had to choose between two-week iterations without access to real users, or giving up the short iterations and delivering only quarterly but having real users come into the development lab twice in the delivery period to review growing software, I would choose the latter. This is the reason for the work product *Viewing Schedule*, and the reason that I stress *delivery cycle* over *iteration cycle*.

"Two developers are separated by a hallway and a locked door."

"Oh, come on, Alistair, how much does it really matter? We can't get our people into the same hall space. It just happens that we have a combination lock on that hallway door."

It matters, if you want the safety offered by Crystal Clear. Crystal Clear is predicated on people exchanging small pieces of information at a high frequency over the course of a day, without losing their ability to complete the task at hand. That speed of communication creates a safety net for the project.

Suppose a person can stand up, look over a low partition, ask and within a few seconds get the answer to: "Did you really mean that the users will never have to retype their password? We have three servers in use, each with their own password system." Her total time spent away from the task at hand is perhaps 45 seconds. Her ability to continue with the task at hand is excellent.

Contrast that with the same person thinking, "I don't know if I want to stop this just yet to walk across the hall, punch in the combination. Maybe Pat isn't at her desk right now. I'll see her sometime today." The result is either no answer at all, or several minutes, involving a number of very different sub-tasks and a high chance of getting distracted, spent away from the task at hand at least. Her ability to continue with the task at hand is moderate at best.

When negotiating the hallway, steps, locks, a person has to deal with sub-tasks that intrude on her cognitive state of mind. This breaks her line of thinking for a period and makes getting back into the problem-solving state energy consuming. Dealing with these cognitive shifts slows information flow between individuals several orders of magnitude, blocking the exchanges that make a team productive and successful.

It is true that many projects have shipped software despite developers sitting across hallways. However, team leaders repeatedly tell me, "Give me 3-5 developers in one room, keep the distractions away, and I can get the software out. Don't spread us out." That means they visit each other, draw on each others' whiteboards, look over each others' shoulders at their programs and tests.

"We have this big infrastructure to deliver first."

Sometimes it does take a long time to get the infrastructure set up to deliver just the first, smallest piece of functionality. That is why the allowable increment period is as large as 3 months. But larger than that, and project leaders tell me they can't keep focused on the design at hand, that the possibility for error in the design is just too large. More and more, teams are wanting to deliver new functionality in 3-6 week time-frames, saying that they don't want to wait longer than that to get feedback. They implement a skinny version of the infrastructure first, and then evolve the infrastructure in parallel with the functionality (the *Walking Skeleton* and *Incremental Rearchitecture* strategies).

Crystal Clear is based on getting feedback from running code and active users. The feedback is not just on the code, but on the requirements and the development process, on the ability of your team to deliver systems.

Rapid feedback on the code lets the team further reduce writing down requirements and designs in excruciating detail. If you lengthen the time before you have running, tested, examined code, and you have to put more into writing just to keep the memory of it alive. Rapid feedback on the process allows the team to locate unexpected problems and set up corrective action. It also boosts morale and establishes the habit of delivering that is one of the critical success factors of working teams.

Short iterations and delivery cycles are beneficial for four reasons:

- to avoid having to write down so many design details,
- to get real feedback on the design decisions,
- to find and fix problems in the development process, and

-
- to establish a habit of delivering.

"Our first delivery is a demo of the data tables."

No good. Each delivery cycle should produce running, tested code that is, at least in principle, of some use to some user.

Moving to agile development requires learning how to divide the development assignment into chunks that can be integrated end to end, exercising the full development process and as much of the architecture as is needed for the growing functionality.

It is a comforting trap to develop the user interface design, or the tables, or a small prototype, and call that "a cycle" (of some sort) after which a *Reflection Workshop* is held. I have heard of teams that called writing all the requirements "an iteration," and then reflected on their requirements-gathering process. This does not work for the dual reasons that they cannot know how well they gathered and documented the requirements until they see how it all meshed with the rest of their development process (i.e., after delivery).

"No user is available, but we have a Test Engineer joining us next week."

A test engineer is not a user. Nor is a manager or supervisor who "once worked in the field." Your expert user validates and breaks your UI designs. A test engineer can double-check your own tests to tell you if your code is broken, but cannot tell you whether the system is what the users want. In a published study of project managers, each of whom had experience with and without direct links to the users, the project managers overwhelmingly felt that the projects with direct user linkage were more successful than those without (Keil 95). When they had users present, the users alerted them, early on, to mistakes they were making in the concept and layout of the software, allowed them to produce software better suited to the needs of the users, with fewer unneeded features.

A Crystal Clear project can get by with relatively informal intermediate work products because of the ability of the designs to simply ask a user, at intervals no longer than a week apart. The best projects have a user available full time, and some can get a user to visit several mornings a week. At the very least, an hour each week should be scheduled for user interview or shared design time.

Those were the obvious violations. Here are a set of questions and situations that either violate the intention of Crystal Clear or question the boundary.

"One developer refuses discuss his design or show his code to the

rest."

As expert developer Pete McBreen said, "The days of claiming that code is private are long gone." Willingness to discuss the design and find possible faults in his/her own design or code is important.

People being willing to discuss their code is part of the **Personal Safety** property. Programmers often disagree about what constitutes good design, and often are quite nasty about it. Providing people with the needed **Personal Safety** may require that a few people step forward and show their designs so that the team can learn how to discuss the design technically and not emotionally.

A project is unlikely to pull off a Crystal Clear process if they are afraid to trust their designs to their teammates. Ask yourself how you and they can manage a design discussion without insult, letting different people work at their own levels of competence, still sufficient to deliver a working system.

"The users want all of the function delivered to their desks at one time . . ."

"... we don't want to annoy them by having to install new versions so often."

It happens with certain sorts of systems that the users can't be disturbed with many deliveries of changing system, and especially changing user interface. However, you should be able to bring a user in to see and use the system to get feedback on your technical and usage design decisions. Once that portion of the system is designed, tested, viewed and accepted, you can relax on those decisions and move on to the next.

In the situation that it is not appropriate to actually install the software on the users' machines, set the constructed, tested functionality to the side (properly versioned, of course), and wait to add the next increment's functionality. In other words, you will develop exactly as though you had delivered the functions to the users, even though, in this case, you didn't. The key to the Crystal Clear process is that you get closure and feedback on your development and deployment process as well as your software.

Crystal Clear can be used when you cannot keep delivering system updates to all the users. Modify the process by finding a "friendly user," and deploy the different versions to just that one person.

"We have some milestones less than a use case and some bigger."

"... Our milestones are like these: implement new communication commands, implement HTML tags, implement class Blob. Some of them are inside a use case, some of them span use cases. Will those do?"

Implementing class Blob is not an "interesting" milestone in the Crystal Clear vocabulary. Getting a communication command implemented and tested is. Implementing a pair of HTML tags implemented, tested and integrated is.

There is no obvious violation here. Keep asking, "What counts as a usable piece of functionality."

"We wrote down a basic concept and design of the system. We all sit together, so that should be good enough."

This is a reference to "You have a project mission statement, you are using usage-based requirements, probably use cases, and you have a description of the system design using one of the many description techniques you can invent," questioning just how many requirements are needed, and how much design documentation is needed.

I find use cases useful at the start of projects (less critical as the system fills out), and so of course I recommend them. However, a number of teams like feature lists or requirements formats that are neither use cases nor feature lists. Some live almost entirely by verbal agreement. If you all sit together, then it may be enough that you have some combination of an actor-goal list, and/or user stories or use case briefs or feature lists as your requirements "table of contents," and this is sufficient to hold your conversations in place. There is no violation of Crystal Clear here, as long as the *Sponsor* and *Ambassador User* are part of the agreement to work this way.

The same rule of thumb applies for design documentation.

Watch out, though, for one person becoming overly loaded as "the expert" on the requirements or domain or master design, and slowing down the team.

"Who owns the code?"

I used to have a rule in Crystal Clear requiring that "every function, class and work product has a clear owner." During the revisions of this book, that rule dropped low enough in priority not to make the final cut. However, code ownership is still a hot and even dangerous topic on many projects.

On many projects, everyone is allowed to add code to any class, with the consequence that no one feels comfortable deleting someone else's code from the increasingly messy class. The result is something a lot like a refrigerator shared by several roommates: full of increasingly smelly things that *almost* everyone knows should be thrown out, but nobody actually throws out.

I have seen a group of three people sitting around the same workstation for days, working their way through every class, negotiating which lines of code to delete or move. This situation comes from absence of an ownership model, and is expensive.

You have a clear ownership model for your work products if you know who it is that can change, update and, most importantly, delete any part of it. The answer may be a single person, anyone in a given sub-team, or even anyone on the team. You should never have to have the whole team sit around the screen and ask, "Who put this in here? What is it doing there? Can we delete it?" If you have to do this, you know you have a breakdown in the ownership model.

One person wrote me: "Good point. We just found ourselves all sitting around the screen last week, reviewing all the classes we have so far. It seems we had all been adding, and no one removing. It took a full day just to sort that out. Now we are organized with use case ownership and class ownership. Anyone can make a short-term change to a class when we hit panic mode, but that person notifies the class owner, who can double check it."

People often think that XP has no ownership model. This is untrue. XP has a strong ownership model: *Any two people sitting together and agreeing on it can change any line of code in the system*. The ownership model is explicitly communal, with the safety check that two people have to agree to the change (and also that all the tests have to run!).

Most of the Crystal Clear projects I have visited adopt the policy, "change it but let me know."

"Can we let our Test Engineer write our tests? How do we regression test the GUI?"

Regression unit tests are for the *developers* to write and use, not the Test Engineer. They are there so the developers can go home at night knowing that the changes they put in during the day did not unexpectedly wreck the system, make changes with greater rapidity, find mistakes faster, and sleep better at night.

Programmers traditionally dislike writing tests, and have a tendency to pass their code along to someone else called Tester to test as soon as the code "sort of runs." While this is not a strict violation of Crystal Clear, it is a violation of its intention and a bad idea, socially and professionally. Write your own tests.

This raises the question of testing again. Review Property 7, which discusses GUI-less system tests, GUI-driven system tests, and usability tests.

"What is the optimal iteration length?"

I don't know that there is an "optimal" iteration length in general. I have heard adequate defense of everything from one week to one month. It is useful to bear in mind the different negative effects come from them being too-long, too-short, and all the same.

Too long. Most development organizations use iteration lengths of anywhere from three months to two years. This produces the rhythm so neatly described on the Scrum eGroup by Daniel Gackle (who was writing about even a monthly iteration!):

"Meander, meander, meander, realize there isn't much time left, freak out, get intense, OK we're done. Repeat monthly."

This "meander, meander, freak out, get intense" syndrome is one that incremental and iterative development is supposed to remove. The question remaining is, how long is too long for you?

Two good initial choices are two weeks and one month. Two-week iterations could be good, because this cycle length will illustrate to the programmers how much they need to change. It will teach them to estimate in small quantities and develop in micro-increments. It may, however, be too difficult initially. A cycle length of one-month is less of a shock to the system and may allow the team to ease into the new approach. You may find, however, that one-month iterations are not efficient enough for your purposes.

You will, of course, be holding *Reflection Workshops* after each iteration, so you can question the iteration length within a few iterations. Some groups change from two weeks to a month, others from a month to two weeks. I have even met teams that change the iteration length depending on where they are within the project ! (For those interested⁴⁷: they use one-week iterations at the start of the project to avoid the "meander" syndrome described above, move to two-week iterations as they hit their stride, and go back to one-week iterations as they near the end, in order to fine-tune the scope-cutting and deal with late-breaking requirements changes and integration errors.)

Too short. Iteration start-up and shut-down is not free, so if the iterations are too short, the team will get frustrated at restarting so often. Even people who like short iterations sometimes comment on the nuisance of having to do planning and estimation every Monday morning. In addition, if you don't have good automated integration and acceptance tests, the testing burden will be too high.

Also on an eGroup discussion, Patrick Parato noted about one-week iterations:

The only drawback we have seen is that the development team fizzles out close to the end of the iteration. The tasks will be all complete but people don't want to admit it because the one week iterations make them feel constantly under pressure. The developers don't really like the feeling that they can't vary their effort depending on mood, health, or other external stimuli. Some developers believe that one month long iteration allows them more freedom to vary their pace, and that the one week iteration is too much pressure. Very similar to micro management.

⁴⁷ Thanks to Jeff Patton for this note.

Monotony. Patrick's note introduces the topic of monotony. As mentioned earlier, human beings seem built for rhythm, and the need for this shows up in incremental / iterative development. An increasing number of people are reporting that when they do short iterations for a year or two, a feeling of burnout sets in, just from the flatness of their work life. This occurs even when and even though they are delivering good code to their customers regularly.

You can take two steps to prevent this monotony. The first is to attend to your iteration and delivery completion rituals (see the *Process* chapter). At the end of your iteration, decompress or celebrate, whether that means a walk in the woods or a group trip to the pub. Arrange to do something different after delivering new code. Some teams use the period right after delivering to clean up the mess they created in the code just before delivery (Hohmann ref???), others allocate time for some personal development. People use the *Reflection Workshop* and iteration planning to create a change of pace.

The second step applies if you are using shorter iteration lengths, such as one week. Bundle these into some form of super-cycle, perhaps every four or six iterations to a super-cycle. Arrange a rhythm shift to occur at the boundaries of the super-cycle, a team outing, picnic, treat, trip to the movies, half-day off; or some days for development of personal topics.

Some teams to allocate a quota of time for people to work on mini-projects not directly related to the project. Some put marks in the code or on the whiteboard about ugly places in the code they'd like to have time to clean up. Some allocate time for people to investigate new technologies. Both of these are "investment" activities, which improve the programmers and the system, and also provide a change of pace.

A final idea is to allow programmers to spend time with real users of their system in their work environment. This not only breaks the monotony, developers, as Tom Poppendieck and others have pointed out to me, find it highly motivating.

Chapter 7

Questioned (Frequently Asked)

Readers may be curious about how these ideas arose, compare to others in the industry, how far they can be stretched, and what to do when they don't seem to apply. The chapter is presented in question-and-answer form to allow for "talking about" the ideas, everything from philosophical foundations to "how do I get started?"

Crystal Clear works because software development is an economically constrained cooperative game of invention and communication and the rules of Crystal Clear improve both the invention and cooperation aspects of the team's work.

Paradox though it may seem, it is precisely the lack of requirements and planning documents *together with* having team members in the same room that improve the odds of success. The team members get quick feedback on both the process and the product, and have only a short distance between them and the people who need to know the information. A person who learns something can pass it along to the others with very little communication cost. A person needing to know something can get an answer with similarly little communications cost. By making it easier to coordinate, with less to coordinate, the team reduces both cost and error.

In this chapter, I start right from the very beginning; the research that gave rise to the recommendations, the underlying idea of software development being an economic-cooperative game, and the philosophical heritage of the embodied world-view. These conversations are fairly heavy and not for every reader. As one person put it, Crystal is "short on practices and long on principles."⁴⁸ These questions allow me to talk about the principles. After the principles, I review the boundary conditions of Crystal Clear, as distinct from Crystal Orange, for example, and the lack of required techniques. I discuss the structure of methodologies in general, and how that influences the structure of this book and Crystal Clear in general. I include a summary chart for Crystal Clear.

After the first four, heavy questions, I discuss the way in which this book is laid out, how Crystal compares with Scrum, XP, and with ISO 9001 and CMM(I) requirements, distributed and larger teams. Finally, the last question, "How do I get started?"

⁴⁸ Thanks to Andy Pols of the UK.

Question 1. What is the grounding for Crystal?

The first and best grounding is empirical. The empirical evidence started showing up after I was employed in 1991 by then-fledgling IBM Consulting Group to create a methodology for their forthcoming object technology projects. I didn't know anything special about methodology at the time, so my boss at the time, Kathy Ulisse suggested that I visit and debrief project teams to find out what they had learned (thank you, Kathy!). From then to now, I visited projects both inside IBM and outside IBM, in all parts of the world. I continue these project debriefings even now, because what people report is both informative and surprising.

What I learned was that most of what had been written in the methodology books was quite irrelevant to life on the project and had very little to do with the project's likelihood of success. In fact, until the arrival of XP, the correlation was inverted, that the more occupied the group was with following the strictures of the methodology, the less likely it was that they would deliver the software in a timely fashion.

The projects I interviewed that had succeeded often had a sloppy-looking process. These successful people talked about topics that were not method-related, but communication- and delivery-related: sitting close together, having quick access to expert users, delivering frequently and getting feedback quickly.

Initially, I took down all these notes, and didn't know what to do with them -after all, I had been charged with writing down a "methodology," which was supposed to consist of roles, work products, milestone standards and the like. It was only after several years of interviews, three or four attempts to write down a useful methodology, and working directly on projects that the message started to sink in properly: Those really are the core properties of success; the techniques and work-product details of the "methodology" are really a second order success factor. Get the people close together, communicating with good will, delivering often and getting quick feedback from the users, and the team will probably sort out the rest on their own, using whatever technology and techniques they know.

Looking back over my notes from ten years of interviewing, these same recommendations surfaced over and over again. I slowly recognized that they form a methodology, one that contains few specific rules, but allows the people to act as educated professionals in their fields and work out the best path to success on their own. The problem that has occupied me for the years since that realization has been just how to write it down in a way that another team could follow, how to identify which pieces are more critical than others, and how to describe the allowable variances. My best current guess is this book.

The second grounding consists of a set of principles distilled from watching software projects in social-anthropologist fashion, from reading literature on human communication and process design. I described these principles at length in *Agile*

Software Development (Cockburn 2002??), in a pair of articles called "Learning from Agile Software Development" (Cockburn 2002a,b??) and in my doctoral dissertation, "People and Methodologies in Software Development" (Cockburn 2003a??). I list and summarize the principles here:

1. **Different projects need different methodology trade-offs.** This should be completely obvious. Judging from the ongoing initiatives at large companies around the world to create and standardize a single methodology for all their projects, however, it is evidently not obvious. In "Selecting a Project's Methodology" (Cockburn 1999), I proposed two axes for selection: the number of people needing to be coordinated, and the degree of damage that could be caused by the system malfunctioning. These two axes explain why Crystal is not a single methodology but a family of methodologies, and serve as a basis for selecting a particular Crystal methodology to start from. Of the two axes, I consider the number of people being coordinated as the most significant to start with; the team can often shape the methodology for the other axis.
2. **Larger teams need more communication elements.** Team size is the critical distinguishing element between members of the Crystal family. Crystal Clear, for example, is only recommended for teams that can achieve **Osmotic Communication**. Once the team grows to more than a dozen people, or sitting more than 30 seconds walk from each other, different communication and coordination modes are needed.
3. **Projects dealing with greater potential damage need more validation elements.** A small team of six or fewer people programming the movement of boron rods in a nuclear reaction should, I hope, use more care in their work than the same six people programming a free-time system, perhaps to organize their recipe collection, keep track of the neighborhood's football team scores, or order food for late night work. The difference is in the verification and validation dimension, not in the coordination and communication dimension. For the nuclear reactor, there should be public reviews and walk-throughs of the requirements, the design, the code, and the tests . . . it should be manifestly and publicly clear that the system works. That amount of verification is unlikely to be worth the time, energy and monetary cost on the free-time projects. Crystal Clear does not natively address the life-critical level of damage. A team working on such a project can only use Crystal Clear as a starting point, and must add additional verification elements to fit their situation.
4. **A little methodology does a lot of good; after that, weight is costly.** Not much work has been done on the diminishing returns aspect of methodologies. The surprising news is that a little goes a long way, and that diminishing returns set in very quickly. This is surprising because there seems to be a reflex response that "more methodology is better," meaning that more process, more documents, more

status reporting will improve the likelihood of timely delivery. My interviews indicate the reverse, that less is generally better, as long as one covers the gaps with personal communication. Jim Highsmith gives the advice to start with less than you think you need ... that's probably all you need; it is easier to add some as you go along than to remove as you go along. That advice is incorporated into Crystal Clear.

5. **Formality, process, and documentation are not substitutes for discipline, skill, and understanding.** This principle from Jim Highsmith (2003) articulates what is different between agile and traditionally predictive or plan-driven projects. Software is built by people who need to invent and communicate their ideas, based on their skill and understanding. Large corporations often make the mistake of thinking that having people fill out forms, follow particular steps in development and document the current state of their code, will confer upon them skill in inventing solutions. However, the three elements that Jim highlights, discipline, skill and understanding, are all *internal* properties of a person and cannot be substituted for by the external forms.
6. **Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.** Two or three people standing at a whiteboard, diagramming and talking, are able to take advantage of a wealth of communication channel and get near-instantaneous feedback. They can see each others' expressions and change sentences in mid-utterances. They can move closer and farther, point, gesture, make facial movements, draw and talk all at the same time, interrupting each other and adding to the drawing as they go. This sort of exchange situation is known as *warm*, meaning that it is rich in information and also in emotion. As the people move to telephone, email, and eventually paper, the richness, speed of interaction and emotional content diminish, and the communication is known as *cooler*. Although there are certain advantages to using cooler communications channels (asynchrony and isolating social-emotional reactions are two of them), there is a tremendous loss in communication efficiency, which on a software project translates to lost time and increased mistakes. Since most projects are time- and cost sensitive, Crystal Clear maximizes rate of progress for time and money expended by keeping the communications close and rich (**Osmotic Communication**)
7. **Increasing feedback and communication reduces the need for intermediate deliverables.** Many methodologies are full of what I call "promissory notes." These are documents that promise that the team *will* build such-and-so, which *will* be built as such-and-such a time and *will* look like this or that. These promises are primary needed because there is such a long time delay between when the promise is made and when the software is delivered. With such a long time delay, it is of course natural that the sponsors, examiners and users want to know that progress

is being made and what the final system will look like. The problem with these promissory notes is that very often the neat plans and designs don't work out as promised. If, in contrast, the team builds something and shows it to the users every month, the time delay is short enough that they don't have to make elaborate promises - they simply show the month's results and learn directly and correctly what is right and what is wrong. Software development is still slow enough and expensive enough that some promissory notes are needed - the delivery plan and use cases being two - but the principle is still that the shorter the time delay, the fewer intermediate, "promissory" notes are needed.

8. **Concurrent and serial development exchange development cost for speed and flexibility.** This principle says that properly executed concurrent development can speed development, at a possibly higher salary cost, compared to correctly executed serial development, in which each task is run to completion before the next task is started⁴⁹. The problem with the lower-cost serial development is that any mistake in understanding causes a ripple-effect of rework, which is expensive. It is predicated upon having very stable requirements and a full and stable understanding of the domain and the implementation technology. It is very rare in software projects that the requirements and the technology are well enough understood to permit these efficiencies to be harvested. The Crystal family is therefore built upon concurrent development, which permits real-time discovery of mistaken assumptions, but at the same time requires better communications between the people (which gets back to **Close Communication** and **Osmotic Communication**).
9. **Efficiency is expendable in non-bottleneck activities.** The book, *The Goal*, by Elihu Goldratt, identifies that every process has a "bottleneck" station, one that constrains the speed of the entire enterprise. Software development is no different. What the book does not draw upon, but the Crystal family does, is the corollary, that the non-bottleneck stations can help in certain ways, operating at lower efficiencies. Efficiency becomes an expendable commodity. In software development this manifests itself when people with spare capacity help to write requirements, tests or documentation; or, when the developers (assuming they have spare capacity) start work before the requirements are finalized, taking upon themselves the likelihood of later rework. Examples of these situations are described in *Agile Software Development*. Identifying the bottleneck station and making use of spare capacity should become a core part of the team's methodology shaping workshops.

⁴⁹ The third chapter of *Simultaneous Management* (Laufer 1998) describes how to mesh concurrent and serial development on the same project to take advantage of each. As interesting as the techniques he described is the fact that his stories come from civil engineering projects (e.g., building a hospital, or an airstrip in the jungle).

10. **"Sweet spots" speed development.** The best of all possible worlds is having (1) dedicated, (2) experienced people who (3) sit within earshot of each other, (4) use automated regression tests, (5) have easy access to the users; and (6) deliver running, tested systems to those users every month or two. Such a project is clearly in a better position to complete successfully than one missing those characteristics (it is surprising that sponsoring executives do not pay more attention to these important success factors).

Crystal Clear is built upon and legislates the last four of those sweet spots. Obviously, it is a rare team that can be staffed with experienced people, and so Crystal Clear works with the assumption that the project is staffed with a mixture of experienced and novice developers (although it does require that the *Lead Designer* is experienced). When the team cannot hit one of those sweet spots, then they need to invent a way to get closer to it. The **Focus** property is about allowing to team to get close to the first sweet even when they are not fully dedicated to just one project. The question of what to do when some of the other sweet spots are missed is taken up in later questions.

Since it is hard to remember ten principles, I derive them from one idea, that of the economic-cooperative game (described at length on my web site⁵⁰, in *Agile Software Development*, and in "The End Of Software Engineering And The Start Of Economic-Cooperative Gaming" (Cockburn 2004a).

The cooperative game manifesto says:

"Software development is a series of resource-limited, goal-directed cooperative games of invention and communication. The primary goal of each game is the production and deployment of a software system; the residue of the game is a set of markers to assist the players of the next game. People use markers and props to remind, inspire and inform each other in getting to the next move in the game. The next game is an alteration of the system or the creation of a neighboring system. Each game therefore has as a secondary goal to create an advantageous position for the next game. Since each game is resource-limited, the primary and secondary goals compete for resources."

The advantage of this phrasing is that it highlights a couple of critically important aspects of software development:

- It is the *people* and their *invention* and *communication* that make software happen. Everything that relates to their speed of invention and communication affects the outcome of the project. This specifically includes proximity, community, amicability, conflict, and trust, as have been discussed at length in this book.

⁵⁰ <http://alistair.cockburn.us>

- There are two goals in motion at every instant: delivering the present system and setting up for the following game. They conflict, requiring some non-trivial combination of short-term "greedy" and long-term "investment" actions.
- Every decision, by every person involved, has economic consequences. Since the situation is over-constrained and under-resourced, this typically means you will hurt no matter what you do; you only get choose in which way to get hurt. If you spend too much time in investment actions, you get hurt in the short term (and maybe the long term); if you do too many greedy actions, you get hurt in the long term; and there is probably no middle safe ground. Don't fall asleep.

I generate most of my methodology recommendations and project management strategies from the cooperative game manifesto. Considering the state of the project, the two goals, and the nature of human-to-human communication, I am able to choose where to abbreviate, where to spend extra effort, when to put people closer together, and when to put them further apart. (Yes, there are indeed times to separate them and *reduce* communication! See "The Cone of Silence" (Cockburn 2004b).)

* * *

The Swedish researcher Pelle Ehn and Danish developer Peter Naur provide additional grounding to these ideas based on the philosophers Descartes, Marx, Heidegger, Wittgenstein and Ryle. Ehn examines the first four of these philosophers in his excellent but sadly out of print *Work-Oriented Construction of Computer Artifacts*⁵¹. I summarize his extensive discussion in a brief and necessarily coarse fashion as follows:

- Descartes gives us a world-view that there is an objective reality that can be described. This world-view underlies and informs mainstream traditional software development: The assignment of the development team is to capture that reality in their requirements, in their code, and then again in their analysis models, designs and documentation.
- Wittgenstein's gives us an opposing world-view, that there is no objective reality that we can articulate and agree upon. Rather, we are all engaged in ongoing "language games" that evolve over time and that externalize a meager part of our understanding at any moment in time. The development team never "captures" the requirements, because neither the requirements nor the people's understandings are either expressible or shared. What happens instead is a sort of dance in which the various people perform actions and make utterances, and react to the actions and utterances of everyone else and also themselves. Although this description may seem untidy and vague, it matches my particular understanding of what

⁵¹ An extract from Ehn's writing is included in *Appendix B* of *Agile Software Development*.

-
- happens on projects, and forms the deepest underlying basis for the Crystal family and Crystal Clear most particularly.
- Marx gives us the world-view that every new software system will change the social-power structure. Accordingly, the users, the executive and the developers should include in their development process an inquiry into how the system will shift power, and how to handle that shift. This world-view is taken into account to a small extent by the agile methodologies, who give a larger voice to the users. It was taken deliberately into account by Kristin Nygaard, Pelle Ehn and the Scandinavian school of development in the 1970s and 1980s, when they were working with the local trade unions on where and how to incorporate computer technology into the unionized workplace.
 - Heidegger discussed the nature of tools, as being either visible to the wielder as a tool object, or invisible to the wielder, simply an extension to his body that accomplishes the needed task. While the tool is visible to the wielder ("Gee, this hammer is heavy," or, "Where's that hyphenation key, again?"), the tool interferes with the performance of the task. When the wielder is performing the task at full speed, he does not notice its existence at all. This discussion is important to the designers of the user interface and also of the domain model, both of which can interfere with the system's usability.
 - Ryle's views are discussed by Peter Naur in "Programming As Theory Building" (Naur 1990). Naur incorporates Ryle's idea of "theory" to describe the way in which a designer-programmer has to understand the domain at hand, the requirements, the technology and the proposed design solution. For a designer to make sympathetic changes requires him to have an understanding rich enough to be mined for implications and causes. For a group of people to make harmonious design decisions, they have to have in their separate heads, theories that are "close" to each other. To me, this explains very neatly the metaphor element of XP: the metaphor is a short phrase that points to the theory of the system's design; the better the team's agreement on the metaphor, the more likely they are to make harmonious design decisions (see *Agile Software Development*, pp. xxxx).

Naur's idea of each person having his own theory of the system, and the difficulty of aligning the multiple theories across the development team fit naturally with Ehn and Wittgenstein's notion of language games and the dance of actions and utterances that move people to greater alignment of language and understanding.

Ehn discusses the idea of people laying down markers of their current understandings, so they can refer back to them later. Naur's writing leads one to recognize that the subtlety of people's understandings change over time, and that the current design, beautiful or ungainly, really is a mirror of their current

understanding⁵². As their theory matures, it becomes more subtle, and they get a more concise and natural design expression.

There is a negative side to having a more subtle theory. The design becomes more subtle over time, which means that the maintainer also needs to develop a similarly subtle theory to take over the code! Really experienced developers find this frustrating, because it means that their best, cleanest, most concise code is virtually unmaintainable! They watch in horror as the people who follow them unpack the design to become bigger, clumsier, and *more evident to the casual observer!*

This observation brings into question the very idea of quality in design. One of the very many dimensions of quality is maintainability and evolvability. To the more experienced developer, the subtle design is "better." He can, with his subtle understanding, adjust the system to new circumstances with just the smallest of changes. That same code is nearly unmaintainable to the person with a less developed theory. With the less-subtle design, the second person will take longer and have to make more extensive changes, and at the same time find that less subtle design "easier" to maintain and extend. I see no way out of this dilemma; it is just another of those contradictory situations that infest software development.

Experiential, cognitive and philosophical groundings suggest that the best way to develop software is to use so few people that everything can be communicated in face to face discussions, letting small groups finesse problems by relying on common sense and skill; hold the behaviors and artifacts together by adopting only a few principles of development, such as direct user involvement and tracking by finished products. These are the foundations of Crystal Clear.

Readers who care about such things should examine Ehn and Naur's writings to better understand the philosophical and cognitive underpinnings of the Crystal approach.

* * *

There is a mostly unstated bias in Crystal, that of generally trusting people. That bias comes from me. Psychological baggage inevitably seeps from a methodology author into his or her recommendations, and this one is mine.

It is not a blind trust, though. Although teams perform better and use less energy when they can work in trust, not all people are able to trust others, nor do all people deserve trust. Some people, playing their own game of career survival, are quite willing to take advantage of their teammates or sub-optimize the project results if it will enhance their careers. Others are not capable of fulfilling their assignments. Still others simply don't care. In general, every person is watching all the others to see

⁵² Ward Cunningham said almost identical words to me back in 1994. I might not have noticed it in Ehn and Naur's writings if Ward had not already mentioned it.

when the trust will fail. This is one reason that the targeted property of Crystal Clear is the weaker **Personal Safety**, and not full **Trust**.

The manager of a "cooperative gaming" team must notice and attend to failures in trust. One dilemma that periodically shows up is this: The *Executive Sponsor* asks the programmers how long a set of tasks will take. They answer, the *Executive Sponsor* accepts, and the programmers start work. As time passes, the *Executive Sponsor* gets this uncomfortable feeling that the programmers are taking advantage of her. Does the assignment really take that long (the programmers are working diligently), or are they taking advantage of the situation and padding their time estimates so they can relax along at an arbitrarily slow pace?

There is no way to resolve the question, either in Crystal or in any other methodology. Either could be true and I have encountered what I believe are examples of both.

Software development is a game of people working with and against other people. All human motives and emotions apply and must taken into account, even in Crystal.

Question 2. What is the Crystal Family?

Crystal is an approach to software development based on a genetic code. That genetic code allows specific instances of Crystal to be generated for different circumstances; all instances will share a family resemblance.

The genetic code consists of

- the economic-cooperative game model just described,
- a set of *priorities* and *principles* for making choices,
- selected *properties* to steer toward,
- sample *strategies* and *techniques*,
- sample *instances* to copy.

The three priorities are: project *safety*, development *efficiency*, *habitability* of the resulting conventions.

The *safety* priority is to get a reasonable business result from the project, given the project's priorities and resource constraints. *Efficiency* in development is a top priority because so many projects are economically overconstrained.

Habitability became a top priority when I started noticing in my project interviews that most programmers have the power to ignore whatever methodology is mandated by their organization. All they have to do is say that it slows them down and they'll miss their deadline if they do all that extra work. That is usually sufficient basis for ignoring it. To be fair, most methodologies are so inefficient that they are correct in their statement, so it is not clear whether they are doing the organization a disservice or a favor by ignoring the methodology (my estimate is that there is an immediate 15%-30% efficiency to be gained in most organizations simply by throwing out most of their process⁵³). If the programmers are told they have to do it anyway, they usually find ways to circumvent it. It doesn't matter what the methodology is if it doesn't get used. In other words, *habitability* is a critical success factor of methodology adoption.

Part of habitability is tolerance for human variation. Some like to make lists, others don't; some work best visually, others with text; some like to work alone, others in groups. Some stare at the design for hours, days or weeks making sure it is right, while others like to "feel" the code taking shape. It is for this reason that Crystal has so much tolerance around techniques – not only do they change all the time, but different people are drawn to different techniques. The *habitability* priority means that the rules need be such that the people on the team can live with them.

The priorities interact: *Habitability* means that I accept that a chosen set of rules may not be as *efficient* as possible. The *safety* priority means that the chosen rules should be

⁵³ As Luke Hohmann points out: "When you want your boat to go fast, it is easier to cut anchors than add horsepower."

good enough to get an adequate result (making exception, of course, for poor project managers and greedy sponsors, both of which still exist).

Part of efficiency and habitability is dealing with the fact that every project is slightly different, and so properly needs its own, tailored methodology.

This means that no one methodology will suffice for the range of projects that even a mid-sized company will encounter.

A company need not define dozens of methodologies, though. One company with widely diverse projects was able to get away with just three base methodologies: one for under-four-person projects, another for projects of eight to twenty people, and a third one for systems that have to pass the U.S. Food and Drug Administration process certification. Each team tailors the nearest base methodology to their project's particular requirements, expertise and technology characteristics.

Of course, they have to do this tuning quickly enough to fit within the business' cost-value horizon. If methodology shaping takes half the project budget, then it would indeed be better to use a suboptimal but standard methodology. The need for efficient methodology shaping is why I describe the *Methodology Shaping and Reflection Workshop* techniques in this book.

Those are the priorities. The properties are just those described in chapter 2. No strategies or techniques are required for a team, other than incremental development. The set of allowed strategies and techniques is very large, and so I present just a few of the more interesting and less known ones in chapter 3. Methodology samples are included in *Surviving OO Projects* (Cockburn 1998), *Agile Software Development* (Cockburn 2002), and of course, this book.

To help with choosing a set of rules for a particular project, I constructed the two-dimensional grid shown in Figure 7-1.

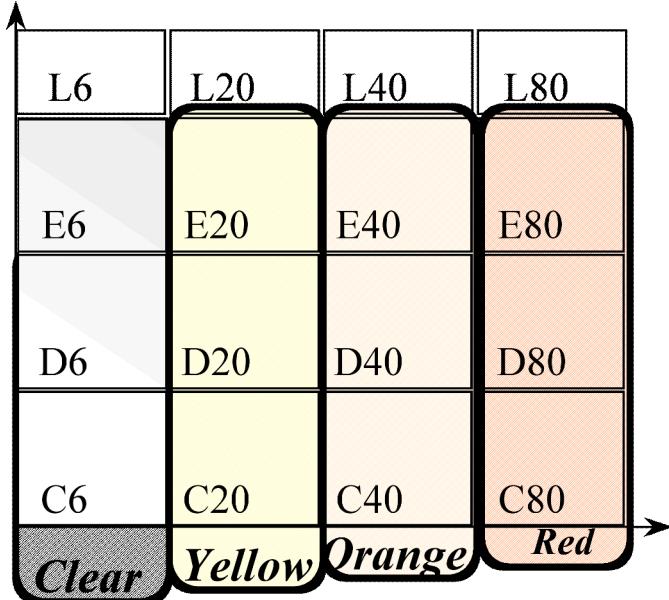


Figure 7-1. Crystal's coverage of different project types.

This grid⁵⁴ identifies *team size*, the number of people being coordinated, as the primary axis with respect to defining methodologies. A second important dimension is *criticality*, the potential damage caused by an undetected defect: loss of comfort (C), loss of discretionary moneys (D), loss of essential moneys (E), and loss of life (L). We should expect to see the communication and coordination components of methodologies change noticeably as the team keeps doubling in size. We should expect to see the validation and verification components change noticeably as the criticality increases up to loss of life (where the FDA process approval can become relevant). In Figure 7-1, the box labeled C6 signifies projects with up to six people working on a loss-of-comfort system.

The name *Crystal* derives from these two dimensions, in an analogy with geological crystals, which also are characterized in two dimensions: color and harness. I let the team size (more generally, the communication and coordination complexity) correspond to the darkness of the color: clear, yellow, orange, red, maroon, blue violet and so on, and the criticality correspond to mineral harness: soft as quartz for loss-of-comfort systems, hard as diamond for loss-of-life systems.

Figure 7-1 shows 16 boxes, implying the need for at least 16 methodologies to be defined. My experience is that if a group starts from a previously documented methodology and shapes on the fly with the *Methodology Shaping Workshop* and the *Reflection Workshop*, things can simplified. One suggestion is to put the life-critical

⁵⁴ Described in (Cockburn 1999ieee) and in *Agile Software Development*.

projects into their own area, and cluster the rest by team size. This leaves one primary dimension - color - as the distinguishing characteristic for non-life-critical methodologies. I don't have enough experience yet to know how the life-critical methodologies cluster, but the company mentioned above decided that all their FDA-approval projects with less than 20 people could be clustered together. They didn't have any projects at all with more than 20 people, so they simply had three categories, which we might call *clear*, *yellow*, and *diamond*.

Question 3. What kind of methodology description is this?

The predominant way of conveying a methodology is through an element-by-element description of approximately one dozen different types of elements. Figure 7-2 shows a generic and typical set of methodology element types. A little multiplication shows that even a fairly simple methodology requires the description of over a hundred separate elements, plus how each fits with all the others. Most methodologies contain several hundred elements, and Extreme Programming has about four dozen. You can immediately see why methodology descriptions are so long (and boring).

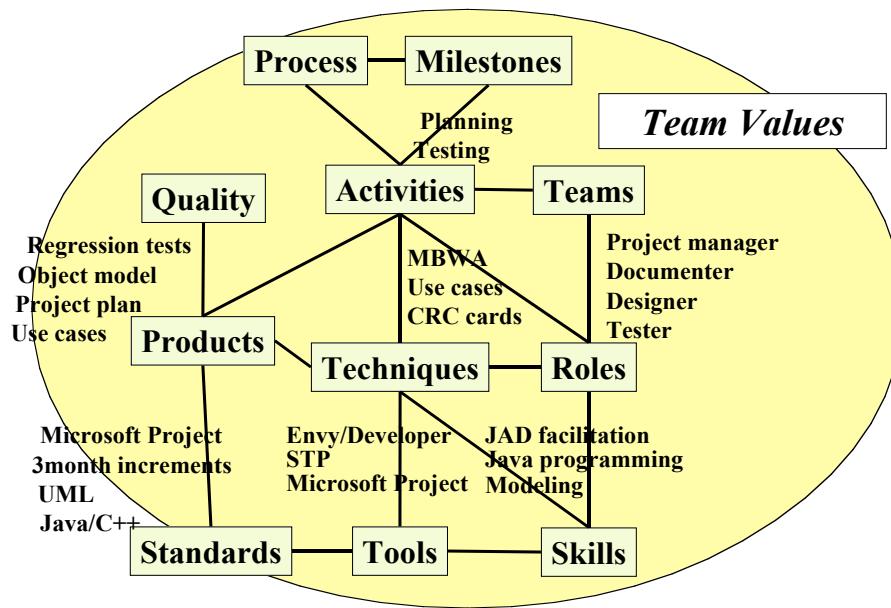


Figure 7-2. Element types of a methodology with sample elements.

Part of the methodology text describes the "factory" where the software is produced (I don't like the historical connotations of that word but can't find another). The non-factory part of the methodology describes how the work products are to be described. Figure 7-3 shows the separation of methodology elements into "factory" and product descriptors (with time flowing downward in the diagram).

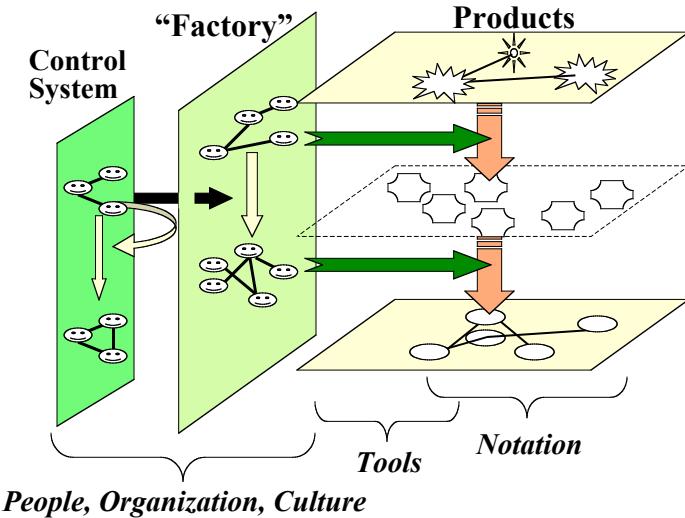


Figure 7-3. Division of methodology into "factory" and product descriptors.

On the right, we see the original, fuzzy and imprecise descriptions of the product (vision & mission statement, conceptual prototypes) at the top, evolving over time into intermediate work products such as requirements and plans, and eventually into source code, tests, packaging, user manuals. The elements of the methodology covering this evolution line in the realm of notations and techniques. Thus we see that UML is a product-description notational standard (which explains both why it is useful, and still affects the outcome of projects so little).

The two vertical slices on the left are the "factory" and its control system, both made of people. The people producing the software live in an organizational structure, which changes over time. The people causing those changes live in their own organizational structure, which is also changing over time. The elements of the methodology covering these concern people, organization and culture. Historically, software development groups have been averse to explicit discussion of their organizational and cultural elements, which has been a continual source of dysfunction in our field.

The horizontal arrows indicate the tools that the people use to create and alter the work products.

I have used these frameworks to describe methodologies for years. Two flaws in them are, however, slowly becoming both apparent and painful.

One flaw is the idealization of people into roles. A methodology description is, by its very nature, a formula intended to cover multiple projects, and the main part that gets abstracted away is *personality*. The problem is that people are simply stuffed full of personality, so that the *people* who show up at work don't have the correct fit with the *roles* that are supposed to be in operation. It is not a "tester" who comes to work in the morning, it is Jim; it is not a "project manager," it is Annika; it is not a "designer," it is

Peter, Annika may or may not have the personality needed for a good project manager, Peter may or may not have the skill set needed to be a good designer (see Figure 7-4).

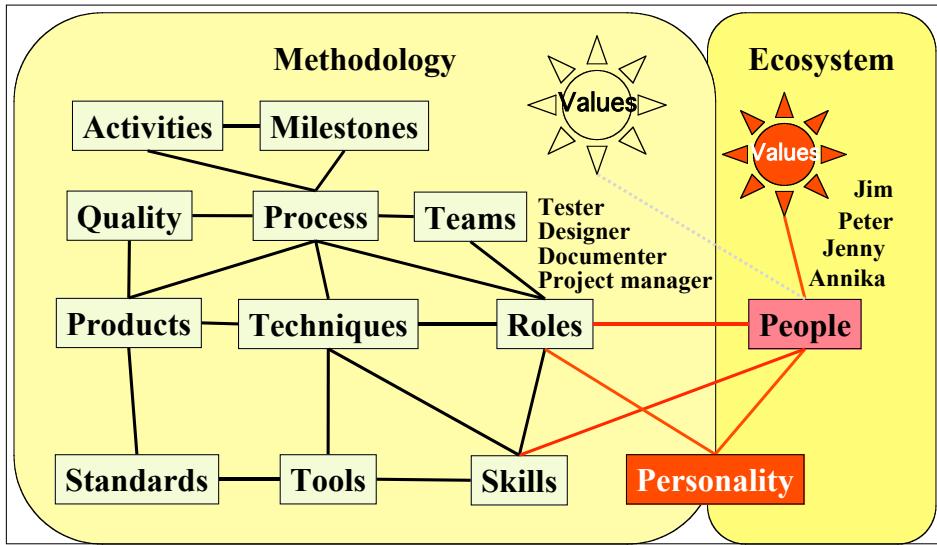


Figure 7-4. Methodologies idealize personality, but people are stuffed full of it.

In practice, individual personalities dominate. Peter may get offended by Annika and refuse to turn in reports; Annika may find Jim hard to talk to and ignore him for weeks at a time; or, as often happens, Peter may simply take over the project, ignore Annika and drive the project in any direction he wants.

The second flaw in the standard presentation of methodology is that there are an arbitrary number of unclassified items that don't fit into the methodology diagram but turn out to be important to the way the project runs. I have encountered as critical "methodological" items: the shapes of the desks, the layout of rooms, the presence or absence of ventilation, a cultural tradition of Thursday afternoon volleyball games or Friday night beer busts, the establishment of a cultural signature such as a logo, saying, ritual or a value such as excellence, working late, or always being laid back, and rules such as "salespeople are not permitted in the programming area after 10:00 a.m." or "no phone calls or conferences between 10:00 and 12:00."

To summarize, I found four knotty issues in methodology descriptions:

- They are long, intricate and boring;
- "Factory" description gets intertwined with product description;
- They miss many small rules and issues that have large effect;
- They miss the effects of individual people with personalities.

I decided to deal with these four issues in the following way:

Different people come to a methodology description from different backgrounds and able to notice different things (similar to the blind men feeling different parts of an elephant and trying to form their impression of the whole). I therefore have written the chapters in different formats to give different views for different reader backgrounds. I do not legislate techniques, but leave those up to the individuals as professionals in their respective fields. The work products are presented via samples, so that I don't get tied into notational standards that are likely to change every few years. As a result Crystal Clear is primarily a description of the "factory": where people sit, how they interact, and so on.

To cover the many small rules that are important to projects, I experimented in describing methodologies using *patterns* instead of the methodology 12-tuple. Much to my delight, I found that I could describe many aspects of a methodology that were important, including the odd little rules, in a way that was easily understood and applied by busy project teams. Those aspects that didn't fit as patterns could simply be expressed as project conventions.

Out of those experiments I learned that *a methodology is nothing more than the conventions the team agrees to adopt!* These conventions drift over time, and so there is nothing more natural than that the team should convene regularly and review their operating conventions. This is of course the *Reflection Workshop*. Also, conventions are no harder to write down than simple sentences, which means they can be written and posted. This is shown in the example of the *Team Conventions* work product.

Studying the patterns literature, I discovered that good patterns fall into two sorts: properties and strategies. Some are ambiguously both, but usually one can decide that any one is more one than the other. Patterns fans might like to reconsider Christopher Alexander's patterns in this light. For example, *Light on Two Sides of the Room* is more a property and *Attic Window* more a strategy (Alexander 1977???). The book *Design Patterns* (Gamma 1995???) contains strategies – it could as well have been called *Design Strategies*. In *Surviving Object-Oriented Projects*, I decided to call the appendix "Risk Reduction Strategies" instead of "Risk Reduction Patterns." On the other hand, when we were writing *Patterns for Effective Use Cases*, we deliberately searched for *properties* of effective use cases and use-case-writing processes, rather than strategies.

When it came time to write this book, I tried describing the methodology with patterns. However, they were an unsettling mix of properties and strategies. This is not at all a problem for any one project team – they only care about what they should do themselves. It is problematic when aiming for a wide set of projects, because the strategies should be the discretionary choices of the team. Therefore, I separated the patterns into two chapters: **Osmotic Communication** and **Frequent Delivery** went into the properties chapter, and *Exploratory 360°*, *Walking Skeleton* and *Incremental Rearchitecture* into the strategies chapter. *Reflection Workshop* got split out as a technique, a means to achieve the property **Reflective Improvement**.

The next issue involves the idiosyncrasies of the specific people who show up in the specific buildings to work with specific technologies on the specific problem at hand. This instance of people, project and environment I refer to as the project "ecosystem" (Cockburn 2002asd). Dealing with the ecosystem comes in two steps. The first is to recognize that the Properties have to be the heart of the matter, and everything else advisory. Thus, the Distilled chapter captures the visible properties (six to eight people in one room or adjacent rooms, etc.) plus the properties the group sets into place (**Frequent Delivery**, etc.). It doesn't mention the strategies, techniques, process cycles or work products. Instead, the group is required to periodically get together and discuss together what their local conventions should be. When someone with a strong personality joins or leaves, the group gets together again to revisit their conventions.

The second part of dealing with the ecosystem is to recognize that a methodology can't solve all of the group's problems. No methodology can correct for the absence of competent and experienced people, or people who dislike each other, or can't make a decision. These are simply exposures, and not just to Crystal Clear, but to every methodology. It is quite possible that a group, otherwise following every recommendation of Crystal Clear, will get together in their *Reflection Workshop*, have an argument and come out not being able to work with each other. That is outside the reach of methodology.

What Crystal Clear can do is to name the rules that give a project team better than even chances of success, and room to maneuver.

Question 4. What is the summary sheet for Crystal Clear?

Project size	Up to 6 or possibly 8 colocated developers. Can be shaped with care to up to 12 people. Not intended for larger teams, as it is missing group coordination.
System's potential for damage	Loss of comfort or loss of discretionary moneys, as in invoicing systems. Can be shaped, with additional testing, verification and validation rules, up to "essential" moneys. Not intended for life-critical systems as it is missing verification of correctness.
Types of systems	Mainframe, client-server, web-based, using any type database, central or distributed. Can be shaped for hard real-time systems with additional rules for planning & verification of system timing issues. Not intended for fail-safe systems, as it is missing hard architectural reviews for fault tolerance and fail-over.
The values	Strong on communications, light on deliverables. Short, rich, informal communications paths, including with the sponsoring and user communities. Frequent deliveries with product and process feedback. Reduced overhead, and fewer intermediate work products. Tolerance for variations in people's working styles.
Tolerances	The policy standards are mandatory unless an equivalent substitution can be made. For example, use of the "Scrum" or Extreme Programming techniques for project scheduling and staging is acceptable; use of the TSP project launch and re-launch techniques is acceptable.
	Every deliverable has a designated owner or owners. "None" and "Everyone" are acceptable, if explicitly stated. In each of these cases there must be additional justification as to how the model works (for example, Extreme Programming declares that everyone owns all code when (and only when) they work in pairs; pair programming creates moment-to-moment dual ownership of code).
Use of Precision	Low precision at first, high precision only in delivered artifacts. Low precision includes designers talking with users and sponsor, two-paragraph use cases or user stories or feature lists; design drawings on the whiteboard, user screens drawn as pencil sketches,

release schedule as short lists and phrases (initially).

High precision includes demo and production screens, final code, test cases, user manuals or help text.

Question 5. Why the different chapter formats?

Methodology books are generally written as a set of rules telling the reader what to do. There are several reasons not to use that format in this book. The first is that Crystal Clear is not so much a set of instructions about how to write software as properties, conditions, conventions and techniques regularly used to bring a certain class of projects home safely (this is the *safety* priority). The intersection of those successful projects is so small as to not be safe itself, and the union of all of them is too fat. I am trying, in this book, to indicate how to find subsets that are both safe and usable.

Readers come from various backgrounds, which makes a difference in what they look for in the book. Our practice of programming has changed drastically since 1987 and the arrival of objects, and again since 1998, when the object and XP communities brought patterns, refactoring, pair programming and test-driven development to the fore. Some programmers have kept up with the most recent changes and need to see proper references to those latest practices. Others still operate using the older practices, and need to be given detail to catch up with the latest vocabulary. Some readers, already experts at running small-team projects, are looking for new ideas or new words for old ideas. Others, new to leading a small team, want detailed instruction.

Here are some of the categories of reader I tried to deal with:

- Those arriving cold to the topic of agile development might benefit from the lighter and very contextual email discussion in the first chapter.
- Those already very familiar with small-team agile development may want to focus only on the safety properties of the second chapter and use their own ways to get to those properties.
- Those getting started with a Crystal Clear way of working will want a starter set of techniques. Some of these techniques will be new even to experienced people, things they can add to their practice set.
- Very few people coming from a traditional software development process history have an operational grasp of the cyclical development processes. Those with a process focus should benefit from seeing the process unrolled in the various ways, and the cycles isolated.
- Methodologists, and owners of development processes in general, operate directly from the list of work products, evaluating a methodology based on the list of who produces what and who checks that. The collection of work samples could still give an experienced agile developer some new ideas.
- Those coming to the book with a varied set of projects on their active list will want to know how to vary the rules and when to shift to a different methodology entirely.

- Those who have read widely will be interested in the economic and philosophical foundations, and discussion of how these ideas fit with related and competing ones.

It is not my ambition that every reader should like every chapter and format. Rather, my hope is that through the use of different perspectives, each reader, coming from his or her particular background, can find some chapter format that conveys what they need in order to understand Crystal Clear's main ideas.

Question 6. Where is Crystal Clear in the pantheon of methodologies?

There are too many methodologies to compare against, so I'll address this question by showing *how* I answer it and giving just two examples.

Each methodology author selects a couple of priorities to be the methodology's focus or center of attention. That focus could be on *defect reduction*, as in Cleanroom (ref ???) and the Dijkstra/Gries' program-derivation method (Gries 1983), *repeatability* as in ISO 9000 (ref ???), *predictability and control* as in the CMM(I) (ref???), or *productivity* as in XP. I have heard people discuss how to stay "laid back" as they expand, indicating that as a priority. Whatever the author prioritizes becomes a filter for selecting and rejecting possible elements of the methodology. Beware methodology authors who assert that their methodology design satisfies all priorities.

When comparing methodologies, investigators usually list the methodologies' techniques and rules in a table, and lets the readers evaluate the list according to their (the readers) personal priorities. This presupposes that the reader already understands the implications of priorities in methodology design. If the reader does understand, then of course he can read the comparison and make a meaningful choice. If not, however, (and most readers don't know about methodology priorities) there is an increased chance that the reader will select a methodology not aligned with the team and organization's actual priority set.

Crystal, as a family of methodologies, focuses on *efficiency* and *habitability* as critical elements of *project safety*. All Crystal methodologies try to accomplish as much as possible with as little as possible. However, that drive for efficiency is subject to the habitability imperative. Teams around the world have different backgrounds and values and are faced with projects of differing characteristics and priorities. The *habitability* priority is to give each team and each team member the maximum free choice in working his or her own way (we want the team not only to succeed, but also be willing to work in a similar way again).

Obviously, the two conflict at some point, and so Crystal allows for teams to choose less-than-optimal ways of working if they so choose. As long as they keep delivering software successfully, I am satisfied⁵⁵.

Crystal Clear is the version of Crystal for small, colocated teams. It has the same priorities and focus of attention, being efficient and tolerant.

As a point of comparison, XP prioritizes for programmer *productivity* and code *evolvability* (note: these are my interpretations). XP reduces intermediate work products (increasing productivity), and raises the discipline required of the people (increasing

⁵⁵ This is, I believe, a fundamental difference between Crystal and XP.

productivity again), but reduces tolerance for individual work styles. The code should be refactored to become easier to evolve, and there are an extensive set of unit tests to help the next person along.

Project sponsors requesting a development team to have ISO-9000 and CMM(I) certification are focusing primarily on *predictability* and *repeatability*. Note that it is not the methodology that gets assessed at the various CMM(I) levels, but the organization itself. To pass certification, the organization needs a documented methodology, and that methodology should be designed in a way to facilitate getting certified.

CMM(I), in particular, carries the concept that variation across projects can be reduced through proper attention to statistical control of the process. For CMM(I) Level-4 certification, the organization must show that it measures the process, and for Level-5 certification that it uses the measurements to optimize the process. (Crystal turns the CMM(I) pyramid upside down in a sense, but that discussion belongs in a later question).

I don't know enough to answer what the focus of attention is for the Rational Unified Process (Kruchten 1999??), DSDM (DSDM Consortium 2002??), or Scrum (Schwaber 2002??).

What priorities drive the selection of methodology elements in your organization? When you understand that, you will be in a much better position to decide how to choose or blend other methodologies.

Crystal Clear and XP

XP is similar to Crystal Clear in many ways, particularly with its attention to colocation, short iterations, frequent delivery, and close contact with the sponsors and end users. It is stricter than Crystal Clear in several ways and looser in a few⁵⁶.

- XP's iteration cycles are required to be shorter: one day to one month. XP doesn't make any rules about how long the delivery cycle must be. This is the reverse of Crystal Clear, which requires deliveries to be no longer than three months at worst, but allows the iteration to be as long as the delivery. I have seen otherwise excellent XP projects get into trouble simply because the time to delivery was six months to a year. This is clearly a violation of the intent of XP, which was *implicitly* predicated upon frequent deliveries. To correct those situations, Crystal Clear is explicit about the need for real deliveries. My sense is that changing from annual to quarterly deliveries calls for the greatest mental shift in the organization, and once the organization gets to quarterly deliveries, it can see the purpose in shortening both the iteration and delivery cycles, and with the reflection workshops, can find a set that works for it.

⁵⁶ These comments are based on the 1998-2004 definition of XP. Kent Beck is busy changing the definition of XP, so the answers in this section may change over time. Use the same line of reasoning as shown to derive the up-to-date answers as XP evolves.

- XP requires programming in pairs, which Crystal Clear doesn't. The difference is fundamental: XP aims to be the most effective methodology possible, and consequently reduces the degrees of freedom of the team in making these sorts of choices. Crystal Clear aims to be a tolerant and *adequately* effective methodology, giving the team as much freedom as possible in constructing the local set of conventions.
- XP requires **Automated Unit Testing** with a rich set of automated unit tests, and integrations multiple times a day. The first is only a recommended property of Crystal Clear, and the integration period is allowed to vary significantly across projects.
- XP requires a customer on site, where Crystal Clear only calls for easy access to expert users, with an indicated minimum of an hour a week rather than full time. This is a situation where more is better; for Crystal Clear I am deliberately looking for the minimum that satisfies the *project safety* priority.
- XP requires the team to decide and adhere to common coding conventions, something not required or even mentioned in Crystal Clear. XP's convention becomes understandable when one considers that the programmers change programming partners frequently, and have to get to an agreement about coding conventions in order to avoid constant fights and confusion.
- XP talks explicitly about simplicity in design and refactoring, as part of the development episode and also in the longer term. These are only recommended in Crystal Clear, as discussed in the strategies *Walking Skeleton* and *Incremental Rearchitecture*. While I am clearly in favor of simple designs with ongoing refactoring, the question is, should it be a required standard in the methodology? My conclusion is that learning to simplify and refactor designs is part of the normal growth path of a professional programmer, and not something for Crystal Clear to attempt to legislate across all project types and programming technologies.
- XP does not require documentation and Crystal Clear does. In terms of the two goals of the cooperative game, XP is loud about its focus on the first goal, delivering the software, and silent on the second goal, setting up for the second goal.

XP does provide half of the needed investment for the future: through pair-programming rotations, a new person joining a seasoned team can come up to speed on the design and code very quickly. However, as people move on to other projects over time, there is a leakage of team understanding, and it is quite likely that the newcomers will not master and internalize the knowledge as fast as it leaks out. This decay is almost inevitable, which means the team should create other, externalized informing markers, to support the influx of new people. Crystal Clear asks the team to deliberately perform both "greedy" and "investment" actions, balancing the longer term communication paths with the short term.

Looking at these, we see that where XP regulates, it regulates stricter rules, and where it is looser, it is because of absence of a rule. This has two consequences:

- You are welcome to adopt any of the Extreme Practices in Crystal Clear. This includes on-site customer, the planning game, programming in pairs, three-week iterations, test-first programming, fully automated unit tests, continuous integration, strict common coding conventions, and simple designs with ongoing refactoring.
- If you want to do full-on XP and still meet Crystal Clear, you must add two things: add a commitment to not just iterate, but also *deliver* code regularly; and add reminding and also informing documentation, as part of the investment activities involved in setting up for the next game.

From a techniques point of view, XP provides a rich library of effective techniques for the professional developer to master.

Crystal Clear and Scrum

Scrum focuses on three key practices as well as the properties of **Close Communication** and **Focus**:

- *Time-boxing*. The subset of the requirements being developed during each one-month iteration is frozen into the "iteration backlog" at the beginning of the iteration. This gives the team the peace of mind (**Focus**) to work on them without fear of them changing. Scrum requires a demo or equivalent after each time-box, not an actual delivery.
- *Dynamic backlog*. To compensate for locking the requirements during the iteration, a list ("product backlog") is kept of everything remaining to do. The sponsors are allowed to change the backlog however they wish, whenever they wish. The development team and the sponsors look at the backlog at the start of a new time-box period and select the subset to be frozen for the time-box. That is, they *reprioritize the backlog* before each iteration so each iteration targets the highest priority work each month⁵⁷.
- *Daily standup*. The team meets briefly (literally standing up) each day to let each person announce what he worked on yesterday, is planning on working on today, and what is holding him back. This binds the team with social and technical interchange and creates an early-warning system for when they get off track.

These combine well with the three key practices of Crystal:

⁵⁷ Jeff Sutherland is evolving "Continuous Scrum." Using automated planning and tracking tools, the contents of the time-box are allowed to vary daily, not just monthly (see <http://wiki.scrums.org/index.cgi?ContinuousScrum>). This requires a different approach to managing user relations and sponsor expectations. I look forward to his writings on this.

-
- *Frequent deliveries*, not merely demo'ing the system, but actually putting it in the hands of users, a "friendly user" at the minimum.
 - *Close communication* for Crystal in general, and *Osmotic Communication* for Crystal Clear, ease of question-and-answer at any time during the day, not just during the daily standup, and overhearing of information in the small-team setting.
 - *Reflection Workshop*, tuning of the rules allows rapid adaptation to local conditions within the project timeframe.

Scrum and Crystal fit neatly together, producing what one wit called the *No-Process process*. The No-Process process says, roughly, start anywhere, work in short cycles with high communication and reflective feedback, and eventually you will end up with what you need (a genetic algorithm, if you like).

- Time-boxing with real deliveries is the motor to the process.
- Periodic reprioritization of the backlog keeps the product on track over the long term.
- Post-iteration reflection workshops keep the team and process on track over the long term.
- Osmotic communication and daily stand-ups keep the team on-track over the short term.

You can probably add the No-Process process to your organization's process, no matter what that is. It would be a really wonderful genetic algorithm, except that your project probably has a 4-, 6-, or 10-month deadline, and you don't have time to start just *anywhere* and evolve to optimal. You need to start *pretty close*.

Crystal helps you start *pretty close* with its properties, principles, techniques and examples.

Crystal Clear and RUP

The Rational Unified Process (RUP) is very similar to the Crystal family of methodologies in structure, but in a very different way than XP is.

Both RUP and Crystal are "methodology generators." Neither is a methodology or process on its own (yes, I know it is called the Rational Unified *Process* but it is not a process, anyway, it is a process generator, or "process framework" in RUP language). Both RUP and Crystal are predicated upon the view that no single process can fit all projects, and therefore, the project team has to customize the methodology to fit their local situation.

Both RUP and Crystal contain advice about how to custom-tailor the methodology to the organization and project; both describe core techniques and contain work product samples and templates. Both recommend incremental growth of the system using iterative development, good testing practices, close contact to the users, risk management and feedback from running code.

RUP's *inception* phase is very similar to the *chartering activity* described in this book. RUP's *development cases* for process customizing are just as core to RUP as the methodology shaping workshop is to Crystal (if you aren't tuning RUP to your company with development cases, then you aren't doing RUP just as if you don't do methodology shaping and reflection workshops then you aren't doing Crystal).

At this level of discussion, there is very little to distinguish the RUP process generator from the Crystal methodology generator.

Three core elements differ. RUP sets *architecture*, *visual modeling*, and *tool usage* as core elements. Crystal considers architecture be a local matter, although I give the same recommendation to build an early, executable architecture. Crystal differs strongly from RUP on the subject of visual modeling, using first the idea that documentation is to be locally decided matter, and second the idea that documentation should attempt to convey the "theory" of the system (see Question 1 in *Questioned*), and visual models are only a small part of conveying that theory. Crystal in general is tool-agnostic, and Crystal Clear even somewhat reticent when it comes to modeling tools. The ones that serve Crystal projects best tend to be configuration management, communication, programming, test harness, test code-coverage, and performance profiling tools.

Crystal contains one core practice that easily could be (and should be!) added to RUP: the *Reflection Workshop*. There is nothing to stop every team using a RUP instance from simply getting together after each iteration and reflecting on how to do better. The RUP authors can (and should) simply add a technique and work sample showing teams how to do some form of **Reflective Improvement**.

There are other differences, some of which are essential and some of which are more mental associations about the methodology's practices and its "feel" in practice. Those mental associations generate their own reality in organizations adopting a methodology and so are not to be ignored.

Light and tolerant. Each Crystal methodology is intended, in its core values, to be as light, efficient, non-bureaucratic, and tolerant as the project situation will allow. Those are not declared elements of RUP. This gives the result that Andrea Branca describes: "It is possible to make any process *look* agile, but hard to make it *feel* agile."

Shaping principles. Crystal contains a set of principles (described in *Questioned*) intended to give guidance on how to make shaping choices. Those principles are part of the Crystal "package." RUP recommends tailoring, and lists milestones and templates to tailor, but provides no theory about how to make the shaping choices (these can, of course, be added, building on the theory provided in Crystal).

Speed of methodology shaping. Crystal calls for the methodology shaping to be done in very short order, usually on the order of days, so that cost of that workshop is repaid by the new efficiencies *within the project timeframe*. A Crystal methodology is intended to evolve within the course of the project, which means that methodology shaping has to fast. Shaping RUP need not take months, but often does, which may be

why so many companies avoid development cases and adopt the whole package (which is doubly bad, because they get the inefficiency of an untuned methodology with the cost penalty of an overly heavy one).

Cut-down versus build-up. The approach in Crystal is to start from whatever your team comes up with in the methodology shaping workshop, or whatever you used last time, and build up from there, adding to the methodology only when experience reveals a problem (typically, during the *Reflection Workshop*). Crystal is deliberately underspecified, committing "errors of omission," if you will. The idea is that it is easier to detect when you are missing something you need than it is to detect that you have something extra you don't need. Crystal is deliberately "stretch to fit."

RUP, in contrast, is "shrink to fit." RUP contains an encyclopedia of work product templates (among other contents). Just as you wouldn't read an encyclopedia from front to back, or prepare all the recipes in *The Joy of Cooking* in sequential order, you shouldn't fill in all the templates in the RUP encyclopedia from front to back. Rather, just as you look into a cookbook or encyclopedia for what you need today, you should look into the RUP encyclopedia and pull out the specific elements you need on this particular project.

The approach in RUP, then, is to start from the full encyclopedia and throw things out during the shaping activity. In this sense, it is overspecified.

The two err in different ways for different organizations. I expect groups to have trouble with Crystal because it is too empty to start with; many have trouble RUP because it is too full to start with. I see no escape from a dilemma of this sort.

Visual modeling. As described above, visual modeling is a core element of RUP, where it is an elective element in Crystal.

Tools. RUP is designed to be supported by tools, where Crystal is deliberately agnostic on tools, and even slightly against investment in visual modeling tools. My recommendation since 1992 has been to buy whatever tool lets you draw connected shapes the fastest; that includes pencil-and-paper, whiteboards and cameras, Powerpoint, Visio, and CAD-CAM tools (some of which, I am told, have outstanding usability).

Concurrent development. Crystal has concurrent development as a core recommendation, with guidance for incorporating just-in-time requirements when it suits your project. RUP contains recommendations for incremental growth of the system, but no guidance on how to incorporate just-in-time requirements and concurrent development.

Corporate support. There is no escaping the difference that RUP is supported by a dedicated design team along with marketing, tools development, training and consulting arms. This is not a difference in methodology definition, but it creates a very great difference in adoption. You can't buy a copy of Crystal; you can only buy a copy of two books (*Agile Software Development* and this one) and raid my web site

(Alistair.Cockburn.us). After that you and your team have to get together and *think*, *discuss* and *reflect*. (OK, for a very limited number of teams, a Crystal methodology expert can be hired to do some coaching). That is very different from placing an order for books, tools, courses and consulting to show up at your organization's doorstep.

Can you do Crystal Clear within RUP? Would Crystal Clear be a valid implementation of RUP, assuming of course a colocated team of three to eight people?

Recall that only the first three properties of Crystal Clear are mandatory. Everything else is advisory or at least up for discussion. Those three properties are not conceptually difficult; the question is whether a team doing RUP will actually bring themselves to deliver to a real user every few months, whether the team will move from their private offices to a common area to get **Osmotic Communication**, and whether they will sit down, discuss in a *Reflection Workshop* and achieve **Reflective Improvement**. If they can do these three things, then probably the rest of what they do will fall within Crystal Clear.

It is not obvious to me that Crystal Clear, as written, is a valid implementation of RUP. There may not be enough visual modeling or architecture done or explicit risk management to pass the RUP test. Whether it does or doesn't probably depends on how your development cases come out.

Question 7. What about the CMM(I)⁵⁸?

Crystal Clear was not designed with the intention of meeting requirements of the CMM(I), and it is difficult see how it would move up that ladder. Crystal Clear does address many of the activities called out in the CMM(I), although perhaps not in the way the standard CMM(I) assessor is used to. However, Crystal Clear has a project focus, and the CMM(I) has an organizational focus.

On the other hand, many organizations already certified at CMM(I) level three or above should be able to either introduce Crystal Clear or borrow from it to increase efficiency. Let's look at that closer by imagining an organization already doing software development at Level 3 of the CMM(I) and wanting to use Crystal Clear.

- They have to find a "friendly" user willing to let them deploy the system every few months. This should violate none of their process rules. However, the groups I have encountered that prioritize CMM(I) certification have put their developers so far away from the users and burdened their process so much that it is unlikely that they can deliver running, tested software every quarter.
- They have to colocate the team to get **Osmotic Communication**. This should break no process rules. However, many CMM(I) organizations are committed to distributed teams.
- They have to adopt **Reflective Improvement**. It should violate no process rules that people get together and look for ways of working better. The trouble it may cause is that the team is likely to suggest process changes. Many organizations who have gone to the trouble to get CMM(I) certified did not leave room for the process to change. The shift to an evolving process, though difficult, will be a healthy one.

In all three cases, the needed change is not in the process rules, but in mindset (the harder of the two to change).

Attending to **Focus**, **Personal Safety**, and **Easy Access to Expert Users** should not jeopardize any part of their declared process, but only help the people work better.

Similarly, the **Technical Environment with Automated Testing, Configuration Management and Frequent Integration** may take some work, but shouldn't interfere in any way with the certified process.

The work products of Crystal Clear are advisory to start with, and the organization certified at level 3 is likely to have all of the ones I name (and others besides). The organization might adopt some of the ideas shown in the work products section to simplify or improve their work product set (replacing the Gantt schedule charts with earned-value or *burn-up* charts would be a good start).

⁵⁸ CMMI is broader in scope and more flexible than the "CMM for Software." Indeed, the CMM for Software is no longer officially supported. For the purposes of this discussion, though, there is no significant difference between CMM and CMMI.

There are several other differences between Crystal and the CMM(I). The most obvious of them is that the CMM(I) is hosted and supported by a sizable organization that acts as a resource for training and examination, while Crystal is supported by three few books and a handful of expert consultants around the world. Even though with good luck Crystal will get more support, it is unlikely ever to have organizational support at the same level as the Software Engineering Institute (if it did, it would have to be called the Software Gamesmanship Institute!).

The real core of the differences comes from the underlying values and assumptions about software development. The CMM(I) is fundamentally about consistency, predictability and repeatability. Crystal is fundamentally about efficiency, habitability, and locally optimized rule sets.

Process engineering literature categorizes processes as either *defined/theoretical* (not the CMM(I) meaning of "defined" I hasten to add!) or *empirical*. A *defined/theoretical* process is one that is well enough understand to be automated. An *empirical* one needs human examination and intervention. A textbook in the field has this to say:

"It is typical to adopt the defined (theoretical) modeling approach when the underlying mechanisms by which a process operates are reasonably well understood. When the process is too complicated for the defined approach, the empirical approach is the appropriate choice." (Ogunnaike 1992)

To pull off a *defined/theoretical* process, one must know and be able to adequately measure all the relevant parameters, but more significantly, the system must be non-chaotic. In a chaotic system, even the slightest perturbation in a key parameter sends the system off in a different direction; no amount of measurement and control is enough. The CMM(I) ladder is built on the assumption that these conditions are met.

Personally, I doubt that either the first or the last assumption is met in software development. First, I don't think we know the relevant parameters to measure. Second, I think it quite likely that any team-based creative activity is chaotic, meaning that a tiny perturbation can cause an arbitrarily large effect. Here is a short example:

In a meeting about a new initiative, the people in the room were tense, but seemingly well-behaved. When we took a short break, the woman who was to lead the new initiative turned in her resignation! It seems that one of the (to me) mildly phrased barbs offered by the future CFO was simply "one too much" for her, and she quit on the spot.

Think of the last time that you made what your thought was an innocuous comment and your listener got surprisingly upset. Think of when a key person missed a meeting. Either of those can cascade into an arbitrarily large consequence for the project.

The need to have people detecting things going wrong and intervene in unexpected ways was driven home to me in an email from Glen Alleman, reporting on an organization (CH2M HILL Communication and Information Solutions), which provides nuclear material stewardship applications, ERP, cyber security and other

services to several thousand users under the guidelines of the Defense Nuclear Facilities Safety Board (which controls thousands of buildings contaminated with chemical and radioactive waste). Glen wrote:

We've just come from a "safety stand down," (there's always some safety issue with our business including the IT part). In the lessons learned process one of the questions is, "If you followed the procedure correctly would the problem not have occurred?" The answer from the safety guys is essentially - there are empirical processes that must be present.

This single question allows the organization to detect new, previously unsuspected parameters of importance. At any level of accomplishment, a team should have the humility and healthy skepticism to ask this question and follow where it leads.

For the above reasons, I find it unlikely that the issue will ever seriously arise of a CMM(I) Level-3 organization doing Crystal Clear, although a CMM(I)-certified group may borrow ideas from Crystal Clear.

* * *

This is the time to explain what I meant in the previous section by Crystal turning the CMM(I) pyramid upside down.

Crystal is built on the concept that *every* methodology and organization needs to be *self-evolving*. What the CMM(I) places at the top level, optimizing, I have as a starting requirement. Every Crystal project uses **Reflective Improvement**, guaranteeing that the rules will change over time. If Crystal were to have a certification activity, then even at "Crystal Level 2," the team would have to show that they *changed* their process within and across projects!

There is a less distinct inversion of the pyramid, having to do with measurement precision. In order to step into statistical process control, the CMM(I) calls for numerical measurements to take place at Level 4. The concept I find missing in this approach to measurement is the idea of *making the measurements more precise at higher levels*, starting off with "fuzzy measures" that are nothing more than adjectives or expressions of feelings.

When a person says in a *Reflection Workshop* or status meeting, "that went well," "I'm not comfortable," "I didn't like," "I'm not completely certain," or "there was a wobbliness at the start," she is offering up the integrated response of a professional who has been in many situations in their lives. It is a rolling up of a lot of information gathered from a number of sources, into a single expression. It is a "fuzzy number" in the mathematical sense (Fuzzy Logic ref??), occupying a range of values with a probability distribution function.

These are great measurement values for many situations, particularly when the important parameters are numerous or unknown, as with a software development project.

The pyramid that would be interesting to me would start with personal, fuzzy measurements of the project at the lower levels. Most projects would benefit greatly from recording and analyzing them. As the organization progresses, they might learn which ones are more significant to track, and find ways to sharpen the measurement value, making it more *precise*, eventually ending up with numbers (which I consider sharp, pointy things when applied to such things as project mood).

Question 8. What about UML and Architecture?

The Unified Modeling Language (UML) is a drawing notation standard, not a methodology. In terms of the standard methodology framework (roles, techniques, standards, milestones, etc.), it is one of the standards for one or two of the roles. It is a component of a methodology, but not likely to be a very large factor in the project's outcome.

The UML standard does not make prescriptions or even recommendations about when, where, and how much you draw. With respect to the standard, you can draw out the entire design before starting to program, or you can "think a little, draw a little, code a little," a strategy long recommended by object technology experts and supported by Scott Ambler's book *Agile Modeling* (Ambler 2002).

Discussion of drawing models quickly gets caught up in discussion of architecture and how much design to do when. You will not be surprised to learn that my view is that different people have different preferences and like to think about designs for different amounts of time before starting to program.

Having said that, it is also my view that a design is a "theory," which desperately needs a matching "experiment" to validate it, or more importantly, reveal its weaknesses. Usually, the design doesn't survive its first encounter with real code. This means that the sooner the design gets tested in implementation, the sooner the designer learns where and how it needs improvement.

Some people have a very short thinking time horizon before they desperately feel the need for an experiment: on the order of five or ten minutes. Others have a medium time horizon, on the order of several hours or a day or two, during which they mentally explore all the pitfalls, weakness and potential change requests they may have to adapt to. Others have quite long thinking time horizons, on the order of a week or two, during which they do the same. My experience is that when designers spend more than a few weeks working out a (software) design, they usually have passed the optimal trade-off point, where it would have been useful to create an experiment with real code to get some real feedback⁵⁹.

Taking all those together, I feel that most architects take too long before they make their first experiment, but it is not necessary to adopt the view that design can be replaced just by refactoring and attention to XP's "once and only once" rule.

⁵⁹ I did hear one story of a programmer assigned to program an operating system kernel element, who stared at his design for months, studying its symmetry and aesthetics, until he was totally convinced no symmetry was missing and it couldn't be made simpler, at which point he typed it in (and it worked). Such a situation is extremely rare.

The subjects of UML, drawings, tools, architectural design, and thinking time horizons tend to get confused all together. If you can pull them apart, it frees you up to work in new combinations.

It is, for example, *not* the case that all drawings are UML drawings (see the work products for some examples), nor must architectures be described in UML, or even in drawings, nor must fancy software tools be used to capture either drawings or UML (imagine UML drawn by hand, see the work products for examples), nor does spending time thinking mean sitting at a tool. Finally, and especially, nor is it the case that sitting in front of a UML drawing tool means that any architectural thinking is going on.

The approach typical on a Crystal Clear project is to use the *Walking Skeleton* and *Incremental Rearchitecture* strategies. An early architecture is sketched, programmed and tested in the first iteration, and completed, extended and evolved over subsequent iterations. UML might or might not be used to think through it and document the result.

Question 9. Why aim only for the safety zone? Can't we do better?

Yes, you can do better, and if you're up for it, I encourage you to do so. For example, you can apply all of the XP practices, with test-driven development, full-time pair programming in rotations, automated acceptance tests as well as unit tests, expert users and domain experts on-site full time, continuous and automated builds, shortening the iteration cycle to one or two weeks, and delivering monthly.

You can, and should, if at all possible, institute a weekly one-hour discussion group. People who have done this report back on the increase in the trust, capability and the ability of people to discuss their topics with each other. Work your way through several books, including *The Pragmatic Programmer* (Hunt 2001), the books on test-driven design (Beck 2003, Astels 2004), refactoring (Fowler 2002) and any number of the design pattern books. In these sessions, show and discuss snippets of code from the project, learning to analyze and compare design-programming techniques. Discuss ways of testing code. Use the sessions to try out new languages and tools.

You can and should pay extra special attention to the **Personal Safety** property, and build the basis for **Trust** between people – technical as well as personal trust. You can and should allocate time and money for professional training of the team members, so they stay up to date with our fast moving field. For example, I have totally revised my programming techniques every ten years – I am on my fourth relearning cycle, and that is only for the programming. I haven't even touched usability or user interface design yet, and am woefully out of date on configuration management systems. Where are you? If you haven't significantly changed your habits for design, programming, UI design and testing in the last five or seven years, you are almost certainly far out of date.

The list of how to do *better* than what I have required in Crystal Clear is long. However, each of those things adds difficulty in adoption, and reduces the number of people who can manage the sum total.

The Crystal family of methodologies is not targeted to be "optimal" for any one dimension of possible project priorities. It does not optimize for productivity, or for traceability, legal liability, freedom from defects, laid-backness, distributed teams, maximum use of junior staff, or any of the other priorities you come up with.

Crystal has as priorities: project safety, efficiency and habitability. The efficiency priority says, play the economic-cooperative game well, allowing good speed of delivery on this game while still paying attention to the next game. The habitability priority says, take into account the natural strengths and weaknesses of humans working together, and to make the result reasonably pleasant to live in. Project safety means that that software comes out the door in a fair timeframe, with the project staff intact and willing to go through the same process again.

The efficiency and habitability priorities conflict (as so many issues do that we must attend to in software development). One can be more efficient by being less tolerant. One can be more tolerant by requiring less efficiency.

Hopefully, this book provides enough rigor for your team to get into the project's safety zone, and enough latitude so that you can add whatever next priority you need in your projects.

Question 10. What about distributed teams?

This is a much more interesting question now (2004) than it was when I first formulated Crystal Clear.

Crystal Clear is predicated on having **Osmotic Communication**. It takes me six seconds to push my chair away, walk out of the office, to an office one down and across the hall, and stick my head in to ask the person a question.

In six seconds, I can still discover the person isn't there and get back to my workstation and keep working without losing my train of thought. Or ask the question asked, face to face. The point is to limit the amount of time and energy used to get the question placed, and have short, rich conversation about it.

Richard Herring reported on an experiment in which the team, sitting on different floors of the same building, used microphones and web cameras to simulate "presence and awareness" (Herring 1999). Each person put the webcam photo of each other person as small images on their workstations. With these small images, they could each see whether the other person was busy with heads-down typing, in discussion, or absent. With the microphones and speakers, they could ask and answer questions immediately. Using chat technology, they could ask and answer questions silently and asynchronously (reducing the disturbance factor even more). They even copied code to and from their development environment, so they could exchange real code and not just suggestions about it.

Richard Herring satisfied my concerns about **Osmotic Communication**, and it is clear that his group took care of their social **Personal Safety** and trust issues as well. This was a very creative manner to deal with one of the "sweet spots" (see the first section in *Questioned* for a discussion of the sweet spots and simulating them). If you have an only mildly distributed team, try Herring's ideas.

Victoria Einarsson, in Sweden, described using something essentially like Crystal Clear, except that the developers worked out of their homes. Victoria described spending hours each day on the phone, asking and answering questions and keeping the social connections in place. Several things became clear during our conversation. First, they were indeed effective as a team. Second, they mimicked Crystal Clear in almost every aspect. Third, it wasn't actually Crystal Clear because they didn't have **Osmotic Communication**. Fourth, their success was heavily dependent on Victoria's tremendous social-technical skills of the team leader. In other words, if they wanted to set up a second project team, they would have to be very careful in interviewing the new group leader for having outstanding social and technical skills. They were successful, but it is not a project setup I would easily recommend, because it was so dependent on Victoria's skills.

The success of this team raises a relevant point. It is not the case that you have to do everything in Crystal Clear to be successful. A number of teams I visit are successful

while dropping elements of Crystal Clear. They operate outside of the general safety zone that Crystal Clear describes, and make up for that by having some outstanding people in key positions on the team. Having such people is also a way to win the game, just not the subject of this particular book.

Teams distributed across more than just one city are more dependent on communication technology and outstanding individuals. There are ways for them to succeed, but those ways aren't Crystal Clear.

Question 11. What about larger teams?

I had the pleasure of visiting and interviewing Jan Siric, then of Nordea in Denmark, about his 16-person project, which matched all the characteristics of Crystal Clear except for team size.

The twelve developers, programming both mainframe and workstation software, sat at three long tables, two per side of each long table. Each table had a workstation at each end. The center section of each table was a whiteboard lying on its side. The monitors were mounted on rails so they could be slide into the center section. Jan and three other team leader / manager people sat at tables to the side. A long conference table lay further down the room. The room was airy, spacious and with plants. I immediately noticed the presence of carpeting on the floor and acoustic tile on the ceiling to dampen the noise. As a result, noise was not a problem.

With this arrangement, Jan was able to get the six-second timing even with 16 people, and people could work alone, in pairs, in clusters or as a group as they needed.

What impressed me the most, however, was Jan's ability to detect cracks in the communication. Hearing an odd comment in the cafeteria one day, he interviewed the programmers and found that a mainframe programmer who had only recently joined their team, hadn't fully adjusted to their open conversations with frequent questions and interactions. Jan rearranged the seating, swapped the team managers so that the one with the best social and communications skills was in charge of that particular subteam, and then paid special attention to how the interactions proceeded.

Jan's actions perfectly illustrate Crystal Clear's concept of judging the state of the project in good measure by the quality of the communications.

However, I would not recommend Crystal Clear for teams of 16 people as a general rule. Jan was able to pull it off because he was outstanding at the social/communication issues involved, and able to get both **Osmotic Communication** and **Personal Safety** in place with that particular team.

Under common circumstances, **Osmotic Communication** is very hard to achieve with more than about eight people. XP groups can achieve it with 12 people, using six workstations and two people per workstations. After that, it gets quite difficult.

Crystal Clear is not intended to scale. It is an efficient way to work when you can achieve colocation and **Osmotic Communication**. If you can't achieve that sweet spot, then you have to work differently.

Question 12. What about fixed-price and fixed-scope projects?

There is a tendency to think that agile processes only apply to exploratory projects. This is simply not true. All of my early experiences were on fixed-scope projects. I started using these techniques simply because there was no other way to meet the project deadline than to be super-efficient, as these ideas permit one to be. You should be able to use these, or adaptations of these techniques on almost any small project (many of the strategies and techniques can be used or adapted for almost any project).

Let's see how *Blitz Planning* adjusts for fixed-time and fixed-scope projects.

The fixed-time project. You lay out the cards on the table and walk through the tasks and timing. The answer comes out larger than the time allowed in the fixed-time schedule. One of two things may happen:

- You and your *Executive Sponsor* get creative with the cards and find a way to meet the deadline.
- Despite all your creativity, you can't find a way to meet the timeline.

The first situation shows a good application of the *Blitz Planning* technique. It is possible that the technique allowed you to find a strategy that you might otherwise have overlooked.

The second situation does not show a failure in *Blitz Planning*, Crystal Clear or agile techniques. It shows that someone made a wrong bid on the project. Whether you can convince the sponsors to change the bid depends on factors local to your situation. The result may in fact be that you have to live with the given schedule and simply work overtime. The difference is that everyone in the room knows the situation early.

On one project we went and got the customer-side sponsor and showed him the cards. After working with the cards, he agreed that the project simply couldn't be done in the time given, and changed the terms of the project. I can only hope that you are so fortunate.

The fixed-scope project. You are starting the second, third or fourth iteration. You lay out the cards on the table, and . . . *you don't change the scope of the project!* There is nothing in agile development that says you *have to* change the requirements each iteration. Crystal Clear and XP say that *if* you are in a situation where you need to change scope, this is the time and manner in which to do it. For the fixed-scope situation, you simply leave the scope alone

Question 13. How can I rate how "agile" or how "Crystal" we are?

Proponents of agile development dread measuring how "agile" a project team is. Nonetheless, the manager of a portfolio of projects will want to track how different teams adopt elements of Crystal Clear or some other agile method. Here is how we answered the question at one company. (Figure 7-5 shows the results for a series of projects at different companies, at different levels of adoption.)

- We characterized each project by how many people were being coordinated and the iteration length. Iteration length is secondary to frequency of delivery and rate of user viewings in my view, so we included iteration length only to characterize the projects, not to evaluate them.
- We listed the properties, starting with the seven properties from Chapter 2. We unpacked them to expose their components. Thus, user viewings, automated testing, configuration management, integration frequency, and collaboration across boundaries become separate tracking items.
- We identified which strategies and techniques might be interesting to try. In Figure 7-5, I show a dozen worth considering.

We filled out this table, marking *how frequently* the deliveries, the reflection workshops, the viewings and integrations had happened. Osmotic communication got a Yes if the people were in the same room, otherwise we wrote down the team's spread. Personal Safety got '0', '1/2' or Yes, depending on a subjective rating. (If you don't have Personal Safety on your projects, can you post a '0'?).

For focus, we put down "priorities" if that was achieved, "time" if that was achieved, or Yes if both were achieved. For automated testing, we considered unit and acceptance testing separately (none had automated acceptance testing).

Figure 7-5 reflects the idea that is desirable to achieve all of the properties, but the list of strategies and techniques indicates only what the team might try next. I'm not sure whether it is possible or desirable to come up with an overall score, but I do think that this table may help a team or department manager steer one or more projects. I would, of course, post it as an information radiator.

Project	EB	SO	EV	GS	TH	Ideal
# People	25	25	16	6	2	<30
Iteration length	2 weeks	1 month	3 weeks	1 month	1 month	<2 months
Frequent Delivery	2 weeks	1 year!	1 year!	1 month	4 months!	<3 months
User Viewings	2 weeks	1 week	1 year!	1 month	1 month	<1 month
Reflection Workshops	2 weeks	1 month	3 weeks	0	1 month	<1 month
Osmotic Communication	1 floor	1 floor	yes	yes	yes	yes
Personal Safety	1/2 priorities	yes	1/2 yes	yes	yes	yes
Focus (priorities,time)	priorities	yes	yes	priorities	yes	yes
Easy Access to Expert Users	0	1 day/week	0	0	yes	yes
Configuration Management	yes	yes	yes	yes	yes	yes
Automated Testing	0	0	unit	0	unit	yes
Frequent Integration	3/week	3/week	daily	1/week	1/day	continuous
Collaboration across Boundaries	yes	yes	0	0	1/2	yes
Exploratory 360°	0	yes	0	0	yes	yes
Early Victory	yes	yes	yes	0	yes	yes
Walking Skeleton	yes	yes	yes	0	yes	yes
Incremental Rearchitecture	yes	yes	yes	0	yes	yes
Information Radiators	yes	yes	yes	yes	yes	yes
Pair programming	0	0	yes	0	0	maybe
Side-by-side programming	0	0	0	0	0	maybe
Test-first development	0	0	yes	0	0	maybe
Blitz Planning	yes	yes	yes	0	yes	yes
Daily stand-up meeting	yes	yes	yes	yes	0	yes
Essential Interaction Design	0	0	0	0	0	yes
Burn charts	0	yes	0	0	yes	yes

Figure 7-5. A way to track projects' use of the agile ideas.

Question 14. How do I get started?

Do these four things immediately:

1. Review the constraints for Crystal Clear (the *June 9* email from Crystal to Alistair). If you can't meet these constraints, go ahead and continue with the next steps, but bear in mind you will need additional rules or practices to compensate for the missing project characteristics.
2. Hold a practice *Reflection Workshop* with a few friendly colleagues (this is a *Process Miniature* for the *Reflection Workshop*).
3. Proceed through the *Project Interviews* and proper team *Reflection Workshop* as described in the techniques section. In that *Reflection Workshop*, walk through this book's description of process cycles, and the work products. Review the Team Structure and Conventions work product, in particular.
4. Identify who will serve as the "friendly user" to your group, to review your system as it grows, before it becomes something your can deploy to all users.

After these steps, you should have discussed what it means to be doing Crystal Clear, where your team's strengths lie, and where the team and project's weak spots lie. You will naturally have discussed **Automated Tests**, **Configuration Management**, **Frequent Integration** and ways to attain **Focus**. It will take some time for **Personal Safety** to be established.

5. Run a short iteration. Make it artificially short, one or two weeks, so that you can hold your first *Reflection Workshop* right away and revise your working habits already (this short iteration provides you with a *Process Miniature* on Crystal Clear itself). It will also cement the *Reflection Workshop* as a group technique, provide an outlet for questions and ideas, and increase **Personal Safety** (people will see how their actions are interpreted).

At this point, you are underway. Use this and other books as encyclopedias of ideas to try during successive iterations. Finally,

6. Hold *Reflection Workshops* every two weeks for a month or two, and then decide on the interval to use between the next *Reflection Workshops*.

There. You are doing Crystal Clear.

Chapter 8

Tested (A Case Study)

Stephen Sykes of Thales Research and Technology in the UK experimented with an early version of this book and tried it out. Here is his report on the experience, along with the ISO 9001 auditor's recommendations. Many thanks to both Stephen and Thales.

It is a rare treat to be able to include in this book a field report from someone who tried Crystal Clear. It is an even greater treat to read the report of the ISO 9001 auditor who analyzed their project.

Thales Research and Technology UK is an ISO 9001 certified organization. Wanting to know more about agile development as part of their corporate process improvement activity, they decided to try Crystal Clear on a program for calibrating the camera position for a natural scene recognition system. Stephen Sykes, the initial lead designer for the project, downloaded and worked an early (2002) version of this book for that experiment. The concluding part of the experiment was to have one of their ISO 9001 auditors analyze the project for what should be done differently for when it wouldn't be an experiment.

The top priority for the project team was to complete the prototype by the end of January, 2004. The ideal would be to have a deliverable-quality product. In fact, they ended up with a good-quality demonstrator that could be turned into a product at reasonably short notice. The sponsor terminated the project at that point, electing to drop from the project scope the trial of a second calibration algorithm, and certain project documentation activities.

The ISO auditor noted the difference between the final project scope and the original with the description: ". . .the project was unfortunately terminated early. However, due to the nature of the CC method, the early termination did not prohibit a useful software product."

In terms of Crystal Clear's attention to the sponsor's priorities, this counts as a satisfactory project outcome (" . . . at the last reflection workshop, the sponsor also said that the advantages greatly outweighed the disadvantages. He's suggested we try it again, on a larger project, provided the fit with our existing procedures can be resolved"). The methodology has become classified as an allowable candidate for projects in the "rapid application development" category, and is slated to be used again in the near future.

Here is the report, trimmed to fit within this chapter.

The Field Report

Executive Summary

This report describes a pilot project using the Crystal Clear methodology. The report represents part of a body of knowledge about Agile methodologies built at Thales Research and Technology (TRT UK), as part of its Small Systems Software Engineering activity. Crystal Clear itself was developed by the methodologist Alistair Cockburn.

Crystal Clear is an Agile methodology. 'Agile' is an umbrella term for methodologies such as XP, DSDM and Scrum. In general, Agile methodologies aim to reduce the risk of building the wrong thing, and to deliver value as early as possible. A literature survey of the Agile methodologies was conducted during 2003, and it was decided that Crystal Clear was the one most likely to be useful at Thales⁶⁰. Crystal Clear was selected because it emphasizes

- **Efficiency.** Crystal Clear aims to minimize waste, while focussing the team on the important features.
- **Habitability.** Crystal Clear is designed to be comfortable to use, so that teams do not feel a need to avoid following it.
- **Safety.** The methodology aims to increase the probability of successful delivery.

Crystal Clear is intended for small, low-criticality projects. Compared with XP (which is the most well-known Agile methodology), Crystal Clear is more tolerant of individuals' different ways of working, but includes stronger planning activities.

We ran a pilot project in order to evaluate the methodology. We aimed to test the methodology against the claims made for it, and to determine any issues that might prevent adoption. Project 'CamCal' was chosen as the pilot because it was small and non-critical (which is the kind of project the methodology was designed for), and because its members were willing to be part of the trial. Unfortunately, the project was halted before its planned end date, for reasons outside the scope of the pilot project. The project still succeeded in delivering a product to the satisfaction of its customers.

Our conclusions are summarized as follows. The methodology met its objectives of being efficient, habitable and safe:

- **Efficiency.** Performance metrics showed high productivity.
- **Habitability.** The team would be happy to use the methodology again.
- **Safety.** The project was converging on its agreed end date until it was curtailed. The project still created a usable product.

⁶⁰ This was Stephen's view. It is not a corporate Thales position.

The following issues need attention in adopting the methodology:

- It may require established roles to change
- It challenges established documentation practices
- It requires a broadly-scoped contract

The TRT UK Quality Assurance team conducted an informal assessment of the project from the perspective of ISO 9001 and TickIT, concluding that the methodology could be made compliant with these standards, with the addition of a few extra practices and records. The QA team proposed that Crystal Clear be incorporated into the Quality Management System at TRT UK, and proposed further studies using the methodology.

An analysis of the methodology from the CMMI perspective is ongoing at the time of writing. We aim to trial the methodology on a project with more developers.

Introduction

The project developed a program for use in connection with a computer vision system called ZYX. Our pilot project, CamCal, developed a camera calibration tool for ZYX. The purpose of camera calibration is to inform ZYX of where the camera is, and where it is pointing. This information is encoded in a calibration matrix that maps 3D world coordinates, to 2D image coordinates. The matrix is needed by ZYX in order to understand what it sees.

The starting point was a prototype camera calibration tool that had previously been developed in MATLAB. The purpose of the project was replace the prototype with something less fragile, more suitable for presenting to customers, and more suitable for use at customer sites. We decided to write our new version in Java, with some of the mathematical components implemented in MATLAB.

CamCal was a small project, with only 1 developer, and a total cost of just over six man months. The work was funded internally by TRT UK and there was no immediate external customer.

Project Chronology

The key dates on the project are shown in **Error! Reference source not found.** below. In order to keep this section reasonably short, we will present only the most significant events on the project. These include events where there was a change to the requirements or to the schedule.

The project involved the following people. With the exception of Stephen, who is the author of this document, their names have been changed.

- Sam, the project manager in charge of the wider project of which CamCal was a part
- Liam, a computer vision consultant

-
- Stephen, an engineer with a programming / algorithms background
 - Desmond, an engineer with a Java background

Liam's prototype

Liam conceived the idea of a camera calibration tool. He wrote a prototype, mostly in his own time. The prototype included all the key features required, but it was fragile and not particularly easy to use. What was needed was a tool written to professional standards that we could deliver to customers.

Proposal

Sam and Liam began work on a proposal for a camera calibration project. They would need at least one software engineer. Stephen would be released to work on the camera calibration project on a part-time basis, provided the project use an Agile methodology.

A proposal document was written that laid out the project's initial requirements, the development approach, an initial work breakdown structure, and an estimate for the work. The estimate was for 105 days of effort. The initial allocation of roles was that shown below.

Name	Roles
Sam	Sponsor
Stephen	Lead Designer , Co-ordinator, Writer, Tester
Liam	User, Designer , Business Expert, Writer, Tester

Figure 8-1. Initial allocation of roles

The proposal document was put to Senior Management and accepted. The project started on August 18th 2003.

The proposal document was based on an early (Jan 2002) version of Crystal Clear. This did not include project start-up activities. But it was clear that there were issues we needed to resolve before we could begin the iterative development. So, we borrowed the idea of an 'Exploration phase' from XP. This was the first phase of the project and is described next.

	Event	Effort (days)	Dates
Project Startup	Liam's prototype		Dec 2002 to Feb 2003
	Proposal		June 2003

	Exploration		Mon 18/8/03 Fri 5/9/03
	Desmond joins project		Tues 9/9/03
	First Planning meeting		Wed 10/9/03
Increment 1 Walking skeleton	Development		Thurs 11/9/03 to Fri 17/10/03
	Viewing		Fri 17/10/03
	Reflection Workshop		Mon 20/10/03
	Planning meeting		Mon 20/10/03
Increment 2 CP entry and edit	Development		Mon 20/10/03 to Fri 7/11/03
	Viewing		Fri 7/11/03
	Reflection Workshop		Thurs 13/11/03
	Planning meeting		Thurs 13/11/03
Increment 3 Calibration	Development		Mon 10/11/03 to Fri 5/12/03
	Viewing		Mon 8/12/03
	Reflection workshop		Mon 8/12/03
	Planning meeting		Mon 8/12/03
Increment 4 consolidate for year end	Development		Tue 9/12/03 to Tue 23/12/03
	Viewing		Tue 23/12/03
	Reflection workshop		Tue 23/12/03
	Planning meeting		Mon 5/1/04
Increment 5	Development		Mon 5/1/04 to Fri 23/1/04

	Viewing		Mon 26/1/04
	Reflection workshop		Mon 26/1/04

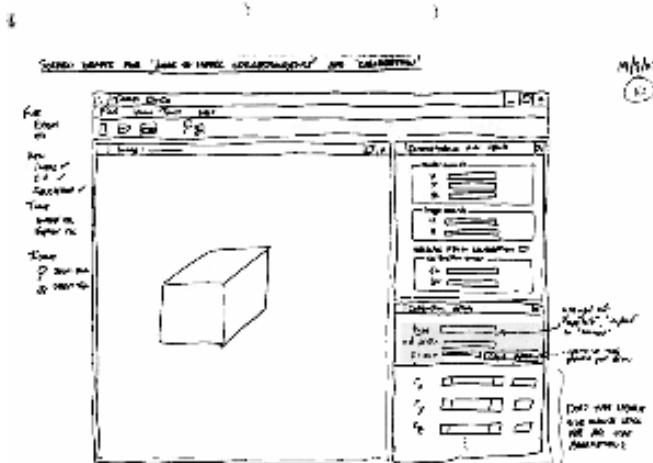
Figure 8-2. Project chronology

Exploration

The 'Exploration phase' was a two-week long task in which Stephen and Liam captured more detailed requirements, investigated the key technical issues, and firmed up the estimates⁶¹.

The proposal document gave a set of high-level requirements, but more detail was needed. Stephen and Liam worked together to flesh out the details, with Liam (as User) being the main source of requirements. They captured the requirements using a printing whiteboard, drawing several draft screen-shots such as the one shown below (Figure 8-3).

The printouts were numbered and kept together in a folder (the printout below is number 10). They also started a requirements document.

**Figure 8-3.** An example white-board printout

The proposal document had left open the issue of which language the software was to be written in. This needed to be resolved quickly because it affected the composition of the team. After some investigation it was decided to use Java for the GUI, and MATLAB for the underlying mathematics. The MATLAB source-code would be compiled into a DLL compatible with the Java Native Interface.

⁶¹ Readers will recognize this as similar to the Exploratory 360° strategy described in Chapter 3.

Crystal Clear expects at least one team member who is thoroughly familiar with the technology, but neither Liam nor Stephen had used Java before. So we asked for another team member, and received Desmond, a strong Java developer. Desmond relocated to sit with Liam and Stephen in the same bay.

At the end of the Exploration phase, the project roles were as shown in **Error! Reference source not found.** Desmond was allocated to the project 100%. Stephen had a commitment to another project so was allocated 50%. The other team members were called in when necessary. Stephen is shown as Lead Designer because that was the original plan. But Desmond had the more relevant skill-set, and it was clear that he would lead the development work once he was up to speed with the project.

Name	Background (partial)	Role(s)
Sam	Project Manager	<u>Sponsor</u>
Liam	Technical Consultant	<u>User</u> , Business Expert, Writer, Tester
Stephen (50%)	Software engineer, engineering maths	<u>Lead Designer</u> , Coordinator, Writer, Tester
Desmond (100%)	Software engineer (Java expert)	<u>Designer</u> , Tester

Figure 8-4. Project roles at end of exploration

First planning meeting

We conducted a planning meeting using the project planning 'jam session' technique suggested in CC. Sam, Desmond and Stephen sat round a table and contributed task cards. An example task card is shown below, in Figure 8-5. Each task card had a title, a brief description, the name of the person, and an estimate in man days. The description did not need to be very long, nor understandable to anybody outside the project. The estimate was in the top right hand corner in man days. The team adopted the convention that the estimate included any associated unit testing effort.

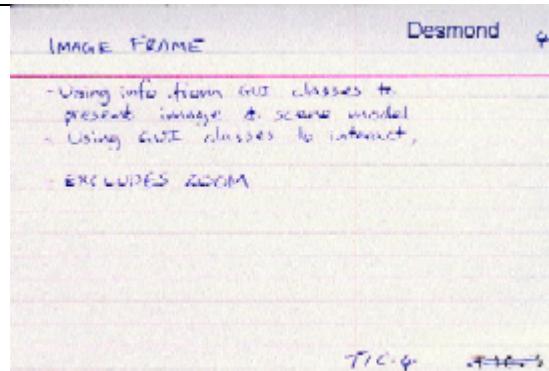


Figure 8-5. An example task card

The key tasks went to Desmond, as the person with the strongest Java experience. We spread the task cards over the table and agreed an ordering that made sense. (A schedule that 'makes sense' is one that maximises the chances of achieving the project objectives, as stated in the mission statement). We then gave each card a label (at the bottom of the card) so that we could reconstruct the ordering. After the meeting, the tasks were copied into Microsoft Project, which is the standard scheduling tool at TRT UK.

In fact, the majority of the task cards were created before the meeting, by Stephen and Liam, who had gone through this process already. Desmond essentially revisited the task descriptions, technology choices, and estimates. Nevertheless, the first planning meeting was a key point in the project. The project had a clear direction, and all had contributed.

By adding up the totals on the task cards, we arrived at an improved estimate for the remainder of the project: 134 man days. Adding the cost of exploration (already spent) gave 154 man days. This was higher than the estimate in the proposal document, but the sponsor was happy for work to continue on the basis of the schedule we had agreed.

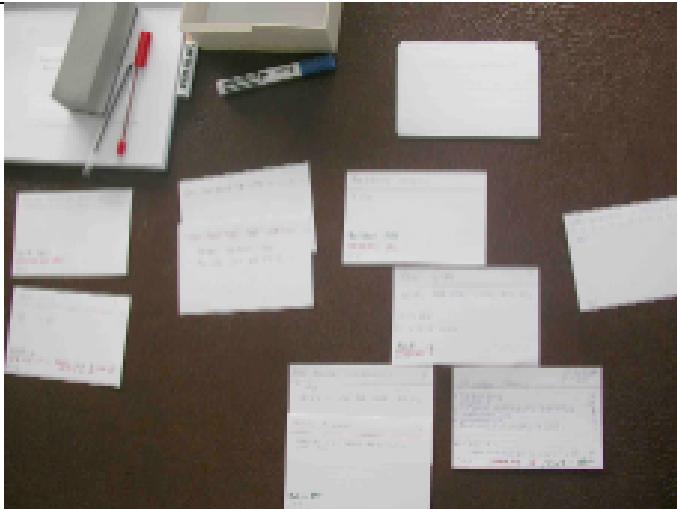


Figure 8-6. One end of the table that was covered in task cards

Increment 1: Walking Skeleton

The purpose of the first increment was to create a skeletal application incorporating the main architectural features and some core functionality. In Crystal Clear parlance this is called a 'walking skeleton'. Key tasks would be the creation of the object model, and implementation of the software for projecting a scene model onto the screen.

First we had to select and configure our tools. At TRT UK, projects mainly decide for themselves which tools they will use. The tools were chosen by the team members, especially Desmond, on the basis of his experience on other projects. The tools were as follows.

- Design tool. Together 5.5 was selected. This generates skeletal Java code from UML diagrams. Subsequent changes to the software are automatically reflected in the UML representation.
- Programming tools. NetBeans 3.5 was chosen as our Java IDE, running on J2SE 1.4.2. MATLAB 6.5 had already been chosen for the calibration software, with Microsoft Visual C++ for the DLL 'glue logic' that would connect the compiled MATLAB to Java.
- Unit testing tool. We selected JUnit.
- Version control tool. We used Microsoft SourceSafe 6.0, which is standard at TRT UK.

Once the tools were selected and configured, Desmond began creating the architecture for the software, discussing design issues with Stephen using the printing

white-board. The application was designed using the 'Model-View-Controller' paradigm.

At this point we need to say a little more about the application. The application uses a 3D scene model, which describes the objects in the region observed by the camera. The scene model consists of a set of 3D points, which are joined together to form the edges of the objects. The scene model is projected onto the 2D screen according to the location and orientation of a virtual camera. The picture formed by the virtual camera is called the 'projected scene model'.

During increment 1, Stephen wrote the Java code for transforming the 3D scene model into 2D screen coordinates.

Here, there was a change of plan. The original schedule had called for this software to be written in MATLAB, compiled to DLL and accessed over the JNI. It soon became clear that we were not going to achieve that level of integration in the time available for increment 1. So Stephen wrote the code in Java instead: this way, Desmond would have something to integrate within increment 1. If the software proved to be too slow, it could be rewritten in MATLAB later. (MATLAB is optimized for matrix operations, so we would expect higher performance). Stephen and Desmond re-planned their work by conducting a brief planning meeting.

Simultaneously, the team continued to refine the requirements, with Liam and Stephen working together to produce updated screen drafts. Most of the requirements discussed at this stage concerned usability issues. Liam made one request for completely new functionality (hidden line removal), and this was turned down because it was found to complicate the design too much. We knew that Sam's main concern was to deliver on time, rather than to deliver the most comprehensive product.

By the end of Increment 1, the team had succeeded in creating the walking skeleton.

The scene model was only a dummy, consisting of a few geometric shapes (a box, a prism etc). The final application would use a scene model that represents the region being viewed, in this case part of a railway station. The user could click and drag the camera icon around the plan view, and the application automatically updated the projected scene model.

It was only a start, but enough to demonstrate some core functionality.

Towards the end of the increment, Sam gave the team its 'mission statement'. This is one of the work products required by Crystal Clear. It is the only work product asked of the Sponsor.

Camera Calibration Mission Statement

The mission of the camera calibration project is to:

- Develop a maintainable, simple to use, portable and extendable software tool that will enable intelligent video surveillance system installers and repairers to rapidly,

- accurately and efficiently calibrate their system cameras with respect to a pre-defined three-dimensional scene model.
- Establish a basis for developing a future tool set that can manage all aspects of intelligent video surveillance system installation and repairers activities. Such a tool set will include the camera calibration tool, and will add scene model creation capabilities. It will be able to grow with the technology growth of the video surveillance market.
 - Develop and trial the use of new software development methodologies and processes, with a view to improving the software development process used within Thales, for small to medium sized software projects.

The development priorities are:

Sacrifice others for this: Complete by end of January 2004.

Ensure accuracy and quality

Retain if possible: Usability, potential to grow into a more extensive tool

Sacrifice these first: Execution speed, portability, additional interfaces, internal format storage.

Figure 8-7. CamCal Calibration mission statement

At the end of increment 1 we held a viewing and a reflection workshop.

Viewing 1

The Viewing meeting was held on Fri 17/10/03 at the end of increment 1.

Desmond demonstrated the application to Liam and Sam. The meeting was cordial, with Liam and Sam taking a keen interest in the details of the product. Several issues were raised in connection with the software and these were recorded on the printing whiteboard. The whiteboard printout is shown below in Figure 8-8. At the end of the viewing the team had a view of the product status that was shared, complete, and reasonably positive.

On CamCal, we chose to have a Viewing at the end of every increment. This is more than Crystal Clear demands, the methodology only calls for a minimum of two per release.

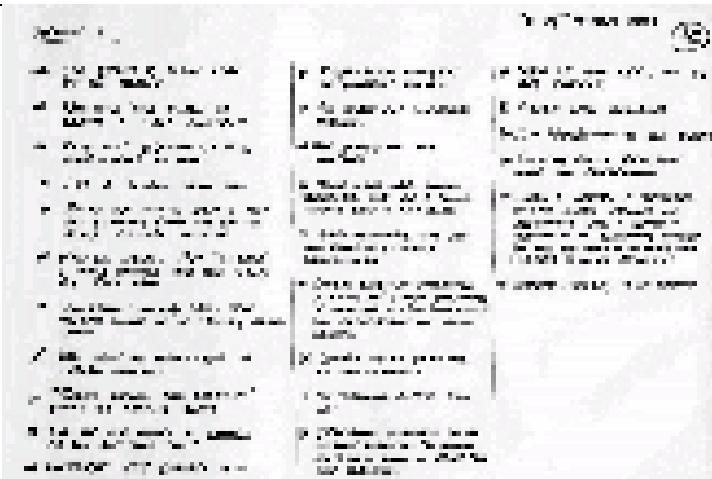


Figure 8-8. Whiteboard printout from Viewing 1

Reflection Workshop 1

At the end of increment 1, the team held a Reflection Workshop. Its purpose was to identify which aspects of the process were working and which needed to be changed. The notes from the reflection workshop are shown on the whiteboard printout below. The main points were

- **All to spend a day reading CC.** The team had not been trained in using the methodology. The only training material that was available was the manual on CC, so it was suggested and agreed that each team member should spend a day reading it.
- **Desmond as Lead Designer.** Desmond had been making the key technical decisions, because of his experience with Java. Stephen had just been acting as 'process champion'. It was proposed to acknowledge this situation by nominating Desmond the Lead Designer. Desmond was unsure, but agreed to think about it.
- **'Keep this' statements.** The team found much about the methodology that they wanted to keep. In particular, the team members were working well together. Several of the team members had worked together before on projects that had been run differently: they were finding it easier to work together on this project.
- **Absence of team members.** Apart from Desmond, all of the team members had significant commitments to other projects. The team recognized this as a problem. Agile methods rely on tacit knowledge, so are vulnerable if the people are not present.

- **Mismatch with TRT UK procedures.** Sam was uncomfortable with the fact that he was formally responsible for the project, but not in control of its schedule⁶².

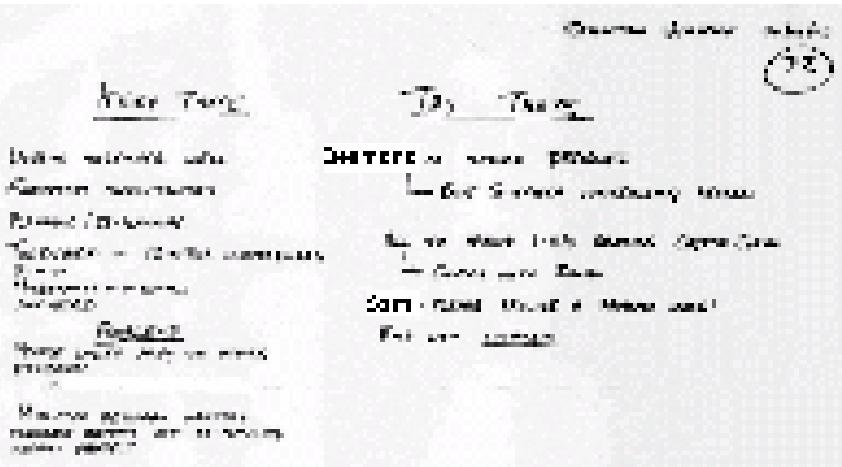


Figure 8-9. Whiteboard printout from reflection workshop 1

Increment 2: CP Entry and edit

The objective for increment 2 was to enable the user to enter and edit 'Correspondence Pairs' (CPs). A correspondence pair specifies an association between a vertex in the scene model, which is three dimensional, and a point in the image, which is two dimensional. CPs are entered into CamCal by mouse operations. The user clicks on a point in the image, and drags to the intended vertex in the projected scene model. The CPs are ultimately used as the input to the calibration algorithm, which computes the calibration matrix required by ZYX.

Increment 2 began with a planning meeting (Monday 20/10/03) in which the project schedule was rebuilt based on the current situation. The main change from the original plan was that Stephen would not be available for increment 2. He had a commitment to another project and would not have time. After discussing the options, the decision was made to delay his contribution until increment 3.

In the new schedule, some time was allocated to address the issues captured at Viewing 1 (**Error! Reference source not found.**): but the priority was to press ahead with the high-risk aspects of the project, such as calibration and the integration with ZYX.

By the end of the increment, the facility to add and edit correspondence pairs was ready for demonstration. It used a real scene model, instead of just geometric shapes.

⁶² (We'll see discussion of this point at the end of the field report.)

We held another viewing session, and another reflection workshop. The viewing generated more ‘issues’, which were recorded on a whiteboard printout. Again, the team had achieved a shared view of the status of the product.

This reflection workshop generated fewer points than the first one had. The main outputs from the reflection workshop were

- At the previous reflection workshop, the team members had agreed to read up on the methodology: but they had not found the time. Desmond agreed to do this in increment 3.
- The project was not meeting certain parts of the methodology. The peer reviews had not happened. The risk list had not been maintained. The requirements document had not been firmed up. We agreed to address these in increment 3.

Increment 3: Calibration

The increment began with a planning meeting. The tasks for this increment are summarized below. They were all completed by the end of the increment.

- Peer code review
- Implement the calibration algorithm in MATLAB, and integrate this with Java
- Implement GUI facilities supporting calibration
- Initial system test
- Desmond to read CC

The integration between MATLAB and Java had been considered a risk area, so it was a relief to achieve it.

We tried posting the task cards for the increment on a whiteboard in the team’s working area. This proved to be very effective at communicating the schedule, and we retained the practice for the remainder of the project.

In addition to these scheduled tasks, the requirements continued to be refined on an ongoing basis. Liam’s original requirement had been that CamCal support two particular calibration algorithms. But when integration was discussed, we learnt that only one algorithm was compatible with our target system. Since the purpose of the project was to develop a calibration tool for that system and the team had a tight schedule, the decision was made to omit the second algorithm.

Stephen firmed up the requirements document, and had it reviewed by Liam and Desmond. There was also more discussion about detailed GUI behaviour. We will relate some details of the discussion, to give a feel for the kind of decisions that were taking place.

During Increment 1, Stephen and Liam had sketched out a user interface that was wizard-like, so that the user would be guided through the activities that calibration involves. Each page of the wizard would have panels of fixed size and position. Desmond, who was more experienced at GUI design, felt that this would be too

restrictive and favoured a more flexible design, giving the user the ability to move, resize, and show/hide the windows. He had implemented the software so that this aspect of its appearance could readily be changed. The difficulty was that by giving the user this level of control, the wizard-like nature of the application was lost, and it might not be obvious to the user what he needed to do.

What was needed was a resolution to the question of how the GUI should behave. At the planning meeting, Desmond had taken a task to sort the issue out. Desmond created a design in which the user could select between views using radio buttons. The views were

- **Manual alignment.** Here, the projected scene model reflected the position and orientation of a virtual camera that the user could control.
- **Algorithmic calibration.** Here, the projected scene model was formed using the calibration algorithm.
- **Xxxxx calibration.** This was a dummy option. Desmond left it in place to show that the user interface could be extended to contain more than one calibration algorithm.

Liam felt this arrangement was still not ideal, because the controls for the 'manual alignment' view were visible when the first calibration was selected.

We presented the system at a Viewing on Mon 8/12/03. At this stage, most of the core functionality of the application was present. One could load an image and a scene model, add correspondence pairs, and run the calibration algorithm. The calibration algorithm finds a transformation that fits the scene model over the 2D image, so that the wire-frame projection of the scene model is aligned to objects in the image.

For this Viewing, a QA representative had been invited. This was the first QA involvement with the process. Subsequently, a QA representative was invited to every meeting on the project. [The QA perspective is presented separately, following this field report.]

Increment 4: Consolidate for year end

The main objective for this increment was to bring the project to a state where it could be closed relatively painlessly. There was a possibility that issues outside the scope of the project might lead to it being closed at the end of the increment (which coincided with the end of the year). In fact, this requirement did not affect the planning too much. The key tasks that needed to be done for our target of end of January 2004, needed to be done anyway. The tasks for this increment are summarized below. All were complete by the end of the increment.

- Create a CD of the application, including an installation tool
- Integrate with ZYX. This involved having CamCal create a calibration file, and testing that ZYX could make use of it.
- Start a User Manual. (It could not be finished until the user interface was complete)

-
- Code reviews and associated actions.

At this point, the project had a page of ‘issues’ from each Viewing. Liam prioritized them, and they were copied into an issues list that Desmond had created in Microsoft Outlook. Using Outlook meant that we would all have instant access to the most recent list.

The increment terminated with a Viewing and a Reflection Workshop. For this Viewing, we ran the software directly from the installation CD, to demonstrate that this worked. Only a few points were raised in the viewing, everybody knew that there wasn’t time for big changes. The reflection workshop was quiet as well, with nobody suggesting any changes to the process.

Increment 5: Complete

The Christmas holiday period provided a natural break between increments, and the fifth increment began on Mon 5/1/04.

At the end of 2003 there was a decision, made outside of the scope of the project, that CamCal should be completed cheaply at the start of 2004. To achieve this, the decision was made to omit the acceptance tests and code reviews that had previously been scheduled. The justification was that the software had already passed various test and review activities, and this had created a degree of confidence in the code. The main tasks for the increment were

- The prioritized issues list. After some negotiation, it was agreed that Desmond would work down the issues list, addressing all the issues until either he ran out of time, or until a certain point in the prioritized list was reached, whichever came first.
- Liam to finish the user manual
- Desmond and Stephen to write a maintenance handbook

At the end of the increment, we held a final Viewing. Compared with the previous increment, it was generally a bit tidier and more polished. This was the result of Desmond’s work addressing the items in the prioritized issues list. Sam accepted the application as ready to demonstrate to customers, and asked that it be installed on the ZYX presentation system.

We held a final reflection workshop. This turned into a general discussion about how the methodology went overall. The team members’ views were generally very positive. The only substantial issue raised was the fit with TRT UK procedures. If this could be satisfactorily resolved, then all the participants would be happy to work this way again.

(In fact, there is a procedure at TRT UK for addressing QMS issues of this sort: any employee may raise suggestions for process improvement).

Compliance with Crystal Clear

In this section we summarise the extent to which our pilot project was consistent with Crystal Clear. We evaluate compliance against the following aspects of Crystal Clear : the properties, the roles, the life-cycle, and the work products.

- Properties

The CC properties were achieved.

- Roles

The pilot project allocated roles to the people as Crystal Clear asks. There were some deviations from the ideal:

The 'Lead Designer' role had to be shared between Stephen and Desmond. Stephen took the methodology aspect of the role, and Desmond took the technical leadership. This division was necessary because the Lead Designer role requires knowledge both of the technical issues and the methodology. While Desmond was the 'Lead Designer' in the literal sense, he had not been trained in the methodology and so Stephen had to take that part of the role. This division of responsibility caused some confusion in places, and the project would have worked better had the role been taken by a single person.

There was a deviation in the role of User. The product had two intended uses: (1) as a demonstration tool to be used by senior figures at TRT UK, and (2) as an installation tool to be used by installation engineers. Liam is a real user of the former category. We did not have a user in the latter category.

- Life cycle

The pilot project initially used an early draft of the methodology, that did not include the Crystal Clear project start-up activities. These were therefore omitted. In other respects the project followed the Crystal Clear life-cycle. We note the following:

- a) The methodology calls for frequent release of software to users, so as to obtain feedback. Our pilot project achieved that by having a user on the team. The user was given an updated copy of the program every so often.
- b) Crystal Clear calls for a minimum of two viewings per release. We had more viewings than this, having made the decision to have a viewing at the end of each increment. All of the viewings were useful, but the later viewings were less useful because the product had changed relatively little between increments.

- Work products

The pilot project created most of the work products required, apart from the following

1. Since there was only one release, the 'release sequence' and 'release schedule' reduced to a single end-date.

2. The 'migration code' work product is for database applications and was not applicable for our pilot project.
3. The acceptance tests were removed from the schedule when the project was curtailed, so there were no acceptance test results.

Analysis

Performance against estimates

The priorities for the pilot project, as stated in the mission statement were

Sacrifice others for this:	Complete by end of January 2004. Ensure accuracy and quality
Retain if possible:	Usability, potential to grow into a more extensive tool

The project was steered so as to meet these priorities. Note that the primary objective was the end-date, not the final cost. So whenever the end date or usability appeared threatened, the team reacted by spending effort to solve the problem. The team was able to do this because only one of its members was working on the project full time. The others had some flexibility in how much time they committed to the project.

This is reflected in the graph of estimated versus actual effort shown in Figure 8-10 below. The estimates increase gradually from the first planning meeting, to the fourth. The fifth planning meeting shows a reduced estimate that reflects the decision to curtail the project. (The curtailed project met the underlying business need, which was for a demonstrable product, not a deliverable one).

The proposal phase was not conducted according to the Crystal Clear process, so does not form part of our experiment. It created an estimate that was found, within a few weeks of starting the project, to be a considerable underestimate. The main reason for the problem was that the proposal phase did not include sufficient time to determine the technology that would be used or the staff who would do the project.

In fact, the figures shown in **Error! Reference source not found.** were not computed until after the project. In hindsight we feel it would have been advantageous to have computed the figures on the fly.

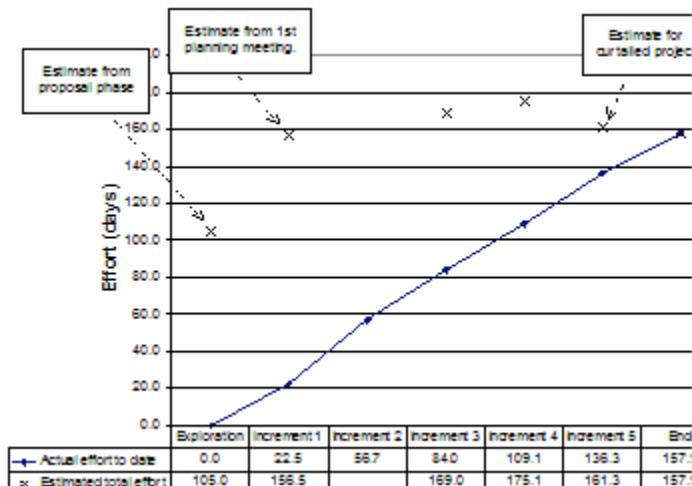


Figure 8-10. Estimation Accuracy (actuals and estimates at the start of each increment)

Analysis from the ISO 9001 viewpoint

[The field report contained an excerpt from the auditor's analysis. A longer excerpt from his report is presented separately, following this field report.]

Conclusions

This section will present our evaluation of Crystal Clear. We evaluate the methodology against the claims that are made for it. We also present a number of issues where care is needed in using the methodology.

Evaluation of Crystal Clear against its claims

The claims made for Crystal Clear are that it is efficient, habitable and safe. On the basis of our pilot project, we support these claims.

- Efficiency

The final productivity metric for the project was 81.7 lines of non-comment code (system code plus unit test code) per day. This metric was computed by dividing the total number of lines of code generated, by the total effort expended, including all the non-coding effort. [Note: Unit test code was 26.4% of the Java code.]

The project did not include acceptance tests but this would not have changed the overall figures very much. Acceptance tests would probably have found some small errors which we would then have corrected: but overall, we knew that the User and Sponsor were already content with the product, and it had already been subject to various tests.

This high level of productivity arose, firstly, from the skills of the designer (Desmond). The methodology helped by keeping the development focussed, and by reducing waste. ‘Waste’, in this context, means anything that does not contribute to the final product.

- Habitability

All of the team members would be happy to use this methodology again. Several positive comments were received from the team, including:

Liam (User). “Overall, taking part was an enjoyable experience, and one which I would be happy to repeat”

Desmond (Designer). “I usually felt very focussed and energised ... I felt generally in control”.

Sam (Sponsor). “Communications were excellent ...the team worked together well.... the software was delivered in a satisfactory working condition.”

Further comments from the team are presented further on. Sam had some concerns about the fit between the role definitions in Crystal Clear and those expected at TRT UK. Sam’s approval of the methodology was contingent on this issue being satisfactorily resolved.

- Safety

A ‘safe’ methodology is one that increases the likelihood of the project succeeding. The pilot project succeeded in that it created a product that satisfied the Sponsor. Beyond that, several aspects of the project were demonstrably safe: the plans were always meaningful, hard issues were resolved in a timely manner, and the team was always focussed on the most important features.

The initial estimate for the project was incorrect, but this was conducted outside of the Crystal Clear experiment.

Crystal Clear does require an experienced and competent Lead Designer in order to be safe. Crystal Clear does not define ‘techniques’ (for configuration management, testing etc), and the intention is that the Lead Designer fills this gap.

Issues requiring attention during adoption

The following aspects of the methodology need attention during adoption:

- It may require established roles to change

Crystal Clear gives people roles that may not match the established organizational roles.

At TRT UK, each project has a project manager who is held responsible for delivering the project on time and within budget. On our pilot project, this was Sam’s role. A TRT UK project manager would expect to own the project schedule. But under Crystal Clear, scheduling was a communal activity led by the Lead Designer. This was a safer arrangement since it brought the skills of all the team members to bear on the

scheduling problem. But it created a situation where Sam had the same responsibility that he would normally have, but with reduced power to influence events⁶³.

- It challenges established documentation practices

Crystal Clear expects initial requirements in the form of a mission statement, an actor-goal lists, and some use cases. These are not expected to be sufficiently precise to define every aspect of the product. The details are intended to emerge through interaction between representatives of the developer and customer organizations.

Crystal Clear also places great reliance on tacit knowledge over written documents. But tacit knowledge is only available to other team members, when the people are simultaneously present. It follows that in order to achieve the greatest efficiency, the team members should work together continuously on the same project.

- It requires a broadly-scoped contract

Crystal Clear expects the detailed requirements to emerge as the project proceeds, but is silent on the question of how this could fit into a contractual framework⁶⁴. One approach would be to adopt the DSDM practice of baselining the requirements at a high level. DSDM has been used successfully in conjunction with the CMMI, so it is unlikely that this approach would create a CMMI or ISO compliance problem.

Recommendations for future work

This pilot project exercised many aspects of the methodology, but was limited by its small size. We recommend a trial on a larger project, having more than 1_ developers, and more than a single release. A larger trial would still need to be within the Crystal Clear scope, having no more than eight team members.

Crystal Clear is designed to accommodate practices drawn from XP, such as test-first design and pair programming. We would recommend that these practices be incorporated into a future trial.

A study on the fit between Crystal Clear and our standard internal methodology is planned for later in 2004. If possible we would also like to conclude the work on the fit between Crystal Clear and the CMMI.

Comments from the team

The following comments are unedited except where indicated.

⁶³ I discuss options that Sam might have in the Commentary afterwards.

⁶⁴ This is discussed in the chapter, *Questioned*.

Liam's comments (user)

11th December 2003:

I have never worked on a project in which team members have such a clear picture of the state of the project's progress. The regular 'viewing sessions and generated 'to-do'/bug lists being posted on a white board, creates a single point of access which all team members can share. The Project Plan, and associated task lists being posted-up also helps.

The only down-side I have experienced is the high-level of conversation/discussion that's takes place around the main board, disturbing thought when one is not involved in a team discussion. An ideal solution would be having the space to move board and discussion away from non-involved team members, during such meetings.

3rd February 2004:

The manner in which the Agile/Crystal Clear methodology roles had to be modified to fit with existing company procedures, led to a degree of reactivity in the evaluation process. For example the senior-designer/coordinator (Stephen), appeared to be questioned over the management of project costs by the sponsor, when in fact only the sponsor ('Sam') had full access to the cost information. If the approach evaluated had been fully integrated with company procedures these issues would not have arisen. Overall, taking part was an enjoyable experience, and one which I would be happy to repeat.

Desmond's comments (designer)**Focus & Efficiency**

As a designer/developer, I usually felt very "focussed" and "energised". During each increment I felt I had a clear idea of which features we would be adding, roughly how they should work, how long they should take, and which were the most important. I felt generally "in control".

The short length of the increments and the clear idea of a goal at the end also helped maintain focus on essentials. Each short increment had a clear goal: a set of functionality to be implemented and a demonstration of it to the users. The overall feeling was that we were being fast and efficient. That we were focussed clearly on valuable features, and that we made efficient progress.

On other developments one might get a chance to "disappear" for a few months, working towards one big goal. Having contrasted this with Crystal Clear, it now seems to me that Crystal Clear is much more efficient: there is always a slight pressure to deliver, to concentrate on the essentials, that feels like it delivers real value. Or rather, that it avoids some of the inefficiencies in the old approach.

This energised, focussed feeling is produced by various factors:

- Involvement in the planning of each increment

-
- Familiarity with the set of requirements & the users' priorities (in value terms) for them.
 - Sense of achievement as each increment is completed, and within increments, as each feature was added.
 - To a lesser extent, being freed from the perceived "drudgery" of heavier processes. In fact, it didn't feel like much was actually being dropped. But the knock-on effects of using the lighter process definitely contributed to the good feeling.

"Deliverability" & Control

At nearly all times on the project it felt like we had a working product, with clear ideas of how to proceed, and confidence that something useful would be delivered. So we did not feel out of control, or worried about whether we would actually produce something at the end.

Differences with 'traditional' methods

From my point of view - from the point of view of someone moving from "traditional" development procedures to Crystal, there are two principles that seem to differentiate Crystal Clear from the old way of working:

- 1) Minimum necessary overhead: do everything in the "lightest" way possible, consistent with achieving the results. It took a while to become comfortable with the lighter weight processes & documentation. But after a while it became quite easy to stick to, and quite liberating.
- 2) View feedback from users in a positive way. Don't view oversights or changes as problems, but as discoveries. I had to keep re-emphasising this to myself.

Sam's Comments (sponsor)

Communications

Communications were excellent: the core team was co-located in a single area. A printing whiteboard was used to record information, which was then held in a file: this proved to be a time saver. However, it would have been good to have more space to post the information for the team to view.

Viewings and reflection workshops were held each month and certainly kept the whole team up to date with the demonstrable progress of the project. However the number of iterations and viewings increased from 3 to 5, and this was excessive.

Planning/Project Management

From a project manager's viewpoint, I felt a little out of control for two reasons:

- The day to day management was delegated to the Lead Designer,
- I was not familiar with the Crystal Clear methodology.

Although not part of Crystal Clear, the project proposal promised that the scope of work could be varied to keep within the original budget. This was not achieved in practice and the project had to be curtailed at a point prior to full completion.

For future projects, the role of project manager needs to be very carefully thought out. Crystal Clear does not fit in well with existing TRT UK procedures and so some changes will be required.

People

The team worked together well, although this project was not typical, in that most of the development was performed by one person. The concept of having little documentation did not work well in the early stages because team members were on holiday or committed to other work. It is a credit to the team that the project was so successful in spite of this.

Deliverables

The software was delivered in a satisfactory working condition. The requirements document, user manual and maintenance handbook are all in draft form, because they were produced late in the project and the project curtailment prevented them from being reviewed and issued. Had the documents been produced earlier, there would have been time to review and approve them. They would also have been of use to guide the development work. In practice, an ISO 9001 audit would have caused some problems for the project.

Because the project was curtailed, the software is not yet ready for release to customers, although can be used for demonstrations.

Summary

For projects that can be clearly specified at the beginning, Crystal Clear does not appear to offer significant advantages over a *well run* lightweight project that follows standard TRT UK processes. However it is accepted that for projects such as this, with a large Human Computer Interface component, where the requirements are difficult to express clearly, Crystal Clear offers some advantages.

The use of paper documents, printing whiteboards and collocated working may not be practicable for widespread use and a better approach would be to use electronic documents as is current practice in TRT UK. The problems with team members being on leave at critical times would have been reduced by drafting earlier, more detailed documentation. However the viewings and reflection workshops were very important and should be retained as face to face meetings.

[End of report]

The Auditor's Report

[CamCal was a process experiment. The auditor analyzed what *would* need to be done in order for a Crystal Clear project to pass an ISO 9001 audit. The auditor's text follows.]

Scope

This audit is supplementary to the Management System Audit Plan and was performed at the request of the 'Lead Designer' on the CamCal elements of the ZYX project.

Crystal Clear (CC) is a software development methodology that is currently being established to enable close, collaborative working of a small software team. The method is considered 'light-weight' when compared to a full mission-critical or safety-critical development (e.g. Mil Std 498, ESA PSS-05-0, etc.) and follows a hybrid incremental/evolutionary lifecycle. This type of approach is described as 'Rapid Application Development' (RAD) within the TickIT Guide.

CC calls for close interaction between the Customer and developers and defines a number of distinct roles.

CamCal had adopted a modified version of the CC method as a conscious experiment in 'light-weight' software development.

The objective of this audit was to assess the CC design methodology as applied to CamCal against the requirements of ISO9001:2000 and the associated TickIT Guide. Inevitably, the specific implementation as applied to CamCal influenced the approach and findings of the audit.

Two checklists were prepared from the TickIT Guide. The first was against Part D 'Guidance for Auditors' and the second was against Part E 'Software Quality Management System Requirements - Standards Perspective'.

Findings

Both ISO 9001:2000 and TickIT are defined within the framework of a full QMS. Consequently, as CamCal (or any other internal project that follows the CC methodology) will operate within the TRT-UK QMS, only the following project-specific elements of ISO 9001:2000 were examined:

Planning of product realisation

ISO/TickIT require planning of processes for product realisation. Whilst CamCal has an overall Project Plan⁶⁵ (which includes the quality requirements) and Schedule, there is no detailed Project Plan for the CamCal development but Proposal document CamCal-01-003 was referenced from Project Plan.

Proposal document CamCal-01-003 had been issued and covered initial planning of CamCal. This gives brief descriptions of the method and a proposed three iteration lifecycle. One of the key features of CC is that each planned iteration (there were five planned later) includes a 'Planning Meeting' which allows the project to be dynamically re-planned. An iteration consists of three mandatory stages: planning; design, code, integrate; and reflection workshop. In addition an iteration may be followed by a Viewing attended by the user and possibly by the sponsor.

In order to be fully compliant with this clause of ISO9001 the QMS would need modification to provide detail of the working practices associated with CC to avoid the need to reproduce this detail in the PP for every RAD project. A dedicated Project Plan would then need to reference the QMS or modify the practices as necessary. The Project Plan would also need to include or reference the following (as appropriate): Customer quality requirements; Risks or risk management process; Methods, tools, standards, etc.; Statutory and regulatory requirements; Deliverables, including documents; Other documents; Verification and validation approach; Approvals, authorisations, acceptance; Roll-out; Prototyping; Release.

CC specifies a 'Risk List' which needs to be maintained. It is recommended that the normal TRT-UK risk management process is followed.

Determination of requirements related to the product

ISO/TickIT require determination of requirements related to the product. CamCal has a Sponsor Mission Statement and an evolving Requirement Document (RD) CamCal-02-003 as required by CC which contained an Actor-goal list and a number of use-cases. The Mission Statement has numbered and prioritised objectives, the RD has been evolving throughout development.

These documents are capable of satisfying the ISO/TickIT requirements.

It was suggested that in future it would be useful to have two separate documents to distinguish between Contractual Requirements (Mission Statement plus Actor-goal list) and the evolving CC Requirements Document which would contain use-cases and lower-level details and could be treated as design detail rather than formal requirements under ISO 9001.

⁶⁵ "Project plan" refers not to the project schedule, but to a plan for the *processes* used..

Review of requirements related to the product

ISO/TickIT require review of requirements related to the product. CamCal involved review and approval of the Mission Statement and ongoing review of the RD. The review of the RD was carried out with the User rather than Sponsor and there had been no formal issues (document remained at Issue 1 Draft C at end, although the project had been prematurely terminated).

In order to be fully compliant with this clause of ISO 9001, the RD should undergo a draft revision (with comments maintained in Visual SourceSafe) for each project iteration or preferably undergo an approved up-issue for each iteration. This would provide visibility of latest stable requirements for each iteration and the mandatory records of review required by ISO 9001.

If Contractual Requirements were treated separately from design detail, as suggested above, Contractual Requirements only should be subject to review at the Proposal stage as required by ISO 9001.

Customer communication

ISO/TickIT require Customer communication to take place. The involvement of the Customer (normally Sponsor and User) is central to CC through 'Viewings' and 'Reflection Workshops' during each iteration.

Consequently this aspect is well covered. CamCal held these meetings and maintained white-board records.

CC requires close co-operation with customer representatives over a long period. . Careful choice of team members can reduce the risk of poor co-operation.

Design and development planning

ISO/TickIT require planning of the design and development activities. In TRT-UK this detail would normally be contained in (or referenced from) the Project Plan and Schedule. For CC, most of the practices and proposed iterations could be in these same documents. CamCal did not have a detailed Project Plan for CamCal but did have a Proposal Document and did maintain the Schedule.

In order to be fully compliant with this clause of ISO9001, the requirements defined in 7.1 would need to be followed. The Schedule would need to reflect the latest 'Planning Meeting' (as it does on CamCal).

In CC plans may be changed at the planning meeting associated with each iteration. This should be recognised in the Project Plan, and the scope of changes that may be made without revision of the Project Plan should be defined.

Design and development inputs

ISO/TickIT require definition of the design and development inputs. CamCal placed this information into a Requirement Document (RD) CamCal-02-003 as required

by CC. The RD included most of the topics required by TickIT. Unfortunately the RD was never formally issued.

If Contractual Requirements were treated separately from design detail, as suggested above, this requirement would be met.

In common with clause 7.2.2, *to be fully compliant with this clause of ISO 9001*, the RD should undergo a draft revision (with comments maintained in VSS) for each project iteration or preferably undergo an approved up-issue for each iteration. This would provide visibility of latest stable requirements for each iteration and the mandatory records of review required by ISO 9001.

Design and development outputs

ISO/TickIT require attributes relating to the design and development outputs. CamCal did not issue all the defined outputs following early termination of the project. Unit code underwent peer review and checking through the Together suite. Acceptance Testing was not carried out because project was curtailed.

In order to be fully compliant with this clause of ISO9001 each project would need to ensure all software tools and embedded COTS had been, or were to be evaluated. Executable code would need to be verified during development and each release would need to be adequately defined and validated.

Design and development review

ISO/TickIT require review of the design and development. CC specifies 'Viewings' and 'Reflection Workshops'. CamCal carried these out and kept whiteboard records.

In order to be fully compliant with this clause of ISO 9001, whiteboard records would need to indicate who was involved with the viewings and workshops. In addition, the Sponsor and/or User should be present at some or all in order to ensure reviewer independence. Actions should be clearly identified to enable tracking at the next workshop. The whiteboard records would need to be kept for the mandatory records of review required by ISO 9001.

Design and development verification

ISO/TickIT require verification of the design and development. CC specifies verification of work products and CamCal did carry out unit test (although no records were kept) and demonstration/review by the User during 'Viewings'. Failures were also not formally recorded during test.

In order to be fully compliant with this clause of ISO 9001, records of tests performed (including test cases/stimuli) and results obtained would need to be taken for the

mandatory records required⁶⁶. Review of performance against current requirements should be carried out at planned stages.

Design and development validation

ISO/TickIT require end-to-end verification of the design and development. CC specifies test cases, test records and defect reports, CamCal did not produce these due to early termination.

In order to be fully compliant with this clause of ISO 9001, records of tests performed (including test cases/stimuli) and results obtained would need to be taken for the mandatory records required. Some form of tracing would be necessary for those requirements not tested or demonstrated⁶⁷.

Control of design and development changes

ISO/TickIT requires control of changes during design and development. Dynamic change is central to CC, which controls changes on two levels; changes to the mission or requirements would be made in the deliverable documents; changes in the design are less formally controlled through workshops and planning meetings. CamCal kept 'Actions Lists' and recorded completion at Viewings. Regression testing was used to verify the effects of changes but formal records were not kept.

In order to be fully compliant with this clause of ISO 9001, whiteboard records would need to indicate who was involved with the viewings and workshops in order to provide the mandatory records required for review of changes. As above, records of regression tests performed (including test cases/stimuli) and results obtained would also need to be taken.

Conclusions and Recommendations

Project CamCal adopted the 'Crystal Clear' rapid development method as a trial and demonstration within TRT-UK of how the process could be used. The Customer for the project was internal (hence protecting any external 'real' Customers from the experiment) but the project was unfortunately terminated early. However, due to the nature of the CC method, the early termination did not prohibit a useful software product.

This audit assesses the CC methodology against ISO 9001:2000 and the associated TickIT Guide, using CamCal as a case study. Inevitably, the detailed processes followed by the project had some bearing on the observations and findings.

CC is generally compliant with ISO 9001, particularly considering that the TickIT Guide acknowledges and accommodates 'Rapid Application Development' (RAD).

⁶⁶ Here is where automated unit- and acceptance regression tests come in handy -- Alistair

⁶⁷ This refers to usability and other requirements that don't have a function test.

Full compliance would require a few extra disciplines/records as described under 'Findings'.

It is recommended that:

- The TRT-UK QMS is enhanced to provide a generic framework for the CC methodology (preferably named something like 'Rapid Development Method' as CC is something of a trademark). The enhancements should include the ISO mandatory requirements.
- Each CC project should still create a Project Plan to adopt or modify the generic approach.
- Further in-house developments are carried out to CC, in order to fully understand the implementation and develop the new procedures.

Once established, external Customers are given the choice of development methodology, including information on the processes, potential benefits and their obligations.

Reflection on the Field and Audit Reports

First of all, I wish to thank both Stephen and Thales Research and Technology for the excellent field report, complete as it is with photos and interviews, and for permission to publish it, even in excerpted form. Speaking as someone who has read a lot of experience reports, I find it unusually clear and informative, with useful detail and candor.

I left all of the participants' questioning and worrying about Crystal Clear in place because I suspect these will be common reactions in many organizations, and it is useful for you to see them laid out in front of you.

In this section, I consider two of their worries and discuss how to deal with them in different contexts. With luck, this will help Thales in the future, and also you.

Reallocation of Power

The project manager and sponsor, Sam, mentioned several times that he felt somewhat disempowered, not having control over the task time estimates and day-to-day management. Responsibility without authority is a worrisome thing. Sam quite reasonably requests a review of the project manager's role in Crystal Clear.

In discussing this with Stephen, we discovered a number of issues and strategies surrounding this issue.

First, power really does get reallocated, in one of two directions.

The sponsor has the responsibility to provide clear priorities and preferred outcomes when direction tradeoffs need to be made. The developers have the responsibility of estimating how long the tasks will take. Together, they have joint responsibility to name all the tasks, and to generate the optimal strategy that will produce the best outcome for the project given the task list, task estimates, and project priorities.

In some companies, the project manager runs the show, naming the tasks, the estimates and the strategies. Having been in this position for years, I can say that this is a losing proposition. It is almost impossible for the project manager to correctly guess all the tasks that need to be done and their estimates. Starting from the day I first saw Jens Coldewey run a joint planning session, I realized just how impossible.

In other companies, the programmers run the show, telling the sponsor what they will do and when they might be done, never minding what the business's priorities happen to be. This is also incorrect.

In each case, someone is likely to feel displaced by the reallocation of power in Crystal Clear (and other agile methods). The key to success is the recognition that everyone is in this together, the project is a joint activity, and both groups need to provide the information they possess in order to get the best outcome.

Second, the sponsor is still not without power.

Suppose that the planning session produces a timeline that the sponsor considers unacceptable. He has three recourses:

- Remove some of the items to be delivered or extend the time allowed.
- Get more creative about the plan: offload the team members, do more in parallel, or buy packages that accomplish part of the work. Much can be done with some creative brainstorming.
- Reduce all the time estimates by a percentage. Yes, this is a drastic measure, but it is still an alternative. Imagine that the sponsor says, "All very well, but we can't live with that schedule and we can't cut any more scope. I'll give you 80% of the time listed on each card, so that we can make the final date." At this point, everyone has seen the cards, so they know the original estimates. The team can track to both the original and reduced estimates and report against both in their status charts. Visibility will be kept high. I suspect that if the sponsor feels he must resort to this alternative, then there isn't a different development methodology that will produce a better result anyway. I also suggest that the team keep brainstorming for a better strategy.

"It requires a broadly-scoped contract"

The field report concludes, "Crystal Clear expects the detailed requirements to emerge as the project proceeds, but is silent on the question of how this could fit into a contractual framework. One approach would be to adopt the DSDM practice of baselining the requirements at a high level."

CamCal was based on an early draft of Crystal Clear. I hope that I have managed in this final version to show how it can work in a contractual framework.

Crystal Clear does not *require* the detailed requirements to emerge as the project proceeds, although it *permits* them to. You are not at all obliged to change the requirements at the start of an iteration. If you have detailed requirements at the start that are not permitted to shift, then simply don't shift them!

It may be that you have committed to a scope, price and time without having had time to properly investigate the requirements and plausible architectures (this is distressingly normal in our industry). You discover part-way through the project that the original bid was simply wrong. You have several options, none of them pretty:

- Go back to the sponsors/buyers and renegotiate. This is not a matter for Crystal Clear or any other methodology to deal with. This has everything to do with how good your executive's relations are with the buyers and what they can change. They may be able to rebid the project, or they may not.
- Carefully prioritize your work to deliver the maximum business value in the time available, so that the features left out at the deadline are obvious to everyone as being the least important ones. Having delivered what is clearly

the most valuable business functionality is the best bargaining chip you can give your executives to work with⁶⁸.

- Be very creative in your strategy generation and find a way to deliver the bid.
- Get ready either to work overtime or change jobs. Even the best agile methodology can't change your social rules (but it may make you a more attractive hire).

* * *

I should round out this reflection with a final note about "Sam," the sponsor. Stephen writes, ". . . at the last reflection workshop, the sponsor also said that the advantages greatly outweighed the disadvantages. He's suggested we try it again, on a larger project, provided the fit with our existing procedures can be resolved."

Relevant to the tests for methodology design, the project delivered an adequate result for the sponsor, and the people (including the sponsor) are willing to use it again.

⁶⁸ See the discussion in *Early Victory* (p. 67), *Burn Charts* (p. 118) and *Essential Interaction Design* (p. 105).

Chapter 9

Distilled (The Short Version)

At the end, it is time to roll it all back up again: What is the core of Crystal Clear, and what are the add-on practices that get the team farther into the safety zone? This chapter is very short

Crystal Clear is a highly optimized way to use a small, colocated team, prioritizing for *safety* in delivering a satisfactory outcome, *efficiency* in development, and *habitability* of the working conventions.

The brief description of Crystal Clear for Level-3 practitioners is just this:

The lead designer and two to seven other developers
in a large room or adjacent rooms,
using information radiators such as whiteboards and flipcharts,
having easy access to expert users,
distractions kept away,
deliver running, tested, usable code to the users
every month or two (quarterly at worst),
reflecting and adjusting their working conventions periodically.

The people set in place the safety properties below using the techniques they feel appropriate. The first three properties are required in Crystal Clear; the next four get the team further into the safety zone.

- 1. Frequent Delivery**
- 2. Reflective Improvement**
- 3. Osmotic Communication**
4. Personal Safety
5. Focus
6. Easy Access to Expert Users
7. A Technical Environment with Automated Tests, Configuration Management & Frequent Integration

All the other pages in this book only expand on this page.