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*[1] 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.

[1] 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 for production use. 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, that is, 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, for *trial* (not production) usage. 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 a 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 three months. 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 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 Object-Oriented Projects* (Cockburn 1998)). 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 twenty-three of the twenty-four client-side programmers to go back to their old jobs, hired twenty-three 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 again were behind 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 encountered a catastrophe early on. 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 Chapter 3 to a straight discussion of people's values while developing.

I like to use the reflection workshop described in Chapter 3. Norm Kerth's (2001) 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.

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 quickly. 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; however, adjacent rooms with two or three people in each confers many of the benefits. Herring (2001) reported the use of high-speed intranet with Web cameras, microphones, and chat sessions to trade questions and code, to simulate the single room to (some) extent. With good technology, teams can 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 to 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"; see Figure 2-1), repeatedly shown as being very effective (Olson 2000).

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, a sofa, or, in some cases, their own meeting rooms.

Figures 2-2 and 2-3 show what one group came up with. Note that although they had fewer square feet per person than before, they ended up with longer sight lines and a conversation area with soft chairs.

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 whoever was still programming, and also to leave their design notes and plans up on the wall.

Figure 2-1 *Osmotic communication. (Courtesy of Tomax)*

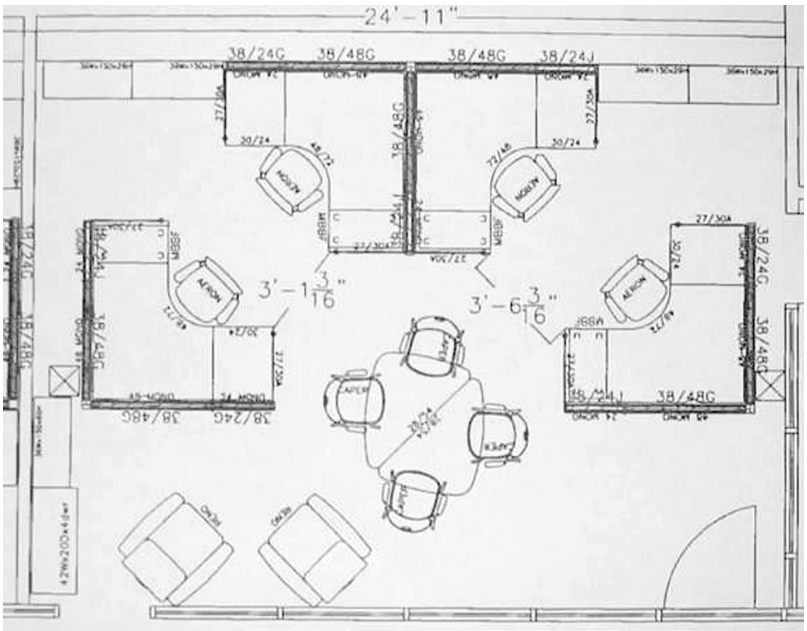

Figure 2-2 *Floor plan. (Courtesy of Schlumberger)*

Figure 2-3 *Photo of same office. (Courtesy of Schlumberger)*

Figure 2-4 *Group work room attached to shared office. (Courtesy of Schlumberger)*

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 e-mail. 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.

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 2002) contains additional information on "convection currents" of information flow within a group, osmotic communication, the value of colocation, and examples of office layout.

* * *

Figure 2-5 *Group discussion area. (Courtesy of Darin Cummins and ADP's Dealer Services)*

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 and 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* (Cockburn url eie), a special use of osmotic communication. (Andrews url) has a blog entry about creating such a seating arrangement accommodating twenty people.

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 to make progress on anything, the lead designer needs a workplace 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 6:00 P.M. to 2:00 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 2003b).

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 (Mishra 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 is using different forms of trust. It is neither realistic nor necessary to ask everyone on a project to trust each other in all forms. It is important that people can speak and act freely—they need to trust each other with respect to damaging actions and betrayal.

When there is no evidence of betrayal or damage, people 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 percent, or that you just received a tempting job offer? Can you disagree with your boss about the schedule in a team meeting? Can people end long debates about each other's 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
- 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 also that the leader and the team as a whole will act to support the person.

> One project leader[2] 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 to interpret each other's mannerisms as nonthreatening, even in heavy argument.
- 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.

* * *

---

[2] Thanks to Victoria Einarsson in Sweden.

Personal safety goes hand in hand with *amicability*, the willingness to listen with goodwill. The project suffers when any one person on the team stops listening with goodwill, 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 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.[3] 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 overamicability described in *Agile Software Development* (Cockburn 2002, p. 101).

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), Kramer (1996), Costa (2002), and Adams (2002).

---

[3] 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 fifty-person company sat down one night, prioritized the seventy pending company initiatives, and announced the results to her managers the following day. She went around to each developer individually and made sure they each knew the top two items for them.

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 twenty minutes and considerable mental energy to regain her 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 the train of thought after working on a different project.

Among the experienced project managers that I interview, the consensus is that about one and one-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 seventeen 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 convention is that once a person starts working on a project, that person 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
- Up-to-date requirements

Researchers Keil and Carmel published results showing how critical it is to have direct links to expert users (Keil 1995). 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 the time 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 semiweekly 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 a 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.

* * *

"Okay, 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" (e.g., fast and efficient 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 a section of 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).

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.

PROPERTY 7. TECHNICAL ENVIRONMENT WITH AUTOMATED TESTS, CONFIGURATION MANAGEMENT, AND 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 noncompiler 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 Control[4] 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 to 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 2003) *Configuration Management Patterns* (Berczuk 2003), 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?

[4] http://cruisecontrol.sourceforge.net/

Besides *usability tests,* which are best performed by people outside the project,[5] I find three levels of tests hotly discussed:

- *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
- *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, the programmer will not lose her concentration while they run, which means that the tests are actually run as the programmer 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 the code is checked into the configuration management system. At this point, the programmer 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.

---

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

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 2003).

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 one of the books on test-driven development (Beck 2003, Astels 2003) 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 FITnesse.[6] 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. The Mac made the idea of scriptable interfaces mainstream for a short while (Simone url) 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.

---

6 http://fit.c2.com and http://fitnesse.org respectively.

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!

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 twenty-five 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 of collaboration across organizational boundaries into his project. 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 (frequent delivery). The *La Poste* 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 and 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, and 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 a small-team project successful *should* be very similar to making any project successful, but should be 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 teams will set up osmotic communication within subteams and close communication across subteams.