1

A Cooperative Game of Invention and Communication

A fruitful way to think about software development is to consider it as a cooperative game of invention and communication.

The first section asks the question, "What would the experience of developing software be like if it were not software we were developing?" The purpose of the section is to get some distance from the subject in order to explore other ways of talking about it.

The second section reviews the broad spectrum of activities called games and finds the place of software development within that spectrum. If you are already familiar with zero-sum, positional, cooperative, finite, and infinite games, you might skim rapidly through the first part of this section. The section continues with a comparison of software development with another team-cooperative game—rock climbing—and two common comparison partners, engineering and model building.

The third section examines the idea of software development as a cooperative game of invention and communication more closely. It considers the primary goal of the game—delivering working software—and the secondary goal—or residue of the game—setting up for the next game. The next game is altering or replacing the system, or creating a neighboring system.

The final section in the chapter relates the ideas to everyday life.

A Cooperative Game of Invention and Communication

Software and Poetry

What if software development were not software development? Then what would it be, and what would the experience be like? I suggest that it is like a community writing epic poetry together. I make this comparison not because I think you have experience in community poetry writing, but because I think you don't. Your imagination will supply you with the sorts of contradictions I am interested in evoking.

Imagine 50 people getting together to write a 20,000-line epic poem on cost and time. What would you expect to find? Lots of arguments, for one thing. People trying to be creative, trying to do their best, without enough talent, time, or resources.

Who are the players in this drama? First, the people who ordered the poem. What do they want? They want something they can use to amuse themselves or impress their friends, not too expensive, and *soon*.

Next we have the key poem designers.

As you might imagine, this began as a one-person project. But our mythical poet found herself promising *much* more than she could deliver in the given time frame. So she asked a few friends to help. They designated her the lead poet and poem designer. She blocked out the theme and the poem's sequencing.

Her friends started to help, but then they ran into problems with synchronizing and communicating their work. It also turned out that they couldn't get it all done in time. So they added a couple of clerical people, more friends, and, in desperation, even neighbors. The friends and neighbors were not real poets, of course. So our lead designers blocked out sections of the poem that would not require too much talent.

What do you think happened?

There was good news: One person was good at descriptive passages, another was good at the gory bits, and another was good at passages about people. No one was good at emotion except the lead poet, who by now was pulling her hair out because she didn't have time to write *poetry*, she was so busy coordinating, checking, and delegating.

Actually, a couple of people couldn't leave well enough alone. Two of them wrote pages and pages and pages of material describing minor protagonists, and our lead poet could not get them to cut it down to size. Another few kept rewriting and revising their work, never satisfied with the result. She wanted them to move on to other passages, but they just wouldn't stop fiddling with their first sections.

As time progressed, the group got desperate and added more people. The trouble was that they were running out of money and couldn't really afford all these people. Communications were horrible, no one had the current copy of the poem, and no one knew the actual state of the poem.

Let's give this story a happy ending . . .

As luck would have it, they engaged a wonderfully efficient administrator who arranged for a plan of the overall poem, an inventory of each person's skills, a time frame and communication schedule for each part, standards for versioning and merging pieces of the poem, plus secretarial and other technical services.

They delivered the poem to satisfied clients, well over budget, of course. And the lead poet had to go on vacation to restore her senses. She swore she would never do this again (but we know better).

Groups surely have gotten together to write a long poem together. And I am sure that they ran into most of the issues that software developers run into: temperamental geniuses and average workers, hard requirements, and communication pressures. Humans working together, building something they don't quite understand. Done well, the result is breathtaking; done poorly, dross.

Balance in Software Design

As I sat in on a design review of an object-oriented system, one of the reviewers suggested an alternate design approach.

The lead designer replied that the alternative would not be as *balanced*, would not *flow* as well as the original.

Thus, even in hard-core programming circles, we find designers discussing designs in terms of balance and flow.

Software developers have a greater burden than our hypothetical poets have: logic. The result must not only rhyme; it must behave properly—"accurately enough," if not correctly.

The point is that although programming is a solitary, inspiration-based, logical activity, it is also a group engineering activity. It is paradoxical, because it is not the case, and at the same time it is very much the case, that software development is:

- Mathematical, as C. A. R. Hoare has often said
- Engineering, as Bertrand Meyer has often said
- A craft, as many programmers say
- A mystical act of creation, as some programmers claim

Its creation is sensitive to tools; its quality is independent of tools. Some software qualifies as beautiful, some as junk. It is a meeting of opposites and of multiple sets of opposites.

It is an activity of cognition and expression done by communicating, thinking people who are working against economic boundaries, conditional to their cultures, sensitive to the particular individuals involved.

Software and Games

Games are not just for children, although children play games. Games are invented and used by many people, including novelists, mathematicians, and corporate strategists.

Kinds of Games

If you are sitting around the living room on a winter's evening and someone says, "Let's play a game," what could you play?

You could play charades (play-acting to uncover a hidden phrase). You could play tic-tac-toe or checkers, poker or bridge. You could play hide-and-seek or table tennis. You could play "When I took a trip, . . ." a game in which each person adds a sentence onto a story that grows in the telling. You could, especially if you have younger children, end up having a wrestling match on the living room floor.

Games fall into many categories: zero-sum, non-zero-sum, positional, competitive, cooperative, finite, and infinite, to name a few (see Figure 1-1). As a way to help identify what kind of game software development could be, let's look at those choices.

Zero-sum games are those with two sides playing in opposition, so that if one side wins, the other loses. Checkers, tic-tac-toe, bridge, and tennis are examples. Software development is clearly not a zero-sum game.

Non-zero-sum games are those with multiple winners or multiple losers. Many of the games you would consider playing on that winter's evening are non-zero-sum games: poker, parcheesi, and hide-and-seek. Software development is also a non-zero-sum game.

Positional games are those in which the entire state of the game can be discovered by looking at the markers on the board at that moment. Chess and tic-tac-toe are examples. Bridge isn't, because the played cards don't show which person played them.

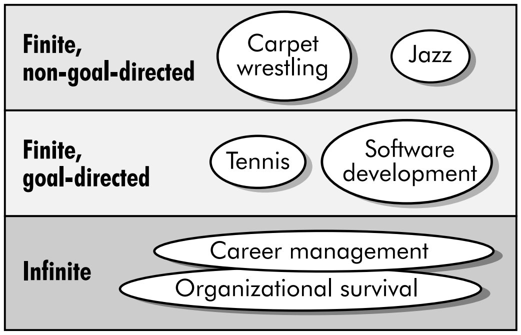

Figure 1-1 Different categories of games.

Some people try to play software development as a positional game, requiring that the documentation reflect the history *and* current state of the project. They intend that, should anyone leave the project, a replacement person will be able to join the team, read the documentation, and pick up where the other person left off. We will come to see that this is not an effective gaming strategy for software development.

(Positional games are actually far more interesting than the simple description here implies. John Conway, in his book *On Numbers and Games* (1976), was able to show how two-person, positional games form a superset of *all* numbers: real, imaginary, finite, and transfinite. He constructs the notion of *number* directly from two-person, positional games.)

All of these are *competitive* games, in which there is a clear notion of winning and losing.

In *cooperative games*, the people work either to win together or to continue the game as long as they consider it worth playing. The former are *goal-seeking* cooperative games, the latter *non-goal-seeking* cooperative games. Storytelling, playing jazz, and carpet wrestling are non-goal-seeking cooperative games. In these latter games, the players do not seek to end the game by reaching a goal as fast as possible. They come to an end only when enough people get tired of playing and step out.

Charades, rock climbing, and software development are goal-seeking cooperative games (see Figure 1-1 again).

All of these are *finite* games—games intended to end. *Infinite* games are those in which the players' primary intention is to keep the game going. Organizations, corporations, and countries play these. Their core purpose is to stay in existence.

A person's profession is also an infinite game. The person, wanting to continue the profession, makes a set of moves that permit her practice of that profession to continue.

Often, a person or company aims to play well on a particular project in order to get the best position on the next game. As with the card game appropriately named "So long, sucker," these sorts of teams and alliances change continually and without notice.

Software and Rock Climbing

Of all the comparison partners for software development that I have seen, rock climbing has emerged as the best. It is useful to have such a comparison partner, to get some distance from the subject and open a vocabulary that we can reapply to software development. Rock climbing is not a metaphor for software development but a comparison partner, another member of the same class of games.

Let's see how some of the words and phrases associated with rock climbing relate to software development.

Cooperative and goal-seeking. A team of rock climbers work together to reach the top. They will evaluate the climb based on how well they climbed together and how much they enjoyed themselves, but the first measure of success is whether they reached the top. Reaching the endpoint is a primary goal, and the game is over when they reach the top.

(If you are a rock climber, you might well interrupt me here. For many rock climbers, the moment of reaching the end of the climb is a sad one, for it signals the end of the game. That is true of cooperative games in general. The game comes to an end when the endpoint is reached, but if the players have been enjoying themselves, they may not want to stop. Similarly, sometimes software developers do not want to finish their design, because then the fun part of their work will be over.)

Load bearing. The climbers must actually support their weight on their hands and feet. This is a particularly valuable point of comparison between the two: Software must run and produce reasonable responses. While multiple solutions are possible, not just any solution will do.

Team. Climbing is usually done in teams. There are solo climbers, but under normal circumstances, climbers form a team for the purpose of a climb.

Individuals with talent. Some people just naturally climb better than others do. Some people will never handle certain climbs.

Skill-sensitive. The rock climber must have a certain proficiency. The novice can approach only simple climbs. With practice, the climber can attack more and more difficult climbs.

Training. Rock climbers are continually training on techniques to use.

Tools. Tools are a requirement for serious rock climbing: chalk, chucks, harness, rope, carabiner, and so on. It is important to be able to reach for the right tool at the right moment. It is possible to climb very small distances with no tools. The longer the climb, however, the more critical the tool selection is.

Resource-limited. A climb usually needs to be completed by nightfall or before the weather changes. Climbers plan their climbs to fit their time and energy budget.

Plan. Whether bouldering, doing a single-rope climb, or doing a multiple-day climb, the climbers always make a plan. The longer the climb, the more extensive the plan must be, even though the team knows that the plan will be insufficient and even wrong in places.

Improvised. Unforeseen, unforeseeable, and purely chance obstacles are certain to show up on even the most meticulously planned climbing expeditions unless the climb is short and the people have already done it several times before. Therefore, the climbers must be prepared to change their plans—to improvise—at a moment's notice.

Fun. Climbers climb because it is fun. Climbers experience a sense of *flow* (Csikszentmihalyi 1991) while climbing, and this total occupation is part of what makes it fun. Similarly, programmers typically enjoy their work, and part of that enjoyment is getting into the flow of designing or programming. Flow in the case of rock climbing is both physical and mental. Flow in the case of programming is purely mental.

Challenging. Climbers climb because there is a challenge: Can they really make it to the top? Programmers often crave this challenge, too. If programmers do not find their assignment challenging, they may quit or start embellishing the system with design elements they find challenging (rather like some of the poets mentioned in the epic poetry project).

Dangerous. Probably the one aspect of rock climbing that does not transfer to software development is danger. If you take a bad fall, you can die. Rock climbers are fond of saying that climbing with proper care is less dangerous than driving a car. However, I have not heard programmers express the need to compare the danger of programming with the danger of driving a car.

Software development has been compared with many other things, including math, science, engineering, theater, bridge building, and law. Although one can gain insight from looking at any of those activities, the rock-climbing comparison is the most useful for the purpose of understanding the factors involved in the activity.

A Game of Invention and Communication

We have seen that software development is a *group* game, which is *goal seeking*, *finite*, and *cooperative*. The team, which consists of the sponsor, the manager, usage specialists, domain specialists, designers, testers, and writers, works together with the goal of producing a working and useful system. In most cases, team members aim to produce the system as quickly as possible, but they may prefer to focus on ease of use, cost, defect freedom, or liability protection.

The game is finite because it is over when the goal is reached. Sometimes delivery of the system marks the termination point; sometimes the end comes a bit later. Funding for development usually changes around the time the system is delivered, and new funding defines a new game. The next game may be to improve the system, to replace the system, to build an entirely different system, or possibly to disband the group.

The game is cooperative because the people on the team help each other to reach the goal. The measure of their quality as a team is how well they cooperate and communicate during the game. This measure is used because it affects how well they reach the goal.

If it is a *goal-directed cooperative game*, what does the game consist of? What constitutes moves in the game?

The task facing the developers is this: They are working on a problem that they don't fully understand, one that lives in emotions, wishes, and thoughts and that changes as they proceed. They need to

- Understand the problem space
- Imagine some mechanism that solves the problem in a viable technology space
- Express that mental construct in an executable language, which lacks many features of expression, to a system that is unforgiving of mistakes

To work through this situation, they

- Use props and devices to pull thoughts out of themselves or to generate new ideas that might help them understand the problem or construct a solution
- Leave trails of markers for those who will come later, markers to monitor and test their progress and their understanding, and they use those markers again, themselves, when they revisit parts of their work

Software development is therefore a cooperative game *of invention and communication*. There is nothing in the game but people's ideas and the communication of those ideas to their colleagues and to the computer.

Looking back at the literature of our field, we see a few people who have articulated this before. Peter Naur did, in his

1985 article "Programming as Theory Building," and Pelle Ehn did, in "Scandinavian Design: On Participation and Skill" (1992) and in his magnificent but out-of-print book *Work-Oriented Design of Software Artifacts* (1988). Naur and Ehn did this so well that I include those two articles in near entirety in Appendix B. Robert Glass and colleagues wrote about it in "Software Tasks: Intellectual or Clerical?" (1992), and Fred Brooks saw it as such a wickedly hard assignment that he wrote the article "No Silver Bullet" (1995).

The potential consequences of this cooperative game of invention and communication are outlined in the remainder of this chapter. The remainder of the book examines those consequences.

Software and Engineering

Considering software development as a game with moves is profitable, because doing so gives us a way to make meaningful and advantageous decisions on a project. In contrast, speaking of software development as *engineering* or *model building* does not help us make such advantageous decisions.

The trouble with using engineering as a reference is that we, as a community, don't know what that means. Without having a common understanding of what engineering is, it is hard to get people to work "more like engineering." In my travels, people mostly use the word *engineering* to create a sense of guilt for not having done enough of something, without being clear what that something is.

The dictionary is clear as to what "engineering" is: "*The application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people*" (*Merriam-Webster's Collegiate Dictionary*, Eleventh Edition, 2003).

That definition does not explain what *doing* engineering is about. In my experience, "doing engineering" involves creating a trade-off solution in the face of conflicting demands. Another person, though, wrote to me and said, "A basic concept of engineering is to address problems in a repeatable and consistent manner." This confusing the *act* of doing engineering work with the *outcome* of doing engineering work is a common mistake.

The *outcome* of doing engineering work is the factory, which is run while specific people watch carefully for variations in quantity and quality of the items being manufactured.

The *act* of doing engineering work is the ill-defined creative process the industrial engineer goes through to invent the manufacturing plant design. That process is not run with statistical controls, measuring quantity and quality of output. Like software development, it runs as a cooperative game of invention and communication, with individual people of different backgrounds huddling together to come up with a workable design.

When people say, "Make software development more like engineering," they often mean, "Make it more like running a plant, with statistical quality controls." But as we have seen, running the plant is not the act of doing engineering.

The other aspect of "doing engineering" is looking up previous solutions in code books.

Civil engineers who design bridges are not supposed to invent new structures. Given a river and a predicted traffic load, they are supposed to take soil samples and use the code books to look for the simplest structure that handles the required load over the given distance, building on the soil at hand. They base their work on centuries of tabulation of known solutions.

This only marginally fits the current state of software development. We are still in the stage where each team's design is supposed to be better than the neighbor's, and the technologies are changing so fast that few code books exist. As time goes by, more of these code books will be available. Today, however, there are still more variations between systems than there are commonalities.

Let's return to considering "engineering" to mean "thinking and making trade-offs." These are appropriate phrases. We would like our software developers to think, and to understand the trade-offs they select. However, these phrases do not provide guidance in running projects.

Software and Model Building

Many people have advocated model building over the last decade, including Ivar Jacobson, who declared, "Software development *is* model building."

Characterizing software development as engineering may not provide much guidance for running projects, but characterizing it as *model building* leads directly to inappropriate project decisions.

If software development were model building, then a valid measure of the quality of the software or of the development process would be the quality of the models, their fidelity to the real world, and their completeness. However, as dozens of successful project teams around the world have told me:

> "The interesting part of what we want to express doesn't get captured in those models. The interesting part is what we say to each other while drawing on the board.
>
> "We don't have time to create fancy or complete models. Often, we don't have time to create models at all."

Where I found people diligently creating models, software was not getting delivered. Paying attention to the models interfered with developing the software.

Constructing models is not the purpose of the project. Constructing a model is only interesting as it helps win the game.

The purpose of the game is to deliver software. Any other activity is secondary. A model, as any communication, is *sufficient, as soon as* it permits the next person to move on with her work.

The work products of the team should be measured for *sufficiency with respect to communicating with the target group*. It does not matter if the models are incomplete, drawn with incorrect syntax, and actually not like the real world *if* they communicate sufficiently to the recipients.

As Jim Sawyer so colorfully wrote in an e-mail discussion about use cases (Cockburn 2001c):

> ". . . as long as the templates don't feel so formal that you get lost in a recursive descent that wormholes its way into design space. If that starts to occur, I say strip the little buggers naked and start telling stories and scrawling on napkins."

The *effect* of the communication is more important than the form of the communication.

Some successful project teams have built more and fancier models than some unsuccessful teams. From this, many people draw the conclusion that more modeling is better.

Some successful teams have built fewer and sloppier models than some unsuccessful teams. From this, other people draw the conclusion that less modeling is better.

Neither is a valid conclusion. Modeling serves as part of the team communication. There can be both too much and too little modeling. Scrawling on napkins is sufficient at times; much more detail is needed at other times.

Understanding how much modeling to do, and when, is the subject of this book. Thinking of software development as a cooperative game that has primary and secondary goals helps you develop insight about how elaborate a model to build or whether to build a model at all.

A Second Look at the Cooperative Game

The Cooperative Game Principle

Software development is a (resource-limited) cooperative game of invention and communication. The primary goal of the game is to deliver useful, working software. The secondary goal, the residue of the game, is to set up for the next game. The next game may be to alter or replace the system or to create a neighboring system.

Programmers as Communications Specialists

Saying that "software development is a cooperative game of invention and communication" suddenly shines a very different light on the people in our field.

Programmers are typically stereotyped as noncommunicative individuals who like to sit in darkened rooms alone with their computer screens.

This is not a true stereotype, though. Programmers just like to communicate about things *they* like to communicate about, usually the programs they are involved in. Programmers enjoy trading notes about XML-RPC or the difficulties in mapping object-oriented designs to relational databases. They just don't like joining in the chitchat about things they consider irrelevant.

There has been a surprisingly high acceptance of programming in pairs, a technique in which two people sit together and co-write their program (Beck 2000). I say "surprising" because many programmers first predict that they won't be able to work that way and then find they actually prefer it, after trying it for a week or two (Cockburn, 2001b).

As far as the stereotype is true, it accents the "invention" portion of the cooperative game. Programming has, until recently, been more focused as a game of invention than as a game of communication. The interest of programmers to discuss programming matters with each other gets in the way of discussing business matters with sponsors, users, and business experts.

We can attribute part of the cause for this to our educational curricula. Imagine some people thumbing through a university's curriculum guide. They see two tracks. One calls for a lot of reading, writing, and speaking, and some programming. The other calls for less reading, writing, and speaking, and more of working alone, building artifacts. We can easily imagine the verbally oriented people selecting the first curriculum and the less verbally oriented people selecting the second.

Historically, success in our profession came from being able to sit alone for long hours without talking to anyone, staring at papers or screens. Those who didn't like that mode of work simply left the field. Newer, and particularly the "agile" methodologies, emphasize communication more. Suddenly the people who elected to join a profession that did not require much interpersonal communication are being asked to become good at it.

Only the universities can reverse the general characteristics by creating software development curricula that contain more communication-intensive courses.

At the University of Aalborg, in Denmark, a new Informatics major was defined that involves both software design and communication skill (Mathiassen 1999). The department head, Lars Mathiassen, reports that the difference in people's personalities is noticeable: The new curriculum attracts those who are willing to accept the communications load, and the old curriculum attracts those who have less interest in communication.

To the extent that software development really *is* a game of invention and communication, we will have to see a greater emphasis on communication in the university curricula.

Gaming Faster

We should not expect orders of magnitude improvement in program production.

As much as programming languages may improve, programming will still be limited by our ability to think through the problem and the solution, working through the details of how the described solution deals with the myriad cases it will encounter. This is Naur's "programming as theory building" (Appendix B).

To understand why exponential productivity growth is not an appropriate expectation, we need only look at two other fields of thought expression: writing novels and writing laws. Imagine being worried that lawyers are not getting exponentially faster at creating contracts and laws!

In other words, we can expect the game of invention and the business of communicating those intentions to a computer to remain difficult.

Markers and Props

Intermediate work products help with Naur's "theory building" and Ehn's "language games," as *reminders for our reflection*. They provide shared experiences to refer to or serve as support structures for new ideas.

The former need only be complete enough to *remind* a person of an earlier thought or decision. Different markers are appropriate for different people with different backgrounds.

The latter act as props to incite new thoughts.

Laser Printer Mock-Ups

Ehn's team considered introducing laser printers to a group that had no experience with them, back in 1982. They constructed cardboard mock-ups, not to remind the participants of what they already knew, but to allow them to invent themselves into the future by creating an inexpensive and temporary future reality to visualize.

These mock-ups are not second-class items, used only due to some accidental absence of technology. Rather, they are a fundamental technique used to help people construct thoughts about new situations. Any work product that helps the group invent a way forward in the game is appropriate. Whether they keep the mock-up around as a reminder of the discussion is up to them in the playing of their game.

Diminishing Returns

Because the typical software development project is limited in time, people, and money, spending extra of those resources to make an intermediate work product better than it needs to be for its purpose is wasteful. One colleague expressed it this way:

Diminishing Returns

It is clear to me as I start creating use cases, object models, and the like, that the work is doing some good. But at some point, it stops being useful and starts being both drudgery and a waste of effort. I can't detect when that point is crossed, and I have never heard it discussed. It is frustrating, because it turns a useful activity into a wasteful activity.

The purpose of each activity is to move the game forward. Work products of every sort are sufficiently good as soon as they permit the next move.

Knowing this permits a person to more easily detect the crossover from value adding to diminishing returns, to hit the point of being *sufficient-to-purpose*. That point has been nicknamed "satisficing" (Simon 1987, Bach, www.satisfice.com).

Sufficiency for the Primary Goal

Intermediate work products are not important as models of reality, nor do they have intrinsic value. They have value only as they help the team make a move in the game. Thus, there is no idea to measure intermediate work products for completeness or perfection. An intermediate work product is to be measured for *sufficiency*: Is it sufficient to remind or inspire the involved group?

These three short stories illustrate how quickly sufficiency can be reached:

Sufficiency in a Meeting

On a project called "Winifred" (Cockburn, 1998), I was asked partway through the project to review, for the approximately 40 people on the project, the process we were following and to show samples of the work products. The meeting would be held in the cafeteria.

I copied onto overhead transparencies a sample of each work product: a use case, a sequence chart, a class diagram, a screen definition, a fragment of Smalltalk code, and so on.

As luck would have it, the overhead projector bulb blew out just before my little presentation. As I was wearing a white shirt that day, I asked everyone to move closer and held up the sample use case in front of my shirt.

"I can't read it!" someone called out, not too surprisingly, from the back.

"You don't need to read it," I said. (The group laughed.) "All you need to see is that a use case is paragraphs of text, approximately like this. There are lots of them online for you to look at. We write them as requirements..." and I described who was writing them, who was reading them, and how they were being used.

I held a sample class diagram in front of my shirt.

"I can't read it!" someone called out again.

"You don't need to read it." (The group laughed again.) "All you need to see is that it is a diagram with boxes and lines. It is written by ..." and I discussed the role of the class diagram in the project.

I went through the work products this way. In each case, all that the group needed was a visual image of what one of these things looked liked, who wrote it, who read it, and how it served the project. Real examples were all online and could be examined by anyone on the project.

This was communication *sufficient* to the purpose that people could have a visual memory of what each product looked like, to anchor the sentences about how they were used.

We did have a drawing showing the process we were following, but as far as I know, nobody other than the project managers and I ever looked at it.

Sufficiency of Work Products

Project "Winifred" was a fixed-time, fixed-price project costing about $15 million, lasting 18 months, with 24 programmers among 45 people total. We ran it with the cooperative game principle in mind (the principle hadn't been defined back then, but we knew what we wanted), with as much close, informal communication as possible.

At the time, use cases weren't very well defined, so the writers wrote just a few paragraphs of simple prose describing what was supposed to take place, and some of the business rules involved.

The analyst responsible for a use case usually went straight from each meeting with the end users to visit the designer-programmers, telling them the outcome of the meeting. The designer-programmers put their new knowledge directly into their programs, based on the verbal description.

This worked effectively, because the time delay from the analyst's hearing the information in the meeting to the programmer's knowing of its effect on the program was just a matter of hours.

There was an odd side effect, however. Halfway through the project, one of the programming leads commented that he didn't know what purpose the use cases were supposed to serve. They certainly weren't requirements, he said, because he had never read them.

The point of the story is that the casual use cases were "sufficient to the task" of holding the requirements in place. The communication channels and the shared understanding between the writers and readers was rich enough to carry the information.

Chrysler's Ultralight Sufficiency

Chrysler's Comprehensive Compensation project (C3 1998) ran even lighter than project Winifred. The 10 programmers sat together in a single, enormous room, and the team tracker and three customers (requirements experts) sat in the next room, with no door between them.

With excellent intra-team communications and requirements available continuously, the group wrote even less-than-casual use cases. They wrote a few sentences on an index card for each needed system behavior. They called these "user stories."

When it came time to start on a user story, the programmers involved asked the customer to explain what was needed and then designed that. Whenever they needed more information, they asked the nearby customer to explain. The requirements lived in the discussion between the participants and were archived in the acceptance and unit test suites.

The design documentation also lived in a mostly oral tradition within the group. The designers invented new designs using CRC card sessions (Wilkinson 1995). In a CRC-card design session, the designers write the names of potential classes on index cards and then move them around to illustrate the system performing its tasks. The cards serve both to incite new thoughts and to hold in place the discussion so far. CRC cards are easy to construct, to put aside, and to bring back into play, and are thus perfectly suited for an evolving game of invention and communication.

After sketching out a possible design with the cards, the designers moved to the workstations and wrote a program matching the design, delivering a small bit of system function.

The design was never written down. It lived in the cards, in memories of the conversations surrounding the cards, in the unit tests written to capture the detailed requirements, in the code, and in the shared memories of the people who had worked together on a rotating basis during the design's development.

This was a group highly attuned to the cooperative game principle. Their intermediate work products, while radically minimalist, were quite evidently *sufficient* to the task of developing the software.

The team delivered a new function every three weeks over a three-year period.

Sufficiency in the Residue

Thus far, the topic of discussion has been the *primary* goal of the game: delivering working software. However, the entire project is just one move within a larger game. The project has two goals: to deliver the software and to create an advantageous position for the next game, which is either altering or replacing the system or creating a neighboring system.

If the team fails to meet the primary goal, there may be no next game, so that goal must be protected first. If the team reaches the primary goal but does a poor job of setting up for the next game, they jeopardize that game.

In most cases, therefore, the teams should create some markers to inform the next team about the system's requirements and design. In keeping with Naur's programming as theory building and the cooperative game principle, these markers should be constructed to get the next team of people *reasonably close* to the thinking of the team members who completed the previous system. Everything about language games, touching into shared experience, and sufficiency-to-purpose still applies.

The compelling question now becomes this: When does the team construct these additional work products?

One naive answer is to say, "As the work products are created." Another is to say, "At the very end." Neither is optimal. If the requirements or designs change frequently, then it costs a great deal to constantly regenerate them—often, the cost is high enough to jeopardize the project itself. On the other hand, if constructing markers for the future is left to the very end of the project, there is great danger that they will never get created at all. Here are two project stories that illustrate this point:

Continuous Redocumentation

Project "Reel" involved 150 people. The sponsors, very worried about the system's documentation becoming out of date and inaccurate, mandated that whenever any part of the requirements, design, or code changed, all documentation that the change affected had to be immediately brought up to date.

The result was as you might expect. The project crawled forward at an impossibly slow rate, because the team members spent most of their time updating documentation for each change made.

The project was soon canceled.

This project's sponsors did not pay proper attention to the economic side of system development, and they lost the game.

Just Never Documentation

The sponsors of the Chrysler Comprehensive Compensation project eventually halted funding for the project. As the people left the development team, they left no archived documentation of their requirements and design other than the two-sentence user stories, the tests, and the program source code.

Eventually, enough people left that the oral tradition and group memory were lost.

This team masterfully understood the cooperative game principle during system construction but missed the point of setting up the residue for the following game.

Deciding on the residue is a question that the project team cannot avoid. The team must ask and answer both of these questions:

- How do we complete this project in a timely way?
- When do we construct what sorts of markers for the next team?

Some people choose to spend more money, earlier, to create a safety buffer around the secondary goal. Others play a game of brinksmanship, aiming to reach the primary goal faster and creating as little residue as possible, as late as possible.

In constructing responses, the team must consider the complexity of both the problem and the solution, the type of people who will work on it next, and so on. Team members should balance the cost of overspending for future utility against the risk of underdocumenting for the future. Finding the balance between the two is something of an art and is the proper subject of this book.

A Game within a Game

Although any one project is a cooperative and finite game, the players are busy playing competitive and infinite games at the same time.

Each team member is playing an infinite game called *career*. These individuals may take actions that are damaging to the project-as-game but that they view as advantageous to their respective careers.

Similarly, the company is playing an infinite game: its growth. To the company, the entire project is a single move within that larger game. In certain competitive situations, a company's directors may deliberately hinder or sabotage a project in order to hurt a competitor or in some other way create a better future situation for the company.

Watching military subcontracting projects, it sometimes seems that the companies spend more time and money jockeying for position than developing the software. Thinking about any one project in isolation, this doesn't seem to be sensible behavior. If we consider the larger set of competitive, infinite games the companies are playing, though, then the players' behavior suddenly makes more sense. They use any one project as a playing board on which to build their position for the next segment of the game.

The cooperative game concept does not imply that competitive and infinite games don't exist. Rather, it provides words to describe what is happening across the games.

Open-Source Development

Finally, consider open-source projects. They are grounded in a different economic structure than commercial projects: They do not presume to be resource-limited.

An open-source project runs for as long as it runs, using whatever people happen to join in. It is not money-limited, because the people do not get paid for participating. It is not people-resource-limited, because anyone who shows up can play. It is not time-limited, because it is not run on a schedule. It just takes as long as it takes.

The moves that are appropriate in a game that is not resource-limited are quite naturally different from those in a resource-limited game. The reward structure is also different. Thus, it is to be expected that an open-source project will use a different set of moves to get through the game. The creation of the software, though, is still cooperative and is still a game of invention and communication.

One may argue that open-source development is not really *goal seeking*. Linus Torvalds did not wake up one day and say, "Let's finish rewriting this UNIX operating system so we can all go out and get some real jobs." He did it first because it was fun (Torvalds 2001) and then to "make this thing somewhat better." In other words, it was more like kids carpet wrestling or musicians playing music than rock climbers striving to reach the top.

While that is true to some degree, it is still goal-directed in that a person working on a section of the system works to get it to "the next usable state." The people involved in that section of the system still work the cooperative game of invention and communication to reach that goal.

What Should This Mean to Me?

As you practice this new vocabulary on your current project, you should start to see new ways of finishing the job in a timely manner while protecting your interests for future projects. Here are some ideas for becoming more comfortable with the ideas in this chapter:

Read "Programming as Theory Building" in Appendix B. Then, watch

- The people on the design team build their theories
- The people doing last-minute debugging, or program maintenance, build their theories
- The difference in the information available to the latter compared with the former
- How their different theories result in different programs being produced
- How your understanding of your problem changes over time and how that changes your understanding of the solution you are building

Look around your project, viewing it as a resource-limited cooperative game of invention and communication. Ask

- Who are the players in this game?
- Which are playing a finite, goal-directed team game?
- Which are playing their own infinite game instead?
- When are my teammates inventing together, and when they are laying down tracks to help others get to where they are? Track this carefully for five consecutive workdays to see them move from one to the other.

View the project decisions as "moves" in a game. Imagine it as a different sort of game, crossing a swamp:

- Recall the project setup activities as a preliminary plan of assault on the swamp, one that will change as new information emerges about the characteristics of the swamp and the capabilities of the team members.
- Watch as each person contributes to detecting deep or safe spots and builds a map or walkway for other people to cross.

Reconsider the work products your team is producing:

- Who is going to read each?
- Is the work product more detailed than needed for that person, or is it not detailed enough?
- What is the smallest set of internal work products your team needs to reach the primary goal?
- What is the smallest set of final work products your team needs to produce to protect the next team?
- Notice the difference between the two.

Consider running the project as two separate subprojects:

- The first subproject produces the running software in as economic a fashion as possible.
- The second subproject, competing for key resources with the first, produces the final work products for the next team.

Think about developing a large, life-critical, mission-critical system:

- Will that project benefit more from increasing the invention and communication or from isolating the people?
- Notice which sorts of projects need more final work products as their residue and which need fewer work products.

Finally, notice the larger game within which the project resides. Notice

- The distractions on your project, such as giving demos to visitors, taking the system to trade shows, and hitting key deadlines
- How those "distractions" contribute to the larger game in play
- That moves in the larger game jeopardize your local game
- How you would balance moves in the project-delivery game against the moves in the larger game

The point of all this watching and reconsidering is to sharpen your sense of "team," "cooperative game," "moves in a game," "invention and communication," "theory building," and "sufficiency."

After watching software development for awhile, reexamine the engineering activities around you:

- Identify where they too are cooperative games of invention and communication and where they are more a matter of looking up previous solutions in code books.

When you have achieved some facility at viewing the life around you as a set of games in motion, practice

- Adding discipline on your project at key places
- Reducing discipline at key places
- Declaring, "Enough! This is sufficient!"