

The End of Software Engineering and the Start of Economic-Cooperative Gaming

Alistair Cockburn, *Humans and Technology*, arc@acm.org,
HaT TR 2004.02, Jan 14, 2004

Paper submitted to the *ComSIS Journal*, *Computer Science and Information System*
(<http://www.comsis.fon.bg.ac.yu>)

Abstract

"Software engineering" was introduced as a model for the field of software development in 1968. This paper reconsiders that model in the light of four decades of experience, and finds it lacking in its ability to explain project success and failures, predict important issues in running projects, and help practitioners formulate effective strategies on the fly.

An alternative underlying model for software development is described: Software development is a series of goal-directed, resource-limited, cooperative games of invention and communication. The primary goal of each game is the production and deployment of a software system; the residue of the game is a set of markers to assist the players of the next game. People use markers and props to remind, inspire and inform each other in getting to the next move in the game. The next game is an alteration of the system or the creation of a neighboring system. Each game therefore has as a secondary goal to create an advantageous position for the next game. Since each game is resource-limited, the primary and secondary goals compete for resources.

The cooperative-game model provides the benefits that the software engineering model misses: It raises to the proper priority level issues crucial to successful software projects; it explains how, historically, teams with messy-looking processes sometimes outperform others with tidier processes; and it helps busy practitioners decide how to respond to unexpected situations. Finally, it is seen that much of engineering in the general belongs in the category of resource-limited, cooperative games.

I. Introduction

Software development is not "naturally" a branch of engineering. It was proposed as being a branch of engineering in 1968 as a deliberate provocation intended to stir people to new action [Naur-Randell]. As a provocation, it succeeded. As a means for providing sound advice to busy practitioners, however, it cannot be considered a success. After 35 years of using this model, our field lacks a notable project success rate [Standish], we do not find a correlation between project success and use of tidy "engineering" development processes [Cockburn 2003a], and we do not find practitioners able to derive practical advice to pressing problems on live projects.

One might defend the software engineering model by asserting that this is due to our not yet doing enough "software engineering." However, project debriefings show that some very messy-looking projects succeed quite nicely while many process-oriented projects fail quite badly. Opposites of both occur as well, making it difficult to see what does correlate with project success. Whatever does correlate with project success, it does not appear to be tidiness of the process, or the amount of "engineering" used [Cockburn 2000a, 2003a].

The term "software engineering" fails a crucial test, that of suggesting good actions to the busy practitioner. If told to "do more *software-engineering*" on their current projects, would project managers, executives, programmers and testers

- produce similar interpretations of this phrase?
- be correct with respect to what the term *software engineering* really calls for?
- produce good advice for the project at hand?

When I ask people what it would mean for them to "do more *software-engineering*," they usually return a blank look. The answer, when it comes, usually involves their doing more intense modeling of the system to establish *a priori* its correctness and verisimilitude (match to the real world), spending more time on estimation of cost and time, and in general looking to construct a software equivalent to mass-production manufacturing facilities. As discussed below, these are not actually what is called for in *engineering* at all, and worst of all, they are often not good strategies for leading the project to success.

The failure of the software engineering model leaves our field needing an underlying model that

- explains why successful projects succeed and failing projects fail,
- intrinsically names topics known to be important to project success, and
- leads the person on a live project to derive sensible advice as to how to proceed.

In this paper, I describe a new model for software development: A series of resource-limited, goal-directed, cooperative games of invention and communication. The new model explains historical data, can be converted intuitively by people on projects into meaningful strategies, identifies a lexicon of terms that correlate well to project success and failure, and points to future topics of research for our field as well as pointing to what results to borrow from other fields.

The primary use of the new model is in creating strategies for managing software development. It creates space for, without completing, an adjunct model that should cover the thought processes of the designer-programmer while creating and manipulating the design and expression of the program. This adjunct model, which may derive from craftsmanship [McBreen], will have a natural fit into the framework created by the economic-cooperative game model.

The paper is structured in five sections:

Section II, *Cooperative Games*, breaks apart the terms in the cooperative game model, showing their relation to software development projects. The new model is presented first in order to highlight concepts that are important to have in view when reviewing the historical record.

Section III, *Software Engineering*, examines the origin of the term software engineering, discusses the failure of the term to either correlate to project success or to offer good advice to busy practitioners, and presents project experience reports that are anomalous to the software engineering lexicon but natural in the cooperative game lexicon.

Section IV, *Engineering in Action*, reexamines engineering itself, showing that if the term *engineering* were properly interpreted, then the term *software engineering* would have a very different connotation today. Engineering is seen also to be an economic-cooperative game of invention and communication. The poisoning of the term "engineering" after WWII can be seen to contribute to the failure of the term "software engineering."

Section V, *Future Research*, looks at how the cooperative game lexicon points to research topics for our field.

Section VI. *Summary*, recapitulates the essential points of the paper.

I. **Cooperative Games**

Viewing software development as a "*series of resource-limited, cooperative games of invention and communication*" meets the objectives for an underlying model of our field:

- It lets us make sense of historical successes and failures.
- It names at the top priority level a set of topics that are known to be important to project success but do not normally have a place to live in discussions of software development, topics such as community, amicability, morale, talent, trust, proximity, and sufficiency.
- It offers immediately usable advice to people busy on live projects.

Let us look at the core concepts arising from the model and how they relate to successful software projects.

Games, Cooperative Games, Series' of Games

Although the Merriam-Webster dictionary defines *game* only as "an activity engaged in for diversion or amusement" [Webster], the term has grown considerably in scope over the last century. Some managers resist the idea of software development as any kind of 'game' because, as one manager retorted, "We are not here for amusement". Managers obviously do not want frivolous, non-productive use of time on their projects, for reasons that will become more clear as we examine the economically overconstrained game that

is software development. While fun generally is implicit in common uses of the term (and, indeed, it is a good thing to find people having fun on a project; see the discussion of *morale* below), "games" are no longer only about frivolous or non-productive activities.

The range of what currently falls into the category of games is so broad that it is very difficult to find something in common across all of them. Wittgenstein [1953] discusses *rule following* as essential to the concept, in that a 'game' ceases to exist at the moment the players decide to stop following its rules (a game is therefore a voluntary activity to the extent that the players have the choice to act in another way). Games consist of "moves" toward or away from some target, with some sort of measurement against the distance from the target.

Some games are solo, some are group-based. Some center around achieving a goal, while others center on interactions between the participants. Some only last minutes or seconds, while others last years, even lifetimes. In finding a home for software development, I find three categorizations of a game useful:

- It may be finite or infinite in nature.
- It may be competitive or cooperative.
- It may be terminated after a goal is achieved (including time-termination of the game), or it may have no distinct endpoint (it just ends whenever it ends).

The purpose of an *infinite* game is to keep playing the game [Carse]. Individuals, organizations and countries play infinite games of survival. Individual people play the infinite game of managing their careers (so they may act to enhance their market value at the expense of the project). Government contractors arrange and rearrange staff on any one contract, possibly creating sub-optimal results for that contract, as moves in an infinite game of obtaining more contracts. It can be seen that these infinite games interfere with optimal play of a single project, and the project leaders have to contend with that interference as part of their project strategy.

Among *finite* games, some come to an end as soon as a goal is achieved, others come to an end just whenever they happen to stop. Within each category, some are competitive, others are cooperative. Tennis and chess are competitive, goal-terminating games. The children's game "king of the hill" (fighting to defend the top of a hill from the other children) is competitive but not goal-terminated. The children keep playing after, and particularly *after*, one of them becomes the new king of the hill. They stop only when it gets too dark to continue or they are called in. Poker is similarly a competitive game with an arbitrary ending point. In the *open-ended cooperative* games category we find jazz music and dancing. In both these cases, people focus on the quality of their interactions and the group performance, and the game stays in play for as long as the people decide to keep going.

The category remaining is the *goal-terminated cooperative* game. In this category we find rock and mountain climbing, exploration expeditions of all sorts, and software

development. The terminating goal for rock climbing is reaching the top of the cliff face; it is with respect to that that the team first evaluates its efforts. After that, they ask whether it was a good climb, or a pleasant one, or an interesting one. First and foremost, though, the team needs to complete the climb.

A software project has much in common with rock climbing. The primary goal is to deploy the system. It is with respect to this goal that the team is first evaluated. After that, one may ask whether it was a fun project, or well-run, or the program is aesthetic or maintainable. If the team does not deliver the software, however, the project is generally considered a failure. (*Note: it is possible – and often a good idea – to abandon the game in the middle when detecting that the goal no longer worthwhile.*)

One notable difference between software projects and rock-climbing is that software projects come in series and build upon each other in ways that climbing trips do not. In software, a team will alter the deployed system, or develop an adjacent system that interfaces with it.

Thus, unlike a rock-climbing trip, a software project has *two* goals:

- To deliver the system;
- To set up for the next game.

The full evaluation of the project therefore includes, first, whether the system was delivered, and second, to what extent an advantageous position was created for the next game.

These two goals compete for resources. The team can deliver the system more quickly if the system will not have to be extended in the future. (It can deliver the system much, *much* sooner if bugs will not have to be fixed!) Or, the team can set in place a better software architecture and more training and documentation for their successors at the expense of delaying or even preventing the current delivery.

Since project teams are limited in time, money, and people, it is not possible to perfect both goals. Most project teams are happy to achieve the first goal and any modicum of the second. Even project teams having extensive resources find the second goal to be essentially unbounded in scope. In general, a team can at best hope to deliver an acceptable system in a named time-frame, with acceptable but imperfect preparation for the future games.

To understand the shift of strategies that occurs when working with games series', let us construct another resource-limited cooperative game and examine it. Consider a race across an uncharted swampland in which some particular (unknown) artifact must be produced at some particular (unknown) place in the swamp. A team in this race would employ scouts and specialists of various sorts, and would create maps, route markings, bridges and so on. They racers would not, however, construct commercial quality maps, roads or bridges, since doing so would waste precious resources. Instead, they would

estimate how much or little of a path must be cleared for themselves, how strong to build the bridge, how fancy of markings to make, how simple a map, in order to reach their goal in the shortest time.

If the race is run as part of a series, there will be new teammates coming after them to pick up the artifact and move it to a new place. The first team will therefore be well served to make slightly better paths, maps and bridges, always keeping in mind that doing this work competes with completing the current stage of the race. They also will be well served if they leave some people who understand the territory to be part of the next team. Thus, the optimal strategies for a series of races are different than for a single race.

There is no closed-form formula for winning the game. There are only strategies that are more useful in particular situations. That realization alone may be the strongest return for using the economic-cooperative game language: people on live projects see that they must use their minds at all times to observe the characteristics of the changing situation, to collect known strategies, to invent new strategies on the fly; and that since a perfect outcome is not possible in an overconstrained situation, they much choose which outcome to prioritize at the expense of which others.

Cooperating and Communicating

At its core, software development is people inventing and communicating,

- solving a problem that they do not fully understand, which keeps changing underneath them,
- creating a solution that they do not fully understand, which also keeps changing underneath them, and
- expressing their thoughts in artificial and limited languages that they do not fully understand and that also keep changing underneath them, to an interpreter that is unforgiving of error,

where every choice has economic consequences, and resources are limited. Software development is a resource-limited, goal-directed, cooperative game, whose moves consist of *invention* and *communication*. The people, who are inventing, manipulating and communicating information across multiple heads, must share their information in order to produce the solution.

This means the speed of the project is proportional to the speed at which information moves between people's heads. *Every obstacle to detecting and moving information between heads slows the project.* Understanding and attending to this issue is essential to playing the game effectively.

The following terms are elements of the cooperative game lexicon, and are concepts that help people both understand and steer projects:

- *Ability*, a combination of *talent* and *skills* development. Raw talent provides the ability to invent better solutions or see patterns faster. People can increase their skills in an area according to their talent. The combination of talent with developed skills produces a person's *ability* in an area. Teams with greater ability in key areas have the potential to do better (whether they actually will do so depends on their strategies, as well as issues of community, communication, and motivation).
- *Community*, involving *amicability*, *trust*, *morale* and *shared experience*. Amicability is "the willingness to listen with good will." As amicability decreases, people withhold information from each other or do not listen when provided information. Amicability is fed by trust and morale. Some people are initially trusting, and have high trust until they are hurt. Others are initially untrusting, and will only reach high trust after many demonstrations of competence and non-damaging behavior by the people around them. Their experiences as a group feed their ongoing trust, morale and amicability levels [Brown] [Tyler].
- *Communication*, based on *shared vocabulary*, *proximity*, and communication *technologies*. The team members' shared experiences give them not only a basis for trust, but also a vocabulary of references that they use to speed their communication. Communication involves not only deliberate but also accidental signaling of information, as comes from worried, happy, or relaxed body expressions. The closer people are physically, the more verbal and non-verbal cues they pick up from each other, which speeds the communication. As they move farther away, the greater the role of communication technologies in simulating proximity and the more inventive they have to be in using it (see, for example, [Herring]).
- *Individuals*. With hundreds of thousands of software development specialists now in the world, statistical characteristics apply to general ability levels available. However, software development is an activity of building and passing along understanding, which is sensitive to the chemistry between individual pairs of people and within groups of people. Thus, on any one project, people operate and must be managed at the level of *individuals*. On any given day, they are *individuals* in action, as opposed to *roles* in action.

It is easy to think that "communicating a design" is nothing more than capturing its current shape in a particular descriptive format, such as the Unified Modeling Language. However, communication is not mere "transmission of information" [Maturana]. Communication, which may be thought of as "touching into a shared experience" with another person [Cockburn 2002a] takes forms that depends on the experiences shared between the individual people involved. People who have worked together before communicate through abbreviated references to previous designs and previous situations. Those with only a broad common understanding of algorithms and design patterns communicate more laboriously through references to those algorithms and design patterns. The remainder are reduced to using simple alphabetic documentation forms such as UML or comments in the code.

As Peter Naur showed in his discussion of "Programming as Theory Building" [Naur] what must be transferred is *understanding*, which is not conveyed through documentation languages and design snapshots, but is built internally by each person individually, based upon their previous knowledge. Conveying understanding is aided by showing what had been but got changed, what was rejected, and the rationales involved. This often requires direct dialogue between the people.

Since communication happens through every perceptible body movement of each participant, it happens faster when line-of-sight and multiple modalities of communication, such as gesturing, speaking, drawing, and moving, are available [McCarthy] [Cockburn 2002a]. Thus, in a resource-limited game of intensive communication, a superior strategy will often be to let people talk to each other in the same room, and photograph whiteboards or videotape their discussions, managing to be at once faster, richer, less expensive, and better suited to conveying understanding.

Interestingly, in playing this constantly shifting game, even contrary strategies do become appropriate on occasion; see for example the *Cone of Silence* strategy for an example of deliberately making communication more difficult [Cockburn 2003b].

Inventing

Invention is required at every level of the game. Users invent what they believe will prove useful in their future work, developers invent designs, testers invent tests for the system, managers invent overall project strategies for shifting situations.

Some work has been done on invention techniques: brainstorming techniques [Bordia], paper-based prototyping for user interface design [Ehn] [Snyder], CRC cards for object-oriented design [Beck], or simply discussing ideas at whiteboards, possibly using standardized design notations during discussions [Ambler]. There are some non-obvious factors in setting up invention environments, such as concealing the relative social status of the participants in order that ideas not be unduly promoted or tainted by knowledge of the suggestor's social rank [Bordia] [Weisband] [Markus]. More research of this sort is needed.

During invention, people use specialized props and specific communication modes so that ideas in their minds become externally available for examination by themselves and others. That may call for informal props such as flipcharts, drawn timelines, index cards or sticky notes, or it may call for formal graphs, tables, and models. It is important to note that these items are *transient*: the value they provide is *during* the session. Archiving them for future use is a separate strategy in the economic game of communication. People often conflate these transient props with the more permanent markers used to remind or inform.

Props and Markers

A person may create a *prop* to help in making a move, or may create a *marker* for the next person (who might be him- or herself some time later). Each prop or marker has one of three functions and optimal forms:

- To *remind* the participants of something they decided or discussed. This is communication to themselves in the future. Photographs of whiteboard discussions, rolled up flipchart drawings, napkins from restaurants and the like are very effective, not just because they are inexpensive, but also because the very imperfections in the materials serve to bring to mind the context of the earlier discussion. Obviously, the material conveys less information to people who were not present. Knowing that the purpose of these markers is only to remind themselves, the team can decide to use very rough and inexpensive markers for this purpose. Agile software development approaches [Highsmith] emphasize the use of these sorts of inexpensive *reminder* markers.
- To *inspire* a new thought in the participants. This is the invention or exploration component. Tactile props, such as paper-based user interface prototypes [Snyder], CRC cards [Beck], brainstorming cards, even stuffed animals, are intended to stimulate new neural connections in the handler through the involvement of multiple modalities. Visual-analytical props, which include simulation outputs, graphs charts, specifications, and analytical or descriptive (UML-type) models allow the handler to examine and reflect on the state of their current understanding. Not intended for communicating across time, the results of the exploration session must be recast into a reminder form. Just which reminder form to cast them into is an economic decision. The team decides after each session for whom, for what purpose and how to retain the results.
- To *inform* a newcomer. This marker is intended to bring the newcomer some distance toward the group's total understanding of the situation. It is not possible to bring the person to a complete understanding of the situation [Naur] [Cockburn 2002a], so the economic decisions to be made are how much time to spend on constructing it and how much information to try to put into it. This is the most expensive marker, since the newcomers have the least amount of shared understanding with the rest of the team. The marker has to bring them from a more remote starting point, using a more generic set of references. The software engineering field has traditionally emphasized the *informing* category of marker, with an eye to allowing total staff changeover.

Markers include people as well as artifacts. The richest marker a team can leave in place for the next team is a person from the previous team, who can inform the new participants in ways and with efficiencies that no artifact can match. For just this reason, it is a common strategy to leave some number of people from the former team in place for the next game. Without those people, informing the next team is generally too expensive and too slow for the economics of the next game.

The inevitable economic tension comes from the degradation of understanding that occurs with each change in team members, placed against the need to allow changeover of the team. Reminding markers are optimal as long as the same team stays in the game,

informing markers are critical if there will be a drastic team change. Over the course of a few games, the organization may be able to keep the same team in place, but eventually, of course, there will be no one left from the original team. Thus, it is generally faulty strategy to use only reminding markers or only informing markers, only artifacts or only people as informing markers. The organization's executives and the team have to choose between various imperfect solutions to this problem.

Team Evolution in the Game Series

Occasionally, a group of people works through the strengths and differences of the various individuals involved, and becomes a "jelled" team [DeMarco].

There are several reasons to keep jelled teams intact.

- They have an internal memory of the paths taken, and so require much smaller set of reminding markers and very little in the way of informing markers, thus saving on marker-construction costs.
- They have developed very rich shared experiences, so their communication is very much faster than another group.
- Finally, they have sorted out their issues of community and trust, which each new team must work through as part of learning to work together.

It is tempting for the manager to split the team into pieces so that each person can "communicate" to other teams whatever it is that the team had learned. Doing this operates from the idea that one person can "seed" other teams or that it is possible to "graft" one piece of a team onto another, as one does with a tree or vine.

However, playing the analogy game, a jelled team is not so much like a plant as it is like a racehorse, all of whose parts have slowly been brought to optimal function. Transferring a person from one team to another is therefore more like cutting off one of the racehorse's legs and grafting it onto a second horse. Experienced managers recognize that each group of people must work through their individual issues of personality and community themselves, in order to build their own communication pathways and shared experiences.

Maintaining a working team while introducing new people into it is a third overconstrained problem facing the manager. A manager might introduce new people onto the team in a slow stream, or according to the *Progress Team / Training Team* (alias *Day Care*) strategy [Cockburn 1998], so that each person can come to know the workings of the community without disrupting the others. Or the manager might use an apprenticeship model, such as pair programming [Williams] allowing efficient, person-to-person mechanisms to inform the new person about the project's situation.

Sufficiency-in-Communication

Software development is resource-limited and overconstrained in several dimensions:

- Delivering the system soon and inexpensively competes with creating an advantageous position for the next game.
- Creating inexpensive markers competes with creating them to work for a wider range of new people.
- Keeping the team intact competes with introducing new people.
- Using a smaller number of highly qualified people (with lower communication costs) competes with using more people of more average capability.

A project is always off balance in one of these dimensions. While making maximum forward progress, the team reduces documentation or training; while bringing the documentation up to date or training new people, it reduces forward progress. The static equilibrium that is so often sought, is not possible. There is at best dynamic equilibrium, where each move corrects one out-of-balance quality by putting some other quality out of balance. The team is continually playing a game of brinkmanship with its resources, producing results that are *adequate* or *sufficient* with respect to their respective purposes.

The effective game player recognizes that a model or piece of documentation need not be complete, current or even correct to be useful. A reminding marker need only be sufficient to remind the recipients, a prop need only be sufficient to allow people to create an interesting next move, and an informing marker needs only to be sufficient to allow the new person to ask a good question or look at another informing marker. "Adequate" is a fine condition for a communication device if the team is in a race and short on resources.

The notion of sufficiency-in-communication allows us to explain the success of many projects that succeeded despite their "obviously" incomplete documents and sloppy processes (several examples are presented in the next section). They succeeded because people made good choices in stopping work on certain communications as soon as they reached sufficiency, and before diminishing returns set in.

This is the second place where busy practitioners get usable advice from the cooperative game model. Weighing the cost of alternatives, people on live projects understand they must choose which activities to amplify and which to stop, given their current position and priorities. They shift their requests. They accept that design documents can be hints into the code, instead of up-to-date with the code. They use the project plan for strategizing, rather than expecting it to always match the current state of the project. They ask the requirements gatherers to capture just enough information to communicate to the specific designers present on the project (as opposed to some idealized set of designers). They replace typing with faster communications, such as visits in person or short video clips. Above all, they make different choices depending on whether the designers are expert and sitting close by each other, or novice and working in different time zones, whether the system is likely to kill someone if it fails, or just cause inconvenience [Cockburn 2000b].

In theory, their choices will take into account the needs of both the current game and the following games. However, it often happens that the people making these decision are

those accountable for delivering *this and only this* system. They naturally optimize for the current game at the expense of the succeeding game (which is a good strategy over a short number of games, but eventually self-destructive). The cooperative game model offers a response to this fallibility – in order to protect the interests of the succeeding games, a decision-maker should be present who both has a direct influence on the present game and a direct interest in the outcomes of the succeeding games. Whether such a person actually is present is just another aspect of playing the game well or poorly. The game concept does not prevent a group from playing poorly; it does provide an explanation of what constitutes better or worse play.

Economics and Games

Casting software development into the language of economically limited cooperative games brings into play fields of research not normally not associated with software development: economics and options theory. Some limited work has been done in viewing software decision-making in the light of buying and selling stock options [Sullivan] [Denne] [Reinertsen] [Cockburn 2002b]. We might hope that informatics researchers can make some of this material relevant to practical project management; in raw form, the results are out of reach of the average practitioner. The body of economic theory targeting choice-making under imperfect circumstances is a promising but still untouched area waiting to be investigated in the context of developing software.

Precursors: Pelle Ehn's Language Games

In 1988, Pelle Ehn built upon Wittgenstein's notion of "language games" [Wittgenstein] to develop the idea of software development as a language-game itself [Ehn]. To Ehn, that ongoing language-game involves not only verbal communication, but also activity-based communication, specifically learning-by-doing and communicating-by-doing. Ehn describes software development as making moves in this language-game to evolve a common understanding of the forthcoming system. He discusses artifacts as *markers*, including both the final system as well as intermediary design artifacts as markers. He writes (his italics):

Every *move* in the language-game of designing is a local experiment, where the initial moves often must be *reframed*, as the changed situation most often deviates from the initial appreciation. . . . In the conversation with the materials of the situation, the designer can never make a move that has only intended *implications*. The design material is continually talking back to him. This causes him to apprehend unanticipated problems and potentials, which become the basis for further moves. (p. 230).

Artifacts can support both *communicative* and *instrumental* activities . . . toward other humans or towards 'objects.' (p. 162)

In the language-game of design we use these artifacts as *reminders* and *paradigm cases* for our reflections on future computer artifacts and their use. The use of

design artifacts brings earlier experiences to our mind and it 'bends' our way of thinking of the past and the future. . . . this is how they 'inform' our practice. If they are good design artifacts, they support good moves within a specific design-language-game. (pp. 109-100)

. . . models of computer systems architecture, information system models, program specifications etc. . . . These kinds of artifacts support reflection. (p. 169)

. . . prototypes, mock-ups, scenarios with role playing, etc. . . . also allow for involved practical *experience*, not just detached reflections. . . . they can be used as reminders or paradigm cases . . . of practical understanding. (p. 169)

Recognizing the impossibility of perfect communication between people, Ehn primarily focuses on the communication between developers and users of a system:

. . . paradoxical as it sounds, users and designers do not really have to understand each other in playing language-games of design-by-doing together. Participation in a language-game of design and the use of design artifacts can make *constructive but different sense*, to users and designers. (p. 118).

. . . it is hard to see how we as designers of computer artifacts for page make-up could manage to come up with useful designs without understanding how the knife is used or what counts as good layout. For this purpose, we had to have access to more than what can be stated as explicit propositional knowledge. This we could only achieve by at least to some extent participating in the language-games of use of the artifacts. (p. 116)

Hence, what designers (and users, I would like to add) do and know, to a great extent has to be experienced in practice, not for some romantic or mystical reason, but because it is literally indescribably in linguistic terms. (p. 214)

Ehn's ideas are clearly source to, and an intrinsic part of the cooperative game concept described in this paper. However, even though he writes,

The rule-following behavior of being able to play *together with others* is more fundamental to a game than explicit regulative rules. Playing is interaction and cooperation (p. 106),

he does not develop the economics of the cooperative and group communicative aspects of the language-game, the fact that a team consists of multiple minds using multiple languages, all out of sync with each other. Not only does each move in the game involve multiple people, but there is a cost to pulling the minds closer (but never fully) together. Also missing is the notion of achieving a goal. The team is not simply "getting together to design" or to "build language"; the team is charged with producing something specific in a time-frame.

If we add three elements to Ehn's language-game concept,

- the goal-directedness of the assignment,
- the economic constraints, and
- minds separated by experience and distance,

we derive the economic-cooperative game model of this paper.

Cooperative Gaming at the 1968 NATO Conference

The term "software engineering" was coined for the 1968 NATO Conference on Software Engineering [Naur-Randell]. We look at their construction of the term in the next section, but it is useful here to extract from those proceedings a few sample quotes that illustrate to what extent they were already referencing the above concepts.. Here are sample statements from attendees at that conference, marked with terms from the lexicon:

Individuals: "I would never dare to quote on a project unless I knew the people who were to be involved." (Fraser, p. 50)

Communication: "We could use more and better and faster communication in a software group as a partial substitute for a science of software production. We cannot define the interfaces, we do not really know what we're doing, so we must get in a position where we can talk across the interfaces. (Buxton, p. 53) [*Ed. note:* compare with the description of talking across interfaces at Lockheed's Skunk Works engineering facility, in *Section IV*]

Communication and communication technologies: "An attack on the problem of communication is crucial for successful production. We are not using automation (remote consoles, text editing, etc.) as much as we should." (Gillette, p. 53)

Amicability: ". . . if I'm setting up a software group to carry out a project I'm extremely careful that all the people working on it are close personal friends, because then they will talk together frequently, and there will be strong lines of communication in all directions. One in fact uses personal relationships to support technical communication." (Buxton, p. 53)

Shared experience: ". . . if I were suddenly to recruit you lot and form a rather good software house it would be excellent publicity, but it would not actually work. It certainly wouldn't work at first, because you do not have a sufficient level of communication. One way to obtain this is by a commonality of experience. This is a major difficulty because it leads exactly to the point made by Buxton. It encourages you to work with your friends. But you have to remember that those who are incompetent find each other's company congenial." (D'Agapeyeff, p. 55)

Detecting information: "It is relatively easy to set up a communication system, manual or automatic, which will let me find information that I already realize I need to know. It is more difficult to make sure I also get information which I need, but of whose very existence I am ignorant." (Randell, p. 55).

Detecting information: (An unintended response to Randell's problem) "There was a fourth communications mechanism which every project has, and which perhaps does not get encouraged as much as it should be. There are certain people in any organization who are remarkably effective at passing gossip. Many of the potential troubles in a system can be brought into the open, or even solved, by encouraging a bit of gossip." (Fraser, p. 55)

Strategies in unknown territory: "Only one thing seems to be clear just now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies. (Fraser, p. 11)

I. **Software Engineering**

To understand the failure of the software engineering model and the need to replace it, we need to understand how that model originated, how it is failing, and what anomalies need explaining by the new model.

The 1968 NATO Conference and Software Engineering

The software engineering model came as a "provocative" action in chartering the 1968 NATO Software Engineering Conference [Naur-Randell]. In the words of the conference organizers:

1. BACKGROUND OF CONFERENCE

...

In the Autumn of 1967 the Science Committee established a Study Group on Computer Science. The Study Group was given the task of assessing the entire field of computer science, and in particular, elaborating the suggestions of the Science Committee.

The Study Group concentrated on possible actions which would merit an international, rather than a national effort. In particular it focussed its attentions on the problems of software. In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering. (p.8)

Despite having the term as a focal point for the conference, the participants showed little understanding of either the term "software engineering" or engineering in general, and provide little guidance as to just what readers are supposed to infer from the term "software engineering." Alan Perlis' keynote speech contains the following:

This is the first conference ever held on software engineering and it behooves us to take this conference quite seriously since it will likely set the tone of future work in this field in much the same way that Algol did. We should take quite seriously both the scientific and engineering components of software, but our concentration must be on the latter. (p.78)

Unfortunately, that is all he offers on the intention of the term. We pick from the following sentence the hint that making people interchangeable is a core part of its success criteria.

Stability in our goals, products and performances can only be achieved when accompanied by a sufficient supply of workers who are properly trained, motivated, and interchangeable. (Perlis, p. 79)

Ross offers the following thought on why software development is "engineering":

I agree very strongly that our field is in the engineering domain, for the reason that our main purpose is to do something for somebody. (Ross, p.74)

Since all of volunteerism is about "doing something for somebody," the above sentence does not advance our understanding much. We would not want for all volunteer activities to be brought inside the engineering discipline. It is possible that Ross was thinking of the dictionary definition of engineering ("the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man" [Webster]). However, that definition runs into a problem straight away, since programming is not about harnessing the "properties of matter" nor the "sources of energy in nature."

David introduces the "utility" aspect of engineering:

Software engineering and computing engineering have an extremely important and nice aspect to them, namely that people want to work on things that meet other people's needs. They are not interested in working on abstractions entirely, they want to have an impact on the world. This is the real strength of computing today, and it is the essence of engineering. (David, p.74)

David also worries about the direction that engineering education is taking:

Incidentally, I think that a lot of engineering education in the United States is stuck in the mud. (p. 74).

We shall see, in Section IV, "*Historical Origins of the 'Engineering Myth'*," why he might feel this way.

In short, the conference attendees were not asserting that software development is *actually* engineering (whatever that might mean), but rather, they *presuppose* that it will be fruitful to *consider* software development as engineering, for whatever benefits that might bring.

Their provocative phrase has had a good run of 35 years. It is quite reasonable that after this length of time we reconsider whether it really is the most appropriate and fruitful term to use for our practitioners' activities.

Failure of "Software Engineering" in the Present Day

If the term they coined in 1968 had performed properly, we should be able to find after 35 years of use that

- people in the industry have a similar interpretation for what the term intends;
- the interpretation provides good advice on live projects; and
- the correct application of the term correlates to more successful projects.

I find people using the term "engineering" to mean

- building models [SEED],
- looking up the answers in code books.
- balancing design trade-offs in the face of conflicting demands.
- predictable and repeatable methods and outcomes.
- that a project runs like a modern factory with statistical controls.
- "the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man" (Webster's 1977).

The model-building interpretation was put in strongest form by Ivar Jacobson as "software development *is* model building" [Jacobson, public talks]. This view carries with it the implication that the more model building one is doing, the greater the completeness and verisimilitude of the model, the better a job one is performing. In this interpretation, lots of model building should correlate to project success. Experience runs to the contrary, however. One experienced designer raised the appropriate objection this way,

"I feel when I start modeling that I am doing something useful. However, after a time, I find that I am more fiddling with the model than making real progress, and it is never clear when I passed the point of diminishing returns. Nobody ever talks about when I should *stop* modeling." (Colaizzi, private communication)

In other words, building models may be useful, but also may be counterproductive. How is a person to decide which? The term "software engineering" does not give a useful clue in the way that sufficiency-in-communication does.

Several of the interpretations of the term "engineering" confuse the activity of doing engineering with the results after having done engineering. People often mean, "Make software development more like running a factory, with statistical quality controls." However, running the factory is not the act of doing engineering, it is what comes after the engineering activity is finished. Designing the plant was the act of doing engineering, and it was a creative act, fundamentally non-repeatable and very sensitive to the characteristics of the people doing the work.

Nor is the dictionary definition useful. Converted to verb form, it advises us to "apply more of science and mathematics." Indeed, it is clear that a portion of software development depends on mathematics, and early efforts in software engineering did usefully revolve around maximizing the contribution of mathematics to solve programming problems. This work led to efficient algorithms for searching, sorting, encryption, distributed control and compiler generation among others. The mistake lies in thinking that software development is *mostly* mathematics. It may well be that the role of mathematics in software development has passed its peak.

The phrase, "do more *software-engineering*," besides generating confusion, seems mostly to generate guilt. People are sure they have not done enough of something, without being clear as to what that something is. This notion of guilt was vividly illustrated in a recent interview with a programmer I'll call NJ, one of two programmers in a four-person company. He asked me for help in discovering what they needed to do more of in their development methodology. Here is an approximate transcript of our discussion:

NJ: I think we're not doing enough of something, but I do not know what.

AC: Are you getting software out every few weeks or each month?

NJ: Yes.

AC: How are the bugs – are the bug counts high?

NJ: No, they're fine.

AC: What about the company owners – are they happy with the rate of progress and what they see being delivered?

NJ: Yes.

AC: Are there any particular problems that you can see now, or in the near future, either with the software, its quality, or the documentation you have for it?

NJ: No.

AC: If there is nothing wrong with the way you are working, why are you asking for something to change?

NJ: It just seems too simple. I feel like we should be doing *something* more, and I thought you could tell us what we're not doing enough of.

We as a community do not know what the term "software engineering" means after 35 years. Without that common understanding, it is of course hard to get people to do more of it.

Nor has project success become standard over the years, despite 35 years, establishment of "software engineering" as a valid university curriculum around the world, and the creation of the Software Engineering Institute. The Standish Report of 2003 indicates a success rate of only 34% of projects, with 15% outright failures and 51% of project in what they consider the "challenged" state [Standish]. In my own comparisons of projects [Cockburn 2000a, Cockburn 2003a], I found that

- almost any process can be made to work on some project;
- any process can manage to fail on some project;
- heavy processes can be successful,
- light processes are more often successful, and more importantly, the people on those projects credit the success to the lightness of the process.

The software engineering model does not predict the high success rate of lightweight (low-ceremony) process and the low success of very-high-ceremony process. Obviously, poor management is a non-methodological factor of greatest significance, but even normalizing for that does not give meaningful predictions.

Explaining Anomalies

Many experienced developers are not surprised by the above results. It is exactly that lack of surprise that deserves investigation. What is the experienced developer looking at to gauge the likelihood of success of any given project, if not similarity to "engineering"?

In 1991, I began interviewing and debriefing project teams as part of constructing a new methodology for the IBM Consulting Group. In each interview, I asked people at several different levels of control (project manager and programmer, for example) what they had done, what they thought worked well for them, what they would do differently, and what their priorities were [Cockburn 1998, Cockburn 2003a]. What caught me most by surprise was that they did not talk much about the subjects I had expected them to, particularly modeling tools, and modeling in general. In fact, those tended to be the items they put lowest on the list of priorities [Cockburn 1998, pp. 62-63].

Instead, I encountered sentences that did not make any sense to me at the time I wrote them down. One successful leader said:

"Give me a maximum of four people working in one big room, and access to our users and we'll deliver software to them every month or two. That's all I need. If you make me have more people, I could use eight people in two rooms. But not more."

It is tempting to suggest that this person is not a competent manager, being unable to work with more than eight people. But that is not what he is expressing. He is describing what *does* work, at least for him.

Another sentence that took a long time for me to notice, was the reference to pride-in-work:

"I mean, it works. It's not broken. But it's not as though I drive home feeling proud of the work I've done during the day."

However, the sentence that kept showing up and eluded my understanding was this one:

"At key moments, a few key people stepped in and did whatever was needed."

I could not find a satisfactory way to understand this. Was it heroism in a form that signaled poor project management? Why did so many successful project teams refer to it with pride instead of embarrassment? Should it be stamped out, or harnessed? The "software engineering" model did not provide any advice here.

These sorts of anomalies show up in the oldest case studies. In the late 1960s, Gerald Weinberg described the negative effects of removing a bank of vending machines. Note how his, and the other excerpts in this section, are naturally matched by the cooperative game lexicon:

[At] at large university computing center . . . a large common space was provided near the return window, so that the students and other users could work on their programming problems. In the adjoining room, the center provided a consulting service for difficult problems, staffed by two graduate assistants.

At one end of the common room was a collection of vending machines . . . the noise from the revelers congregating at the machines often became more than some of the workers could bear. . . . [The computing center manager] went to investigate their complaint. . . . Without more than fifteen seconds of observation he went back to his office and inaugurated action to have the machines removed to some remote spot.

The week after the machines had been removed—and signs urging quiet had been posted all around—the manager received another delegation. . . . They had come

to complain about the lack of consulting service; and, indeed, when he went to look for himself, he saw two long lines extending out of the consulting room into the common room. He spoke to the consultants to ask them why they were suddenly so slow in servicing their clients . . . For some reason, they said, there were just a lot more people needing advice than there used to be.

The manager spent two weeks checking for a possible source of the increased load, but all courses and other users were carrying on normally. . . . After some time, he discovered the source of the problem. It was the vending machines!

When the vending machines had been in the common room, a large crowd always hovered around them—but not necessarily for fol-de-rol, as the manager had so quickly assumed. True, they were drinking coffee and chatting, but they were chatting about their programs. . . . Since most of the student problems were similar, the chances were very high that he could find someone who knew what was wrong with his program right there at the vending machines. Through this informal organization, the formal consulting mechanism was shunted, and its load was reduced to a level it could reasonably handle. ([Weinberg], pp. 49-50)

Dee Hock describes how the first VISA credit card clearing system was developed by a group of people who did not seem to have the qualifications to do the job and used a spectacularly messy process:

We decided to become our own prime contractor, farming out selected tasks to a variety of software developers and then coordinating and implementing results. Conventional wisdom held it to be one of the worst possible ways to build computerized communications systems.

We rented cheap space in a suburban building and dispensed with leasehold improvements in favor of medical curtains on rolling frames for the limited spatial separation required. ...

Swiftly, self-organization emerged. An entire wall became a pinboard with every remaining day calendared across the top. Someone grabbed an unwashed coffee cup and suspended it on a long piece of string pinned to the current date. Every element of work to be done was listed on a scrap of paper with the required completion date and name of the person who had accepted the work. Anyone could revise the elements, adding tasks or revising dates, provided that they coordinated with others affected. Everyone, at any time, could see the picture emerge and evolve. They could see how the whole depended on their work and how their work was connected to every other part of the effort. Groups constantly assembled in front of the board as need and inclination arose, discussing and deciding in continuous flow and then dissolving as needs were met. As each task was completed, its scrap of paper would be removed. Each day, the cup and string moved inexorably ahead.

Every day, every scrap of paper that fell behind the grimy string would find an eager group of volunteers to undertake the work required to remove it. To be able to get one's own work done and help another became a sought-after privilege. Nor did anyone feel beggared by accepting help. Such Herculean effort meant that at any time, anyone's task could fall behind and emerge on the wrong side of the string.

Leaders spontaneously emerged and reemerged, none in control, but all in order. Ingenuity exploded. Individuality and diversity flourished. People astonished themselves at what they could accomplish and were amazed at the suppressed talents that emerged in others.

Position became meaningless. Power over others became meaningless. Time became meaningless. Excitement about doing the impossible increased, and a community based on purpose, principle, and people arose. Individuality, self-worth, ingenuity, and creativity flourished; and as they did, so did the sense of belonging to something larger than self, something beyond immediate gain and monetary gratification.

No one ever forgot the joy of bringing to work the wholeness of mind, body, and spirit; discovering in the process that such wholeness is impossible without inseparable connection with the others in the larger purpose of community effort. Money was a small part of what happened. The effort was fueled by a spontaneous expansion of the nonmonetary exchange of value. ...

No one ever replaced the dirty string and no one washed the cup. ... The BASE-1 system came up on time, under budget, and exceeded all operating objectives." ([Hock], p. 205-207)

The standard software engineering lexicon would predict that this project should have been a disaster. In the cooperative game lexicon, however, it is clear that these people capitalized on the key factors of rapid communication, cooperation, trust, community, and pride-in-work.

The 1968 NATO conference itself is so rich with anomalies that question the engineering lexicon and support the cooperative game lexicon that I cannot include them all. Here are a representative set:

Fraser: "Design and implementation proceeded in a number of stages. . . . Each stage produced a useable product and the period between the end of one stage and the start of the next provided the operational experience upon which the next design was based. . . . The first stage did not terminate with a useable object program but the process of implementation yielded the information that a major design change would result in a superior and less expensive final product. During the second stage the entire system was reconstructed; an act that was fully justified by subsequent experience. . . . The final major design change arose out of

observing the slow but steady escalation of complexity in one area of the system." (pp. 11-12)

Smith: I'm still bemused by the way they attempt to build software. . . . All documents associated with software are classified as engineering drawings. They begin with planning specification, go through functional specifications, implementation specifications, etc., etc. This activity is represented by a PERT chart with many nodes. If you look down the PERT chart you discover that all the nodes on it up until the last one produce nothing but paper. It is unfortunately true that in my organisation people confuse the menu with the meal. (p. 52)

Kinslow: The design process is an iterative one. I will tell you one thing which can go wrong with it if you are not in the laboratory. In my terms design consists of:

1. Flowchart until you think you understand the problem.
2. Write code until you realize that you do not.
3. Go back and re-do the flowchart.
4. Write some more code and iterate to what you feel is the correct solution. (p. 21)

Ross: The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. (p. 21)

Perlis: A man can communicate with about five colleagues on a software project without too much difficulty. Likewise he can supervise about five people and know pretty well what they are doing. One would structure 120 people in three levels, in which no man is talking to more than about eight people, both across his level and up and down . . . (p. 51)

Opler: I think I know how to organise reasonably successful communication for projects of between 10 and 50 people. . . . every member of the staff receives a three-ring binder and perhaps half-a-dozen pages stating the very first decisions and ground rules for the project, including an index. As the project proceeds everybody contributes sheets, which must be countersigned by their management. . . . This had interesting side-effects. I noticed that one part of the book was not filling in very fast — this led to early discovery of a worker who was lagging behind, and who eventually had to be dismissed. (p. 55)

Fraser: The question of what methods should be used for organising information flow between members of a production team depends largely on the size of the team.

I was associated with a 30-man project . . . We had three, or rather four, forms of information flow. The first was based on the fact that the compiler was written in a high-level language and hence provided, in part, its own documentation. The second form of information flow was based on documentation kept in a random access device which was regularly accessed by every member of the team. This was a steel filing cabinet kept in my office. . . . This was probably the most important form of communication we had. Its merits were that there was only one set of authoritative information, and that the indexing scheme, albeit crude, was sufficient to allow one to find, in most cases, the relevant information when you needed to make a decision. . . .

There was a fourth communications mechanism which every project has, and which perhaps does not get encouraged as much as it should be. There are certain people in any organization who are remarkably effective at passing gossip. Many of the potential troubles in a system can be brought into the open, or even solved, by encouraging a bit of gossip. (p. 55)

Even in 1999 we find the same issues in play. Here is an excerpt from a team working at the top level of the Software Engineering Institute's Capability Maturity Model. Note the importance given to issues of trust, communication, pride-in-work and personal, individual interactions:

. . . To be most effective, engineers must be motivated and energetic; they need to be creative and concerned about the quality of their products, and they should enjoy their work and be personally committed to its success. This can only be achieved if management trusts the engineers to work effectively and the engineers trust their management to guide and support them. . . . Management also needs to ensure that the engineers consistently follow disciplined methods and that the teams do not develop interpersonal problems. [Webb]

I. **Engineering in Action**

The previous section raised the question of what professionals do while they are *doing engineering* and why people do not automatically think of craft and cooperation issues with respect to the term. In this section we look at the degradation of the term "engineering" in the last half century, and consider its more appropriate constituent parts.

Historical Origins of the "Engineering" Myth

Engineering once incorporated craft as an aspect, but lost it following WWII, as discipline envy flowed from applied physics to engineering and thence to software development. Schön recounts:

After World War II, in the glow of engineering triumphs which would have been impossible without the contributions of physics, and later on under the shadow of

Sputnik, the advocates of engineering science had succeeded in transforming the engineering curriculum into an education in applied physics. By the late 1960s, however, leading practitioners and educators were beginning to have second thoughts. Harvey Brooks, the dean of the Harvard engineering program was among the first to point out the weakness of an image of engineering based exclusively on engineering science. In his 1967 article, "Dilemmas of Engineering Education," he described the predicament of the practicing engineer who is expected to bridge the gap between a rapidly changing body of knowledge and the rapidly changing expectations of society. The resulting demand for adaptability, Brooks thought, required an art of engineering. The scientizing of the engineering schools had been intended to move engineering from art to science.

Aided by the enormous public support for science in the period 1953-1967, the engineering schools had placed their bets on an engineering science oriented to "the possibility of the new" rather than to the "design capability" of making something useful . . . Practicing engineers are no longer powerful role models when the professors of highest status are engineering scientists. . . . by 1967 engineering design had virtually disappeared from the curriculum, and the question of the relationship between science and art was no longer alive. . . . (pp. 171-172)

The result of this inflation of the infallibility of mathematical prediction was that people started expecting things made of "engineering" – and by implication, software development – to be predictable in cost, time and quality. However, even practitioners of the oldest field of engineering, civil engineering, fail in the same way as the average software developer when put in a similar situation. The project to build a highway under the city of Boston, to take one example, was estimated in 1983 as costing \$2.2 Billion and being completed in 1995. At the time of this writing in 2003, it is scheduled for completion in 2005 at an approximate cost of \$14.6 Billion [Cerasoli] [Chase]. The cost overrun is ascribed to the fact that it is larger than previous projects of this sort, and employs new and untried technologies. Martin Fowler quips [public talk], "Compared to civil engineers, software developers are rank amateurs at cost overruns".

Popular expectations for engineering are faulty because the popular understanding of engineering-as-an-activity is itself faulty. If we reframe engineering as another entry in the economic-cooperative game category, along with framing laws and constitutions, the difficulty in accurately predicting the trajectories of engineering projects becomes more understandable. Predicting those is in the same category as predicting the time and cost for lawmakers to frame a new law or constitution (or any other activity in the economic-cooperative game category).

Doing Engineering

"Doing engineering" involves doing direct work in a situation, reflecting on the lessons learned in doing that work, and generating theories local to the problem at hand.

In *The Reflective Practitioner*, Donald Schön (1983) considers a key aspect of professional practice to be the engagement in a "reflective conversation with the situation." Schön gives examples of both novice and experienced engineers gaining intimate knowledge of materials, actions and consequences, setting those next to their personal theories about the problems and solutions encountered to construct their next actions. Schön's observations are incorporated in Pelle Ehn's description of the language-game of development: "In the conversation with the materials of the situation, the designer can never make a move that has only intended *implications*. The design material is continually talking back to him. This causes him to apprehend unanticipated problems and potentials, which become the basis for further moves." [Ehn, p. 230]

The leader of Lockheed's famed "Skunk Works" facility harnessed rather than fought the need for guessing, experimentation, feedback and communication that are crucial to effective engineering. Kelly insisted on people sitting close together and taking accountability for decisions all the way from design to testing [Rich]. This can be seen both as effective "reflective conversation with the situation" and as effective play in a resource-limited cooperative game of invention and communication:

Kelly kept those of us working on his airplane jammed together in one corner of our [building]... My three-man thermodynamics and propulsion group now shared space with the performance and stability-control people. Through a connecting door was the eight-man structures group. ... Henry and I could have reached through the doorway and shaken hands. . . .

I was separated by a connecting doorway from the office of four structures guys, who configured the strength, loads, and weight of the airplane from preliminary design sketches. ... The aerodynamics group in my office began talking through the open door to the structures bunch about calculations on the center of pressures on the fuselage, when suddenly I got the idea of unhinging the door between us, laying the door between a couple of desks, tacking onto it a long sheet of paper, and having all of us join in designing the optimum final design. ... It took us a day and a half. ..."

All that mattered to him was our proximity to the production floor: A stone's throw was too far away; he wanted us only steps away from the shop workers, to make quick structural or parts changes or answer any of their questions.

The similarities in team set up between Kelley's expert group and Dee Hock's *ad hoc* group are not accidental, but essential to accomplishing the group assignment

[Allen]. Contrast Kelley's and Schön's understanding of engineering with that proposed by the 1968 NATO conference attendees:

Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes — build the whole thing, push it off the cliff, let it crash, and start over again. (Graham, p. 12)

If we investigate the Wrights' methods, we find, quite to the contrary, that they practiced engineering in the best sense, namely, reflective conversation with the situation, using guesses, experiment, theory, and, of course, feedback. Their own account reads as follows [Wright]:

In order to satisfy our minds as to whether the failure of the 1900 machine to lift according to our calculations was due to the shape of the wings or to an error in the Lilienthal tables, we undertook a number of experiments to determine the comparative lifting qualities of planes as compared with curved surfaces and the relative value of curved surfaces having different depths of curvature. . . . In September we set up a small wind tunnel in which we made a number of measurements. . . but still they were not entirely satisfactory. We immediately set about designing and constructing another apparatus from which we hope to secure much more accurate measurements. . . . we made thousands of measurements of the lift, and the ratio of the lift to the drift with these two instruments. (pp.15-18)

Engineering as a Cooperative Game

Much of "doing engineering," along with doing research, and developing software, is playing a goal-directed and resource-limited cooperative game of invention and communication. Thomas J. Allen of MIT researched and documented the essential role of proximity and communication in research and development organizations in the 1970s [Allen]. He found, as Kelly's Skunk Works team and other researchers, engineers and software developers find, that the energy and time expended in detecting and transferring ideas between minds is a key factor in the team's progress. Proximity, accountability, morale, community and trust are aspects of reducing delay in detecting and transferring ideas. Rapid feedback across the total process is part of Schön's "reflective conversation with the situation."

We could attempt to bring the world back to a pre-WWII appreciation of the craft elements of engineering. Even if we managed that, though, we would still not address a primary *desiderata* for an underlying model for our field: to evoke a reaction in busy practitioners to attend to community, cooperation, amicability, trust and sufficiency-in-communication. The cooperative game vocabulary does that.

II. **Future research**

The cooperative game model indicates several areas of research. The first centers around people's abilities to create software:

- the mechanics and economics of inventing,
- the mechanics and economics of communicating over various media for various purposes, including optimal occasions to *avoid* using face-to-face communication,
- how and why people cooperate,
- what affects trust and pride,
- what affects morale.

A second area of research centers around theories of decision making with bounded rationality and imperfect communication. A third area centers around project funding, perhaps borrowing from venture capital financing and strategies for options trading.

One caution is called for here. There is an old story about a man looking for his wallet at night under a lamp post. When a passer-by stops to help and asks him where he lost it, he points into the darkness and replies, "Over there somewhere." The passer-by asks, "But then why are you looking for it over here?" The man replies, "Because the light is better here." When suggesting research for our field, I often hear the response. "We do not research people issues because we are computer scientists. It is not that these topics are irrelevant, but they aren't for us." These speakers are comfortable under their lampposts and do not wish to venture into the darkness.

However, we work in the arena of software development, and so it is incumbent on *us* to learn how to do a better job at creating software. If we must learn something about people to accomplish that, then indeed, we must learn something about people. We needn't learn everything about people, only those things surrounding invention and communication on cooperative games. That will, of course, include motivation, reward, fear, trust, amicability, pride, ego, community, modalities in communication, and solo and group idea creation.

It may well be that we will do as we have done in the past with other areas of enquiry, and learn things that other specialists do not know. It may be that we will incorporate into our field knowledge from other fields, as we have done in the past with fair success. Whichever way it goes, it is thoroughly part of our job to learn more about the active component in our arena: *people*.

I. Summary

When "software engineering" was introduced in 1968 as a model for the field of software development, it was introduced as a provocation rather than as a model deduced from experience [Naur-Randell]. This paper reconsidered the model in the light of four decades of experience, and found the model lacking in four respects:

- The model does not intrinsically generate topics known to be important to project success, topics such as talent and skill, team cohesion and interpersonal communication [Boehm].
- The model fails to explain the historical record of successful and failing projects [Cockburn 2003a]. In particular, it fails to explain the success of so many low-ceremony, even sloppy-looking projects, and the declared preference of experienced, successful developers with those processes.
- After 35 years of use, different people still interpret the term in very different ways, leading to conflicting recommendations for behavior on projects.
- The term, and the model, do not lead practitioners on live projects to derive effective advice as to how to proceed.

This paper introduced a new model:

Software development is a series of resource-limited, goal-directed, cooperative games of invention and communication.

The primary goal of each game is the production and deployment of a software system.

The residue of the game is a set of markers to assist the players of the next game.

People use markers and props to remind, inspire and inform each other in getting to the next move in the game.

The successor game is an alteration of the system or the creation of a neighboring system, and so each game has as a secondary goal to create an advantageous position for the next game.

The primary and secondary goals compete for resources in a resource-limited situation.

This model intrinsically names issues known to be important to project success: cooperation, communication, cost-of-, rate-of-, and sufficiency-in-communication. It quickly hints at other relevant issues: individual talent and skill, relations between people-as-individuals in pairs and in groups, the value of retaining jelled teams, the diminishing returns with extended modeling and documentation, and the importance of learning and applying different strategies for different circumstances.

With those topics brought to the fore, the new model was shown to fit project experience reports from untrained groups in the 1960s and 1970s up through a CMM Level-5 organization in 1999. Practitioners on live projects find the model useful because it illuminates the key issues and the trade-offs they have to deal with in their overconstrained situations.

The model introduces the possibility of usefully borrowing results from other fields to help in creating project management strategies. Economic theory, theories of decision-making with bounded information, and options trading seem particularly relevant.

A brief retrospective of engineering in the general sense indicated that much of engineering also belongs to the category of resource-limited, goal-directed, cooperative games of invention and communication.

The economic-cooperative game model serves primarily in building project strategies. It model does not capture the thought processes of the designer-programmer while creating and manipulating the design and expression of the program. An adjunct model, picking up aspects of a *mental craft*, needs to be added. This adjunct model will want to incorporate Peter Naur's consideration of programming as "theory building," Donald Schön's idea of "reflective conversation with a situation," and the issues of efficiency, manipulability and aesthetics of the program [Coplien]. The adjunct model will be subject to and have a natural fit with the economic-cooperative game model.

I. References

1. Allen, T., *Managing the Flow of Technology*, MIT Press, 1984.
2. Ambler, S., *Agile Modeling*, John Wiley & Sons, 2002.
3. Beck, K., Cunningham, W., "A laboratory for teaching object-oriented thinking," *ACM SIGLPLAN* 24(10):1-7, 1989.
4. Bordia, P., Prashant, K., "Face-to-face versus computer-mediated communications: A synthesis of the literature", *The Journal of Business Communication* 34(1), U of Illinois, Champaign, IL, Jan 1997, pp. 99-120.
5. Boehm, B., *Software Cost Estimation with COCOMO II*, Prentice Hall PTR, 2000.
6. Brown, K., Klastorin, T. & Valluzzi J., "Project Performance and the Liability of Group Harmony," *IEEE Transactions On Engineering Management*, 37(2), May 1990, pp. 117-125.
7. Carse, J., *Finite and Infinite Games*, Ballantine Books, 1987.
8. Cerasoli, R., Office of the Inspector General, Commonwealth of Massachusetts, "A History of Central Artery/Tunnel Project Finances 1994 – 2001: Report to the Treasurer of the Commonwealth", available online at <http://www.state.ma.us/ig/publ/cat01rpt.pdf>.
9. Chase, T., "Revelation 13: The Big Dig" <http://www.revelation13.net/bigdig.html>
10. Cockburn, A. [1998], *Surviving Object-Oriented Projects*, Addison-Wesley, 1998.
11. Cockburn, A. [2000a], "Characterizing People as Non-Linear, First-Order Components in Software Development", 4th International Multiconference on Systemics, Cybernetics, and Informatics, Orlando, FL, July, 2000. Online as Humans and Technology Technical Report, TR 99.05, at <http://alistair.cockburn.us/crystal/articles/cpanfocisd/characterizingpeopleasnonlinear.html>.

12. Cockburn, A. [2000b] "Selecting a Project's Methodology", *IEEE Software*, 17(4), July/Aug 2000, pp.64-71.
13. Cockburn, A. [2001], "The Expert-in-Earshot Project Management Pattern", in Succi, G., Marchesi, M., *Extreme Programming Examined*, Addison-Wesley, Boston, MA, 2001, pp. 245-247.
14. Cockburn, A. [2002a], *Agile Software Development*, Addison-Wesley, 2002.
15. Cockburn, A. [2002b], "Learning from Agile Development, Part 1," in *CrossTalk: The Journal of Defense Software Engineering*, October, 2002, pp. 10-14
16. Cockburn, A. [2003a], *People and Methodologies in Software Development*, Dr. Philos. dissertation, Faculty of Mathematics and Natural Sciences dissertation no. 264, University of Oslo, 2003, online at <http://alastair.cockburn.us/htdocs/crystal/books/pamisd/peopleandmethodologiesinsoftwaredevelopment.pdf>.
17. Cockburn, A. [2003b], "The 'Cone of Silence' and Related Project Management Strategies," Humans and Technology Technical Report 2003.01, online at <http://alastair.cockburn/crystal/articles/cos/coneofsilence.htm>.
18. Coplien, J., "Software Development as Science, Art and Engineering." In Rising, L, ed., *The Patterns Handbook: Techniques, Strategies, and Applications*, pp. 321-332. Cambridge University Press, New York, January 1998.
19. DeMarco, T., Lister, T., *Peopleware: Productive Projects and Teams, 2nd Ed.*, Dorset House, 1999.
20. Denne, M., Cleland-Huang, J., *Software by Numbers: Low-Risk, High-Return Development*, Prentice Hall PTR, 2003.
21. Ehn, P., *Work-Oriented Development of Software Artifacts*, Arbetslivscentrum, Stockholm, 1988.
22. Herring, R., Rees, M., "Internet-Based Collaborative Software Development Using Microsoft Tools", in *Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics SCI'2001*. 22-25 July, 2001. Orlando, Florida., online at <http://erwin.dstc.edu.au/Herring/SoftwareEngineeringOverInternet-SCI2001.pdf>.
23. Highsmith, J., *Agile Software Development Ecosystems*, Addison-Wesley, 2003.
24. Hock, D., *Birth of the Chaordic Age*, Berret-Koehler, 1999.
25. Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
26. Markus, M., "Asynchronous technologies in small face to face groups," *Information Technology & People*, Apr 92, 6(1), p.29.
27. Maturana, H., Varela, F., *The Tree of Knowledge*, Shambhala Publications, 1988.
28. McBreen, P., *Software Craftsmanship*, Addison-Wesley, 2001.
29. McCarthy, J., Monk, A., "Channels, conversation, cooperation and relevance: all you wanted to know about communication but were afraid to ask," in *Collaborative Computing*, Vol. 1, No. 1, March 1994, pp. 35-61.
30. Naur, P. Randell, B, *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7th to 11th October 1968, Naur, P., Randell, B., eds., 1969, online at <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/>

31. Naur, P., "Programming as Theory Building", pp.37-48 in *Computing: A Human Activity*, ACM Press, 1992.
32. Ohno, T., *Toyota Production System: Beyond Large-Scale Production*, Productivity Press, 1988.
33. Reinertsen, D., *Managing the Design Factory*, Free Press, 1997.
34. Rich, B., Janos, L., *Skunk Works: A Personal Memoir of My Years at Lockheed*, Little, Brown and Company, 1994.
35. Schön, D., *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, 1983.
36. SEED, "Object-Oriented Software Engineering", The SEED Project, online <http://seed.edrc.cmu.edu/SC/requirements/SC-req-2-development.html>.
37. Snyder, C., *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*, Morgan Kaufmann, 2003.
38. Standish, *2003 CHAOS Chronicles*, The Standish Group International (<http://www.standishgroup.com>), summarized at http://www.findarticles.com/cf_dls/m0EIN/2003_March_25/99169967/p1/article.jhtml
39. Sullivan, K, Chalasani, P., Jha, S., Sazawal, V., "Software Design as an Investment Activity: A Real Options Perspective," in *Real Options and Business Strategy: Applications to Decision Making*, L. Trigeorgis, ed., Risk Books, December 1999.
40. Tyler, T., Kramer, R., eds., *Trust in Organizations: Frontiers of Theory and Research*, Sage Publications, 1996.
41. Webb, D., Humphrey, W., "Using TSP on the TaskView Project", in *CrossTalk: The Journal of Defense Software Engineering*, Feb 1999, pp. 3-10, online at <http://www.stsc.hill.af.mil/crosstalk/1999/feb/webb.asp>
42. Webster, N., *Webster's New Collegiate Dictionary*, 1977.
43. Weinberg, G., *The Psychology of Computer Programming, Silver Anniversary Edition*, Dorset House, 1998.
44. Weisband, S., Schneider, S., Connolly, T., "Computer-mediated communication and social information: status salience and status differences", in *Academy of Management Journal* 38(4), Mississippi State, Aug 95, pp. 1124-1143.
45. Williams, L., Kessler, R., *Pair Programming Illuminated*, Addison-Wesley, 2002.
46. Wittgenstein, L., *Logical Investigations*, Basil Blackwell, Oxford, UK, 1953.
47. Wright, O., *How We Invented the Airplane: An Illustrated History*, edited by F.C. Kelly, Dover, 1953.