ABSTRACT

This is a position paper describing several rules for the start-up of a new software development project. Management methods and attitudes are suggested which reduce the problems of misestimating people skills and task difficulty, appointing the wrong person as team leader, measuring conformity to specifications and adjusting to unforeseen changes. Establishing an environment which encourages continual, on-job education through the informal and formal review mechanism is advocated in favor of formal training programs. Team building and the concept of organizing for self-organizing teams is discussed.

"Remember, it's always darkest in the middle of the night."
—The Fonz

INTRODUCTION

Our topic, as originally assigned, was "What is the best way to organize, recruit, and train for a new software development project?" The best way to organize this paper is by discarding that topic for one or two others:

a. What are better ways of getting started than we now often do?
b. What are some of the errors that ensure a bad start?
c. Given limited resources (and presentation time), what are the most important areas for concentration of management attention?

Our organization, Ethnotech, is a small one, with limited resources for consulting and training in software development. Like many of our clients, we have neither the time nor money to accumulate sufficient experience in software development to state the "best" way of doing any non-trivial aspect—and getting started properly is most assuredly non-trivial. We have, therefore, concentrated our own work in several areas—areas in which present practice is the poorest and which therefore promise greatest returns for a small investment. These areas are largely in the human side of the business, as our name suggests. Our paper will concentrate on these areas.

RULE 1—START WITH WHO YOU HAVE

Any successful project begins with a realistic appraisal of problems and resources. Put another way, more projects fail for lack of realism at the outset than for any other reason. There are two principal forms of this misestimation:

a. misestimating your people
b. misestimating the task

Let us first consider the people.

Because programming work is traditionally hidden from the view of all but a single person, there is no reliable way to estimate the capabilities of each technical person. Some programmers are considered aces because they talk a good program or spend a lot of time finding fantastic bugs that never should have been created in the first place. Others quietly and competently go about their work, unrecognized by their management. When the time comes to organize a project, the manager is working largely at random in matching people to responsibilities. We have seen, for example, numerous failures of chief programmer teams because of misplacement of individuals, such as

a. placing an inadequate person as chief
b. appointing the more adequate person as backup, even though the chief may be reasonably qualified

In order for a chief programmer team to work, the best person—who must also be an adequate person—must be the chief. If someone inadequate is appointed, the team quickly finds out—if they don’t already know. In true team work, the open inspection of one another’s work reduces the misestimation of people. Therefore, once teams have been well and truly established, the problem of assignment of responsibility diminishes.

This worthwhile feature of team organization is, unfortunately, lacking when an organization first decides to use the
TEAM BUILDING

TEAM BUILDING

RULE 2—NEVER ALLOW THE PROBLEM TO BECOME UNDEFINED

To someone outside the computing business, it must seem astonishing that people would be asked to make reliable estimates for building a product when

a. They don’t know what it is they’re building.
b. They don’t have any reliable way of measuring how much has been built.

If the task to be done is misestimated at the outset, how can we expect the project to be well managed?

Much has been spoken and written about the importance of obtaining accurate specifications early in a project, so we need not labor that point. Suppose, for the moment, that we did somehow obtain accurate and complete specifications at the outset. Once the project gets into full swing, everyone knows

a. The specifications will change.
b. People will be too busy, personally too interested, or technically too ignorant to appraise the conformity of the partial project to the specifications.

The only known way to ensure a coherent project in the face of these problems is through regularly scheduled formal reviews of conceptually manageable portions of the project. When a specification is passed from one group to another, a formal review marks its passage. When a change is proposed to the specification, it must invariably be filtered through a formal review. When a team claims to have completed a certain portion of assigned work, it must be reviewed by outsiders before it can be considered part of the completed system.

Review by qualified outsiders assures that no team can drift away from project goals and accomplishments. Periodic reviews of manageable pieces ensures that the project as a whole cannot drift into undefined—and thus unmanageable—states.

But, of course, the project must be manageable to begin with, or it can hardly be made manageable by reviews or any other strategy. When we drop our assumption of accurate and complete initial specifications, we see immediately that the project begins with reviews. Even before teams are formed, in some cases, review groups can and must be put to the task of validating the completeness and accuracy of specifications—without which the project will certainly fail.

And, as a byproduct of these “pre-natal” reviews, all technical people begin to get sensible appraisals of one another’s technical leadership skills. Thus, in attempting to control the task, you also begin to control the estimation of people.

RULE 3—INVEST EARLY AND KEEP INVESTING IN TEAM BUILDING

For successful software building, we must understand that a team is not the same thing as a group. A group is merely a collection of people in the same place at the same time, whereas a team is a collection of people sharing a past of working together and a future in which they will have to live with the consequences of present actions.

The contrast can be understood by thinking of sports teams. Winning teams are more than collections of talented individuals. Witness the singular lack of success of all-star “teams” when playing the true team that has won a championship. The all-stars, by definition, have far more talent, but seldom win. In programming projects, too much management prestige in the particular choice. No specific “chief” or “backup” is appointed. Team makeup is made conspicuously subject to revision as the team develops. The team members, with occasional gentle assistance from the manager, gradually adapt the team membership and responsibilities to the task at hand, in the light of increasingly accurate appraisals.

The manager selects the members of the initial team by whatever means seem plausibl, but avoids investing too much management prestige in the particular choice. No specific “chief” or “backup” is appointed. Team makeup is made conspicuously subject to revision as the team develops. The team members, with occasional gentle assistance from the manager, gradually adapt the team membership and responsibilities to the task at hand, in the light of increasingly accurate appraisals.

Of course, sometimes a manager is lucky in assigning people to responsibilities. Once off to a lucky start, new projects can succeed by simply following the adage: “Reward the successful with more challenging assignments.” In an organization with a history of unlucky software development, some other strategy is needed. Unfortunately, we know of no other way to reliably select the true technical leader, so we recommend a strategy that doesn’t require reliable selection by the management.

The approach that seems to work here is the adaptive team. The manager selects the members of the initial team by whatever means seem plausibl, but avoids investing too much management prestige in the particular choice. No specific “chief” or “backup” is appointed. Team makeup is made conspicuously subject to revision as the team develops. The team members, with occasional gentle assistance from the manager, gradually adapt the team membership and responsibilities to the task at hand, in the light of increasingly accurate appraisals.

RULE 3—INVEST EARLY AND KEEP INVESTING IN TEAM BUILDING

For successful software building, we must understand that a team is not the same thing as a group. A group is merely a collection of people in the same place at the same time, whereas a team is a collection of people sharing a past of working together and a future in which they will have to live with the consequences of present actions.

The contrast can be understood by thinking of sports teams. Winning teams are more than collections of talented individuals. Witness the singular lack of success of all-star “teams” when playing the true team that has won a championship. The all-stars, by definition, have far more talent, but seldom win. In programming projects, too much attention has been devoted to selecting a cast of all-stars, and not enough to getting the kind of collective effort that causes a team to build itself. No wonder we have seen so few programming teams and so many programming groups, sometimes mistakenly called “teams.”

How does team-building take place? Though details vary, the underlying principle is always the same. Teams build by sharing experiences. You cannot build a Super-Bowl champion by having each player work out on a private practice field, and you cannot build a champion software building team when each member “owns” a private piece of the work. Only through mutual help and criticism can a team develop common understanding of objectives, accurate appraisals of individual abilities, patterns of work that involve the strengths and overcome the weaknesses of each team member, rapid and precise communication among team members, and all the other factors that make a healthy
team so far superior a productive unit to any other group of people.

One way of establishing a team for a new project is to beg, borrow, or steal an already existing team. Building a new team takes time, and often means a fallow period in which the members will seem less productive than they would have been as individuals. Be sure, though, that the team was truly successful at its previous work, which is the best way to measure the progress of team building.

Another way of establishing teams is by keeping them intact, even though they have completed an initial project. IBM's New York Times project was rather large compared with many of the projects undertaken by the average DP organization. Indeed, the kind of programming done with the IBM Corporation is hardly typical of the programming done within the organization of IBM's average customer. If a new team had to be built for each new small project, few organizations could afford the team building effort.

IBM's experience of larger projects quite naturally led to team organizations because team building overhead was easily buried in large development costs. Organizations with many small projects will have to preserve their investment in team building by providing each team with a number of projects, either in serial, in parallel, or both. Though projects come and go, the team abides.

Conversely, throughout the lifetime of a team, members will come and go, rotating through the organization, leaving, retiring, or dying. By continually investing in teams, we invest the project with independence from the vicissitudes of the life of individuals. Rarely will the project find itself lacking someone experienced in some important aspect. To be sure, the comings and goings of team members add to the cost of maintaining teams, but only through teams can we, in effect, achieve redundant programming.

RULE 4—LET YOUR TECHNICAL LEADERS DO TECHNICAL LEADING

Regardless of the particular structure chosen for the team by itself or by management, there are fairly universal roles and responsibilities that must be assumed by someone. In particular, if is the role of "leader," but there are many aspects to the leader role. In the Chief Programmer Team as described by Mills, most of these aspects are bundled together in a single person backed up by another, similar, person. In practice, we more frequently find no single person capable of exercising all these aspects at the same time, for a variety of reasons.

One common reason is the heaping of too much supervision on the few technically competent people. As a project grows, there is a tendency to try to accomplish a growing task by increasing the number of people on a team. Brooks has demonstrated why adding workers late in a project actually destroys working capacity, but there are other reasons to avoid this seductive management move. For one thing, the team, to the extent it is left alone to do so, will develop more capacity as the project progresses—without adding members. For another, each additional person to supervise—especially new persons—puts an additional burden on the leadership capacity of the team.

No team should have more than five members, except during transitions when an old member hasn't yet left but a new one has joined. This limit is well supported both by social science literature and software development experience. When the team exceeds this number, communication overhead rises. Either communication breaks down or nothing but communication takes place.

What about smaller teams? Starting with two, three, or four is a way to allow for growth within the project, but it's better to allow for growth in teamwork capacity, rather than in sheer membership bulk. Get the team members together as early as possible and start them working on something meaningful, so that team development can take place. Smaller teams place an additional burden on their leadership, for there may not be sufficient diversity to solve the diverse technical problems that the team will face.

Given the proper size team—three to six at the outside—the technical leader has a chance to exercise a critical role, including speaking for the group, interfacing with management, making technical decisions, teaching, providing access to resources, keeping the team calm, getting the team excited, being wise, making everybody laugh, and just doing whatever needs doing that nobody else seems able to do at the moment.

These roles may or may not be vested in a single person, but every team member will at times exercise a few of them. Some person on the team may be designated "chief" or similar title, and may even have management responsibilities. Quite often, however, it is best not to burden one person with too many expected roles, as this may lead to insufficient time and energy for adapting to the unexpected.

Managers who tie up the technical leader's time in order to get "informed" are preventing the leader from leading. The formal review process is a more reliable source of much information a manager tries to get from the technical leader through time-consuming personal discussion.

Managers who have formerly held technical skills—especially those who still hold them—are easily tempted into meddling in the technical leader's domain. When a football team has an inexperienced quarterback, the coach may be tempted to send in all the plays. But if that's the best technical leader you can muster, perhaps your project is doomed to failure in advance. In a recent championship game, one of the quarterbacks remarked, "I don't think I'd respond very well to getting plays from the coaches upstairs. There's a certain chemistry that takes place when the quarterback selects the play and tells 10 teammates, 'Here's what we'll do and now let's make it work.' I think that's extremely important to a team.'"

In the early stages of a project, a manager may be relatively well informed technically—at least when compared with the analysts, designers, and programmers. Also, the manager may have considerable spare time to devote to making technical decisions for the team and the would-be technical leaders. As the project progresses, however, the
manager's relative technical skill will diminish, and spare
time will be a vanishing commodity.

When it's no longer possible to make technical interven­tions, the meddling manager—like the coach who sends in
plays from the bench—will find that the quarterback lacks
the experience, confidence, and desire to select the right
plays. Good performance in the game comes from good
management prior to game time—not from sending in the
winning play when the crowd is roaring loudest.

RULE 5—ALLOW ADEQUATE TIME AND SPACE
FOR LEARNING

There will always be some unknown, uncertain areas in a
software project. Every project should start with an assess­ment of its store of ignorance. This ignorance inventory
warns how much uncertainty to allow for, and how much
uncertainty in that uncertainty.

Before the project is finished, someone will have to learn
all those things about which we are now ignorant. We must
allow, from the outset, sufficient time and resources for
that learning to take place. If we cannot, then we are
gambling, not managing. It may be necessary to gamble if
the stakes are high, but when you gamble you have to be
prepared to lose, and lose often.

If we are building a system similar to one with which we
have considerable experience, the margin for learning may
be small. We can make reasonably tight guesses as to
required people, machine time, and other resources. If the
project is not similar to something we've done in the past,
we cannot accurately estimate any of these requirements.
In some cases, we can postpone the project itself in favor of
a "research" project designed to pin down the looser
ends—to educate us about the task before us. If we cannot
politically afford a distinct research effort, we must lend
the project with slack in which the learning can take place.

The need for education on projects with a large research
component explains both the success and failure of certain
efforts that have been called "top down." On well-under­stood projects, the top down work can take place solely in
the design phase without much harm, because the designers
can anticipate what the implementors will face. As the
research component grows, however, the number of prob­lems first recognized in "implementation" grows propor­tionately. By implementing top down, and by postponing
lower level design decisions until they are actually needed
in implementation, we gain the room necessary to learn
what we don't know and to change the project accordingly.

For instance, we can modify the number of teams needed
as the lower levels become successively better defined.
Adding independent teams need not cause any major inte­gration effort, particularly if the teams have already in­ vested wisely in their own development. Implementing
bottom-up, however, we would need the maximum number
of teams at the beginning, long before we were in a position
to make reasonable evaluations of the actual work load.

The learning for which we have made room takes place
primarily on the job. Perhaps it ill-behooves a small com­pany specializing in training to say so, but very little useful
learning—proportionately—comes in classroom situations.
When teams are in use, the proportion and quality of on­the-job learning is even greater.

It has been our experience, and the experience of our
clients, that the best investment in education is that de­voted to getting teams started—analyst teams, programmer
teams, programmer/analyst teams, development teams,
maintenance teams, training teams, documentation teams,
design teams,... it really doesn't matter. The average
project contains more than enough information and wisdom
to be successful—if only it can be brought to bear in the
right place at the right time.

Installations, groups, or individuals tend to develop a
unique style of working—whether in the use of a particular
language, style of development, tools, or work habits.
Wherever there are communication boundaries, useful in­formation is prevented from moving from one work style to
another. We have found installations where people were
totally ignorant of a particular language feature or program­ming method. Even worse are installations where one or
several people know the feature or method but have never
managed to communicate it to those who really need it.

Time after time, when such a shortcoming was pointed
out in an informal review between team members it spread
to the entire team in a few hours. When seen in a formal
review, the new feature or technique became the property
of the entire installation in a few days—without explicit
expenditure for "education." Allowing adequate time and
space for learning, then, is largely a matter of allowing
adequate time and space for teams and formal reviews.
These methods prove much more effective forms of educa­
tion than sending people to schools—with an increased
psychological advantage. Instead of learning a concept in a
vacuum, we learn in the context of solving a particular
problem—a problem guaranteed relevant to our project.

CONCLUSION—ORGANIZE FOR
SELF-ORGANIZATION

The earliest decisions in a software development project
have the greatest potential impact. The approach we have
abstracted here is one of avoiding decisions whose impact
will be limiting, in favor of decisions which will increase the
problem-solving capability of the project organization.
Instead of organizing for the project, organize the people who
will organize the project, unencumbered by premature
restrictions, uninformed intervention, counterproductive
social organizations, and unmeasurable goals.

One major self-organizing structure is the team. Above
the level of the team, and providing a second level of self­
or ganization, is the review process. By organizing the
various members of the project into mutually reviewing
teams, we create a project which will

a. react swiftly and effectively to changes in specifi­cations and constraints
b. learn to become increasingly productive, both individually and collectively

c. produce finished code which is readable and modifiable, progressing in steps which are measurable and controllable

d. enjoy a professional peer group environment, with less turnover and more work satisfaction

Critical to the team is the technical leader, or, rather, technical leadership. In some teams, most of the leadership is bundled into one or two members, as in the Chief Programmer Team concept. The true quality of a programming project environment, however, is measured most accurately by the amount of participation in technical leadership. In a business where one tiny mistake can cost millions, or one tiny idea can be worth millions, there is much to be lost by excluding people a priori from contributing their creative technical talents.

To be sure, a project usually begins—either well or badly—with the selection of a small number of conspicuous technical leaders—individuals who will have a recognizable influence on the shape of the project. If the project is small enough to need no others, well and good. Otherwise, the project’s success will stand or fall on its ability to incorporate other contributors into its problem-solving processes. Certain would-be leaders have personal characteristics which stand in the way of others making contributions. Certain would-be managers have similar characteristics which interfere with the technical leaders. You might think that the one would learn from the experience of trying to work under the other, but nothing is certain when it comes to learning.

With proper social organization—teams and review procedures—the personal characteristics of a few leaders become less critical to the ultimate project success. Moreover, a healthy environment will lead to the growth of more and better technical leaders as time goes on, so later projects will become less and less critical. If only we can make the right start—or restart—now, starting new projects will grow easier and easier as time goes by. Indeed, starting projects will not be a problem at all, for one project will flow smoothly out of another in the process of creating more sensibly integrated, humanized systems. In the future, we’ll know our profession has matured when we’re no longer infected with that adolescent preoccupation with starting new things, or with finding the best way. Instead, we’ll be looking for a better way of doing what we’re doing now, which is needed badly enough.