The Mythical Man-Month and Other Human Factors

Seminar on Software Cost Estimation, WS 2002/03

erstellt von

Irene Bonomo-Kappeler

Angefertigt am
Institut für Informatik der
Universität Zürich

Prof. Dr. M. Glinz
Betreuer: Arun Mukhija

Contents

1 Introduction

There is a tendency in the IT industry to concentrate on technology and neglect people. The main reason why we tend to focus on the technical rather than the human side of work is not because it's more crucial, but because it's easier to do: Human interactions are complicated and the behavior of people is not completely predictable! On the other hand personnel factors have the greatest single impact on the software engineering productivity. Since 1977, DeMarco/Lister have conducted a survey of more than five hundred development projects and their results. Fully twenty-five percent of projects that lasted twenty-five work-years or more failed to complete. They were canceled, aborted, postponed or they delivered products that were never used. For the overwhelming majority of the bankrupt projects they studied, there was not a single technological issue to explain the failure. The major problems of our work are not so much technological as sociological in nature.

Factors which influence project success are:
- time and cost estimation
- (quantitative and qualitative) staffing
- individual skills (training, experience, ability) and mental attitude (motivation)
- social environment (colleagues, managers, organizational structure, culture)
- office environment

Every organization can influence most of these factors and improve its project success by taking appropriate measures. Yet some organizations are doing a lot better than others. Their better performance can not be explained by using any particularly advanced technology, but by their more effective ways of handling people, modifying the workplace and corporate culture. There are no easy solutions, because all the easy solutions were thought of and applied long ago. Managers should specially beware of the following *seven false hopes of software management*:

1. There is some new trick you've missed that could send productivity soaring.
2. Other managers are getting gains of one hundred percent or two hundred percent or more.
3. Technology is moving so swiftly that you're being passed by.
4. Changing languages will give you huge gains.
5. Because of the backlog, you need to double productivity immediately.
6. You automate everything else; isn't it about time you automated away your software development staff?
7. Your people will work better if you put them under a lot of pressure.

2 Estimation

Good cooking takes time. If you are made to wait, it is to serve you better, and to please you.

Antoine's chef

2.1 Why are Projects Late?

More software projects have failed for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?
- Our estimation techniques are poorly developed. They reflect an unvoiced assumption which is quite untrue, i.e., that all will go well.
- Our estimation techniques confuse effort with progress, hiding the assumption that man and months are interchangeable.
- Schedule progress is poorly monitored.
- When a delay is recognized, the natural and traditional response is to add manpower. This makes matters worse as we will see in the next few paragraphs.

2.2 The Mythical Man-Month[1]

The man-month is the unit of effort used in estimating and scheduling. Cost does indeed vary as the product of the number of men and the number of months. Progress does not. Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth. It implies that men and months are interchangeable.

2.2.1 Perfectly Partitionable Task

Men and months are interchangeable only when a task can be partitioned among many workers with no communication among them. This is true of reaping wheat or picking cotton. It is not even approximately true of systems programming.

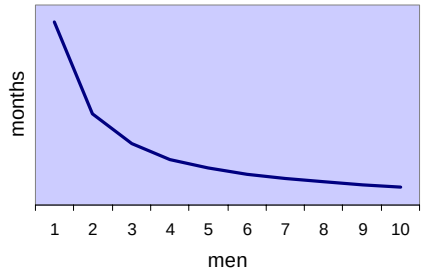

Figure 2-1: perfectly partitionable task

In tasks that can be partitioned but which require communication among the subtasks, the effort of communication must be added to the amount of work to be done.

2.2.2 Unpartitionable Task

When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule. The bearing of a child takes nine months, no matter how many women are assigned. Many software tasks have this characteristic because of the sequential nature of debugging.

[1] Nowadays political correctness would require to call it person-month, but as the title of this essay is very famous I resisted renaming it.

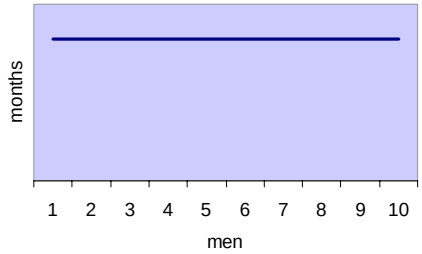

Figure 2-2: unpartitionable task

2.2.3 Task with Complex Interrelationships

The added burden in software projects is made of two parts, training and intercommunication. Each worker must be trained. This training cannot be partitioned, so the effort varies linearly with the number of workers. Intercommunication is worse. If each part of the task must be separately coordinated with each other part, the effort increases as n (n – 1) / 2:

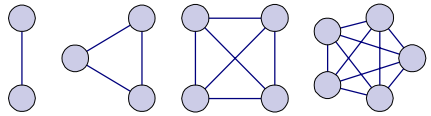

Figure 2-3: intercommunication

Three workers require three times as much pairwise intercommunication as two. Four require six times as much pairwise intercommunication as two.

The added effort of communicating may quickly dominate the decrease in individual task time brought about by partitioning. Adding more men then lengthens, not shortens, the schedule.

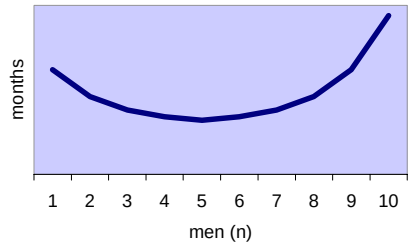

Figure 2-4: task with complex interrelationships

2.2.4 Conclusion

As a result, Brooks comes to the following very famous and often citated conclusion:

> Adding manpower to a late project makes it later.

2.3 Scheduling a Software Task

Testing is the most mis-scheduled part of programming. Because of our optimism, we usually expect the number of bugs to be smaller than it turns out to be. Delay at this point has unusually severe financial and psychological effects.

For this reason Brooks proposes the following rule of thumb for estimating a software task:
- 1/3 planning
- 1/6 coding
- 1/4 component test
- 1/4 system test

The fraction devoted to planning is larger than normal. Even so, it is barely enough to produce a detailed and solid specification, and not enough to include research or exploration of totally new techniques. The half of the schedule devoted to debugging of completed code is much larger than normal. The part that is easy to estimate, i.e. coding, is given only one-sixth of the schedule.

2.4 Parkinson's Law

Parkinson's law states that work expands to fill the time allocated for it. This may be true in a government bureaucracy, but it almost certainly doesn't apply to software engineers. They do not need unnecessary pressure!
In 1985, two researchers at the University of New South Wales, Michael Lawrence and Ross Jeffery, run a project survey that raised some doubts on the applicability of Parkinson's law in the IT industry. They determined the productivity effect of various estimating methods trying to prove that programmers work harder if they are trying to meet their own estimates.

Effort Estimate Prepared by	Average Productivity[1]
programmer alone	8.0
supervisor alone	6.6
programmer and supervisor	7.8
systems analyst	9.5
no estimate	12.0

Table 2-5: productivity by estimation approach

The first three cases confirm the assumption that programmers seem to be a bit more productive when they can do the estimate themselves. If the estimates are prepared by a systems analyst, the programmers work substantially harder. The explanation is simple: Bad estimates are always a demotivating factor, and the systems analyst tends to be the better estimator than either the programmer or the supervisor. The most surprising part of the study are the projects for which no estimates were prepared at all. These projects far outperformed all the others. Projects on which the boss applied no schedule pressure whatsoever had the highest productivity of all.

[1] weighted metric of productivity, similar to the COCOMO productivity metric

3 Staffing

3.1 Shortage

There is an abundance of would-be programmers, but a general shortage of skilled people. Or, as Davies put it 1988, “there is no skills shortage, only skills wastage; many MIS divisions are stifled by bureaucracies, many of their own making”. Nevertheless it is generally agreed that there has been a serious shortage of (experienced) staff. The current magnitude of the shortage is difficult to quantify, because the situation has considerably changed for the last few years.

The main causes of this shortage are:
- *Growth of the IT industry*.
- *Low productivity*. The lower the productivity of the existing staff, the more additional staff are needed.
- *Lack of training*. Few organizations are prepared to take on inexperienced staff. They want people with proven track records, and they are afraid that their staff, having been trained at great expense, will be poached by other organizations: “you are training people for your competitors”.
- *Recruitment*. The emphasis on mathematics as a requirement for trainee programmers may have discouraged suitable candidates. Similarly, the emphasis at universities lie on computer science rather than on commercial computing: “many computer science graduates would rather design a compiler than write the application code needed”.
- *Economic recessions*. Organizations cut down on training, so productivity suffers. During the recession, there is a reduction in the amount of work requested by users. Once the recession is over, they ask for all the work they had been holding back.
- *Poor deployment of staff*: insufficient work, waiting time between jobs, highly-qualified staff doing clerical work, over-qualified staff.
- *People leaving the industry.*
- *Turnover.*

There are many ways to alleviate the shortage, e.g. by increasing productivity, improving deployment of staff, devoting more resources to training, reduing turnover. These measures will not reduce the shortage directly, but will have the same effect as a reduction in the shortage.

3.2 Turnover

What would organizations typically do to improve productivity? Pressure people to put in more hours, mechanize the process of product development, compromize the quality of the product, standardize procedures, etc. Any of these measures can potentially make the work less enjoyable and less satisfying. You may achieve an "improvement" that is more than offset by the loss of your key people.

Some turnover of staff is inevitable and is even beneficial. It can, for example, assist the interchange of knowledge. It is also the consequence of getting rid of misfits. However, excessive turnover is detrimental and very expensive. The recruitment and induction costs of a resignation come to the equivalent of one person-year of work!

We encounter turnover figures in the range of 5-90%. In general, though, turnover has been about 25-35%. This means, that IT personnel stay, on average, only three to four years in their jobs. Turnover varies enormously from one organization to another, and it depends also on other factors, namely job category, region, nation, industry sector, age. Turnover is also related to appraisal scores: the higher the score, the higher the turnover. The best people leave! Fortunately, turnover has generally been decreasing for the last decades.

The obvious effects of an excessive turnover are loss of experience and efficiency. If, as some evidence suggests, it is generally true that turnover is highest amongst the best staff, then the average level of skill in an organization may decline to hazardous levels. A resignation can also trigger a chain effect and result in a vitious circle of turnover and shortage. This merry-go-round image destroys the users’ confidence.

High turnover is caused by different factors as shortage, lacking career-path, large organizations, pressure, desire for personal growth, personal conflicts, salary, location, lack of creativity and autonomy, mergers and takeovers etc.

The time to worry about keeping employees is when they are first hired, not when they suddenly show up in three-piece suits and ask to take two-hour lunch breaks. Cures against high turnover are:
- *Recruitment.* There is a need for better matching – on personality, ability, experience, knowledge, communications skills, etc. – of potential employees with specific employers and jobs.
- *Training* leads to a stable staff who identify with the organization – there is a “psychological contract” between the employee and the organization.
- *Career development*. There is no natural career path for a software engineer – he is „either an engineer or something else which is not an engineer“ – so management is forced to develop “artificial” career paths for software engineers.
- *Job satisfaction*. The most effective way to reduce turnover is to ensure that staff are happy: “good people in good jobs stay in them”. Maslow‘s “hierarchy of needs” can help to adjust your measures according to the staff’s needs. There are specific ways to keeping staff happy: challenges, rewards, esteem, promotions, participation, autonomy, etc. Good management is therefore the key.
- *Personal growth*. Programmers are intelligent. They want to study further and improve their qualifications, gain wider experience, and work with state-of-the-art technology.

You won't be surprised to learn that companies with extraordinarily low turnover seem to excel at many or most of the people-conscious qualities discussed in this paper. They are the best. This provides a community feeling, joint satisfaction, and a strong binding effect. There is a mentality of permanence, the sense that you'd be dumb to look for a job elsewhere.

3.3 Recruitment

The formula of recruiting is simple: get the right people and make them happy so they don't want to leave. Getting the right people in the first place is all-important. Fortunately, you don't have to depend entirely on the luck of the draw. You may get to play a significant part in the hiring of new people or the selection of new team members from within the company.

Circus Manager: How long have you been juggling?
Juggler: Oh, about six years.

Circus Manager: Can you handle three balls, four balls, and five balls?
Juggler: Yes, yes, and yes.

Circus Manager: Do you work with flaming objects?
Juggler: Sure.

Circus Manager: …knives, axes, open cigar boxes, floppy hats?
Juggler: I can juggle anything.

Circus Manager: Do you have a line of funny patter that goes with your juggling?
Juggler: It's hilarious.

Circus Manager: Well, that sounds fine. I guess you'r hired.
Juggler: Umm… Don’t' you want to see me juggle?
Circus Manager: Gee, I never thought of that.

It would be ludicrous to think of hiring a juggler without first seeing him perform. That's just common sense. Yet when you hire an engineer or a programmer, you don't ask to see a design or a program or anything. In fact, the interview is just a talk. You need to examine a *sample* of those products to see the quality of work the candidate does. That may seem obvious, but it's almost always overlooked by development managers.

As mentioned, the IT business is more sociological than technological. The workers' abilities to communicate with each other is as important as their abilities to communicate with machines. So the hiring process needs to focus on at least some sociological and human communication traits. The best way to do this is through the use of *auditions* for job candidates. The candidate is asked to prepare a short presentation on some aspect of past work. The audience is made up of those who will be the new hire's co-workers. As a result, you see the various candidates' communication skills, and you give the future co-workers a part in the hiring process.

Most organizations want the people to fit the system, instead of the system fitting the people. Most hiring mistakes result from too much attention to appearances and not enough to capabilities. The need for uniformity is a sign of insecurity on the part of management. Strong managers don't care when team members cut their hair or whether they wear ties. Their pride is tied only to their staff's accomplishments.

4 Individual Skills

4.1 Training

Training is important because of the rapid evolution of the computer field. The technical skills become obsolete within 5 years! Training leads to fewer mistakes and omissions, more accurate time and cost estimates, higher productivity, better quality products, and better service to customers. It should be considered as an investment producing a measurable return (like research and development). Two person-months of training per person in modern programming practices is expected to raise productivity by 20%, and would therefore pay for itself within a year. This leads to the following conclusion:

> Training is expensive, but no training is even more expensive.

Training can also be detrimental. Excessive training may lead to boredom. On completion of a training course, too much may be expected too soon of the returning student. Training is ineffective, if it is not provided timely, but long before the knowledge is needed.

The training cost can be calculated as about 2% of the IT budget. It contains salaries, travel and accommodation of the training staff, training facilities, opportunity cost – the cost of having the people do training rather than some other job for the organization – and the development of training materials. The amount should be 10-15 days of training per year per person. Small organizations are less committed to training than large ones.

As training methods should be considered:

- *On-the-job training* is time-consuming, costly and inefficient. A graduate of an in-house IT training program can become productive in *one-tenth* the time it would take through on-the-job training. On-the-job training also requires guidance time from experienced personnel in a master/apprentice relationship.
- *Classroom training* still counted for more than 80% of training time in 1988. However, it is notoriously unstandardized, varying with the trainer and the training center. It is also pitched at the level of the average student, who makes up about 65% of the total. More advanced students are bored, less advanced ones are anxious. Training is more effective if it is accessible and timely, i.e. available where and when needed. Classroom training is often centralized, and it is not always timely. Nevertheless, classroom training does provide individual help, for which there is a need. 15 hours of effort is required to produce one hour of conventional training.
- *Computer-based training (CBT)* involves interaction with the computer, which motivates the student. People are used to being entertained, or at least kept interested, and they learn more in those circumstances. CBT is self-paced and can be delivered at any time and any place. It reduces training time by 15–90%. The trainees must be active, i.e. not just see and hear, but also do. On the other hand, CBT is time-consuming and expensive to develop (2-10 times longer than than conventional training). The initial costs will only be recovered if they are spread over a sufficiently large number of students. Furthermore, CBT is not suitable for all training tasks.
- *Hands-on training* can be very effective: people retain only about 25% of what they hear, only 45% of what they see and hear, but 70% of what they see, hear and do.

4.2 Experience and Ability

A person‘s performance depends on his innate ability (or potential) and on his knowledge (gained through training and experience). In practice, these factors are closely linked, and it may be difficult to separate the effects due to each. Experience increases with time whereas innate ability does not.

It has often been demonstrated that productivity improves as familiarity increases. The effect was quantified by Aron in 1974, who gave the following weighting factors:

senior programmer:	0.5
programmer:	1.0
apprentice programmer:	1.5
trainee programmer:	3.0

A senior programmer is, on average, six times more productive than a trainee. The top 20% of programmers produce 50% of the output, while the bottom 50% of them only produce 20%. Furthermore, experienced programmers make less errors, write more compact code, and are better in removing bugs. By comparison, senior programmers are only paid twice as much, on average, as juniors, and are therefore not paid what they are worth. There is thus a very obvious – though limited and short-sighted – way to reduce costs: use only senior staff. This leads us to the *principle of top talent*:

> Use better and fewer people.

It is surprising that ability is taken into account in only three of nine cost-estimation models surveyed by Boehm in 1981.

Of course experience is only valuable if it is relevant to the task in hand: One must be familiar with the tools and techniques to be used, as well as with their application. Too much experience and knowledge can also have disadvantages: People may become over-qualified for their jobs, resulting in boredom and causing them to resign.

An organization should consider two points of view to ensure that it takes full advantage of the experience available:
- *Short-term view*: The *existing* experience can best be used to achieve maximum productivity on *current* work by an optimal organizational structure, ensuring interchange of knowledge and optimal allocation of work.
- *Long-term view*: The *increase* in experience can be accelerated to achieve maximum productivity on *future* work by selective recruitment[1], appropriate training, interchange of knowledge, and carefully planned allocation of work to maximize the rate of learning.

There are different types of ability which are relevant in software engineering: an analytical mind to find errors, a synthetical mind to fix them, considerable persistence to find errors, ability to write for documentation purposes, communication ability for working in teams, etc. However, ability is often very narrow: People who are very good at some things are very bad at others. To take full advantage of the differences in ability, staff would have to specialize more. Extra care must be taken when staffing a project team, to ensure that the abilities of the team members complement each other.

Are good programmers made or born? In other words: is training or ability more important? Should our energy be devoted to creating good programmers by training them or selecting them? The majority believe that if you do not have the aptitude for programming, then neither motivation nor training can compensate. According to Turton, who is a personnel consultant, only 5% of the population have the ability to become programmers.

[1] Selective recruitment means recruiting experienced staff with specialized skills, who transfer their knowledge to the existing staff.

5 Social Environment

5.1 Teams

A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts. With jelled teams, the probability of success goes up dramatically. The team can become almost unstoppable. Managing such teams is a real pleasure. You spend most of your time just getting obstacles out of their way. They don't need to be managed in the traditional sense, and they certainly don't need to be motivated. They just work for joint success and the pleasure of achieving the project goal, any goal, together. Even though there is very little true teamwork required in most of our work, teams are still important, for they serve as a device to get everyone pulling in the same direction.

> The purpose of a team is not goal attainment but goal alignment.

A few very characteristic signs idicate that a jelled team has occurred. The most important of these is *low turnover*. The team members aren't going anywhere till the work is done. Jelled teams are usually marked by a strong sense of *identity*. Teammates may all use the same catch phrases and share many in-jokes. There is a sense of *eliteness* on a good team. Team members feel they are part of something unique. There is invariably a feeling of *joint ownership of the product* built by the jelled team. The final sign of a jelled team is the obvious *enjoyment* that people take in their work. Jelled teams just feel healthy. The interactions are easy and confident and warm.

Unfortunately, you can‘t easily "make" teams jell. You can hope they will jell; you can cross your fingers; but you can't make it happen. The process is much too fragile to be controlled. Teams are not built, they grow. On the other hand, it's very easy to prevent teams from jelling, DeMarco/Lister call it *teamicide*:
- defensive management (don't trust your own people)
- bureaucracy (mindless paper pushing)
- physical separation
- fragmentation of time (assign people to more than one project)
- quality reduction (undermines the people's self-esteem and enjoyment)
- set unreachable deadlines (makes success impossible)
- clique control (break up teams)

5.2 Management

> In my early years as a developer, I was privileged to work on a project managed by Sharon Weinberg (…) She was a walking example of much of what I now think of as enlightened management. One snowy day, I dragged myself out of a sickbed to pull together our shaky system for a user demo. Sharon came in and found me propped up at the console. She disappeared and came back a few minutes later with a container of soup. After she'd poured it into me and buoyed my spirits, I asked her how she found time for such things with all the management work she had to to. She gave me her patented grin and said, "Tom, this *is* management."
>
> *Tom De Marco*

A natural manager has got a subconscious feel for what's good for the team. This feel may govern decisions throughout the project. The entire experience is organized for small, easy joint successes. You have to look twice to see the manager's hand in any of this, it just seems to be happening.

As a good manager you are the exact opposite of a defensive manager. You take no steps to defend yourself from the people you've put into positions of trust. And all people under you are in positions of trust. A person you can't trust with any autonomy is of no use to you.

> One of my first bosses was Jerry Wiener (…) At the time I came along, the company was about to enter into a contract that was larger than anything it had ever done before. The entire staff was assembled as our corporate lawyer handed Jerry the contract and told him to read it and sign on the last page. "I don't read contracts," Jerry said, and started to sign. "Oh, wait a minute," said the lawyer, "let me go over it one more time."
>
> *Tom De Marco*

The most common means by which bosses defend themselves from their own people is direct physical oversight. If you've got decent people unter you, there is probably nothing you can do to improve their chances of success more dramatically than to send them away. Any easily separable task is a perfect opportunity. There is no real management required for such work. Find a remote office, hire a conference room, borrow somebody's summer house, or put them up at a hotel.

The best bosses take some chances. They take chances on their people. None of this says that good managers don't manage, that they don't give direction and make judgements of their own. They have to do this all the time. The suggestion here is that they do this only by exercising their *natural authority*. In the best organizations, there is natural authority working in all directions. The manager is known to be better at some things, perhaps setting general directions, negotiating, and hiring, and is trusted to do those things. Each of the workers is known to have some special area of expertise, and is trusted by all as a natural authority in that area.

Some organizations are famous for their consistent good luck in getting well-knit teams to happen. It isn't luck, of course, it's *chemistry*. What are managers up to in these healthiest companies? A quick surface appraisal might convince you they aren't up to much at all. They don't seem busy. They're not giving a lot of directions. Whatever their relationship is to the work going on around them, they're certainly not *doing* any of it. In organizations with the best chemistry, managers devote their energy to building and maintaining healthy chemistry:

- They make a cult of quality. It binds the team together because it sets them apart from the rest of the world.
- They provide lots of feelings of success, DeMarco/Lister call it "closure". They divide the work into pieces and make sure that each piece has some substantial demonstration of its own completion. They deliver a product in many versions for each new version is an opportunity for "closure". The best success is the one in which there is no evident management.
- They build a sense of eliteness and uniqueness.
- They allow and encourage heterogeneity.
- They preserve and protect successful teams. They never change a winning team!
- They consider team structures as networks and not as hierarchies. The manager is most often outside the team, giving occasional direction from above and clearing away administrative and procedural obstacles. This idea is upsetting to managers who pride themselves on their leadership.
- They provide strategical but not tactical direction.

5.3 Quality

Managers should develop a cult of quality („only perfect is good enough for us"). Since the self-esteem of the workers is strongly tied to the quality of the product they produce, they tend to impose quality standards of their own. The minimum that will satisfy them is more or less the best quality they have achieved in the past. This is invariably a higher standard than what the market requires and is willing to pay for. Although extraordinary quality doesn't make good short-term economic sense, in the long run quality is a means to higher productivity, increased job satisfaction and low turnover. In some Japanese companies, the project team has even an effective power of veto over delivery of what they believe to be a not-yet-ready product.

5.4 Standards

In the spring of 1932, efficiency experts ran a series of tests at the Hawthorne Western Electric Company to determine the effects of various environmental parameters on productivity. They tried raising the light level, and they noted that productivity went up. Then they tried lowering the light level, and they noted that the productivity went up higher still. They speculated that turning the lights off entirely might send productivity through the roof . What seemed to be happening was that the change itself wasn't as important as the act of changing. People were charmed by differentness, they liked the attention, they were intrigued by novelty. This has come to be called the *Hawthorne Effect*. Loosely stated, it says that people perform better when they're trying something new.

A careful study of the literature of productivity improvement could convince you that all productivity improvements are due to the Hawthorne Effect. All papers report on productivity gains of factors when they were first introduced. You almost never hear of a study that analyzes ten-year-old "improvements" to see if they are still worthwhile. They probably aren't.

To allow the Hawthorne Effect to work for you, you have to make nonstandard approaches the rule. Whatever standard there is should be brief and gentle. In this respect, Methodologies in the form of fat books that specify in detail exactly what steps to take at any time, are detrimental. They are an attempt to centralize thinking assuming that project people aren't smart enough to do the thinking themselves.

6 Office Environment

6.1 Coding War Games

From 1984 to 1986, DeMarco/Lister have run an annual productivity survey as a sort of public competition in which more than 600 developers from 92 different companies compete to complete a series of benchmark coding and testing tasks in minimal time and with minimal defects. The participants work in their own work areas during normal work hours using the same languages, tools, terminals, and computers that they use for any other project.

Figure 6-1 shows the performance spread of time to achieve the first milestone (clean compile, ready for test) in the 1984 games:

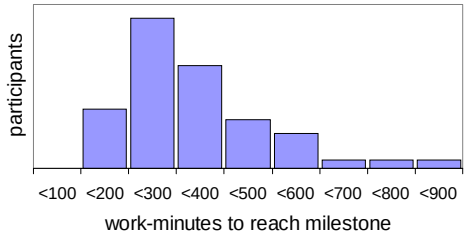

Figure 6-1: individual variation in performance

The best performance was about 2.5 times better than the average. The half above the median outperformed the half below the median by about 2:1. The best people outperformed the worst by about 10:1.

In the analysis of game results, DeMarco/Lister discovered that the language (except assembly language), the years of experience (except developers with less than 6 months' experience with the language used), the number of defects, and the salary had little or no correlation to performance. Slightly more surprising were some of the factors that they found *did* have a substantial effect on performance: People from the same organization tend to perform alike. That means the best performers are clustering in some organizations while the worst performers are clustering in others. This is the effect that Harlan Mills predicted in 1981:

> While this 10:1 productivity differential among programmers is understandable, there is also an 10:1 difference in productivity among software organizations.

The hypothesis that qualities of the workplace may have a strong correlation to developer effectiveness is an easy one to test. DeMarco/Lister had each war game participant (prior to the exercise) fill out a questionnaire about their workplaces. Then they compared the top quarter of finishers with the bottom quarter:

Environmental Factor	Those Who Performed in 1st Quartile	Those Who Performed in 4th Quartile
How much dedicated workspace do you have?	7 m^2	4.1 m^2
Is it acceptably quiet?	57% yes	29% yes
Is it acceptably private?	62% yes	19% yes
Can you silence your phone?	52% yes	10% yes
Can you divert your calls?	76% yes	19% yes
Do people often interrupt you needlessly?	38% yes	76% yes

Table 6-2: environments of the best and worst performers

The top quartile, those who did the exercise most rapidly and effectively, work in space that is substantially different from that of the bottom quartile.

Unfortunately, he environmental trend is towards less privacy, less dedicated space, and more noise.

6.2 Brain Time vs. Body Time

The Santa Teresa pre-construction study looked also into the amounts of time that developers spend in different work modes. For a typical day, they concluded that workers divide their time as follows:

Work Mode	Percent of Time
working alone	30%
working with one other person	50%
working with two or more people	20%

Table 6-3: how developers spend their time

Thirty percent of the time, people are noise sensitive, and the rest of the time, they are noise generators.

During single-minded work time, people are ideally in a state that psychologists call *flow*. Flow is a condition of deep, nearly meditative involvement. In this state, there is a gentle sense of euphoria, and one is largely unaware of the passage of time. For anyone involved in software development, flow is a must. It's only when you're in flow that the work goes well. Unfortunately, you can't turn flow on like a switch. It takes at least 15 minutes of concentration before the state is locked in. During this immersion period, you are particularly sensitive to noise and interruption. A disruptive environment can make it difficult or impossible to attain flow. The total cost of an interruption in flow time is 20 minutes (5 minutes for the interruption plus 15 minutes reimmersion time). Two dozen interruptions, and the work day is gone.

The *Environmental Factor* or *E-Factor* is a meaningful metric of how good or bad your environment is:

$$\text{E-Factor} = \frac{\text{Uninterrupted Hours}}{\text{Body-Present Hours}}$$

Whenever the number of uninterrupted hours is a reasonably high proportion of total hours, up to approximately 40%, then the environment is allowing people to get into flow when they need to.

6.3 The Ideal Office Environment

Before drawing the plans for its new Santa Teresa facility, IBM studied the work habits of those who would occupy the space. They concluded that a minimum accommodation would be the following:

- 9 m^2 of dedicated space per person
- 2.7 m^2 of work surface per person
- noise protection in the form of enclosed offices or 1.8 m^2 high partitions

In the 1984 coding war games, only 16% of participants had 9 m^2 or more of workspace. Only 11% of participants worked in enclosed offices or with greater than 1.8 m^2 high partitions.

The ideal office environment
- achieves the most efficient flow of work and overcomes bottlenecks (e.g. printers),
- increases productivity,
- enhances employee satisfaction and well-being,
- minimizes health and safety risks (e.g. radiation), and
- is free from distractions and interruptions (specially for software engineers).

The following factors have an influence on people's productivity:

- *Physical Environment*: The recommended level of lighting is about 300–500 lux, but when working with displays, a lower level of 100–300 lux is recommended. The *air temperature* should be between 13 and 27°C. If there is continuous *noise* at a volume of 100 dB, or a random intermitted noise of 65–95 dB, then workers make more errors. The noisiest item in many offices is the printer. Speech disturbs more by the distraction it causes than by its volume. Noise can be absorbed by fabric-covered chairs, carpets, and sound-absorbent walls and ceilings. It can be blocked out by wearing earplougs, turning off the telephone bell, uplugging the whole line, or using e-mail!
- *Office Layout*: *Open-plan offices* are very flexible, but also disliked, as the partitions may provide insufficient noise absorption and inadequate privacy, while restricting useful communication and personal contact. The concept may not be suitable for software people with their need for intense concentration and freedom from distractions.

  Office groupings should be made to align with work groups. The worker who needs to spend 50% of his time with one person will spend most of that time with a particular person. These two are natural candidates to share an office. Even in open-plan offices, co-workers should be encouraged to modify the grid to put their areas together into small suites. When this is allowed, people become positively ingenious in laying out the area to serve all their needs: work space, meeting space, and social space.
- *Furniture:* Most office workers spend more than 70% of their time seated at their desks, so seating is very important. The *chair* should be at the correct hight (420–550 mm), and should be designed to ensure correct posture and maximum comfort for people of any shape or size. The *desk* surface should be at a height of 580–730 mm, be large enough to accommodate computer listings, the workstation, and other equipment. It must also have adequate storage compartments.
- *Workstation*: The resolution of the *display* should be big enough. The traditional QWERTY *keyboard* was designed to *prevent* fast typing, so that the mechanism would not jam! For 68% of keystrokes, the user has to reach from the 'home' row to another row. However, it is so widely accepted that it is unlikely to be replaced, although alternative keyboards exist. Many software engineers still use the one-finger-technique. Training can result in a 25–100% improvement in typing speed and a 75% increase in accuracy.
- *Office Automation:* office tools, electronic mail, cellular phones, scanners, teleconference facilities, workgroup computing, electronic copyboards etc.

To provide the ideal environment is „short-term" expensive, but there is evidence that the investment is repaid in improved „long-term" productivity which can improve by (estimated) 15–30%. A Norwegian company reached a return on investment of 10:1! The total investment in ergonomics was $46 000; but the reduction in costs came to $437 000.

7 Conclusion

Focussing on people and their needs can have a considerable positive effect on productivity and project success. This has obviously been known for years. Why are human factors still neglected in the IT industry, although managers should know better? I see three main reasons for this ignorance:

Firstly, as I already mentioned, it is far easier for all of us to solve technical problems than social problems. That is what most managers learnt at universities and practiced before their management jobs. Technical problems might be complicated and challenging, but they are never as complex and irritating as human-related problems. The behavior of people is never completely predictable.

Secondly, providing people with more autonomy and responsibility makes them less controllable. A weak manager feels threatened by demanding, self-sufficient, productive, competent, and unique individuals. He would feel better working with a staff of uniform plastic people, identical, and interchangeable. By making everything uniform and centralized, he exercises and demonstrates control. This behavior is a sign of managerial insecurity and distinguishes the great managers from the merely mediocre. Yet those individuals do more to serve the manager's real goals than any assemblage of interchangeable parts could ever do.

Thirdly, the cost of many of the recommended measures is in a highly visible category (space and services), while the offsetting advantage is in poorly measured and therefore invisible categories (increased productivity and reduced turnover). Once more, the short-term view dominates the long-term view.

8 Appendix

8.1 Glossary

Term in English	Term in German
accommodation	Quartier, Räumlichkeit, Unterkunft
appraisal score	Beurteilung
catch phrase	Schlagwort
clerical work	Büroarbeit
“closure”	“Klick”
deployment	(Personal-)Einsatz
esteem	Wertschätzung
jelled team	verschworenes Team
misfit	schwieriger Fall, Aussenseiter
open-plan office vs. two-person office	Grossraumbüro vs. Zweipersonenbüro
poach	wildern, unerlaubt fangen
resignation	Kündigung
shortage	Mangel
staffing	Stellenbesetzung
trappings	Insignien (der Macht)
turnover	Fluktuation
uppity	arrogant, hochnäsig, frech

8.2 Bibliography

[Bro95] Brooks, F.P.: *The Mythical Man Month*, Essays on Software Engineering, Anniversary Edition, Addison-Wesley, Boston etc., 1995

[DeM87] DeMarco, T.; Lister, T.: *Peopleware*, Productive Projects and Teams, Dorset House Publishing, New York, 1987

[Ste95] Stevenson, C.: *Software Engineering Productivity*, A practical guide, Chapman & Hall, London, 1995