



Extreme Software Cost Estimating

Dr. Randall W. Jensen
Software Technology Support Center

One dominating software development complaint is the inability to estimate cost, resources, and schedule with acceptable accuracy. Several methods of schedule and cost estimation have been proposed during the last 25 years with mixed results due, in part, to the focus and capability limitations of traditional estimation models. A significant part of estimation failures can be attributed to not understanding the inner workings of the software development process and its impact on the parameters used in schedule and cost estimates. For example, poor management can increase software cost, schedule, and quality more rapidly than any other variable, while good project management can decrease development cost and schedule just as rapidly. This article describes a management-centric approach to predicting software development cost and schedule in a modern, or extreme, development environment as opposed to the traditional technology-based approaches. Techniques necessary to produce realistic, reliable software development estimates are introduced, as well as quantitative methods for predicting the impacts of management decisions. This article was awarded the Outstanding Software Paper at the International Society of Parametric Analysts 2003 Annual Conference in Orlando, Fla.

The software estimating tools in widespread use today evolved from models developed in the late 1970s to early 1980s using project data available at the time. These widely used tools include the Constructive Cost Model (COCOMO) II, Price-S, Sage and SEER-SEM. It is important to note these mature tools are as useful today as they were 20 years ago when they were first formulated. Input data parameter sets (analyst and programmer capability, application experience, use of modern practices and tools, etc.) developed for Seer to describe organizations in the early 1980s are, oddly enough, still generally applicable today. Fortunately for the estimating model developer, culture changes very slowly, if at all.

We have been able to learn new things about software development during this period. For example, Barry Boehm wrote the following in 1981:

Poor management can increase software costs more rapidly than any other factor. Each of the following mismanagement actions has often been responsible for doubling software development costs ... [1]

Of course, you have to read the first 485 pages of his book to get to this logical, yet profound statement. Most readers do not seem to get that far. Gerald Weinberg's Second Law of Consulting [2] added a supporting observation: "No matter how it looks at first, it's always a people problem."

There have been several development technology breakthroughs during the past 40 years that have significantly decreased the cost of software products. For example, the introduction of FORTRAN and

COBOL decreased the cost of a given product functionality to one-third of the cost when implemented in Assembler. The transition from C++ to the newer visual languages, and the advent of object-oriented structures created additional large savings in product cost.

However, when we look at the effort required to produce a single line of source code in any given programming language, we see that software development productivity (measured from start of development through software-system integration) has increased, with little blips and dips, almost linearly at the rate of less than one source line per person-month per year as shown in Figure 1. The aged heuristic, which portions the development effort into design, code, and test (40-20-40 = 100%), shows that eliminating the coding activity entirely leaves 80 percent of the work remaining. The advent of powerful programming environments primarily affects only the coding activity.

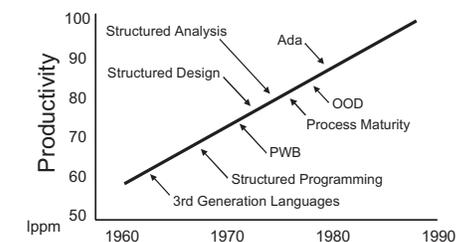
The importance of people shows up in the literature as early as the Hawthorne study by Elton Mayo [3]. This work showed people are primarily driven by esteem and self-actualization, and not by physiological and safety needs (*Rabble hypothesis*). The work of Mayo paved the way for the development of the classic *Theory X – Theory Y* proposal by Douglas McGregor [4] and the Herzberg motivators [5]. W. E. Deming [6] extended these ideas with his *total quality management* work in Japanese and American industry. In spite of the work by these behavioral pioneers and many others, software management remains what Herzberg refers to as a *Theory X* culture. Scott Adams' Dilbert cartoon character and DeMarco's "Covert

Agenda"¹ are two examples of the existence and dominance of this culture.

There are three important dimensions in software management: project, process, and people, as shown in Figure 2 (see page 28). Project was the primary software development focus in the 1960s when the software development discipline was new. The early 1970s brought a shift in focus to the development process. The emphasis on the Waterfall Model in software development, defined and enforced through standards such as Mil-Std-2167A, began a trend that is still flourishing today. The mid-1980s introduced the Software Engineering Institute's Capability Maturity Model[®] as an approach to stabilizing the development process and improving quality and productivity. By focusing energy on process improvement, we can ignore the importance of people in the development process. "Get the process right and people are interchangeable" is a common battle cry. Process is a necessary element of process improvement, but not sufficient to solve the software productivity problem.

Recent developments in teaming concepts led to a focus on management and people issues. The introduction of extreme and agile development methods demonstrated the importance of management

Figure 1: *Software Development Productivity Gains: 1960 to 1990*



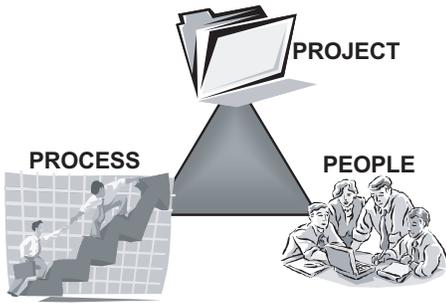


Figure 2: *Project-Process-People Triangle*

and people issues in development productivity and quality. Unless people are considered an important part of the project-process-people triad, software development cost and schedule estimates will continue to be inconsistent and unstable.

Agile Software Development

The Manifesto for Agile Software Development [7], first published Feb. 13, 2001, states the following:

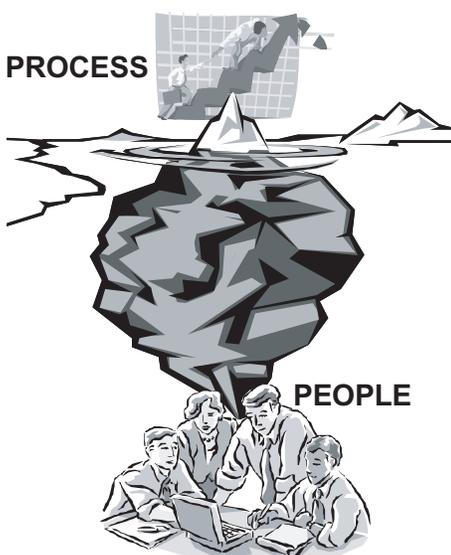
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

That is, while we value the items on the right, we value the items on the left more.

The bulk of the work in the design and test activities (again, 80 percent of the total)

Figure 3: *Process Is But the Tip of the Project Iceberg*



involves a high level of communication that is impacted by the environment, the people, and the organization as well as the development process.

Tonies' Effectiveness Formula

Chuck Tonies introduced the concept that the effectiveness of a software engineer is more than IQ, training, and experience in the 1979 text "Software Engineering" [8]. He pointed out that people in software-related positions in industry work in highly interactive environments. The software development *team* consists of programmers, analysts, test engineers, managers, customers, and users to name a few of the participants.

I italicized team to emphasize the two levels of teams: a group of people assigned to a project (the normal use of the term), and a team in the sense of a professional basketball team. The team in italics suggests the first level: people working as a unit even though their teamness is simply a common charge number and a loose relationship among the players in the project. The second team-level type involves a tight, highly communicative relationship, which is difficult to perceive when all of the members are isolated in cubicles like Dilbert and his cartoon co-worker, Wally.

Members of the development team may be cast in one or more of the roles involved in a project. It is important that people are aware of activities around them and understand their relationship to these activities to achieve their highest effectiveness. They must understand and act in concert with the project management plan, which includes communicating coherently with the other people assigned to the project. However, if the team (either definition) members are unwilling or not motivated to participate in sending or receiving information about the task at hand, the members' technical contribution to the project will be diminished, no matter how gifted or brilliant the individuals are.

Some degree of change is present in almost every development, even the stable projects. The complexity of software development carries with it incomplete and incorrect interpretations of the requirements, interface, and designs. Constant communication among the participants is the only way misunderstanding and errors can be corrected. The danger of emphasizing the process over people and communications is a major point in the Agile Manifesto as shown in Figure 3. Process is only the *tip of the iceberg*, with people making up the bulk of the iceberg.

Tonies postulated that an individual's

value to the development organization in an industrial environment depends on three attributes: computer science skills, communication skills, and management skills. The product gives the effectiveness of the individual in the organization in the following equation:

$$E = CS \times C \times M \quad (1)$$

where:

E = net effectiveness

CS = computer science technical skills (0-1)

C = communication skills (0-1)

M = management skills (0-1)

The effectiveness formula in Equation (1) shows that if any of the three elements are missing, the effectiveness approaches zero. Our experience in the software product-centered environments shows it a realistic model of software engineering performance. It is true that we live in an age of technical specialization. It is also true that software development and engineering is by its nature a complex interactive process that requires careful intensive management. The manager must contribute to the free exchange of information among software development players.

Boehm's list of management problems describes the common software management style for that time – a style that is prevalent today. He states the following:

Poor management can increase software costs more rapidly than any other factor. Each of the following mismanagement actions has often been responsible for doubling software development costs ... Despite this cost variation, COCOMO does not include a factor for management quality, but instead provides estimates that assume the project will be well managed. [9]

G. M. Weinberg [10] extended this discussion by grouping cost impacts described in Boehm's "Software Engineering Economics" to illustrate the relative importance of each impact group. Figure 4 presents Weinberg's results emphasizing the relative importance of organization and management in projecting software development cost. The people impact in Figure 4 represents education, IQ, and experience. Weinberg also points out in this text reference an interesting relationship between the Software Engineering Institute's research publications and relative cost impacts.

The people facet in Figure 2 includes

the most important of the three management facets in terms of productivity and quality gains, and represents the bulk of the communications and management factors that Tonies states.

Traditional Estimating Methods

Traditional estimating methods focus on the technical aspects of software development: project and process. An example of the traditional focus is the intrinsic capabilities of the analysis and programming team members. The principle measures of analyst quality are ability (education, intelligence, and problem solving skills), efficiency and thoroughness, and team communication.

The capability definition deals with capabilities in terms of the *team*; the interpretation generally is a collection of individuals working on a development activity. We abstractly discuss the concept of a team, yet when we look at the project environment, we see a *cube farm*² or a group of people working in isolated offices or widely dispersed locations.

Notice the traditional definition of capability lists cooperation and communication as one of three primary measures, but never mentions the factors that produce esteem and self-actualization; that is, motivation and management.

Extreme Software Estimating Methods

Traditional estimating methods are largely based on Theory X management methods. That is, the soft, or organization aspects of the environment, are difficult to measure and are to be avoided. Boehm said as much in “Software Engineering Economics” [11].

I also avoided the soft factors in the Seer years because of their assessment difficulty. However, I found many projects over a 20-year period that defied reconciling actual cost and schedule results with estimates. It was often impossible to turn the knobs on the estimating models to obtain a cost or schedule match. Once enough data was available to conduct an analysis, I found that all of the abnormally successful projects (higher productivity, etc.) had a common thread – Theory Y managers managed the projects. The problem remaining was to find a way to evaluate organization management. The measures are rather obvious (when outside the box) and easy to measure.

Several factors can be used to assess the capability of an organization: (1) motivation and management style, (2) use

of team methods and proximity of team members, and (3) information flow in the development environment. The remaining traditional capability factors are problem solving ability and programming skills.

Motivation and Management Style

Motivation is one of the most effective and important tasks facing any manager as shown in Figure 5. This task becomes critical in managing a creative, communication-centered activity such as software development. Management style must be considered before other improvement areas since it is the basis for both team concepts and working environment.

Theory X managers manage by control (as directors), closely supervise their employees, and are devoted to structure in both organization and process. Those who search for tools and methods to solve the productivity and quality problems are inherently traditional Theory X personalities. Theory X also underlies the concept that people are interchangeable if the development process is defined and stable.

Human behavior according to Theory Y is quite unlike Theory X behavior. Properly motivated people can achieve their own goals best by directing their efforts toward organizational goals. Theory Y people are motivated at the self-actualization, social, and esteem levels rather than the physiological and safety levels as assumed in Theory X. If the workers have little process ownership, the process is unlikely to change.

The importance of motivation in the development organization is much greater than the space devoted to it here. It is a topic worth additional study by those searching for major gains in quality and productivity.

Team Methods and Proximity of Team Members

A good example of a team approach that did not work is the Chief-Programmer Team [12] introduced by IBM in the 1970s. The team consisted of a chief programmer (creative, good problem solver, intelligent, etc.), a backup programmer (backup and insurance in case the chief programmer became incapacitated or went to the competitor for higher wages), functional specialists (dealt with narrow issues outside the chief programmer’s expertise), coders to implement the architecture and design, and a librarian to keep track of all the stuff being developed.

The team structure was *controlled central-*

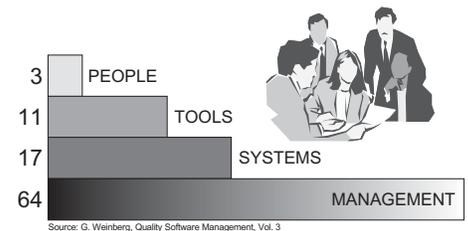


Figure 4: Relative Impacts of Development Environment Elements on Software Costs

ized, meaning top-level problem solving and team coordination are the responsibility of a team leader. Communication tended to be vertical. The chief programmer planned, coordinated, and reviewed all technical activities. This team structure had all the elements necessary to satisfy a high capability rating for the organization.

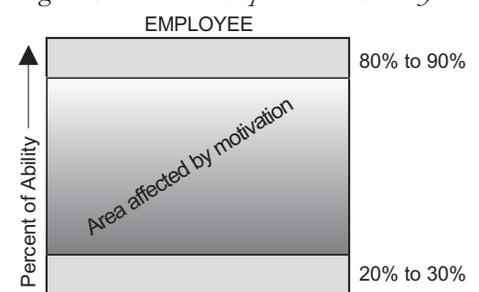
However, the concept failed. Why? First, the team was sensitive to the nature of the chief programmer, which helped to create a low morale environment. Second, the chief programmer team failed to congeal into a team (second definition). It is interesting to note the chief programmer team was still listed as one of the 10 most important ideas in software engineering in 1972 and 1982 by Construx Software Builders, Inc. in a 2002 keynote address³.

Information Flow in the Development Environment

A project’s productivity is tightly related to “how long it takes for information [to flow] from one person’s mind to another’s [13].” There are a number of factors to consider when evaluating information flow (or *convection* as Cockburn describes flow). The obvious measures are distance between developers and noise, including background noise; that is, sound not related to the task at hand. The best representation of good information flow is two people working at a whiteboard. This communication channel contains the best of good communication features: visual cues, visual persistence, sensation of movement, sound, timing (real-time questions and answers), and emotion.

Locating multiple projects (tasks) within the project area creates significant noise.

Figure 5: Motivation Impact on Productivity



Interruptions are also significant flow problems. Other less obvious information flow disrupters are doors and aisles. Telephones and e-mail are useful approaches to decrease information flow. Some programmers are information radiators, such as a programming language consultant located in a project area. Information radiation can take place as information displayed where the developers can readily see it. Walls are common locations for radiators. Note: Web pages are not information radiators.

Other programmers tend to be information sinks. Sinks include people who do not participate in the circulation of information. The infamous lone programmer who works alone, behind a closed door or a closed mind is a typical sink that demonstrates restricted information flow.

Estimating Method Needs

Projects can only be described through input parameters. Estimating tools cannot conjure information that has not been supplied by the estimator. The question we must ask ourselves is, can my estimating tool account for the following:

- Organization and management style (Theory X/Y).
- Motivation.
- Team use.
- Development environment (cube farms, skunk works).

Summary and Conclusions

Traditional estimating methods have worked well in the past because 90 percent of software projects have been developed by traditional organizations. Boehm's assumption that management style and capability could be ignored was generally true in 1981 when COCOMO was initially released. The term *well managed* was an overstatement, and still is, for most development organizations. Consistent management has become a better process descriptor than well managed in the focus era.

Well-managed projects, using the Tonies effectiveness formula, are still the exception rather than the rule. Traditional estimating methods and tools will continue to work in the near future because the style change is risky and very difficult.

Traditional estimating methods also benefit from organization stability. No change in organization style equates to no need for change in estimating approach or tools. Traditional estimating tools use fewer estimating parameters because management and communication effects can be ignored. Last, but not least, there is one estimate area that can be avoided – evaluation of the organization's management

style and effectiveness.

Extreme software estimating methods are needed because accurate software development estimates require more robust estimating models. Ignoring management style and motivation produces high schedule and cost estimates in modern organizations, and produces low estimates in poorly managed organizations. The most important estimating parameter is ignored, or poorly treated, in traditional approaches.

Competitive pressures are forcing organizations to rethink their approaches to effective software development. The number of software projects developed by modern, and possibly agile, organizations is rapidly increasing and driving a need for more estimating flexibility. Extreme estimating methods and tools provide a level of visibility in organization effectiveness that encourages both process and organization improvement. ♦

References

1. Boehm, B. W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981: 486.
2. Weinberg, G. W. The Secrets of Consulting. New York: Dorset House Publishing, 1985: 5.
3. Mayo, Elton. The Human Problems of an Industrial Civilization. New York: The MacMillan Company, 1933.
4. McGregor, Douglas. The Human Side of Enterprise. New York: McGraw-Hill Book Company, 1960.
5. Herzberg, Frederick. Work and the Nature of Man. New York: World Publishing Co., 1966.
6. Deming, W. Edwards. Out of the Crisis. Cambridge, Mass: MIT Press, 1982.
7. Fowler, M., and J. Highsmith. "The Agile Manifesto." Software Development Aug. 2001.
8. Jensen, R. W., and C. C. Tonies. Software Engineering. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1979: 24ff.
9. Boehm 486-7.
10. Weinberg, G. W. Quality Software Management: Congruent Action, Vol. 3. New York: Dorset House Publishing, 1994: 14.
11. Boehm 487.
12. Baker, F. Terry. "Structured Programming in a Production Programming Environment." IEEE Transactions on Software Engineering SE-1.2 (1975): 241-252.
13. Cockburn, A. Agile Software Development. Boston, MA: Pearson Education, Inc., 2002: 77.

Notes

1. *Covert Agenda*: To apply pressure to developers to get them to work longer and harder by the following:
 - Promote an ethic of workaholism.
 - Get project members to sacrifice personal lives.
 - Gull members into accepting hopeless schedules.
 - Hold members' feet to the fire to make them deliver.
2. A *cube farm* is a descriptive term for a facility in which the floor is divided into a large group of cubicles. Another term for this facility organization is a maze. Dilbert works in a cube farm.
3. "The 10 Most Important Ideas in Software Engineering." Construx Software Builders, Inc., 2002 <www.construx.com/docs/open/10MostImportantIdeas-Keynote.pdf>.

About the Author



Randall W. Jensen, Ph.D., is a consultant for the Software Technology Support Center, Hill AFB, with more than 40 years of practical experience as a computer professional in hardware and software development. He developed the model that underlies the Sage and the GAI SEER-SEM software cost and schedule estimating systems. He retired as chief scientist in the Software Engineering Division of Hughes Aircraft Company's Ground Systems Group. Jensen founded Software Engineering, Inc., a software management consulting firm in 1980. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Parametric Estimating in 1984. He has published several computer-related texts, including "Software Engineering," and numerous software and hardware analysis papers. He has a Bachelor of Science, a Master of Science, and a doctorate all in electrical engineering from Utah State University.

Software Technology

Support Center

6022 Fir Ave., Bldg. 1238

Hill AFB, UT 84056-5820

Phone: (801) 775-5742

Fax: (801) 777-8069

E-mail: randall.jensen@hill.af.mil