1987

Software estimating: a description and analysis of current methodologies with recommendations on appropriate techniques for estimating RIT research corporation software projects

Norma Armstrong

Software Estimating: A Description and Analysis of Current Methodologies with Recommendations on Appropriate Techniques for Estimating RIT Research Corporation Software Projects

Thesis Toward Masters Degree in
Computer Systems Management

Norma B. Armstrong

December 1987

Advisors:

_____
Guy Johnson

_____
Rayno D. Niemi

_____
Peter G. Anderson

Software Estimating: A Description and Analysis of Current Methodologies with Recommendations on Appropriate Techniques for Estimating RIT Research Corporation Software Projects

Software Estimating: A Description and Analysis of Current Methodologies with Recommendations on Appropriate Techniques for Estimating RIT Research Corporation Software Projects

Abstract

This thesis investigated, described, and analyzed six current software estimation methodologies. Included were Boehm's COCOMO, Esterling's Work Productivity Model, Putnam's Life Cycle Model and Albrecht's Function Point Determination.

An implementation strategy for software estimation within RIT Research Corporation, a consulting firm, has been developed, based on this analysis. This strategy attempts to satisfy key needs and problems encountered by RIT Research estimators, while providing cost effective, accurate estimates.

Computing Reviews Classification:

D.2.9 Management

K.6.1 Project and People Management

Table of Contents

Chapter 1
Introduction and Historical Perspective

Chapter 2
Presentation of Case Study

Chapter 3
RIT Research Corporation Software Estimation

Chapter 4
Kustanowitz's SLICE Estimation Model

Chapter 5
Esterling's Work Environment Model

Chapter 6
Albrecht's Function Point Effort Predictor

Chapter 7
Walston and Felix's Empirical Estimation Method

Chapter 8
Putnam's SLIM Estimation Method

Chapter 9
Boehm's COCOMO Estimation Method

Chapter 10
Progress and Direction of Estimation Technologies

Chapter 11
Conclusions and RIT Research Corporation Implementation Strategy

Chapter 12
Bibliography

Introduction and Historical Perspective

RIT Research Corporation is a consulting firm which increasingly faces a challenge. In order to gain business within the software development industry, it must accurately estimate the cost and resources required for projects very early in the software development effort.

Because it has a rich resource base of computer scientists, scoping, designing, and implementing software is a function RIT Research can perform very well. Given this competitive advantage, substantially more software development project work would be attractive to RIT Research if accurate estimates were available at project inception.

Across the industry, the task of estimating software projects has challenged project managers for several decades, creating one of the major difficulties in managing software development projects. Farquhar described the significance of the problem in 1970:

> "Unable to estimate accurately, the manager can know with certainty neither what resources to commit to an

> effort nor, in retrospect, how well these resources were used. The lack of a firm foundation for these two judgements can reduce programming management to a random process in that positive control is next to impossible. This situation often results in the budget overruns and schedule slippages that are all too common." [1.1]

During the 1970's, a number of quantitative software estimation models were proposed. Most of these techniques were published because their developers found them to be useful within a particular environment. The authors carefully described their environment, and the limitations and restrictions of their models, realizing that these models may not be useful within other environments.

Even today, estimators typically engage in the largely intuitive process of guessing the number of modules, and the number of statements per module to arrive at a total statement count. Then using some cost per statement relationships which they believe to be appropriate within their organization, they arrive at a total cost for the software development project.

One of the first models proposed by Aron [1.2], attempts to provide empirically determined productivity rates. For 'large' systems, he suggests using an average productivity rate for workers of 20 assembly language source statements per day for 'easy' programs, 10 per day for 'medium' programs, and five per day for 'hard.' These guidelines are often quoted. However, the limitations of this simple model provide its downfall. There is little application within the current software environment for Aron's project definitions of 'large', 'easy', medium', and so on.

Aron's model served as a starting point for productivity rate estimation in a field devoid of many clues. Using his model, project managers could attempt to customize their estimates based on data recorded on their previous projects.

However, as Brooks showed in The Mythical Man-Month [1.3], the underlying assumption of these simplistic estimating techniques is faulty. Most managers viewed the software work to be done as a simple product of a constant productivity rate multiplied by the scheduled time. Brooks argued that manpower and time are not interchangeable, that productivity rates are highly variable, and that there is no industry

standard that can be modified slightly for each application.

Indeed, industry experience validates his point. While small software projects may adhere closely to standardized rates, large projects involving hundreds of thousands of lines of code and multiple programming teams can demonstrate serious departures from constant productivity rates [1.4]. Productivity rate appears to be a function of system complexity.

In 1973, Morin [1.5] studied many of the available software estimating techniques. She concluded,

> "... I have failed to uncover an accurate and reliable method which allow programming managers to solve easily the problems inherent in predicting programming resource requirements. The methods I have reviewed contain several flaws - chief among them is the tendency to magnify errors ..."

The late 1970's saw the publishing of three estimation methods which will be reviewed in detail in this paper. Kustanowitz [1.6] presents a simple, theoretical approach

based on the software life cycle. His technique, which he calls SLICE, guides the estimator through a seven step procedure. For several of the steps Kustanowitz presents theoretical values, which he recommends the estimator customize or change to fit the local environment.

In 1977, Walston and Felix [1.7] presented what has been called a landmark study and the method of programming measurement and estimation that they had developed. By carefully collecting data from over 60 projects and linearizing the results, they built an empirical model. Stressing that their results are preliminary, they isolated 29 factors that affect productivity. Attempting to show the effect of these factors they presented methods of estimating programming projects duration, staff size and computer cost.

Putnam [1.4] proposed a dynamic life-cycle model which assumes a specific manpower distribution over the life of a project. The empirically determined efforts can then be customized through modification of constants, resulting in alteration of the life cycle curve. Putnam's model has been well received, resulting in, a decade later, the production

of a software product, SLIM, which assists the estimator in his work.

While several of the methods presented in the 1970's showed promise, reviewers of the field at that time maintained that no credible, applicable method of software estimation had been identified. In 1981, Mohanty [1.8] noted,

> "Even today, almost no model can estimate the true cost of software with any degree of accuracy."

The 1980's, revealed some exciting new concepts. Three of the most prominent will be reviewed in detail in this paper.

Albrecht [1.2, 1.9] presents a 'function point' method of estimation. He proposes examination of the software in terms of function points, instead of the standard 'lines of code' utilized by most other methods. Function points are functions of the system such as number of inputs, outputs, user inquires, and files, each of which are weighted by complexity. Once the number of function points have been determined he presents an 'early bridge' to a determination of lines of code (LOC). From there, the estimator can

utilize any method for determining the anticipated resource levels and schedule. Albrecht anticipated that empirical studies would be published in the ensuing years that would validate the application of function points across various applications, removing the need to bridge to LOC.

A work productivity model was presented by Esterling [1.10] early in the 1980's. This unique approach attempts to account for the microscopic characteristics of the work environment. The model examines such factors as the number of people working on a project, average amount of overtime, number and duration of worktime interruptions, average time to regain thought, and direct and indirect costs for workers. Esterling contends that the most productive work occurs during overtime because administrative and other 'nonproductive' functions consume standard hours.

In 1981, Boehm [1.11] presented a series of estimation tools called COCOMO. The most complex of these methods utilizes an empirically determined base effort expressed in lines of code, adjusted by a set of cost drivers which are subjective assessments of product, hardware, personnel and project attributes. These estimates are then allocated across the

project phases of analysis, design, programming, integration, test, etc. providing effort and scheduling information.

COCOMO is an impressive empirical model, probably the most comprehensive published to date. Yet it is used with caution. Boehm described a software estimation model as doing well if it can estimate software costs within 20% of actual, 70% of the time, within its own environment. Other people believe that software estimation methods which have errors of up to 50% after completion of the functional description and 10% after system design are still useful.

This thesis will investigate, describe and analyze six software estimation methodologies ranging from Kustanowitz's simple SLICE model to Boehm's COCOMO.

The analysis of each estimation methodology will consist of study of available case histories, independent critiques available within the literature, an assessment of requirements and implication for application both generally and within the RIT Research Corporation environment, a effort estimate when applied to an RIT Research Case Study, critique and suggestions for improvement. The Case Study

is be an actual software project undertaken within RIT Research.

Once each methodology has been described and evaluated, an assessment of the overall progress and direction of estimation technologies is presented. Suggestions for improvement are included.

Finally, an implementation strategy within RIT Research is be proposed. This strategy consists of logical, careful application of the best software estimation model found. Several concepts found in the other models reviewed are recommended for further consideration and investigation.

References

[1.1] Farquhar, J. A., "A Preliminary Inquiry into the Software Estimation Process," Technical Report, AD F12 052, Defense Documentation Center, Alexandria, VA, August 1970.

[1.2] Albrecht, A. J., "Measuring Application Development Productivity," Proc. IBM Applic. Deve. Symposium, Monterey, CA, October 1979, pp 83-92.

[1.3] Brooks, Frederick P., Jr., Mythical Man-Month, Addison-Wesley Publishing Co., 1975.

[1.4] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Trans. Software Engineering, vol 4, no 4, 1978, pp. 345-361.

[1.5] Morin, L. H., "Estimation of Resources for Computer Programming Projects," MS Thesis, University of North Carolina, Chapel Hill, NC, 1973.

[1.6] Kustanowitz, A. L., "System Life Cycle Estimation (SLICE): A New Approach to Estimating Resources for Application Program Development," Presented at COMPSAC '77, IEEE Computer Soc. 1st Int. Computer Software and Applications Conf., Chicago, IL November 8-10, 1977.

[1.7] Walston, C., and C. Felix, "A Method for Programming Measurement and Estimation," IBM Systems Journal, vol 16, no 1, 1977, pp 54-73.

[1.8] Mohanty, S. N., "Software Cost Estimation: Present and Future," Software Practice and Experience, Vol II, 1981, pp 103-21.

[1.9] Albrecht, A. J., and J. E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," IEEE Trans. Software Engineering, November 1983, pp. 639-648.

[1.10] Esterling, R. "Software Manpower Costs: A Model," Datamation, March 1980, pp. 164-170.

[1.11] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

Presentation of Case Study

The Case Study is a project started and completed during 1987 within RIT Research Corporation. Both the client and application were new to RIT Research. And the work was exciting and challenging for the project team.

The first task was to develop a reasonable estimate of the effort for the work. Both the manpower estimates and the schedule needed to be determined. Once this information is available, pricing any RIT Research project is relatively straightforward.

The time and effort involved in producing this estimate should be fairly low, because RIT Research does not charge clients for proposal development. The estimate must be fairly accurate because RIT Research proposals are fixed price.

The client requested that RIT Research define the functionality and design for a new programming tool. This tool was to generate a test environment, for use within an

applications development area outside of the client's direct authority.

The function of this system was not clearly defined. The client's main objective was to reduce the hardware needed to maintain the present programming environment by installing a more useful, smaller test environment.

The system would interface with a massive data base via a mainframe data base manager, and would be written in a combination of a data base language and, probably, PL/1. The client was anticipating radical changes within the programming and test environment within the next few years, so the system must be defined to support such a transition.

The programmers's requirements were not defined. The system would need to be easy and quick to use. Other individuals had an interest in the functionality and system design, such as the system managers reporting directly to the client.

The system would be very resource constrained. Disk space, memory, processing power would be limited while processing speed and effectiveness must be maximized.

The system should be fully documented using a standardized methodology of RIT Research's choice.

The client would like the functional specification and system design phases of the project completed within 12 - 15 weeks, with the entire system installed within 9 months.

After several conversations with the client, RIT Research assembled a potential project team, and generated more information for the estimate.

The project team consisted of highly skilled individuals, with experience in comparable projects, the data base manager and PL/1. The client's specific environment was still unfamiliar to the team. The team members were available to work as much time as needed on the project up to 40 hours/week.

The ballpark estimate of 13,000 lines of code for the system was developed, based on previous similar projects undertaken by project team members. The error on this project metric was estimated to be $\pm$ 40%.

A ballpark estimate was also developed that development of the functional specification and design was 30% $\pm$ 10% of the complete system development effort.

Description of RIT Research Corporation Estimation Method

For lack of a better method of estimating software projects, RIT Research estimators utilize a standard, intuitively developed technique. Projects are decomposed into their smallest components, and then effort estimates are developed based on experience and what the estimator 'feels' the effort should involve.

Application of Method to Case Study

The Case Study, which involved the two major components of the functional specification and detailed product design was decomposed into the following, roughly chronological activities and tasks:

Activity or Task	Man-days
Functional specification	
Interview and define objectives and requirements of:	
Management	3
Client	4
User	6
Define hard/software configuration	4
Define user environment	7
Define logical and physical test environment views	6
Develop black box functional definition	7
Define user interface	6
Define top level design	7
Document functional specification	6
Discuss functional specification with client	2
Total Man-days for Functional Specification	58
Design	
Develop data flow design	7
Develop module specification	12
Define data types	3
Document design	2
Discuss design with client	3
Total Man-days for Detailed Design	27
Total Man-days for Project	85

This 85 man-days effort was scheduled to take 12 weeks to complete due to the client's schedule.

Critique

There are substantial weaknesses in this estimation method. Most of the effort estimation is subjective, and because the activities are within a unique environment even experience with similar efforts does not offer much assistance.

Very little information concerning the actual project is used, except in a general, "Is this bigger or smaller than projects we have done?" manner.

The estimates are not repeatable, trackable, or applicable to other work. The best that can be said for this technique, and the reason it is in use, is it is familiar to the estimators, is fairly inexpensive to apply, and is more accurate than some other, equally simplistic methods that have been tried.

Kustanowitz's SLICE Estimation Model

Overview

In 1977, Alvin Kustanowitz presented a technique for estimation of manpower requirements for software implementation [4.1]. He also reviewed available estimation methods finding a great need for a simple method to estimate manpower requirements of software for business applications.

He named his technique SLICE, for System Life Cycle Estimation. Prior to this, few if any estimation techniques considered the software life cycle. This became his key to customizing the estimate to the environment. By accurately describing the project profile, or the phases of the project, and incorporating this information into the estimate, Kustanowitz felt estimates would be considerably more accurate.

SLICE describes a step-by-step method for software estimation which is simple to apply and easy to adjust as historical data provides more accurate information. By factoring in the

project life cycle, or project profile as he calls it, productivity rates, and the estimated LOC, the estimate can be created rapidly .

Description of SLICE

Kustanowitz provides a step-by-step approach on how to use SLICE.

STEP 1: DESCRIBE YOUR PROJECT LIFE CYCLE

Based on the expected project plan, the estimator should list the steps involved in some detail. Some possible steps include:

Planning	Coding
Feasibility Study	Compilation
Requirements Definition	Data Base Creation
Conceptual Design	File Conversion
Program Design	Unit Test
Data Base Design	Integration Test
Program Specifications	System Test
Program Flowcharting	Documentation

This list then constitutes a project profile which is customized to the environment but is not impacted by project size. Other projects completed within the same environment should display a similar project profile.

A typical project profile involves 6 - 10 phases or steps. An example, provided by Kustanowitz, for a large, on-line system has the project profile:

1. Functional Requirements Definition
2. Conceptual Systems Design
3. System Design
4. Program Specification
5. Coding
6. Unit Test
7. System Test

STEP 2: ASSIGN PERCENTAGES TO EACH OF THE PHASES OF YOUR SYSTEM LIFE CYCLE

By assigning a distribution of effort to each project step, the estimator develops a life cycle profile. Ideally this life cycle profile is based on historical data for similar projects within the same environment. Without such data, Kustanowitz suggest the use of 'rough guesstimates' with change and refinement of the model as the project progresses.

Continuing with the example, Kustanowitz presents a possible life cycle profile:

1. Functional Requirements Definition	18%
2. Conceptual Systems Design	9%
3. System Design	18%
4. Program Specification	10%
5. Coding	6%

6. Unit Test	9%
7. System Test	30%

STEP 3: SELECT PRODUCTIVITY FACTORS

The estimator must determine the productivity rate, typically average number of instructions per day, that the programmers can achieve. Once again, Kustanowitz recommends the use of historical data; on earlier projects with the appropriate environment, divide lines of code (LOC) by number of man-days.

Caution is advised for LOC determination. The estimator must consistently examine the same base, for example, source, executable source, pages, or machine instructions.

While a guesstimate of productivity rate leaves most estimators feeling uncomfortable, Kustanowitz believes it is mandatory. Also productivity is relatively easy to determine once one project has been completed.

For his example, the productivity factor was concluded to be 18 lines of code per day for COBOL.

STEP 4: ESTABLISH ESTIMATING BASIS

The next step is to determine during which steps of the project life cycle the productivity rate is applicable. Since productivity rate is determined to be LOC per day, the rate is only applicable during programming activities.

Continuing the example, only 25% of the project life cycle will display this productivity rate:

4. Program Specification	10%
5. Coding	6%
6. Unit Test	9%

STEP 5: ESTIMATE THE TOTAL NUMBER OF INSTRUCTIONS IN THE FINISHED SYSTEM

Determination of LOC can only be done after initial design work has been completed. Kustanowitz states,

> ".. if you don't have the slightest idea of where to begin [estimating LOC], you shouldn't be estimating yet."

Of course, the estimate should be refined as more work is done on the specification. A re-estimate is suggested at the

completion of detailed program design and after program specifications and/or flowcharts are done.

Kustanowitz believes that any experienced programmer should be able to predict LOC from previous experience, knowledge of other programs of various sizes, and a look at the specifications or flowchart or narrative of the proposed new program. Once again, a historical basis for estimating is needed, but in this case the estimate will always be relatively subjective.

Assume our example has a 100,000 line of code estimate.

STEP 6: CALCULATE TECHNICAL MAN-DAYS

Divide LOC by the productivity factor to determine the technical man-days required. Then divide that result by the estimating basis identified in STEP 4 above to determine the total number of man-days required on the project.

Following the example:

100,000 LOC / (18 LOC/man-day) = 5,600 man-days

5,600 man-days / 25% applicable time = 22,400 man-days

STEP 7: TRANSLATE INTO TIME-PHASED PROJECT PLAN

The final step is the determination of the manpower loading and schedule. Kustanowitz proclaims that "users of this technique have found that it works best for a 'squareroot' manpower vs. time distribution." In other words, find the square root of the required man-months to determine both the number of personnel and the scheduled number of months. He believes that, with a little experience, estimators can become quite good at adjusting these numbers based on time frame and manpower constraints.

The example concludes with:

22,400 man-days/(20 man-days/man-month)=1,120 man-months

Sqrt (1,120) = approximately 34

The project is estimated to require 34 people for 34 months.

Implementation Requirements

SLICE is the simplest method of software estimation reviewed in this paper and considered for implementation within RIT Research. The seven step procedure is very straightforward to apply and can be done with little trouble.

While it is possible to apply SLICE without historical records, the estimate will surely become more accurate as environmental characteristics are factored into the calculations.

Ideally, the estimator will be familiar with several other similar projects that have been successfully completed in the recent past. All productivity tools, such as compilers and debugging software, that will be used in the new project will have been included in the previous projects. The skill levels of the personnel on earlier projects will be comparable to that expected on the new project team.

From these earlier efforts, the estimator should deduce a project profile, a system life cycle, productivity factors, and resource and scheduling guidelines. Also, either the estimator or a knowledgeable computer scientist should have adequate background to produce a good estimate of lines of code.

Application of SLICE to the Case Study

Unfortunately, the Case Study did not provide much of the information required for a SLICE estimate. The historical

data is not available, so guesstimates will be necessary for most of the critical factors in the analysis.

Because of the lack of historical data, the estimate will have a fair amount of risk. A simple way to view the sensitivity or risk involved in the estimate is to calculate both 'expected' and 'worst case' values as is done below.

The relevant details provided by the Case Study description are:

LOC = 13,000

Schedule = 9 months

STEP 1: DESCRIBE YOUR PROJECT LIFE CYCLE

Expected	Worst Case
Requirements Definition	Requirements Definition
Conceptual Design	Conceptual Design
Program Design	Program Design
	Program Specifications
Coding	Coding
Integration Test	Integration Test
Documentation	Documentation

STEP 2: ASSIGN PERCENTAGES TO EACH OF THE PHASES OF YOUR SYSTEM LIFE CYCLE

Expected		Worst Case	
Requirements Definition	15%	Requirements Definition	10%
Conceptual Design	10%	Conceptual Design	5%
Program Design	10%	Program Design	15%
		Program Specifications	10%
Coding	40%	Coding	25%
Integration Test	20%	Integration Test	25%
Documentation	5%	Documentation	10%

STEP 3: SELECT PRODUCTIVITY FACTORS

Expected	Worst Case
20 LOC/day for PL/1	20 LOC/day for PL/1
15 LOC/day for DB1	5 LOC/day for DB1
Code is:	Code is:
60% PL/1	25% PL/1
40% DB1	75% DB1
Average Productivity Rate:	
18 LOC/day	8.75 LOC/day

STEP 4: ESTABLISH ESTIMATING BASIS

The productivity rate is applicable to the following project phases:

Expected		Worst Case	
Program Design	10%	Program Design	15%
		Program Specifications	10%
Coding	40%	Coding	25%
Integration Test	20%	Integration Test	25%
	70%		75%

STEP 5: ESTIMATE THE TOTAL NUMBER OF INSTRUCTIONS IN THE FINISHED SYSTEM

LOC = 13,000

STEP 6: CALCULATE TECHNICAL MAN-DAYS

Expected

13,000 LOC / (18 LOC/man-day) = 722 man-days

722 man-days / 70% applicable time = 1,031 man-days

Worst Case

13,000 LOC / (8.75 LOC/man-day) = 1,486 man-days

1,486 man-days / 75% applicable time = 1,981 man-days

STEP 7: TRANSLATE INTO TIME-PHASED PROJECT PLAN

In this case, a nine month schedule is imposed on the project for all activities.

Expected

1,031 man-days/(20 man-days/man-month)= 52 man-months

52 man-months / 9 months = 6 people

The entire project is estimated to require 6 people for 9 months.

Worst Case

1,981 man-days/(20 man-days/man-month)= 99 man-months

99 man-months / 9 months = 11 people

The entire project is estimated to require 11 people for 9 months.

However, in the Case Study, only the functional specification and program design are accomplished, therefore only these efforts need to be calculated. The client's schedule of 12 weeks for this project also must be achieved:

Expected		Worst Case	
Requirements Definition	15%	Requirements Definition	10%
Conceptual Design	10%	Conceptual Design	5%
Program Design	10%	Program Design	15%
	35%		30%

Thus,

Expected

52 man-months * 35% = 18.2 man-months

18.2 man-months / 3 months = 6 people

The project is estimated to require 6 people for 3 months.

Worst Case

99 man-months * 30% = 33 man-months

33 man-months / 3 months = 11 people

The project is estimated to require 11 people for 3 months.

Thus, there is a considerable difference between the expected and worst case estimates. The most impact was due to the difference in productivity rates for the two estimates. Without historical data, the estimator has little basis for analyzing the risk in accepting one estimate over the other.

Forced to provide a number, this author would weight the estimates 75% expected, 25% worst case for a final estimate of just over 7 people for the entire 9 month project or, similarily since the effort is constant, 7 people for the 3 month design project of the Case Study.

Critique

SLICE provides an easy-to-use method of software estimating. In essence, it is a formalized method of estimating in the same fashion as is presently being done with RIT Research.

By describing the approach as he does, Kustanowitz clearly outlines easy to obtain, historical data that is needed to improve the estimates: project profiles, project life cycles and productivity rates. Unfortunately, RIT Research projects are very diverse with each project being conducted in a different environment. Collection of this type of data would be of limited use.

Once the estimate has been calculated, it is difficult to evaluate the degree of risk or the potential error.

Summary

SLICE is useful as a simple, ballpark estimation tool where no other method is known or available. Also, when an organization undertakes many, similar software projects with tightly controlled environmental characteristics, SLICE

provides a useful step-by-step method of obtaining fairly accurate estimates.

However, most software projects involve factors that cannot be easily and clearly incorporated into the estimation technique. Such items tend to get rolled up into an

estimated reduction in productivity rates, or increase in LOC. In these cases, SLICE is too simplistic to be of much use.

Reference

[4.1] Kustaniwotz, A. L., "System Life Cycle Estimation (SLICE): A New Approach to Estimating Resources for Application Program Development," Presented at COMPSAC '77, IEEE Computer Soc. 1st Int. Computer Software and Applications Conf., Chicago, IL November 8-10, 1977.

Esterling's Work Environment Model

Overview

Bob Esterling's work environment model [5.1] examines the microscopic characteristics of the work environment. His model is based on the premise that manpower utilization is the easiest factor for management to vary toward affecting project cost and time schedule.

By modeling the characteristics of manpower utilization including overtime hours, overtime costs, and number, duration and recovery time from work interruptions, Esterling provides a tool to evaluate the effectiveness, cost and schedule for different environments. His model predicts that, for example, project schedules cannot be successfully accelerated by adding more people because communication becomes costly, but can succeed if overtime work is encouraged.

Esterling's Work Environment Model

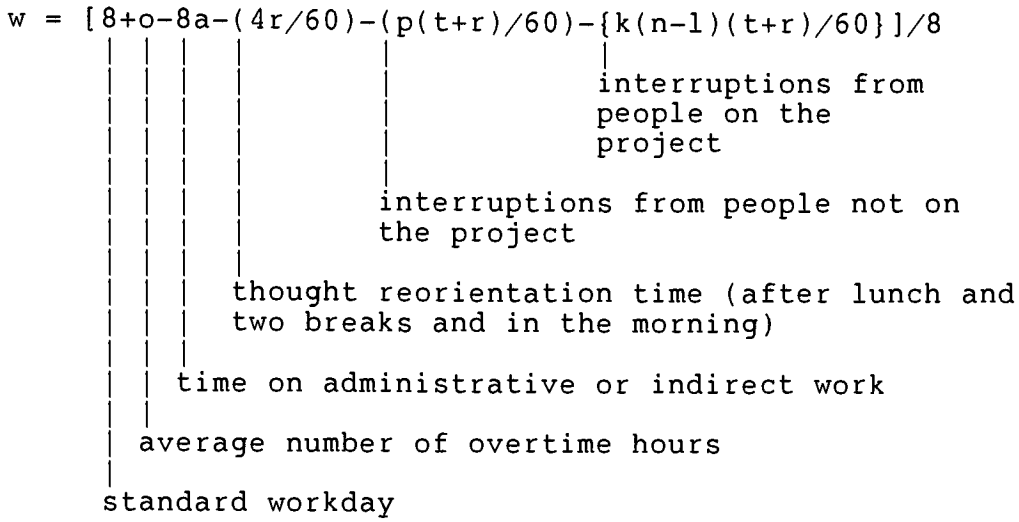
Description of Esterling's Model

Esterling presents a series of equations which attempt to quantify the relationships among costs, time and manpower, in order to describe optimum number of people on a project.

Examining any work environment on the microscopic, individual level, the ultimate productivity, cost and schedule for work is affected by the number of people on a project and how much time they spend actually working. Esterling contends that the best and most productive programming often occurs in the 'wee hours of the morning' when interruptions are minimized allowing the concentration and continuous thought necessary to this creative process. By way of comparison, a factory worker is rarely interrupted and has little difficulty regaining thought after interruptions. In this case, overtime work may not be overly productive.

Esterling's model begins by examining the average fraction of useful time contributed by an employee during the day, w:

Esterling's Work Environment Model



where:

n = number of direct workers on the project

o = average number of overtime hours per workday per person

a = average fraction of workday spent on indirect work

t = average duration of work interruptions (minutes)

r = average recovery time after interruption (minutes)

k = number of interruptions/day from project personnel

p = number of interruptions/day from other causes

w = fraction of useful working time per day per person

Then, given that the number of effective man-days per workday is nw, he examines the required calendar time:

$$T = 7/(5nw)$$

where:

w = fraction of useful working time per day per person

T = ratio of calendar time to man-days required

n = number of direct workers on the project

The labor cost per workday, c, is given by:

$$c = n (8 + 8i + od) s$$

where:

n = number of direct workers on the project

o = average number of overtime hours per workday per person

s = average personnel base salary per hour

i = indirect (overhead) as fraction of base pay

d = differential pay for overtime as fraction of base pay

c = labor cost per workday

Esterling also presents some measures of efficiency, including cost efficiency:

$$e = nw / c$$

where:

w = fraction of useful working time per day per person

c = labor cost per workday

e = effective working time per dollar

n = number of direct workers on the project

The ratio of project cost to man-days is:

$$C = c / nw$$

where:

n = number of direct workers on the project

w = fraction of useful working time per day per person

c = labor cost per workday

C = ratio of project cost to direct project man-days

And the project cost-completion time product, P, another measure of total project effectiveness, is:

$$P = CT = (c / nw) * (7 / (5nw))$$

where:

w = fraction of useful working time per day per person

T = ratio of calendar time to man-days required

n = number of direct workers on the project

c = labor cost per workday

C = ratio of project cost to direct project man-days

P = project cost-completion time product

In order to examine this model, Esterling provides typical model parameters for a factory worker and a programmer and optimistic and pessimistic programmer environmental parameters.

Values for Model Parameters

Parameter	Range	Factory Workers	Programmers		
			Optimistic	Typical	Pessimistic
a	0 - 0.5	0.0	0.05	0.10	0.15
t	1 - 20	3	3	5	10
r	5 - 10	0.5	0.5	2.0	8.0
k	1 - 10	1	2	3	4
p	1 - 10	1	1	4	10
i	1 - 3	0.2	0.2	0.5	1.0
d	1 - 2	1.5	1.0	1.0	1.5

Factory workers spend no time during the day on indirect work, are rarely interrupted, take little time to recover from interruptions, carry little overhead, and earn time-and-a-half for overtime. In contrast, with the programmers, interruptions are more frequent, harder to recover from, and they bear more overhead. Esterling describes the 'optimistic' programmer/programming environment as less intrusive and disruptive to concentration than the 'pessimistic' scenario.

In order to further examine the model, this example can be expanded. Assume the additional data:

s = $10.00 per hour

nw = 20 = required man-days for project completion

	Factory Workers		Programmers					
			Optimistic		Typical		Pessimistic	
n	o (hrs)	c ($)	o (hrs)	c ($)	o (hrs)	c ($)	o (hrs)	c ($)
7	15.3	2278	16.1	*1795	18.4	*2125	26.8	*3933
8	12.5	2268	13.3	1833	15.9	2228	25.1	4296
9	10.3	2259	11.2	1872	14.0	2338	24.1	4695
10	8.6	2253	9.5	1914	12.6	2455	23.5	5130
11	7.2	2247	8.2	1958	11.5	2579	23.3	5601
12	6.1	2244	7.1	2005	10.6	2710	23.3	6108
13	5.1	*2242	6.2	2054	9.9	2848	23.4	6651
14	4.3	2243	5.4	2105	9.4	2993	23.8	7230
15	3.6	2244	4.8	2159	9.0	3145	24.2	7845
16	3.0	2248	4.2	2215	8.7	3304	24.7	8496
17	2.4	2253	3.8	2273	8.4	3470	25.3	9183

* least costly alternative

Because of these microscopic work environment characteristics, the factory worker should perform this job in 13 man-days (or with 13 people in one day), each man-day involving 5.1 hours of overtime to achieve a minimum job cost. For all three types of programmers, interruptions are so costly that in order to minimize cost most of the work has to be done during overtime. In each case the total manpower

cost is least with one man-day of effort and over 150 hours of overtime. The main factor in this ridiculous conclusion is the relative cost of overtime with pure, concentrated effort against the standard work environment.

Obviously, the cost of overtime salaries has a great impact on this example.

	Programmers								
	Optimistic			Typical			Pessimistic		
		d=1.5	d=2.5		d=1.5	d=2.5		d=1.5	d=2.5
n	o	c	c	o	c	c	o	c	c
1	153	2383	3908	153	*1654	3955	157	*2511	*4078
2	73	2370	3822	74	1715	3928	78	2658	4217
3	46	2360	3742	47	1783	*3918			
4	33	2355	3668	35	1858	3925			
5	25	*2352	3600						
6	20	2353	3537						
7	16	2357	3481						
8	13	2365	3430						
9	11	2376	3385						
10	10	2391	3345						
11	8	2410	3312						
12	7	2432	3285						
13	6	2457	3263						
14	5	2486	3247						
15	5	2518	3237						
16	4	2554	*3233						
17	4	2593	3234						

* least costly alternative

Examining each of the programmers with two different overtime salary structures, the results change even if they do not become very feasible. The 'optimistic' programmer with double-time-and-a-half overtime pay only needs to work four hours of overtime per man-day in order to minimize project cost. All other conclusions are impossible, however this example has shown the underlying functionality of Esterling's model.

Esterling did not intend for this model to be fully functional in this simple form. Many of the underlying assumptions he placed on the model are too restrictive to permit practical use. However, without these assumptions, the basic cause and effect of these simple work environment characteristics might become obscured. His assumptions include:

1. There is no learning period adjustment for new people joining a project.

2. Most people want to do a good job at their assigned task.

3. There are no work interruptions, only productive, non-interactive work, during overtime.

4. Overtime is voluntary, and enjoyed by all project team members.

Implementation Requirements

Esterling's work environment model requires substantial information about the environment within which the project will be conducted. The time study nature of this data for the model parameters makes it difficult and expensive to accurately collect.

The simplifying assumptions Esterling has incorporated make the results of application of the model suspect. Other data should be factored into the model to support variation from Esterling's assumptions. For example, uninterrupted concentration during sustained overtime cannot be possible. However, incorporating data concerning the lesser effect of self-induced interruptions during overtime periods, would be fairly straightforward.

Additionally, Esterling's model requires a man-day estimate for the software project. This number is very difficult to estimate. The value of Esterling's model lies in focusing the estimator/project manager's efforts on staffing the project and structuring the environment in a way to reduce cost and increase worker productivity and satisfaction.

Application of Model to the Case Study

The Case Study provides little data which is of use in application of Esterling's model. The lines of code for the project have been estimated to be 13,000. However, no man-day estimate has been developed.

No pay is given for over 40 hours of work at RIT Research Corporation, so the routine replacement of overtime work for interrupted daytime work is unlikely.

The only useful portion of the model, given the information available for the Case, appears to be a calculation of w, or average fraction of useful time of each working day, that programmers can actually concentrate and work most productively. For lack of a better metric, assume

Esterling's 'typical' programmer characteristics. Because number of project team members is not known, an average number of interruptions from people on the project has also been assumed.

Assumed Values for RIT Research Model Parameters

Parameter	Programmers Typical
a	0.10
t	5
r	2.0
p	4
k(n-1)	10
o	0

$$w = [8+o-8a-(4r/60)-(p(t+r)/60)-\{k(n-1)(t+r)/60\}]/8$$

$$w = (8 - 0.8 - 0.13 - 0.47 - 1.17) / 8$$

$$w = 67.9\%$$

where:

o = average number of overtime hours per workday per person

a = average fraction of workday spent on indirect work

t = average duration of work interruptions (minutes)

r = average recovery time after interruption (minutes)

k(n-1) = number of interruptions/day from all project personnel

p = number of interruptions/day from other causes

w = fraction of useful working time per day per person

Thus, Esterling's model predicts that approximately 30% of the programmer's time will be spent on activities which are not fully constructive. This should cause the estimator, once the man-day estimate is known to extend this estimate to take into account these environmental factors.

Critique

This model is too simple to be of much use. Esterling's simplifying assumptions handicap the model for general use.

It requires a great deal of detailed information which normally is not available. If general data is used, such as in application of the Case above, the estimates could be substantially in error. Unfortunately, collection of the

data for a specific environment and application is costly and tedious.

The greatest flaw in this model is that it does not assist in the larger task of determining the size of the project as a whole. However, once these, more useful numbers have been estimated, Esterling's model can offer some generalized assistance in evaluating the efficacy of a local programming environment.

Summary

Bob Esterling has offered some novel ideas to the software estimation field. Normally the factors he examines within his model are ignored, probably to the detriment of the software development effort.

Application of Esterling's model is best done once the software estimation process is well along, and most commonly determined estimates have been made. At that point even if only general parameters are utilized, the model can be a useful tool in the evaluation of the characteristics of the programming environment.

Reference

[5.1] Esterling, R. "Software Manpower Costs: A Model," Datamation, March 1980, pp. 164-170.

Albrecht's Function Point Effort Predictor

Overview

Allan Albrecht [6.1, 6.2] developed an indirect measure of productivity for software development he termed the function point method. Function points are derived using an empirical relationship based on countable measures of the software's information domain and subjective assessments of software complexity.

Function points are a software metric similar to lines of code. Albrecht postulated that some of the problems with lines of code metrics, such as the fact that they are very programming language dependent, can be avoided with functionally derived metrics. Also, the functionality of a proposed project is understood earlier in the development process than lines of code.

Albrecht does not provide a model for utilizing function points to predict software effort. He anticipates that over time more data will be collected and function points will come into more general use.

Albrecht's Function Point Effort Predictor

Description

Albrecht's function point method of quantifying project development effort does not focus on project size, as most other metrics do. Instead function points are determined by examination of program 'function' or 'utility.'

Function points are computed in three general steps:

1. Count and classify the five user function types
2. Adjust for processing complexity
3. Make the function points calculation

STEP 1: Count and classify, to three levels of complexity, the five user function types that are made available to the user. This includes all portions of the system made available through the design, development, testing or support efforts of the development enhancement or support project team. The user function types and their classifications are:

1. External input type

   User data or user control input that enters the external boundary of the application being measured, and adds or changes data in a logical internal file. It should be

counted uniquely if it has a different format or if the external design requires a processing logic different from other external input of the same format. This type includes external input that enter directly as transactions from the user, and those that enter as transactions from other applications, such as input files of transactions.

The complexity levels for external input are:

Simple - Few data elements are included and few logical internal files are referenced. User human factor considerations are not significant to the design.

Average

Complex - Many data elements are included and many logical internal files are referenced. Human interface considerations are significant.

2. External output type

Each unique data or control output that leaves the external boundary of the application. It should be

counted uniquely if it has a different format or if the external design requires a processing logic different from other external output types of the same format. This type includes reports and messages to the user, and reports and messages to other applications such as output files of messages.

The complexity levels are similar to external input types with the following additional complexity definitions for reports:

Simple - One or two columns. Simple data element transformations.

Average - Multiple columns with subtotals, Multiple data element transformations.

Complex - Intricate data element transformations. Multiple and complex file references to be correlated. Significant performance considerations.

3. Logical internal file type

Each major logical group of user data or control information including each logical file or group of data that is generated, used and maintained by the application.

The complexity levels are:

Simple - few record types. Few data element types. No significant performance or recovery considerations.

Average

Complex - Many record types. Many data element types. Performance and recovery are significant considerations.

4. External interface file type

Each major logical files passed or shared between application.

The complexity levels are similar to logical internal files types described above.

5. External inquiry type

Each unique input/output combination, where an input causes and generates an immediate output. This type includes external inquiry types that enter directly from the user, and those that enter from other applications.

The complexity levels are determine to be the greater of the complexity for the input and output part, as described in the external input type and external inquiry type respectively, above.

With the determination of the count and complexity classification for each user function type, Albrecht provides a form to determine total unadjusted function points. Each complexity has a weighting factor which has been empirically determined.

Albrecht's Function Point Effort Predictor

	Complexity			
Description	Simple	Average	Complex	Total
External Input	__* 3 =__	__* 4 =__	__* 6 =__	___
External Output	__* 4 =__	__* 5 =__	__* 7 =__	___
Logical Internal File	__* 7 =__	__* 10 =__	__* 15 =__	___
External Interface File	__* 5 =__	__* 7 =__	__* 10 =__	___
External Inquiry	__* 3 =__	__* 4 =__	__* 6 =__	___
	Total Unadjusted Function Points (FC)			___

STEP 2: Calculate the total degree of influence of the processing complexity. For each of the 14 general characteristics that follow, the degree of influence is estimated using the scale:

Degree of Influence

Not present, or no influence if present = 0

Insignificant influence = 1

Moderate influence = 2

Average influence = 3

Significant influence = 4

Strong influence, throughout = 5

The 14 general characteristics are:

1. Data and control information used in the application are sent or received over communication facilities. Terminals connected locally to the control unit are considered to use communication facilities.

2. Distributed data or processing functions are a characteristic of the application.

3. Application performance objectives, in either response or throughput, influence the design, development, installation, and support of the application.

4. Heavily used operation configuration is a characteristic of the application. The user wants to run the application on existing or committed equipment that will be heavily used.

5. The transaction rate is high and influenced design, development, installation and support of the application.

6. On-line data entry and control functions are provided in the application.

7. The on-line functions emphasize end user efficiency.

8. The application provides on-line update for the logical external files.

9. Complex processing is characteristic of the application such as many control interactions and decision points, extensive logical and mathematical equations, and much exception processing.

10. The application has been specifically designed, developed and supported for reusability in other applications, and at other sites.

11. Conversion and installation ease are characteristics of the application.

12. Operation ease is a characteristic of the application. Protective start-up, back-up and recovery

procedures were provided and manual activities are minimized.

13. The application has been specifically designed, developed and supported to be installed at multiple sites for multiple organizations.

14. The application has been specifically designed, developed, and supported to facilitate change.

Albrecht provides a form for determination of the total degrees of influence for these 14 characteristics:

Albrecht's Function Point Effort Predictor

Characteristic	Degree of Influence None <----------> Strong 0 1 2 3 4 5
Data Communications	
Distributed Functions	
Performance	
Heavily Used Configuration	
Transaction Rate	
On-line Data Entry	
End User Efficiency	
On-line Update	
Complex Processing	
Reusability	
Installation Ease	
Operational Ease	
Multiple Sites	
Facilitate Change	

Total Degree of Influence (PC) _____

STEP 3: Calculate the function points.

PCA = 0.65 + (0.01 * PC)

where:

PCA = Processing Complexity Adjustment

PC = Total Degree of Influence determined in step two

and

FP = FC * PCA

where:

FP = Function points measure

FC = Total unadjusted function points, determined in step one

PCA = Processing complexity adjustment

Once function points (FP) have been calculated they can be used as a measure of software productivity, quality and other attributes similar to lines of code (LOC).

Implementation Requirements

Albrecht's work provides a framework for calculation of a software metric which may, in the long run, provide better estimates of effort than LOC. The basic calculation of FP is relatively straightforward, and can be accomplished in a crude form very early in project development and refined as the project detail becomes available.

The counts and classifications that are required to compute FP require examining software in a manner different from the standard LOC that most software development people are familiar with. Since determination of weighting and complexity is relatively subjective, experience and feedback with the technique will be required to provide accurate FP numbers.

Albrecht does not however, provide a software estimation model which can be immediately installed for the use of RIT Research. There is still a great deal of work to be done in the software estimation field to empirically determine a software estimation tool based on function points. Albrecht

has pointedly placed this challenge before his colleagues, so hopefully more data based on FP will be forthcoming.

Application of Method to Case Study

While the calculation of FP will not directly lead toward detailed software estimation data, it will assist in beginning to build a historical record for RIT Research projects, and will provide another perspective on the project.

STEP 1: Count and classify, to three levels of complexity, the five user function types that are made available to the user:

Determination of Unadjusted Funtion Points for Case Study

Description	Complexity			Total
	Simple	Average	Complex	
External Input		3 * 4 =12	4 * 6 =24	36
External Output		10* 5 =50		50
Logical Internal File		2 * 10 =20	4 * 15 =60	80
External Interface File			4 * 10 =40	40
External Inquiry		4 * 4 =16		16
Total Unadjusted Function Points (FC)				222

STEP 2: Calculate the total degree of influence of the processing and environmental complexity:

Determination of Degree of Influence for Case Study

Characteristic	Degree of Influence None 0	1	2	3	4	Strong 5
Data Communications			2			
Distributed Functions	0					
Performance						
Heavily Used Configuration				3		
Transaction Rate		1				
On-line Data Entry			2			
End User Efficiency			2			
On-line Update		1				
Complex Processing						5
Reusability						5
Installation Ease			2			
Operational Ease			2			
Multiple Sites	0					
Facilitate Change				3		

Total Degree of Influence (PC) 30

STEP 3: Calculate the function points.

PCA = 0.65 + (0.01 * PC)

PCA = 0.65 + (0.01 * 30)

PCA = 0.95

and

FP = FC * PCA

FP = 222 * 0.95

FP = 210.9

Critique

Albrecht has provided the software estimation field with another basis for a metric which threatens to supplant the more commonly applied and researched size-oriented metric, LOC.

Size-oriented metrics are not universally accepted as the best. Critical objections to LOC include:

1. It is highly language dependent
2. It penalizes well-designed but shorter programs

3. It cannot easily accommodate nonprocedural languages
4. Its use in project estimation requires substantial project detail which may be difficult to obtain

Function-oriented metrics, such as Albrecht's FP, do not encounter these difficulties. FP is especially attractive as a basis for estimation because the required detail is available early in project development. However, the computation of FP is fairly subjective, and it is difficult to understand since it is not based on a physical characteristic of the project.

Several studies have examined the relationship between LOC and FP. Albrecht and Gaffney [6.2] found good correlation between LOC and FP for each of three programming languages examined, although the relationship differed for each language. They cited the following average data:

Language	LOC/FP
COBOL	110
PL/1	65
Simple data base language	25

Albrecht suggests using both metrics, relying on FP earlier in project development and using LOC more consistently during project implementation when more detail, and more models based on LOC, are available. And, also, he hopes that substantially more data and models based on FP will become available in the future, making it a more viable tool for software estimation.

Summary

Albrecht's function point effort estimator is a promising concept. Preliminary investigations have found that it is easy to apply, correlates well with standard productivity measures such as LOC, and is unaffected by certain environmental factors such as programming language.

However, the software estimation model based on function points is not fully developed yet. Substantially more

software development projects must be analyzed and modeled using function points as a metric before enough data will be available to make this technique appropriate for general use.

References

[6.1] Albrecht, A. J., "Measuring Application Development Productivity," Proc. IBM Applic. Deve. Symposium, Monterey, CA, October 1979, pp 83-92.

[6.2] Albrecht, A. J., and J. E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," IEEE Trans. Software Engineering, November 1983, pp. 639-648.

Walston and Felix's Empirical Estimation Method

Overview

C. E. Walston and several colleagues began a landmark study in 1972 which hopefully still continues today. This effort is focused on meticulous data collection and analysis for software development projects within IBM Federal Systems division.

As of 1977 when one of their first papers was published on this study [7.1], over 60 projects were included in the data base, and some interesting preliminary findings were available.

While they present a rudimentary estimating model, the major impact Walston and Felix had on the field was identification of 29 factors that were found to impact project productivity.

Description of Method

Walston and Felix focus on programming productivity. They describe productivity as the estimating tool which is easiest

to determine, with schedule, cost, quality and size being more difficult to analyze.

Programming productivity (DSI/MM) is defined to be the ratio of the delivered source lines of code (DSI) to the total effort in man-months (MM) required to produce delivered product.

Applying a least squares fit to the data base, they present a measure of effort:

$$E = 5.2L^{0.91}$$

where:

E = total effort in man-months
L = thousands of lines of delivered source code

In an attempt to explain variation from this model, Walston and Felix examined 68 variables from the data base. Statistically, twenty nine of these were found to have a significantly high correlation with productivity. These variables are described in the table below.

For each of the 29 variables, the estimator answered one or more questions. For example, for the first entry in the table, which describes customer interface, the estimator rated the interface as less than, equal to or greater than normal complexity. All the projects within the data base which rated the customer interface as less than normal complexity displayed a mean productivity of 500 delivered source lines of code per man-month. This table also shows quite clearly the productivity variance for these 29 variables. Customer interface complexity shows a dramatic decrease in productivity as complexity rises, while complexity of program flow does not display such a radical productivity effect.

Question or variable	Response group mean DSI/MM			DSI/MM change
Customer interface complexity	<Norm 500	Norm 295	>Norm 124	376
User participation in the definition of requirements	None 491	Some 267	Much 205	286
Customer originated program design changes	Few 297		Many 196	101

cont'd==========

Question or variable	Response group mean DSI/MM			DSI/MM change
Customer experience with the application area of the project	None 318	Some 340	Much 206	112
Overall personnel experience and qualifications	Low 132	Avg 257	High 410	278
Percentage of programmers doing development who participated in design of functional specifications	<25% 153	25-50% 242	>50% 391	238
Previous experience with operational computer	Min 146	Avg 270	Ext 312	166
Previous experience with programming languages	Min 122	Avg 225	Ext 385	263
Previous experience with application of similar or greater size and complexity	Min 146	Avg 221	Ext 410	264
Ratio of average staff size to duration (people/month)	<0.5 305	0.5-0.9 310	>0.9 173	132
Hardware under concurrent development	No 297		Yes 177	120
Development computer access, open under special request	0% 226	1-25% 274	>25% 357	131
Development computer access, closed	0-10% 303	11-85% 251	>85% 170	133
Classified security environment for computer and 25% of programs and data	No 289		Yes 156	133

cont'd==========

Question or variable	Response group mean DSI/MM			DSI/MM change
Structured programming	0-33% 169	34-66% -	>66% 301	132
Design and code inspections	0-33% 220	34-66% 300	>66% 339	119
Top-down development	0-33% 196	34-66% 237	>66% 321	125
Chief programmer team usage	0-33% 219	34-66% -	>66% 408	189
Overall complexity of code developed	<Avg 314		>Avg 185	129
Complexity of application processing	<Avg 349	Avg 345	>Avg 168	181
Complexity of program flow	<Avg 289	Avg 299	>Avg 209	80
Overall constraints on program design	Min 293	Avg 286	Sev 166	107
Program design constraints on main storage	Min 391	Avg 277	Sev 193	198
Program design constraints on timing	Min 303	Avg 317	Sev 171	132
Code of real-time or interactive operation, or executing under severe timing constraint	<10% 279	10-40% 337	>40% 203	76

cont'd

Question or variable	Response group mean DSI/MM			DSI/MM change
Percentage of code for delivery	0-90% 159	91-99% 327	100% 265	106
Code classified as non-mathematical application and I/O formatting programs	0-33% 188	34-66% 311	>66% 267	79
Number of classes of items in the data base per 1000 lines of code	0-15 334	16-80 243	>80 193	141
Number of pages of delivered documentation per 1000 lines of delivered code	0-32 320	33-88 252	>88 195	125

These variables may have complex interrelationships which are not apparent from this simplified presentation. However, this presentation is useful in providing a qualitative assessment for the relative impact of particular project characteristics on the programming productivity.

Walston and Felix's Empirical Estimation Method

Implementation Requirements

As presented, Walston and Felix's estimating model is quite simplistic and similar to others that have been examined. Implementation is fairly straightforward.

Inclusion of the effects of the 29 variables identified as significantly affecting programming productivity in a software estimate cannot easily be accomplished. Walston and Felix do not provide tools to incorporate the affect of these variable as predicted by the data base. The dedicated estimator would have to begin collecting historical data and model the effects of these variables alone.

Application of Method to Case Study

Given the 13,000 lines of code estimate for the Case Study, application of Walston and Felix's estimation model is simple:

$$E = 5.2L^{0.91}$$

$$E = 5.2(13)^{0.91}$$

$$E = 54 \text{ man-months}$$

where:

E = total effort in man-months

L = thousands of lines of delivered source code

and:

Productivity = DSI/MM

Productivity = 13,000/54

Productivity = 240

A 'rule of thumb' look can be taken at the effects predicted for the 29 impacting variables although it will be very unquantifiable at this early stage in the project. The staff capabilities available for this project are easily predictable, the requirement of structured programming is likely, and the code will not be largely mathematical. Other than these few factors, all other variables must be assumed to have a nominal value, or no affect on programming productivity.

Walston and Felix's Empirical Estimation Method

Question or variable		Effect on DSI/MM	DSI/MM change
Overall personnel experience and qualifications	High	Raise	278
Percentage of programmers doing development who participated in design of functional specifications	>50%	Raise	238
Previous experience with operational computer	Min	Lower	-166
Previous experience with programming languages	Min	Lower	-263
Structured programming	>66%	Raise	132
Code classified as non-mathematical application and I/O formatting programs	>66%	Lower	-79
		Total	140

Walston and Felix have not provided the tools to utilize this total of 140 in any quantifiable manner. However, the conclusion can be reached that their data base predicts there is a significant chance that the productivity calculated above will be higher than 240 DSI/MM.

Critique

Walston and Felix's empirical model is quite simple to apply although it provides little quantifiable information.

Data collection for a qualitative assessment of the 29 significant factors is also relatively straightforward. As detail concerning the project becomes available during the various phases of development, the impact of the 29 variables can be reexamined.

Also, having these factors identified, makes it possible for the project manager to impact the project productivity favorably. Reduction of the customer interface complexity, for example, can significantly increase productivity and, thus, likelihood of project success.

As Walston and Felix are quick to point out, these equations and factors are based on a limited data base of IBM Federal Systems projects. Their successful application to other environments has yet to be demonstrated.

Walston and Felix's Empirical Estimation Method

Summary

Walston and Felix's study has demonstrated several important concepts to the field of software estimation. They carefully established rigorous data collection forms and applied these questionnaires to over 60 projects.

By statistical analysis of hundreds of data points they created a simple model of software development effort, identified 29 project variables which significantly affect programming productivity, and provided a rudimentary tool to qualitatively factor the effect of these variables into software estimation.

Reference

[7.1] Walston, C., and C. Felix, "A Method for Programming Measurement and Estimation," IBM Systems Journal, vol 16, no 1, 1977, pp 54-73.

Putnam's Empirical Model and SLIM Software

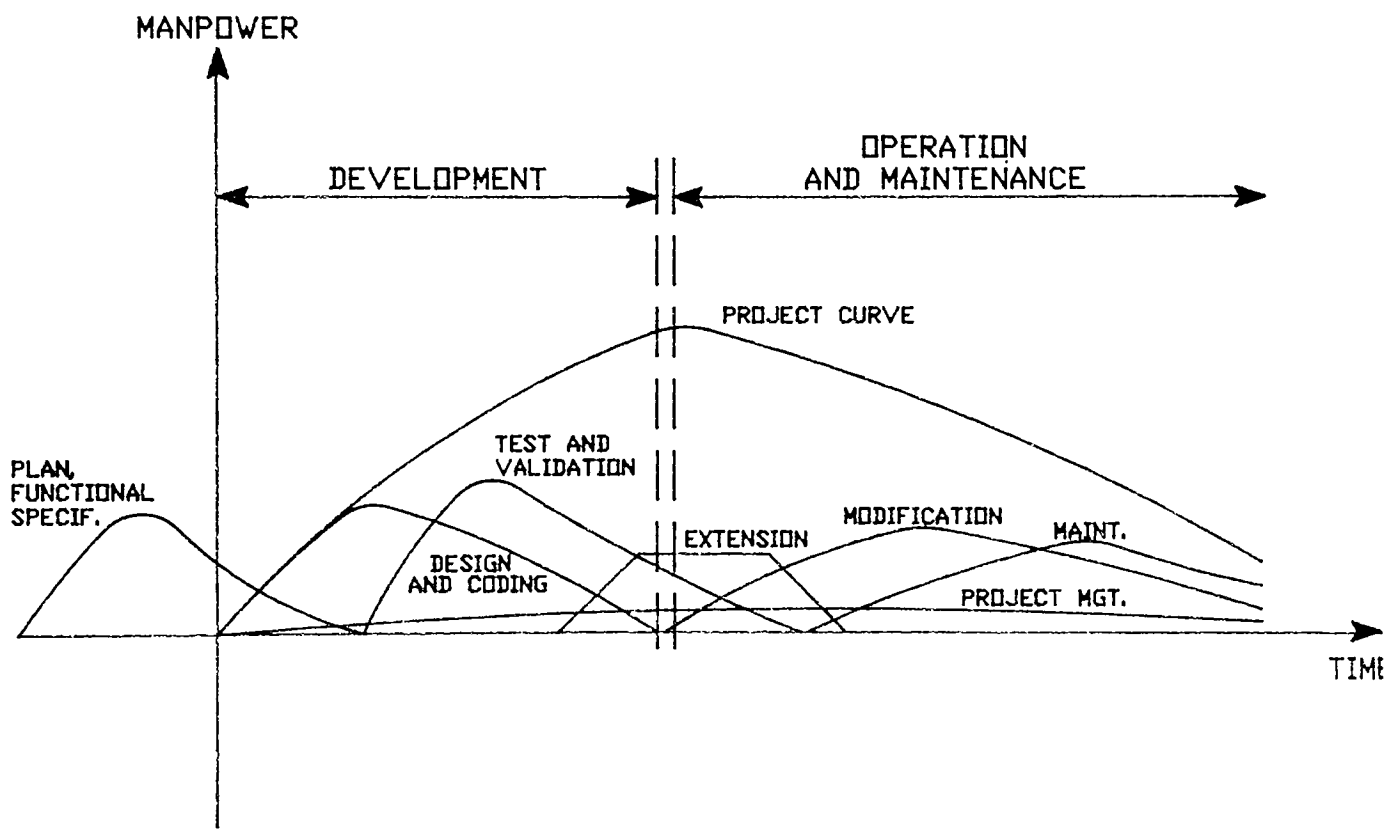
Overview

Lawrence Putnam presents what he describes as a 'general empirical solution to the macro software' estimation problem [8.1]. While too general to be used in specific software estimation efforts, the life-cycle model reveals the underlying relationships between time, effort, and technology factors in software development.

Description

Putnam develops what he terms a dynamic, life-cycle model approach to software estimation through empirical study. His approach is based on the classic life-cycle pattern first generally described by Lord Rayleigh, and later specifically by Norden [8.2]. Lord Rayleigh described a life-cycle pattern applicable to many phenomena. This pattern has been empirically validated by Norden as describing the quantitative behavior of the various cycles of R&D projects. Thus, this model is often termed the Rayleigh/Norden Curve.

Norden found that projects under examination are composed of cycles - planning, design, modeling, release and product support. He linked these cycles to get a project profile or project curve. The project profile he found is shown below:



This life-cycle shows its principal component cycles and primary milestones. All the subcycles, except extension, have continuously varying rates and long tails, indicating

that the final 10 percent of each phase of effort takes a relatively long time to complete.

Empirical evidence suggests that there is scatter, or 'noise' for up to $\pm$25% during the rising part of the curve which corresponds to the development effort. t_d denotes the time of peak effort and is very close to the development time for the system. The falling part of the manpower curve corresponds to the operations and maintenance phase of the system life cycle. The principal work during this phase is modification, minor enhancement, and remedial repair.

Putnam and others examined over 200 systems and found that most exhibit the same basic manpower pattern. The projects that did not adhere to this project curve were found, in general, to have tight control over manpower by management. Management may apply resources suboptimally or contrary to system requirements, for unknown reasons. This can result in a curve that is nearly rectangular, displaying a step increase to peak manpower effort with a steady effort thereafter.

Using the Rayleigh/Norden Curve, Putnam derives a 'software equation' that describes the curve in terms of number of delivered lines of code, effort and development time:

$$S_s = (K^{1/3}) (t_d^{4/3}) (C_k)$$

where:

S_s = Final end-product source statements that will be produced in time t

K = Life-cycle effort in man-years

t_d = Time at which the curve is at a maximum, which can be equated to development time

C_k = state of technology constant

The state of technology constant, C_k, reflects 'throughput constraints that impede the progress of the programmer.' The value of C_k should be determined based on historical data. Typical values established by Putnam are:

C_k = 2,000 for poor software development environment (no methodology, poor documentation and reviews, batch execution mode)

C_k = 8,000 for good software development environment (methodology in place, adequate documentation and reviews, interactive execution mode)

C_k = 11,000 for an excellent software development environment (automated tools and techniques)

Putnam suggests utilization of the equation in a different form:

$$K = (S_s^3) / ((t_d^4) (C_k^3))$$

In this form, K, the development effort for the entire project including development and maintenance is easy to determine, based on lines of code, development time, and the technology constant.

This equation also clearly demonstrates the 'incompressibility' of time and the very unfavorable impact of trading effort for time. Adding people to accelerate a project can accomplish this until the gradient condition is reached, but only at a very high price.

Putnam presents an example of a new system to be built. If the project takes 5 years, it can be done with 5 man-years of effort. Holding S_s and C_k constant, if it has to be done in 3-1/3 years, it will cost about 25 man-years of effort. At

the shortest possible time, 2-3/4 years, it will take about 55 man-years of development effort.

In this way, estimators can simulate, evaluating tradeoffs of cost versus time for their projects.

Unfortunately, Putnam does not go any further in presenting his model in the literature. Determination of the underlying curves for the phases of software development and maintenance is left to the user.

Putnam and Cline [8.3] have introduced a software product called SLIM that assists the estimator with cost and scheduling. SLIM is based on empirically determined life-cycle curves for all phases of software development. These curves can be customized or 'tuned' by the inclusion of a variety of descriptive information about the development environment, the system being developed and the developer's capability to do the job.

Implementation Requirements

An estimate of lines of code must be made prior to use of Putnam's life-cycle model, similar to other models that have been reviewed in this paper.

The state of technology constant should be determined based on historical data. While it is a relatively simple number to determine if similar projects have been undertaken within the environment recently, it cannot be guesstimated. Due to its substantial impact on the model, the lack of a sound state of technology constant destroys the model's value.

Application of Method to Case Study

Given the 13,000 lines of code estimate for the whole project described in the Case Study, and the preferred 9 month (0.75 years) schedule, only one other variable is needed. With no other guidance, C_k is assumed to be 8,000 representing a good software development environment. Therefore,

$$K = (S_s^3) / ((t_d^4) (C_k^3))$$

$$K = (13{,}000^3) / ((0.75^4) (8{,}000^3))$$

$$K = 13.6 \text{ man-years of effort for the total project}$$

This effort is very sensitive to C_k:.

$C_k = 12,000$

$K = (13,000^3) / ((0.75^4)\ (12,000^3)) = 4$ man-years

$C_k = 10,000$

$K = (13,000^3) / ((0.75^4)\ (10,000^3)) = 7$ man-years

$C_k = 6,000$

$K = (13,000^3) / ((0.75^4)\ (6,000^3)) = 32$ man-years

$C_k = 4,000$

$K = (13,000^3) / ((0.75^4)\ (4,000^3)) = 109$ man-years

As it is to development time or schedule:

$t_d = 1$

$K = (13,000^3) / ((1.0^4)\ (8,000^3)) = 4.3$ man-years

$t_d = 0.5$

$K = (13,000^3) / ((0.5^4)\ (8,000^3)) = 68$ man-years

Due to the extreme sensitivity of this model to fluctuation of the factors, it is too unstable to be of general use.

Critique

Putnam presented the very useful concept of life-cycle estimation to the software estimation field. Several other models have been presented in the literature based on the Rayleigh/Norden Curve, and it is very likely that it will be increasingly utilized in future models.

As described in the literature, the Putnam method cannot be implemented because it is too broad. Purchase of the SLIM software cost and schedule estimating model would undoubtedly bring the software life-cycle curve method into practical use.

Summary

Putnam's model presents the software life-cycle concept to software estimation. The entire life-cycle, matching the equation developed by Norden, can be examined as a sum of the underlying phases. This permits study of several concepts

such as Brooks' [8.4] ideas of the limiting factors in effort versus time tradeoffs.

With further development, empirical validation and publication, Putnam's model will be a useful addition to the field of software estimation.

References

[8.1] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Trans. Software Engineering, vol 4, no 4, 1978, pp. 345-361.

[8.2] Norden, P., "Useful Tools for Project Management," Software Cost Estimating and Life Cycle Control, IEEE Computer Society Press, 1980.

[8.3] Putnam, L. H., and R. M. Cline, "The SLIM Software Cost and Schedule Estimating Model," Proceedings of Conference on Software Tools, IEEE Computing Society Press, 15-17 April 1985, pp 18-26.

[8.4] Brooks, Frederick P., Jr., Mythical Man-Month, Addison-Wesley Publishing Co., 1975.

Boehm's COCOMO Estimation Models

Overview

Barry Boehm's COCOMO [9.1] models are well-respected in the software estimation field. Based on a 63 project data base, the empirically determined models are practical and useful.

The models consist of a simple calculation based on lines of code (LOC) which provides man-month and schedule estimates for the 'nominal' project. From there, the estimator has the tools to customize this estimate to the attributes (product, computer, personnel and project) of the effort.

Once this step is completed, COCOMO offers a variety of breakdowns of effort across phases and tasks, which give the estimator further information and customization capabilities.

Description of COCOMO

Boehm's COCOMO models derive their name from COnstructive COst MOdel. He has presented a hierarchical series of three

models each of which allows greater detail and accuracy in its estimate.

BASIC COCOMO: This model is applicable to the large majority of software projects. It is good for quick, early, rough order of magnitude estimates of software costs, but its accuracy is limited because it does not account for critical differences between projects and environments such as hardware constraints, personnel quality and so forth. Basic COCOMO estimates come within a factor of 2 of the project actuals within Boehm's data base only 60% of the time.

Initially the project must be classified by mode:

> organic mode - relatively small project teams, generally stable development environment, little concurrent development of associated new hardware and operational procedures, minimal need for innovative data processing architectures or algorithms, relatively low premium on early completion of project

embedded mode - needs to operate within a strongly coupled complex of hardware, software, regulations, and operational procedures, small team of system designers, very large team of programmers to perform detailed design, coding and unit testing in parallel

semi-detached mode - intermediate stage between organic and embedded either an intermediate level of the project characteristics or a mixture of organic and embedded mode characteristics

The basic equations for each mode are:

Mode	Effort	Schedule
Organic	$MM = 2.4(KDSI)^{1.05}$	$TDEV = 2.5(MM)^{0.38}$
Semidetached	$MM = 2.4(KDSI)^{1.12}$	$TDEV = 2.5(MM)^{0.35}$
Embedded	$MM = 2.4(KDSI)^{1.20}$	$TDEV = 2.5(MM)^{0.32}$

where:

MM = man-months

KDSI = thousands of delivered source instructions

TDEV = development schedule

Then for each mode, Boehm has determined distribution of the effort and schedule across the phases of development for various size projects. For example, for the organic mode:

Phase	Product Size			
	Small (2 KDSI)	Intermediate (8 KDSI)	Medium (32 KDSI)	Large (128 KDSI)
Effort				
Plans & requirements	6%	6%	6%	6%
Product design	16	16	16	16
Programming	68	65	62	59
Detailed design	26	25	24	23
Code & unit test	42	40	38	36
Integrate & test	16	19	22	25
Total	100%	100%	100%	100%
Schedule				
Plans & requirements	10%	11%	12%	13%
Product design	19	19	19	19
Programming	63	59	55	51
Integrate & test	18	22	26	30
Total	100%	100%	100%	100%

Boehm presents similar tables for each mode. For project sizes not represented in this table, interpolation is encouraged.

INTERMEDIATE COCOMO: This model is a compatible extension of Basic COCOMO whose greater accuracy and level of detail make it suitable for cost estimation in the more detailed stages of software product definition. Intermediate COCOMO incorporates an additional 15 predictor variables which account for a great deal of the cost variability Basic COCOMO does not explain. Intermediate COCOMO estimates are within 20% of the project actuals in Boehm's data base 68% of the time.

The basic equations for each mode have different effort coefficients in Intermediate COCOMO than in Basic COCOMO, and the effort multiplier has been incorporated. The equations are:

Mode	Effort	Schedule
Organic	$MM = 3.2(KDSI)^{1.05}(EAF)$	$TDEV = 2.5(MM)^{0.38}$
Semidetached	$MM = 3.0(KDSI)^{1.12}(EAF)$	$TDEV = 2.5(MM)^{0.35}$
Embedded	$MM = 2.8(KDSI)^{1.20}(EAF)$	$TDEV = 2.5(MM)^{0.32}$

where:

EAF = effort adjustment factor as determined below

The 15 predictor variables incorporated into Intermediate COCOMO are called cost driver attributes. These attributes are grouped into software product, computer, personnel and project attributes. Each of these attribute categories contains several factors which are rated from very low to extra high by the estimator identifying that attribute's effort multiplier. The product of these individual effort multipliers then determines the overall effort adjustment factor (EAF) for the project. Typical values for EAF range from 0.90 to 1.40.

The software cost driver ratings and effort multipliers for the 15 cost attributes are:

	Rating	Multiplier
Product attributes		
Required software reliability:		
Effect: slight inconvenience	very low	0.75
Low, easily recoverable losses	low	0.88
Moderate, recoverable losses	nominal	1.00
High financial loss	high	1.15
Risk to human life	very high	1.40
Data base size:		
DB bytes/Prog DSI < 10	low	0.94
10 <= D/P < 100	nominal	1.00
100 <= D/P < 1000	high	1.08
D/P >= 1000	very high	1.16
Product complexity:		
Straightline code with a few non-nested operators, evaluation of simple expressions, simple read, write statements with simple formats, simple arrays in main memory	very low	0.70

cont'd==========

	Rating	Multiplier
Straightforward nesting of operators, mostly simple predicates, evaluation of moderate-level expressions, no knowledge of particular processor or I/O characteristics, single file subsetting with no data structure changes, no edits, no intermediate files	low	0.85
Mostly simple nesting, some intermodule control, decision tables, use of standard math and statistical routines, basic matrix/vector operations, I/O processing includes device selection, status checking and error processing, multi-file input and single file output, simple structural changes, simple edits	nominal	1.00
Highly nested operators with many compound predicates, queue and stack control, considerable intermodule control, basic numerical analysis, basic truncation, roundoff concerns, operations at physical I/O level, optimized I/O overlap, special purpose subroutines activated by data stream contents, complex data restructuring at record level	high	1.15
Reentrant and recursive coding, fixed-priority interrupt handling, difficult but structured numerical analysis, routines for interrupt diagnosis, servicing, masking, communication line handling, generalized parameter-driven file structuring routine, file building, command processing, search optimization	very high	1.30

cont'd==========

	Rating	Multiplier
Multiple resource scheduling with dynamically changing priorities, microcode-level control, difficult and unstructured numerical analysis, device timing-dependent coding, micro-programmed operations, highly couples dynamic relational structures, natural language data management	extra high	1.65
Computer attributes		
Execution time constraint		
<= 50% use of available exec. time	nominal	1.00
70%	high	1.11
85%	very high	1.30
95%	extra high	1.66
Main storage constraint		
<= 50% use of available storage	nominal	1.00
70%	high	1.06
85%	very high	1.21
95%	extra high	1.56

cont'd==========

	Rating	Multiplier
Virtual machine volatility		
Major change every 12 months	low	0.87
Major: 6 months Minor: 2 weeks	nominal	1.00
Major: 2 months Minor: 1 week	high	1.15
Major: 2 weeks Minor: 2 days	very high	1.30
Computer turnaround time		
Interactive	low	0.87
Average turnaround < 4 hours	nominal	1.00
4 - 12 hours	high	1.07
> 12 hours	very high	1.15
Personnel attributes		
Analyst capability		
15th percentile	very low	1.46
35th percentile	low	1.19
55th percentile	nominal	1.00
75th percentile	high	0.86
90th percentile	very high	0.71

cont'd==========

	Rating	Multiplier
Applications experience		
<= 4 months experience	very low	1.29
1 year	low	1.13
3 years	nominal	1.00
6 years	high	0.91
12 years	very high	0.82
Programmer capability		
15th percentile	very low	1.42
35th percentile	low	1.17
55th percentile	nominal	1.00
75th percentile	high	0.86
90th percentile	very high	0.70
Virtual machine experience		
<= 1 month experience	very low	1.21
4 months	low	1.10
1 year	nominal	1.00
3 years	high	0.90

cont'd==========

	Rating	Multiplier
Programming language experience		
<= 1 month experience	very low	1.14
4 months	low	1.07
1 year	nominal	1.00
3 years	high	0.95
Project attributes		
Use of modern programming practices		
No use	very low	1.24
Beginning use	low	1.10
Some use	nominal	1.00
General use	high	0.91
Routine use	very high	0.82
Use of software tools		
Basic microprocessor tools	very low	1.24
Basic mini tools	low	1.10
Basic midi/maxi tools	nominal	1.00
Strong maxi programming/test tools	high	0.91
Add requirements, design, management, documentation tools	very high	0.83

cont'd==========

	Rating	Multiplier
Required development schedule		
75% of nominal	very low	1.23
85%	low	1.08
100%	nominal	1.00
130%	high	1.04
160%	very high	1.10

==========

Once EAF has been determined (by multiplying all the effort multipliers together), MM and TDEV can be calculated using the equations above. Then, as was done in the Basic COCOMO, the distribution of effort and schedule across the phases of development for various size projects can be determined.

Lastly, Boehm provides tables which describe the activity distribution for each phase of software development. For example, the project activity distribution for the programming phase of organic mode projects is shown below (in

this case the distribution is the same for all size projects):

	Product Size			
	Small (2 KDSI)	Intermediate (8 KDSI)	Medium (32 KDSI)	Large (128 KDSI)
Overall Phase Percentage	68%	65%	62%	59%
Programming:				
Requirements analysis		5%		
Product design		10		
Programming		58		
Test planning		4		
Verification & validation		6		
Project office		6		
Config. mgmt & quality assur.		6		
Manuals		5		
Total		100%		

While these figures are presented by Boehm to provide a feel for how the distribution of time, effort and personnel varies with each phase, they can be used in an alternative fashion. They provide the estimator with a final opportunity to customize the model.

Given that Boehm's data base is primarily drawn off of government contracts, estimators for nongovernment contracts may find some of their requirements less expensive then COCOMO estimates show. For example, if certification management and quality assurance is not required at all in a project being estimated above, the project estimate can be reduced by 6%.

Boehm encourages use of Intermediate COCOMO at the component level. In this way cost driver attributes may be determined and incorporated unequally across the various components of a project. This allows the use of COCOMO easily and consistently through all stages of software product definition: as a 'macro' model during the rough early stages, and as a 'micro' model in the later, more detailed stages.

DETAILED COCOMO:

This model was created to overcome what Boehm views as the two primary limitations of Intermediate COCOMO for large software projects:

1. Its estimated distribution of effort by phase may be inaccurate.

2. It can be very cumbersome to use on a product with many components.

Detailed COCOMO provides two main capabilities which address these limitations in Intermediate:

1. Phase-sensitive effort multipliers.
Factors such as required reliability, applications experience, and interactive software development affect some phases much more than others. By providing a set of effort multipliers specific to the various phases of software development, Detailed COCOMO can be used to accurately determine the amount of effort required to complete each phase.

2. Three-level product hierarchy.
The model allows some effects which tend to vary with each bottom level module to be determined at the module level. Some effects which vary less frequently are treated at the subsystem level. Some effects, such as the effect of total product size, are treated at the system level.

Other than these two differences, Intermediate and Detailed COCOMO are basically similar. Detailed COCOMO is not noticeably better than Intermediate COCOMO for estimating complete development efforts. Detailed COCOMO does yield better phase distribution estimates offering a much more detailed breakdown of cost drivers as well as useful low-level project tracking and schedule forecasting opportunities.

These attributes go beyond what is required and data is available for most software estimation efforts within RIT Research. For this reason, Detailed COCOMO will not be covered in depth in this paper. However, Detailed COCOMO should be considered a valuable project estimation and management tool for large, lengthy software projects.

Implementation Requirements

Intermediate COCOMO places three requirements on the estimator. As with most other methods that have been described earlier, an estimate of lines of code (LOC) must be available.

Also, the mode and effort multipliers must be determined. The mode, product, computer and project attributes should be determined by computer scientists familiar with the project. The personnel attributes can be determined by the project manager, who has knowledge of the personnel resources that will be available for the project.

Lastly, an understanding of the tasks within the project and a relative assessment of the distribution of resources and effort over those tasks is advantageous. Historical data would be very useful in determining this estimate.

Application of COCOMO to Case Study

Even early in the project, the Case Study was amenable to application of Intermediate COCOMO. The LOC estimate has been provided as 13,000.

While the project would not involve a relatively static project team and environment, it was not a tightly constrained project either. Thus, it was determined to be semidetached mode.

The effort multipliers are as follows:

	Rating	Multiplier
Product attributes		
Required software reliability:		
Low, easily recoverable losses	low	0.88
Data base size:		
D/P >= 1000	very high	1.16
Product complexity:		
Mostly simple nesting, some intermodule control, decision tables, use of standard math and statistical routines, basic matrix/vector operations, I/O processing includes device selection, status checking and error processing, multi-file input and single file output, simple structural changes, simple edits	nominal	1.00

cont'd==========

	Rating	Multiplier
Computer attributes		
Execution time constraint		
<= 50% use of available exec. time	nominal	1.00
Main storage constraint		
<= 50% use of available storage	nominal	1.00
Virtual machine volatility		
Major change every 12 months	low	0.87
Computer turnaround time		
Average turnaround < 4 hours	nominal	1.00
Personnel attributes		
Analyst capability		
75th percentile	high	0.86
Applications experience		
1 year	low	1.13
Programmer capability		
55th percentile	nominal	1.00

cont'd==========

	Rating	Multiplier
Virtual machine experience		
4 months	low	1.10
Programming language experience		
3 years	high	0.95
Project attributes		
Use of modern programming practices		
Some use	nominal	1.00
Use of software tools		
Basic midi/maxi tools	nominal	1.00
Required development schedule		
100%	nominal	1.00

==========

The major impact of the multipliers is seen in the product and personnel attributes. Because of the low requirement for software reliability, effort is lessened. However, the extremely large size of the data base dramatically increases

effort. The personnel available to this project are very high quality, with the analysts and programmers displaying high skills even though they have little direct experience with this application.

Multiplying all of these individual multiplier together, the EAF is determined to be 0.90. Therefore, for the whole project:

$$MM = 3.0(KDSI)^{1.12}(EAF)$$
$$MM = 3.0(13)^{1.12}(0.90)$$
$$MM = 47.75 \text{ man-months}$$

and

$$TDEV = 2.5(MM)^{0.35}$$
$$TDEV = 2.5(47.75)^{0.35}$$
$$TDEV = 9.7 \text{ months}$$

thus,

$$\text{Average number of people on team} = MM/TDEV$$
$$\text{Average number of people on team} = (47.75)/(9.7) = 4.9$$

Consulting a Project Activity Distribution by Phase Table for the semidetached mode and interpolating a 13 KDSI column:

Phase	Product Size			
	Small (2 KDSI)	Intermediate (8 KDSI)	Case Study (13 KDSI)	Medium (32 KDSI)
Effort				
Plans & requirements	7%	7%	7%	7%
Product design	17	17	17	17
Programming	64	61	60	58
Detailed design	27	26	26	25
Code & unit test	37	35	34	33
Integrate & test	19	22	23	25
Total	100%	100%	100%	100%
Schedule				
Plans & requirements	16%	18%	18%	20%
Product design	24	25	25	26
Programming	56	52	51	48
Integrate & test	20	23	24	26
Total	100%	100%	100%	100%

Thus, plans and requirements will consume and additional 3.3 (7% of 47.75) man-months over 1.8 (18% of 9.7) months, and product design will require 8 (17% of 47.75) man-months over 2.4 (25% of 9.7) months. The total estimate for functional specification and design of this software product is 11.3 man-months over 4.2 months.

Looking at the project activity distribution semidetached mode for each of the two phases and interpolating where required:

Boehm's COCOMO Estimation Models

	Plans & Requirements (13 KDSI)	Product Design (13 KDSI)
Overall Phase Percentage	7%	17%
Requirements analysis	47	12.5
Product design	16.5	41
Programming	3.5	12.5
Test planning	3	5
Verification & validation	6.5	6.5
Project office	14.5	12
Config. mgmt & quality assur.	3	2.5
Manuals	6	8
Total	100%	100%

The Case Study requirements should be less than Boehm's data base dictates. Verification & validation, project office and configuration management & quality assurance are estimated to be half of Boehm's predictions. These estimates are based on

the estimator's familiarity with previous software development efforts at RIT Research. With enough historical data, these reductions could be determined empirically.

Therefore, the plans and requirements phase is reduced by 12%, and product design is reduced by 10.5%. The final, COCOMO estimate for these two activities is:

	Man-months	Schedule (months)
Plans and Requirements	2.9	1.6
Product Design	7.2	2.4
Total	10.1	4.0

Critique

Boehm's COCOMO models are certainly the most comprehensive software estimation tools presently available. By all standards, this method is far superior to the other methods reviewed in this paper.

The estimator must be very cautious, however, in using COCOMO. Each input can have a substantial effect on the final estimate produced by the model. Of course, the major

impact can be produced by varying the effort multipliers and the final adjustment of activity distribution across phases, since each directly affects MM and TDEV.

Because of these impacts, both factors provide an excellent opportunity to customize the model to an environment, to determine sensitivities to various factors, and to constrain the software effort, if required, in a manner that will achieve the objectives.

Summary

The Intermediate COCOMO model is very useful for software estimation. It is easy to understand and use, providing a great deal of detail if needed. With the two techniques available to customize the empirically determined data to a given environment, the model permits tremendous flexibility in application - giving the estimator the power and direction to modify estimates as well as project scope, requirements, and resources in order to improve the chances of project success.

Reference

[9.1] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

Progress and Direction of Estimation Technologies

An examination of two decades of estimating models has revealed increasing sophistication and usefulness of the various technologies.

Barry Boehm's COCOMO model provides the most complete, integrated approach. Most of the features of other models are incorporated within COCOMO.

1. Boehm has empirically determined the underlying equations with the application of nonlinear methods, as was determined by Morin in 1972 [10.1] to be necessary for accurate prediction.

2. COCOMO includes the life cycle and project stages concepts demonstrated by Kustanowitz and Putnam.

3. Boehm also requires inclusion of project customization factors as were described and proven to be statistically significant by Walston and Felix's work.

The simplicity and lack of usefulness of most of the models has made their critique obvious. As models become more

complex in the future, more demands can be made on their robustness. Boehm [10.2] provides a set of criteria for 'goodness of a software cost estimation model', which provide guidelines and expectations for models of the future. The ten criteria are:

1. Definition.
Has the model clearly defined which costs it is estimating, and which costs it is excluding?

2. Fidelity.
Are the estimates close to the actual costs expended on the project?

3. Objectivity.
Does the model avoid allocating most of the software costs variance to poorly calibrated subjective factors?

4. Constructiveness.
Can a user tell why the model gives the estimates it does? Does it help the user understand the software job to be done?

5. Detail.

Does the model easily accommodate the estimation of a software system consisting of a number of subsystems and units? Does it give accurate phase and activity breakdowns?

6. Stability.

Do small differences in inputs produce small differences in output cost estimates?

7. Scope.

Does the model cover the class of software projects whose costs you need to estimate?

8. Ease of Use.

Are the model inputs and options easy to understand and specify?

9. Prospectiveness.

Does the model avoid the use of information which will not be well known until the project is complete?

10. Parsimony.
Does the model avoid the use of highly redundant factors, or factors which make no appreciable contribution to the results?

Two of the models that have been examined, presented ideas that have not, as of yet, made a major impact on the field of software estimation. Esterling's study of the microscopic characteristics of the work environment presented interesting, but hard to implement, concepts. This type of analysis is perhaps best performed once or twice within a given environment. In this way project managers would become aware of the available manpower options which affect project schedule and cost, and the potential impact of these options. However, Esterling's analysis does not provide useful project estimation techniques and has not achieved any real popularity to date.

Albrecht's function point effort estimation metrics are a concept that is gaining popularity. Increasingly, the software development field is becoming aware that "of all the issues that must be dealt with for programming to become a science, measurement is the most fundamental and the most

important." [10.3] Thus, while there is still a great deal of research into software estimation models, recently the study of software metrics has grown. Both of these fields must mature and standardize, before software estimation can be viewed as a science as opposed to an art.

References

[10.1] Morin, L. H., "Estimation of Resources for Computer Programming Projects," MS Thesis, University of North Carolina, Chapel Hill, NC, 1973.

[10.2] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

[10.3] Jones, C., Programming Productivity, McGraw-Hill, 1986.

Conclusions and RIT Research Corporation Implementation Strategy

Overview

A review of the state-of-the-art in software estimation has shown Barry Boehm's COCOMO estimation model to be far superior to other options. However, even this model can be difficult to apply accurately, given the number of customizations and subjective assessments the estimator must contribute. Consistent, diligent application of COCOMO along with careful review of the results will improve RIT Research's ability to apply the model.

Allan Albrecht's metric of functionality, the function point, shows promise for usefulness in future estimation methods and productivity assessment. If RIT Research estimators began calculating function point for current projects, much more could be learned about projects, estimation during early stages of projects and function point versus lines of code metrics.

Esterling's work environment concepts offer some potential productivity gains for RIT Research projects. Application of

his model to the general RIT Research project environment may recommend policy changes which will improve employee morale, reduce project cost, and improve productivity.

RIT Research Estimation Model

As has been shown in the preceding chapters, Barry Boehm's COCOMO is, by far, the most complete, usable model under review. Application of the other models to the Case Study has clearly shown them to be incomplete and inaccurate. They are also not amenable to estimate improvement through incorporation of RIT Research historical data.

Unfortunately, COCOMOs estimates for RIT Research projects may not be very accurate. One of the assets of this model, however, is that it can be customized to the environment through both the cost driver attributes and the activity distribution. And as experience with customizing these factors and calculating the model increases, the estimates provided by the model will improve.

Therefore, Intermediate, or perhaps even Detailed COCOMO if the project is large enough, is the recommended model for

software estimation within RIT Research. This recommendation is made under the assumption that software estimators work with COCOMO on all estimates, and attempt to improve their estimating through critical review of each estimate during and after each project phase.

Additionally, it is recommended that RIT Research calculate the function point metric for each programming project as defined by Allan Albrecht. At this point in time, this metric is not of direct value; there are no models that calculate effort based on function points. However, if RIT Research can gain experience and appreciation with function point metrics, and the level of effort these numbers represent, two potential benefits arise. First, by building an historical record of function point values over several projects, if a function point model were to become available, application at RIT Research would be substantially eased. Second, function points are easier than lines of code to accurately determine early during project development. Once a relationship has been established between the two metrics, function points may serve as an 'early bridge' to lines of code and effort estimates as Albrecht suggested.

Implementation Recommendations

In order to consistently improve RIT Research software estimation accuracy a seven step estimation system is recommended, as outlined by Boehm [11.1]:

1. Establish objectives for the estimate.
Each estimate should be developed under clearly established guidelines as to its and scope and use. For example, does the estimate have to be absolute or can it be made in relation to another? Should it be a generous or conservative estimate?

2. Plan for required data and resources.
Data should be collected as abundantly as possible, and adequate time and resources should be made available to develop the estimate.

3. Pin down software requirements.
All assumption should be thoroughly documented.

4. Work out as much detail as feasible.

5. Use several independent techniques and sources. Both COCOMO, and one or two other estimation models should be applied. The current RIT Research estimation technique

would provide as useful metric, since it has been found to be relatively accurate. Function point calculations should also be made.

6. Compare and iterate estimates.
Reconcile differences between estimates.

7. Followup.
Estimated values should be completely compared and reconciled with costs and schedule actually incurred. Explanations, and ideas for improvement of estimation techniques should be documented.

It is important that RIT Research estimators remain aware of the state-of-the-art in this field. In this fashion they can slowly incorporate new technologies, such as the recommendations for function point implementation, and continually improve the accuracy of their estimates.

A final recommendation for RIT Research is to consider the concepts introduced by Esterling in his microscopic examination of the work environment and its effect on productivity. Esterling's model should be applied to RIT Research's typical programming environment, and an evaluation

made of effectiveness of RIT Research's cost and schedule policies and environment. The benefits of such features as flex-time, overtime compensation, minimized administrative responsibility for programmers, short meetings, and on-line communication can be quantified and considered. Some changes may be appropriate.

Summary

RIT Research Corporation estimators have a powerful tool in Barry Boehm's COCOMO. With careful implementation, and in conjunction with function point determination, a continually improving system of estimation has been recommended.

Esterling's work environment concepts may also contribute additional productivity benefits, after thorough evaluation and consideration by RIT Research staff.

Reference

[11.1] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

Bibliography

Abdel-Hamid, T. K., and S. E. Madnick, "Impact of Schedule Estimation on Software Project Behavior," IEEE Software, vol. 3, no. 4., July 1986, pp 70-5.

Albrecht, A. J., "Measuring Application Development Productivity," Proc. IBM Applic. Deve. Symposium, Monterey, CA, October 1979, pp 83-92.

Albrecht, A. J., and J. E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," IEEE Trans. Software Engineering, November 1983, pp. 639-648.

Arthur, L. J., Measuring Programmer Productivity and Software Quality, Wiley Interscience, 1985.

Basili, V. and M. Zelkowitz, "Analyzing Medium Scale Software Development," Proc. 3rd Intl. Conf. Software Engineering, IEEE, 1978, pp. 116-123.

Basili, V., Models and Metrics for Software Management and Engineering, IEEE Computer Society Press, 1980.

Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

Brooks, Frederick P., Jr., Mythical Man-Month, Addison-Wesley Publishing Co., 1975.

Dauphinais, B. and L. Darnell, "Project Management: One Step at a Time," PC World, September 1984, pp. 240-256.

Bibliography

DeMarco, T., Controlling Software Projects, Yourdon Press, 1982.

Deutsch, D., and E. Fong and J. Collica, "Cost Considerations for Generalized Database Management Systems," Generalized Data Management Systems and Scientific Information, Report of A Specialist Study, 1978, pp 50-70.

Drake, T. A., "Estimating Time and Cost for Software 'Magic or Science'," Proceedings of Micro-Delcon the Delaware Bay Microcomputer Conference, 20 March 1979, pp 119-22.

Dudding, L. C., and S. L. McQuerry, "Expert Software Prices (ESP) an AI/Algorithm Approach to Software Costing," Proceedings of the Twentieth Hawaii International Conference on System Sciences, 1987, pp 466-77.

Esterling, R. "Software Manpower Costs: A Model," Datamation, March 1980, pp. 164-170.

Farquhar, J. A., "A Preliminary Inquiry into the Software Estimation Process," Technical Report, AD F12 052, Defense Documentation Center, Alexandria, VA, August 1970.

Gaffney, J. E., Jr., "Software Metrics: A Key to Improved Software Development Management," Proceedings of Computer Science and Statistics, Proceedings of the 13th Symposium on the Interface, IBM Federal Systems Division, 12-13 March 1981, pp 211-20.

Gehring, P. F. and U. W. Pooch, "Software Development Management," Data Management, February 1977, pp 14-18.

Jones, C., Programming Productivity, McGraw-Hill, 1986.

Bibliography

Jones, C., "Steps Toward Establishing Normal Rules for Software Cost, Schedule, and Productivity Estimating," Proceedings of the NATO Advanced Study Institute, 29 July - 10 August 1985, pp 567-75.

Kustanowitz, A. L., "System Life Cycle Estimation (SLICE): A New Approach to Estimating Resources for Application Program Development," Presented at COMPSAC '77, IEEE Computer Soc. 1st Int. Computer Software and Applications Conf., Chicago, IL November 8-10, 1977.

McCabe, T., "A Complexity Measure," IEEE Trans. Software Engineering, December 1976, pp. 308-320.

Mohanty, S. N., "Software Cost Estimation: Present and Future," Software Practice and Experience, Vol II, 1981, pp 103-21.

Morin, L. H., "Estimation of Resources for Computer Programming projects," MS Thesis, University of North Carolina, Chapel Hill, NC, 1973.

Norden, P., "Useful Tools for Project Management," Software Cost Estimating and Life Cycle Control, IEEE Computer Society Press, 1980.

Paster, D. L., Experience with Application of Modern Software Methodology and Management Control," IEEE Engineering Management Conference, 12-14 November 1980, pp 56-9.

Premo, A. F., Jr., "Computer Software: Estimating Guidelines," 13th IEEE Computer Society International Conference, 7-10 September 1976, pp 146-51.

Pressman, Roger S., Software Engineering, McGraw-Hill Book Company, 1987.

Bibliography

Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Trans. Software Engineering, vol 4, no 4, 1978, pp. 345-361.

Putnam, L. H., "SLIM: A Quantitative Tool for Software Cost and Schedule Estimation," Proceedings of the NBS/IEEE/ACM Software Tool Fair, 10-12 March 1981, pp 49-57.

Putnam, L. H., and R. M. Cline, "The SLIM Software Cost and Schedule Estimating Model," Proceedings of Conference on Software Tools, IEEE Computing Society Press, 15-17 April 1985, pp 18-26.

Putnam, L. H., and R. W. Wolverton, "Quantitative Management: Software Cost Estimating," Presented at COMPSAC '77, IEEE Computer Soc. 1st Int. Computer Software and Applications Conf., Chicago, IL November 8-10, 1977.

Riggs, J., Production Systems Planning, Analysis and Control, 3rd ed., Wiley 1981.

Rubin, H. A., "Macro-estimation of Software Development Parameters: The Estimacs System," Softfair Proceedings, IEEE, July 1983, pp. 109-118.

Rubin, H. A., and R. Jensen and L. Putnam and P. Rook, "A Comparison of Cost Estimation Tools", Proceedings of the 8th International Conference on Software Engineering, IEEE Computing Society Press, 28-30 August 1985, pp 174-80.

Walston, C., and C. Felix, "A Method for Programming Measurement and Estimation," IBM Systems Journal, vol 16, no 1, 1977, pp 54-73.