Software Measurement

Notes by mainly Jo Anne Atlee,
with modifications by Daniel Berry and Richard Trefler

Fall 2012

Software Estimation and Metrics

Dilbert, how long will it take your team to port our inventory control system to NT, as well as fix a few known bugs, make the user interface simpler, and implement just a few new features that the customers have been asking for?

---

i.e., *more than just informal guesses. How did you calculate the estimates?*

Tom DeMarco:

> *"An estimate is the most optimistic prediction that has a non-zero probability of coming true. Accepting this definition leads irrevocably toward a method called what's-the-earliest-date-by-which-you-can't-prove-you-won't-be-finished estimating."*

Our Job is to Estimate:

1. time to develop
2. cost
3. number of developer/months?

Software Cost Estimation

Given an early description of the system, you want to determine as early as possible if a proposed system or requirement is technically and economically feasible. Economically feasible means whether the client is willing to pay what it will cost to develop the project, and whether the developer is willing to devote the resources to the project. Both of these questions have to be answered based on estimates, estimates of the cost of the project and estimates of the development effort.

Why Estimate Software Cost and Effort?

- To provide a basis for agreeing to a job:

  You must make a business case for taking on a job or set thresholds for negotiating a price for performing the job.

- To make commitmemts that you can meet:

  Cost overruns can cause customers to cancel projects. Alternatively, to avoid passing on all extra costs to customer, the project team may work without full financial compensation, swallowing the cost overrun.

- To help you track progress:

  As it is said, you cannot manage what you cannot measure. If you don’t know how long it will take to develop a system, you won’t know if you’re falling behind.

We Can’t Estimate Very Well

Unfortunately, it is very difficult to estimate the cost and effort to build a project when you don’t know very much about that project.

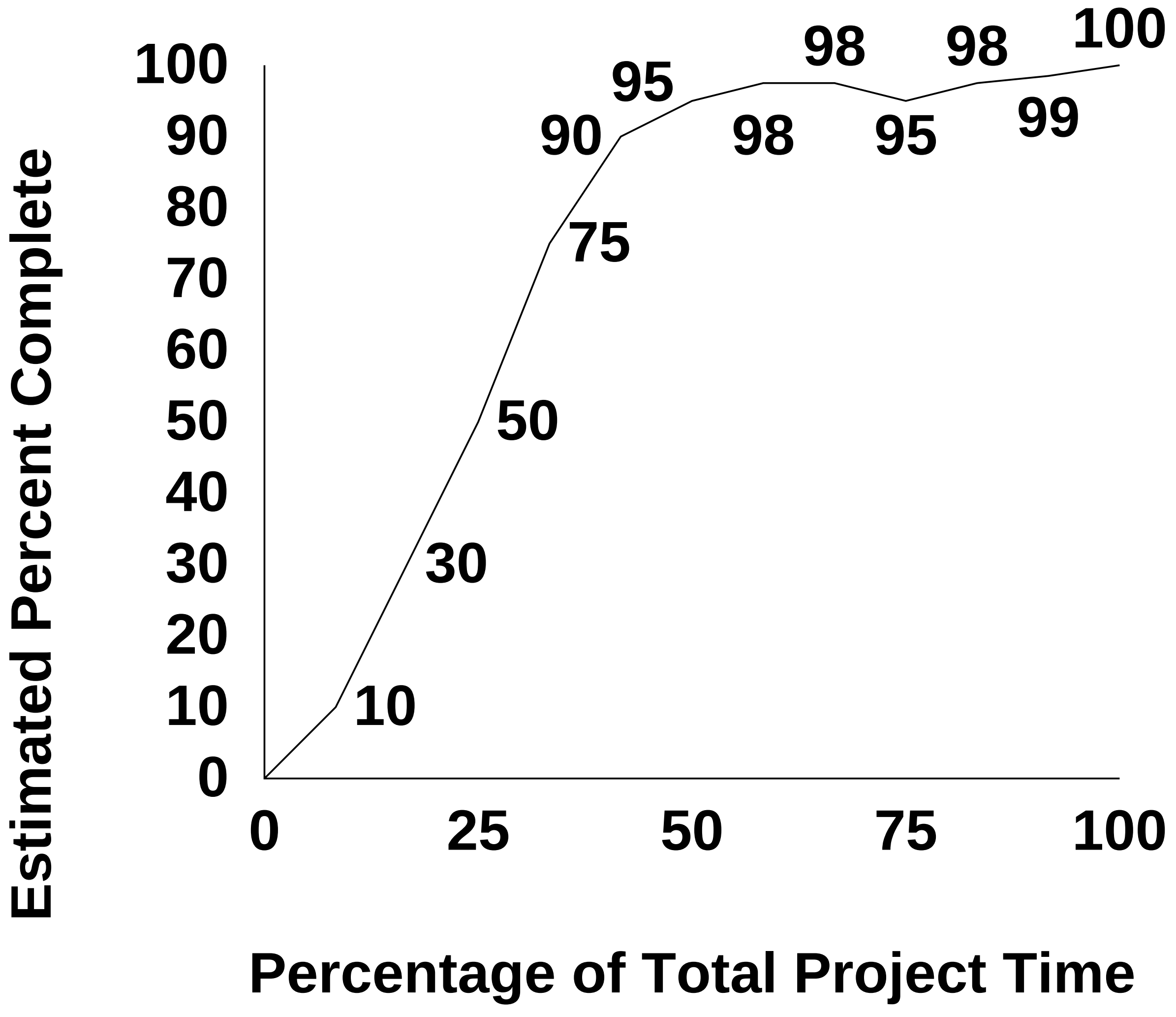
- We’re not estimating repeatable, objective phenomena.
- The earlier the estimate, the less is known about the project.
- Estimates can be biased by business and other pressures.

  The desire to take a job may prompt you to believe an unrealistic estimate.

- A goal to estimate within 10% of actual cost is unrealistic.

  Experience has shown that by the time we know enough about a project to estimate its cost to within 10% of its actual cost, the product is almost complete.

Accuracy of Estimates During Development



Distribution of Code

RE

10–20% of the code = central approximation.

80–90% of the code = exceptional details.

99.99% of execution time is spent in the central 10–20% of the code.

It’s hard to test the exceptional details code, the 80–90% of the code, because it gets executed less than 0.01% of the execution time.

And correcting a defect in this 80-90% of the code likely involves changes in related code scattered all over the code.

Practice Makes Perfect

However, if you practice, you’ll get better.

Steve McConnell equates software cost estimation with estimating how much pocket change there is in a room full of people. Your first attempt is likely to be way off, but you get better. Maybe you start learning how to account for the type of people in the room. Perhaps, students carry more pocket change than business people. Perhaps, men carry more pocket change than women, who are more likely to have purses. However, even while you get better, you will continue to make mistakes. Of course, if you go to the States where there are no $1 or $2 coins, you will probably be way off and will have to learn how to estimate all over again.

Delphi Method

Delphi methods are based on expert judgement

- Each expert submits a secret prediction, using whatever process he or she chooses.
- The average estimate is sent to the entire group of experts.
- Each expert revises his or her prediction privately.

  In some variations of the Delphi method, the experts discuss their rationales before new estimates are made, justifications are circulated anonymously, or no discussion is allowed.
- Repeat until no expert wants to revise his or her estimate, i.e., until a fixed point is reached.

Critical Points About Delphi

- Its success depends on the experts' abilities to determine which past projects are similar and in which ways.
- An expert's experience cannot be transferred to junior developers.

LOC or KLOC

LOC — lines of code
KLOC — thousand LOC

Problems with them:

- How do you measure them?
  - How do you count one line that has several statements?
  - How do you count a statement that is over several lines?
  - How do you count constructs, e.g., conditionals?
- One person's line may be another's several lines

But they are used as the unit of code size with care and ...

with *standards* that answer these questions.

Estimating Resources From Requirements

During requirements analysis, we do not have code with which to make estimates.

We want to be able to estimate cost based on what we know at requirements analysis time, i.e., the requirements.

So we break the problem down into three parts:

1. estimating the number of function points from the requirements,
2. estimating code size from function points, and
3. estimating resources required (time, personnel, money) from code size.

Function Points

- These have a big following
- Probably a better measure than KLOC ... but *caveat emptor*
- Makes some real sense in industrial settings, well understood serious backend kind of systems
- Basic idea:
  - Count # of inputs and output to a function (perhaps multiplying by some scaling factors)
  - Count # of other kinds of references/transactions
  - Add ’em up. That’s the complexity of this function. See the next slide for details on adding ’em up!
- Lots of tools and methods exist that use FPs.

Estimating Function Points From Requirements

Function points are used to predict the size of a system (# “functions”)

Idea is to predict the complexity of the system in terms of the various functions to write, without being as specific as lines of code, which is programming language dependent. So, we are counting the various types of functions, and weighting them according to their types and complexities.

The Basic Model is:

$$\mathbf{FP} = \underset{\textbf{weighting factor for function type}}{\mathbf{a_1}}\,\overset{\textbf{\# of function type}}{\mathbf{P_1}} + \mathbf{a_2 P_2} + \ \mathbf{\dots} + \mathbf{a_n P_n}$$

where P_i is the # of instances of the i^{th} function type

Weightings

S = Simple, N = Normal, C = Complex

Function	S	N	C
user input e.g., input event, data entry	3	4	6
user output e.g., screen, error message, report	4	5	7
user query e.g., simpler request or response function that doesn’t require a change to data or system state	3	4	6
external interfaces e.g., files, other systems; probably want unique weightings for each identified interface	5	10	15

Estimating Code Size From Function Points

There are tables that list for each programming language, the number of statements in it that are required to implement one function point.

Language	Lines/FP
Assembly	50
C	15
C++	12
Java	9
LISP	2

These tables are calibrated for each shop, for each domain, etc.

Note that it was observed a long time ago that a programmer’s productivity in terms of debugged, documented, lines of code per day is constant and is independent of the language he or she is using.

This is why the higher the level the language, the more productive is the programmer; he or she does much more per line he or she writes.

Estimating Resources From Lines of Code

There are several methods.

The most popular is COCOMO.

COCOMO [Boehm]

COnstructive COst MOdel

- Barry Boehm's 1981 book *Software Engineering Economics* is revered by many,
  [but others are skeptical]
- COCOMO is a set of models for performing software estimation, based on Boehm's (and others') experiences building software systems for the US DoD.
- Three levels: basic, intermediate, and advanced. *The difference is in how detailed you want to be.*
- Boehm (who is considered an optimist!):
  "Today [1981], a software cost estimation model is doing well if it can estimate within 20% of the actual costs, 70% of the time, and on its own turf (that is, within the class of projects to which it has been calibrated).

Basic COCOMO Model

$E = a \times KLOC^b \times X$ $\qquad D = c \times E^d$

$E =$ effort in person-months

$D =$ development time in months

$X =$ project attribute multipliers

$KLOC =$ size of resulting system

$a, b, c, d =$ empirically observed coefficients:

Kind of project	a	b	c	d
organic	2.4	1.05	2.5	0.38
semi-detached	3.0	1.12	2.5	0.35
embedded	3.6	1.20	2.5	0.32

organic — smaller project, requirements not rigid, experienced developers

semi-detached — intermediate in size & complexity, mix of rigid and flexible requirements

embedded — tight constraints/requirements on hardware, software, environment

Notes :

KLOC is the estimated size of code, adjusted to account for any reuse of design or code.

a, *b*, *c*, and *d* are *empirically* observed coefficients.

Both are adjusted at each shop for each domain based on historical data.

The *X* stands for multipliers for 16 project attributes that have been observed to be critical.

Project Attributes

- product attributes: reliability, complexity

  required reliability↑, complexity↑, database size↑
- resource attributes: execution time, memory constraints

  execution time↑, memory↑, hardware volatility↑, tight response time↑
- personnel attributes: experience of developers

  quality of analysts↓, quality of programmers↓, experience with product↓,hardware experience↓, programming lang. experience↓
- project attributes: modern techniques, prog. lang.

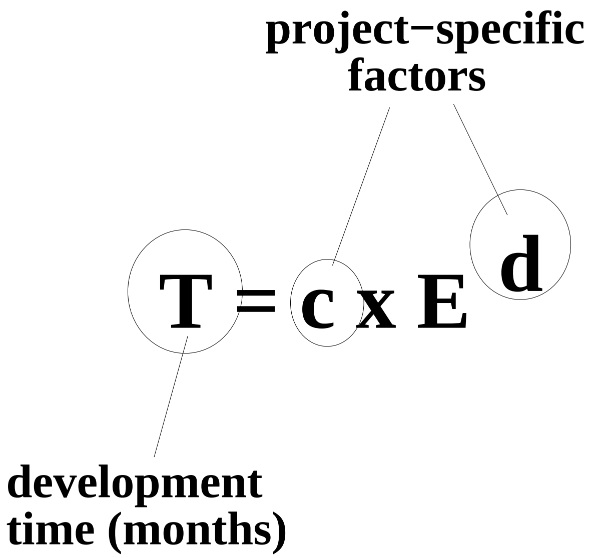
  use of software tools (e.g., debugger)↓, use of modern PL↓, schedule constraints↑

Combine FP and COCOMO

The FPs are calculated from the requirements and are translated into estimated LOCs, which is then used in the COCOMO estimation method.

Estimate time:

Recall the formula



$$T = c \times E^{d}$$

where c and d are based on product type like a and b

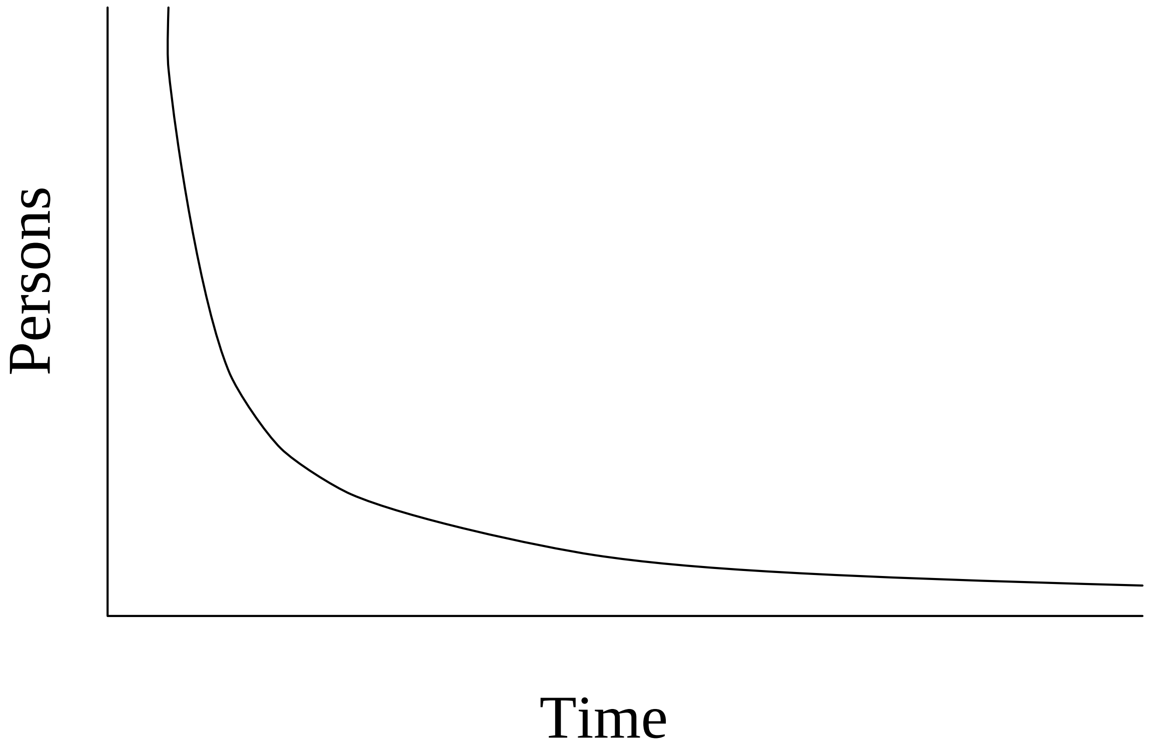
Effort is measured in person-months, and time is measured in months.

Why is formula not linear?

Mythical Person-Month

Freddy Brooks’s famous book: *Mythical Man-Month*

The problem with person months as unit of measure is that it implies the following graph.



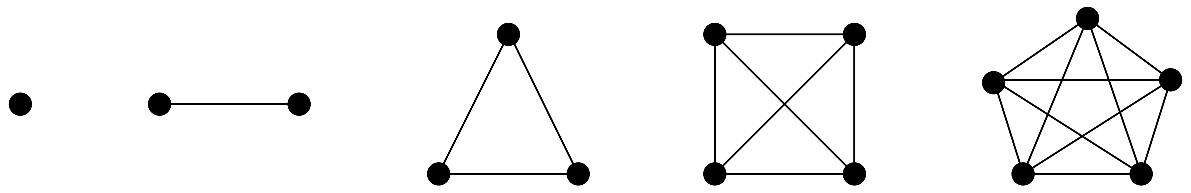
But it does not work: Main counter example:

One woman gestates a baby in 9 months. How many months are required for three women to gestate a baby? Certain tasks require a certain minimum amount of time and throwing more personnel at the tasks does not reduce the time needed.

It does work for painting a fence. Why? Because no communication is required.

It does not work for software development teams. Why?

Importance of Communication in a Group Project:



1,0 2,1 3,3 4,6 5,10

number of persons, lines of communication

Famous quote from Freddy Brooks: “Adding more people to a late project makes it even later.”

At some point, a new person costs in communication more than he or she adds to the work that can be done.

This is not even counting the fact that a new person wastes his or her own and others’ time getting up to speed.

Minimum Personnel Research

The other side of the coin is that any given project needs at least some minimum number *X* of people, and if you don’t have that many people, you need to add more, even though it will cost delay. It’s a choice between delay and never finishing.

Price to Win Techniques

Pricing to win means bidding as low as possible to beat the competition and win the contract.

But then you will not get enough money to pay for the resources needed for development, and you will not show a profit.

This can hardly be called an estimation technique.

The idea is to price low enough to win the contract, but high enough to show a profit.

Experience

- models have to be calibrated to an organization

  Accuracy is perturbed by local factors, such as expertise, process, product type, definition of LOC.

- 100%+ errors are normal

  A software cost estimate model is doing well if it can estimate within 20% of the actual costs and within 70% of the actual time, and this is assuming that the model has been calibrated to this type and size of project!

- model parameters based on old projects/technology

  Weights and coefficients are based on empirical studies of past projects using old technology, and may be completely unlike new projects.

- Predictions can become self-fulfilling

  If estimates are used to generate the project plan, which is used by managers to manage the project, the project ends up having to conform to the estimate!

  Parkinson’s Law!

So why bother?

- You cannot control what you cannot measure

  You need this information to negotiate cost of product. You need this information also to plan project, to determine how many developers to hire or to assign to this project, and to know how long they'll be dedicated to this project and not to others. For these reasons, poor estimates may be better than no estimates.

- Your ability improves with experience.

  Don't get too caught up in an estimate. It's wrong. You'll get better, but you'll never master the problem.

- Some people will be better at estimating than others.

  Cost estimation is not a science. It’s an art, based on intuition and experience. Be wary of any method or tool vendor that claims to predict cost or effort to unrealistic precision, i.e., more than one significant digit!.

Why you underestimate by an order of magnitude

Fred Brooks observes:

Every body thinks *program* when he or she should be thinking *software system product*.

- program—what you write for yourself (and thus what you know)
- system—program that interfaces with other programs, directly or indirectly, costs 3 times as much as central program (more stuff to write)
- product—program written for others, that must therefore be robust, costs 3 times as much as central program
- software system product—program that is system *and* product, costs 9 times as much as central program

RE

Distribution of Code

10–20% of the code = central approximation.

80–90% of the code = exceptional details.

99.99% of execution time is spent in the central 10–20% of the code.

It's hard to test the exceptional details code, the 80–90% of the code, because it gets executed less than 0.01% of the execution time.

And correcting a defect in this 80-90% of the code likely involves changes in related code scattered all over the code.