14. METRICS + EARLY ESTIMATION

Last Mod: 2024-07-20

Metrics are measures.

Software metrics are various kinds of measurements of:

- software,
- software projects, and
- software quality.

They're critical if you want quantitative management information.

We'll also cover early project effort estimating.

Required Readings:

Section 14.9 below.

Optional References:

[Albrecht 79] "Measuring Application Development Productivity", by Allan Albrecht, in Proceedings, Joint SHARE/GUIDE/IBM Application Development Symposium, October 79, pp. 83-92.

[Albrecht 83] "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", by Allan J. Albrecht and John E Gaffney Jr., IEEE Transactions on

Software Engineering, November 1983, pp. 639-648.

[Boehm 81] “Software Engineering Economics” by Barry Boehm, Prentice-Hall, 1981.

[Dreger 89] "Function Point Analysis" by J. Brian Dreger, Prentice Hall, 1989.

[Jones 91] "Applied Software Measurement" by Capers Jones, McGraw-Hill, 1991.

[Jones95] “Determining Software Schedules” by Capers Jones, IEEE Software, February 1995, pages 73-75.

[McConnell95] “Rapid Development: Taming Wild Software Schedules” by Steve McConnell, Microsoft Press, 1996.

[Pressman 97] “Software Engineering: A Practitioner’s Approach, 4th ed.”, by Roger Pressman, McGraw-Hill, 1997. Especially Chapter 23 “Technical Metrics for OO Systems”.

Table of Contents

14.1 Intro to Metrics

Metrics in other manufacturing and design are widespread. E.g.

- temperature,
- voltage,
- physical size,
- flow rate,
- heat/power,
- signal-to-noise ratio.
- A 20-story building under construction is 10 stories up.
  - Or glass on outside is done for first 10 stories.

However, many feel software is harder to measure (especially when only partly complete).

That's since software is ethereal:

- Hidden on someone's hard disk, or
- a design in someone's mind).

But in "Software Engineering Metrics and Models", Conte wrote wryly that **"Not everything about software is not measurable"**.

Software metrics can be collected.

And can be used to assess:

- size,
- quality,
- productivity,
- performance,
- memory use, and
- to form a baseline for estimation of
  - effort
  - schedule
  - cost of future projects.
  - benefits of new languages/tools relative to past.

Demarco has stated that:

- "Costs migrate out of areas they are measured in".
  - E.g. If development costs are budgeted and measured closely, this emphasis will cause release of buggy software that is costly to fix later.

- "You can't control/manage what you can't measure".
  - E.g. Can compare post-release bug costs/dev cost if your company's QA and accounting systems don't collect this data.
  - E.g. Can't measure effectiveness of new programming language tools if productivity before and after for a small subset of your programmers is not measured. Without this, should you get the new tools for all programmers in your company.

14.2 Types of Software Metrics

Two classes of SW metrics:

- Directly measurable.
  - E.g. # of modules, classes, functions per class.
  - Depth of subclass hierarchy.
  - Effort in person-months.
  - Cost.
  - RAM and disk usage.
  - Errors per Kilo Lines of Code (KLOC).
  - Average length of program element names (indicating clear naming of variables and types).

- Indirectly measurable:
  - Functionality
  - Quality
  - Complexity
  - Reliability
  - Maintainability.

Uses for metrics:

- Assess quality
- Estimate size, effort, and cost.

Usually we are trying to quantify:

- Size (before or after coding).
  - E.g. # user commands, # of computed fields, Kilo Delivered Source Instructions (KDSI).
- Productivity
  - E.g. KDSI/person-month or KLOC/person-month.
- Quality
  - E.g. errors per phase, or error density.
- Cost
  - Total, or per KDSI, etc.

Metrics are usually classes as:

- Predictive metrics
  - Can be used to predict result metrics.
- Result metrics
  - E.g. Can we predict project cost before code is written.
  - Project quality.
  - Product code size.

How can we predict? By finding predictive metrics that will help.

Example predictive metrics:

- Size of requirements spec or draft user manual.
- Number of computer output values.
- Number of entities and number of entity attributes in requirements spec.

14.2.1 More Example Metrics

A number of years ago I learned that at Dynapro Systems in Vancouver, significant projects monitored the following 4 metrics in 4 graphs on a wall, updated weekly”

- Software Quality Index.
  - Weighted sum of bugs found per 1000 LOC, with serious bugs weighed more heavily.
- Productivity Index.
  - LOC per person-month.
- Review Effectiveness Index.
  - Average number of development phases delay before finding bug introduced/missed in earlier phase.
- Customer Responsiveness Index
  - Percentage of worthwhile feature suggestions implemented.

14.2.2 Code Size Notes

Code size is a result metric.

Very often you want to predict code size.

And thence use it to predict cost.

But it is not a predictive metric, so how can you use it to predict cost? Sometimes done in two stages.

- From requirements and design specs, predict size of code.
- From known productivity in various programming languages, predict effort and thus cost.

Code size definitions (**memorize**):

- KDSI – Kilo Delivered Source Instructions.
  - Not counting comments.
  - Not easy to count. Usually requires a small source analysis program.
- KLOC – Kilo Lines Of Code.
  - May or may not include comments, depending on your company.
  - In Cmpt-276, does include comments.
  - Very easy to count.

14.3 What Makes a Good Metric

What makes a good metric:

- Measurability.
- Independence.
- Complexity.
  - Is it a weak metric, or a function of many other measures?

In 1980 A.J. Albrecht of IBM suggested that some metrics were deceptive.

For example, in a large project more than half the effort is not coding.

So, if used a more powerful programming language that required fewer statements to write, your overall productivity measured in KDSI/person-month went **down**!

- Equivalently: Effort per line of code went up.

This was because you still had substantial non-coding effort, and less code for the ratio.

14.4 Prediction/Estimation

Estimating something via a prediction function is not a perfect process.

E.g. For parameters 'a' and 'b' you might have measured in your company,

$$\text{Effort} = a(\text{KLOC})^b$$

[Note: Effort is not linear with KLOC. Typically 'b' exponent is 1.12 – 1.30.]

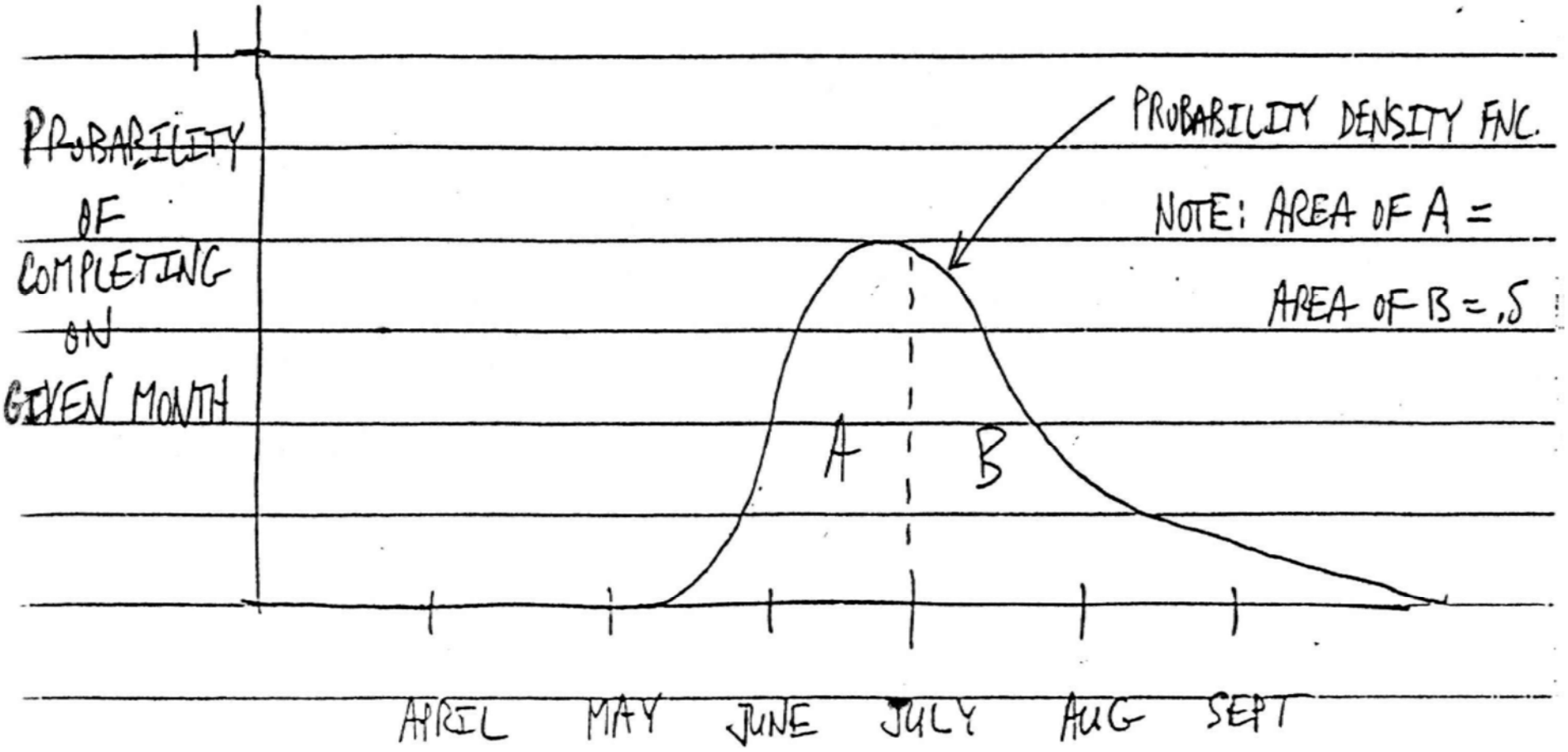
- The predictive metric(s) could be hard to measure. E.g. How commented is your KLOC?

- The predictive function is only correct on average.
  - It may vary by project type, and
  - perhaps you've never tried the project type you are about to embark upon!
- Finally, the result metric you are trying to compare to the prediction may itself be hard to measure.

14.5 What Is An Estimate

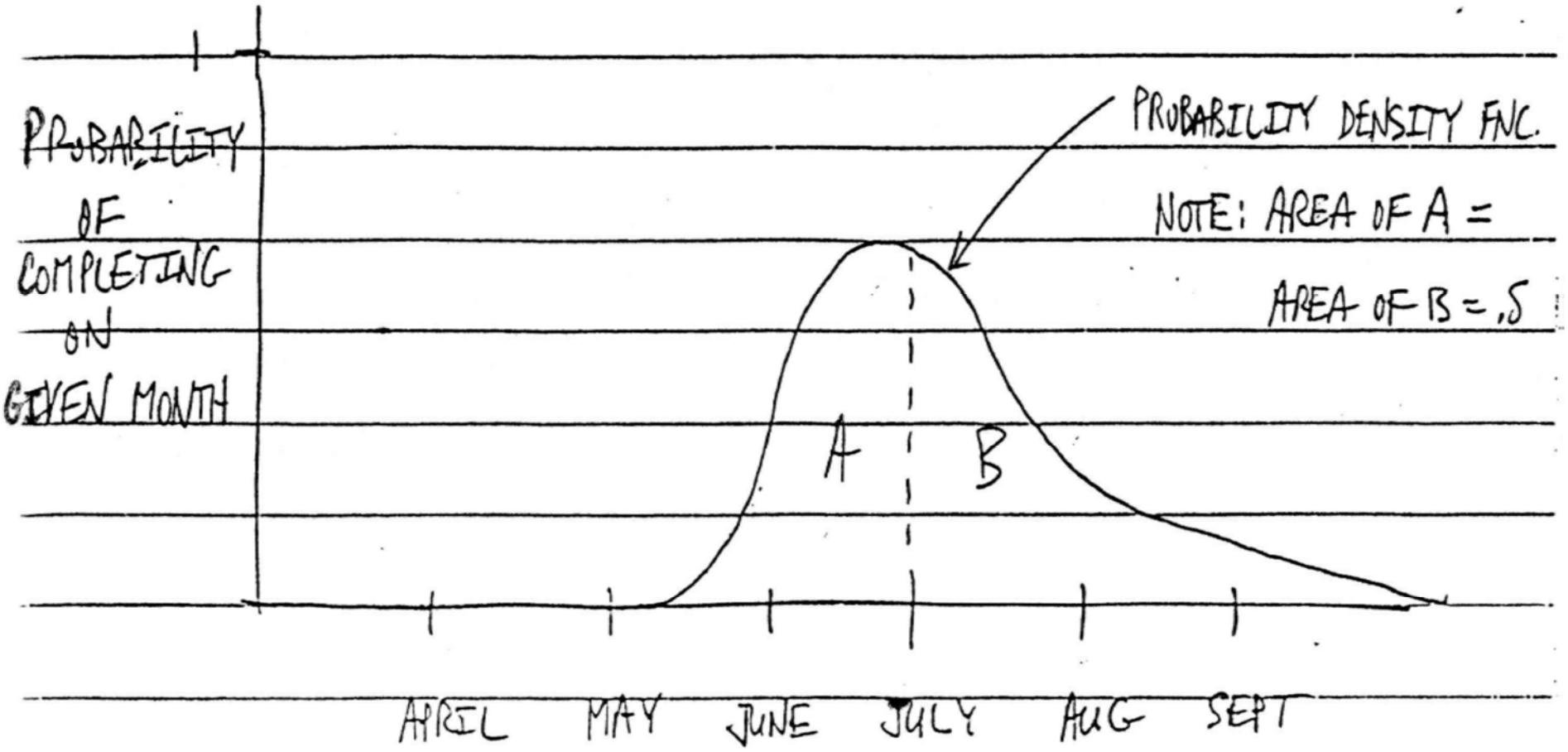
An estimate is a number which has a

- 50% chance of being too low, and a
- 50% chance of being too high.



Surprisingly, most programmers when asked to estimate when they will be done answer when they optimistically thing it would be code.

E.g. May 31 below.



They neglect to consider time for:

- debugging,
- documenting,
- unrelated other meetings,
- holidays, etc.

As a result 4/3 = 33% over time and budget.

I know of one project leader at a very large Vancouver firm whose project seemed to have been estimated to generously by her and her team.

She was asked to reduce her effort estimate somewhat arbitrarily by 40%.

She refused. She was nonetheless asked to manage the project anyway.

She was 3% over her original estimate.

Had she agreed to the 40% reduction, she would have been

103/(100-40) = 72% over budget.

- And poorly regarded for running over budget!

More than a decade ago I heard a Microsoft Vancouver development manager speak.

- He indicated their schedule estimation policy was very conservative.
- They expected effort and schedule to finish early.
- And any extra time to be devoted to extra testing or tiny feature tweeks.

14.6 Early Effort Estimation

Early effort estimation is done when only the

- requirements, or
- possibly the requirements plus some design

is done.

It's necessary to estimate budget needed internally, or accepting contracting jobs from external customers.

[Later in the course we'll discuss monitoring and managing a project.]

14.7 Effort Estimation Generally

Estimating a project often uses a two step process.

- First, estimate the size and complexity of the project.
  - E.g. KDSI.
  - E.g. How complicated the software and algorithms are?
  - Function/Feature Points [TBD]
- Second, from size/complexity, use another formula to estimate effort.
  - And further adjust by how productive your programming language, how experienced your team.

As does the literature, I'll use the term "complexity" to mean both size and complexity.

- Complexity = f(main variables) x CAF
  - CAF is complexity adjustment Factor, to take into account whether application is: GUI, networked, multi-threaded, etc.
  - Complexity might be measured in KLOC, KDSI, or more strange units like Function Points.

Then:

- Effort (in person-months) = g(complexity) x EAF
  - EAF is effort adjustment factor, to take into account programming language, experience, tools. It adjusts for *development process 'nature'*.

Complexity/size tends to be linear with the several predictive variables.

- E.g. If twice the files, likely twice the contribution of file complications to size/complexity.
- E.g. If twice as many reports, likely twice the contribution of report generation to size/complexity.

The following is just illustrative, not real:

Complexity = (1.3 x #ofFiles)

+ (2.6 x #MenuCommands)

+ (0.3 x #ofStates) + etc.

Effort is usually not linearly proportional to complexity/program size.

- Bigger programs take more effort per KLOC!
  - Due to more people, more meetings, more consulting with customer, more stringent quality standards, etc.
  - E.g. Effort = $0.05 \times (\text{KLOC})^{1.3} \times \text{EAF}$.

14.8 Function Points

Estimating size is a problem, because the same system can take 100 KDSI in one programming language and 200 KDSI in a less advanced language.

Thus, we perhaps shouldn't estimate size/complexity in KDSI, but in some other unit.

- Then have a conversion factor for the expressive strength of each language.

A unit called Function Points has been proposed that can estimate a value that represents program size, but not in KDSI.

Note Function Points were designed mainly for information systems.

Later we'll talk about a modification that better for a more widely varied types of software systems.

14.8.1 Calculating Function Points

Good references are [Dreger89], and the International Function Point User's Group (IFPUG) "Counting Practices Manual".

The steps used in calculating the function point rating of an application are as follows:

1. Calculate the number of inputs, outputs, inquires, files, and external interfaces needed to other software applications by your application.

As an example, the number of outputs for the following report is 4:

Monthly Sales for: 1990

Month	Monthly Sales
Jan	$10,250
Feb	$ 8,962
Mar	$ 7,841
Apr	$ 8,916
May	$ 9,666
Jun	$10,100
Jul	$11,110
Aug	$ 8,965
Sep	$ 7,823
Oct	$ 6,543
Nov	$ 5,218
Dec	$ 6,888
TOTAL	$112,927

You'll see there are only 4 types of output: the year, the month, the sales for a month, and the calculated total. The rest is just textural annotation and repetition, which

wouldn't really add much to the complexity/length of the code.

2. Evaluate whether each input, output, inquiry, file, and external interface is simple, average, or quite complex.

   Enter the number of each classification of driving metric into the following table, and multiply by the weighting factor.

Number of Simple Inputs:	_____	x	3	= _____
Number of Average Inputs:	__4___	x	4	= _____
Number of Complex Inputs:	_____	x	6	= _____
Number of Simple Outputs:	_____	x	4	= _____
Number of Average Outputs:	_____	x	5	= _____
Number of Complex Outputs:	_____	x	7	= _____
Number of Simple Inquiries:	_____	x	4	= _____
Number of Average Inquiries:	_____	x	5	= _____
Number of Complex Inquiries:	_____	x	7	= _____
Number of Simple Files:	_____	x	7	= _____
Number of Average Files:	_____	x	10	= _____
Number of Complex Files:	_____	x	15	= _____
Number of Simple Interfaces:	_____	x	5	= _____
Number of Average Interfaces:	_____	x	7	= _____
Number of Complex Interfaces:	_____	x	10	= _____
==========	==========	==========	==========	==========
TOTAL Raw Function Points:			RFP	= _____

Add all the values up to obtain the number of Raw Function Points (RFP) of complexity the product will have.

3. Evaluate the following 14 'product factors' by rating each on a scale of 0 (no influence) to 5 (significant influence). In particular, estimate whether they will likely increase the complexity of the software product and the length of code.

a)	Does the application require on-line data entry?	_____
b)	Do on-line data entry inputs have to be built over several screens?	_____
c)	Are the master files to be updated on-line?	_____
d)	Does the application require data communications?	_____
e)	Must the application be distributed, running on several CPUs?	_____
f)	Is the internal processing performed by the application complex?	_____
g)	Must the code be designed to be re-usable?	_____
h)	Is the system to be designed for multiple installations in different organizations/countries?	_____
i)	Should the application be designed to be user configurable?	_____
j)	Is performance critical?	_____
k)	Must the application run in an existing, heavily used environment?	_____
l)	Does the code need to handle conversion and installation considerations?	_____
m)	Must the application supply its own back-up and recovery function?	_____
n)	Are the inputs outputs, files, and inquiries complex?	_____

==========

TOTAL INFLUENCE **INFL** = _____

4. Calculate the Complexity Adjustment Factor (CAF) of the application by evaluating the following expression using INFL from above:

$$CAF = (0.65 + 0.01 \times INFL)$$

Then

$$FP = RFP \times CAF$$

Note: RFP is calculated as a linear function of the complexity driving variables.

CAF is:

- Not a function of the software language to be used.
- Not of the experience of the programmers,
- nor whether they have advanced CASE tools to improve their productivity.

It's a function of **only** the factors which might influence the length/complexity of the code.

Note that Function Points has these advantages:

- Known early, so predictive.
- Doesn't require hierarchical decomposition.
- Language independent.

It is known though that amount of code to implement 1 FP is:

- 213 LOC/FP for Macro Assembler.
- 128 for C.
- 53 for Java and C++

FP Disadvantages:

- Subjective.
- No physical meaning.
- Non-traditional
- Why are all 14 factors weighted equally?

Could then calculate these:

- # of requirements errors per FP or KDSI.
- # of design errors per FP.
- # run-time errors per FP.
- Etc.

14.8.2 Feature Point Variant

Capers Jones, a prolific software engineering author, has proposed both a simpler Function Points, and a simpler but broader method called **Feature Points**.

- Does away with evaluating every input, et cetera, for simple/average/complex.
- Weights less on Files and adds a weight on algorithms.
- Rather than 14, asks just 2 questions for CAF.

Here is his quick Reference Card on both Feature and Function Points.

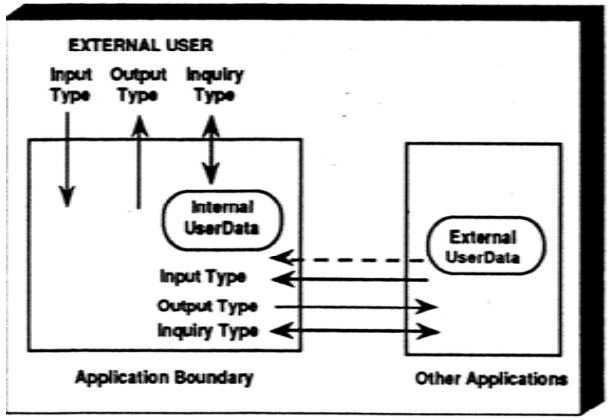
[Side Note: Jones below sometimes refers to interfaces as External User Data Groups (EU).]

Let's look in particular at the last column labelled P.6.

SPR METRIC ANALYSIS

REFERENCE CARD

Counting Rules for Function Points Feature Points



Software Productivity Research, Inc.

SPR

77 South Bedford Street
Burlington, MA 01803
(617) 273-0140

INTERNAL USER DATA GROUP (IU)

(ie. FILES)

Count each major logical group of user data or control information in the application as an internal user data group. Include each logical file, or within a data base, each logical group of data from the viewpoint of the user that is generated, used, and maintained by the application. Count each logical group of data as viewed by the user and as defined in the external design or data analysis rather than on the physical implementation.

COUNTING RECOMMENDATIONS

Do *not* include logical internal files that are *not* accessible to the user through external input, output or inquiry types.

Description	Count
• Logical entity of data from user viewpoint	1 IU
• Logical internal files generated or maintained by the application	1 IU
• User maintained table or file	1 IU
• Files accessible to the user through keyword(s) or parameter(s)	1 IU
• File used for data or control by sequential (batch) application	1 IU
• Each hierarchical path (leg) through a data base, derived from user requirements (include paths formed by secondary indices and logical relationships)	1 IU
• Intermediate or sort work file	0 IU
• File created by technology used (e.g., index file)	0 IU
• A "master file" only read by application	0 IU

EXTERNAL USER DATA GROUP (EU)

(ie. INTERFACES)

Count each major logical group of user data or control information used by the application (but maintained by another application) which crosses the application boundary. Include each logical file, or logical group of data from the viewpoint of the user, that is used by the application.

COUNTING RECOMMENDATIONS

Count each major logical group of user data or control information that enters the application from another application as an external user data group.

Description	Count
• File of records from another application	1EU
• Data base shared from other applications	1EU
• Logical internal file from another application used as a transaction	0EU, 1IT
• Each hierarchical path (leg) through a data base, from another application derived from user requirements (include paths formed by secondary indices and logical relationships)	1EU

INPUT TYPE (IT)

Count each *unique* user data or user control input type that enters the external boundary of the application being measured, and adds, changes, or deletes data in a logical internal file type. An external input type should be considered unique if it has a different format, or if the external logical design requires a processing logic different from other external input types of the same format.

COUNTING RECOMMENDATIONS

The recommendation most closely describing each input type should be used in counting each input type.

Description	Count
• Data screen	1 IT
• Multiple screens accumulated and processed as one transaction	1 IT
• Two data screens with the same format and processing logic	1 IT
• Two data screens with the same format and different processing logic	2 IT
• Data screen that is both input and output	1 IT, 1 OT
• Selection menu with save capability	1 IT
• Data screen with multiple functions	1 IT/Funct.
• Automatic data or transactions from other applications	1 IT
• User application control input	1 IT
• Input forms (OCR)	1 IT
• An update function following a query	1 IT
• Individual selections on menu screen	0 IT
• User maintained table or file	1 IT
• PF Key duplicate of a screen already counted as input	0 IT
• Light pen duplicate of a screen already counted as input	0 IT
• External input types introduced only because of the technology used	0 IT

OUTPUT TYPE (OT)

Count each unique user data or control output type that leaves the external boundary of the application being measured. An external output type should be considered unique 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. External output types usually consist of reports or messages to the user.

COUNTING RECOMMENDATIONS

The recommendations most closely describing each output type should be used in counting each output type.

Description	Count
• Data screen output	1 OT
• Batch report	1 OT
• Screen error message format associated with input type	1 OT

Continued...

COUNTING RECOMMENDATIONS (Cont)

- Data screen output 1 OT
- Start screen display or end screen display 1 OT
- Transaction file crossing the application boundary (at least) 1 OT
- Automatic data or transactions to other applications 1 OT
- Single error message on a screen 0 OT
- Error message sent to an operator as a result of an input transaction 1 OT
- Backup files (only if requested by the user) 0 OT
- Selection menu screen output with save capability 1 OT
- Output to screen and to printer 2 OT
- Output files created for technical reasons 0 OT
- User maintained table or file (at least) 1 OT

EXTERNAL INQUIRIES (QT)

Count each unique input/output combination, where an input causes and generates an immediate output, as an external inquiry type. An external inquiry type should be considered unique if it has a format different from other external inquiry types in either its input or output parts, or if the external design requires a processing logic different from other external inquiry types of the same format.

COUNTING RECOMMENDATIONS

The recommendation most closely describing each inquiry type should be used in counting each inquiry type.

Description	Count
• Online input and online output with no update of data in files	1QT
• Inquiry followed by an update input	1QT, 1IT
• Help screen input and output	1QT
• Selection menu screen input and output	1QT
• ADF Key Selection screen input and output	1QT
• ADF Master Rules screen input and output	1QT

Note: A major query facility or language should be decomposed into its hierarchical structure of IT(s), OT(s), and QT(s) using the existing definitions and current practices.

"BACKFIRE" METHOD

The "backfire" method for estimating Function Points is based on empirical relationships discovered to exist between source code and Function Points in all known languages. This method is based on tables of average values. It is useful for doing retrospective studies of projects completed long ago, and for easing the transition to Function Point metrics for people who are familiar with lines-of-code metrics.

Assembler	320*	DB Languages	40
C	128	Generators	32
COBOL	107	Object Oriented	29
Ada	71	Query Languages	25

*Statements per Function Point

FUNCTION POINT METHODOLOGY

The primary difference between the IBM and SPR Function Point methodologies is in the way they deal with complexity. The IBM techniques for assessing complexity are based on weighing fourteen influence factors and evaluating the numbers of field and file references.

The SPR technique for dealing with complexity separates the overall topic of "complexity" into two distinct questions that can be dealt with intuitively: 1) How complex are the algorithms or equations or problems in the software?; 2) How complex is the data structure of the application?

With the SPR Function Point method, it is not necessary to count the number of data element types, file types referenced, or record types. Neither is it necessary to assign low, average, or high values to each specific input, output, inquiry, data file, or interface.

The SPR complexity questions can be answered quickly by anyone familiar with an application, and they deal with the entire application, rather than with its elements.

FEATURE POINT METHODOLOGY

The SPR Feature Point metric is a superset of the IBM Function Point metric and introduces a new element (algorithms) in addition to the five standard Function Point parameters. The Feature Point method also reduces the Internal User Data Group weight from IBM's average value of 10 to an average value of 7.

Since Feature Points include algorithmic complexity, a definition of "algorithm" is appropriate. An algorithm is defined as the set of user required rules which must be completely expressed in order to solve a significant computational problem. For example, a square root extraction routine, a Julian date conversion routine, or an overtime pay calculation routine are all considered algorithms.

COMPLEXITY FACTOR

The complexity factor and multiplier is used to compute the Function Point Count measure. SPR uses a "quick-fire" method which can adjust the function point count by ±40%.

IBM and SPR use the same principles to identify the five elements (six with Feature Points) of counting. IBM then applies a weighting factor of high, low or average, depending on the number of data elements, file types referenced and record types invoked. SPR's quick-fire method assumes average weight in its methodology.

To adjust the raw FP count, IBM looks at fourteen variables of complexity that will adjust the count by ±35%. SPR simply requires answers to two questions which summarize the intent of IBM's fourteen complexity factors.

By answering 1 through 5 on both questions and adding the two values together, a complexity multiplier is then obtained using the simple chart provided.

Variance between methods only 1-5%

Function Point Counting

Element	Count Weight		Total
Input	___ x 4	=	___
Output	___ x 5	=	___
Inquiry	___ x 4	=	___
Internal User Data Group	___ x 10	=	___
External User Data Group	___ x 7	=	___
		TOTAL	___

• OR •

Feature Point Counting

Element	Count Weight		Total
Input	___ x 4	=	___
Output	___ x 5	=	___
Inquiry	___ x 4	=	___
Internal User Data Group	___ x 7	=	___
External User Data Group	___ x 7	=	___
Algorithm	___ x 3	=	___
		TOTAL	___

Function/Feature Count (FC)

Total Unadjusted Function/Feature Points = ___

Complexity Factor and Multiplier

Problem Complexity?
1) Simple algorithms and simple calculations
2) Majority of simple algorithms and calculations
3) Algorithms and calculations of average complexity
4) Some difficult or complex algorithms
5) Many difficult algorithms and complex calculations

Data Complexity?
1) Simple data with few variables
2) Numerous variables, but simple data relationships
3) Multiple files, fields, and data intersections
4) Complex file structures and data intersections
5) Very complex file structures and data intersections

Sum of Problem and Data Complexity	2	3	4	5	6	7	8	9	10
Complexity Multiplier	.6	.7	.8	.9	1.0	1.1	1.2	1.3	1.4

FP Count Measure:

FC X Complexity Multiplier = ___

Notice on Reference Card P.6, for Feature Points:

- Only 6 types of element counts, which add to a Feature Count.
- Only 2 Complexity Factors, which are added together.
  - E.g. From Jone's Reference Card above, if Problem complexity is 3) and Data Complexity is 4), the sum is: 3 + 4 = 7.
- At bottom, a simple look-up table to convert this sum to a complexity multiplier.

Finally, Feature Points:

FP = Feature Count x Complexity multiplier.

Example: Say,

Feature Counting is 5+20+7+6+7+9 = 54.

Problem and Data Complexity is 3 + 4 = 7.

Complexity multiplier is 7 --> 1.1

Then FP = 54 * 1.1 = 59.4

Interestingly, there was not very much research done to map feature/function points to effort (in person-months). This instructor had to personally analyze graphs in Figure 3.11 and 3.12 of [Jones 91]. The result was this approximation:

$$\text{Effort(in person-months)} = 0.05 \times (\text{FP})^{1.3} \times \text{EAF}$$

There should be an effort adjustment factor (EAF), that takes into account programming language strength, programmer quality, other programming tools, etc. But none was given in [Jone's 91].

However, at least some typical schedule data was published in Jones more recent book [Jones 95]

This indicates that typically staffed projects have a duration of:

Duration (in months) = $(FP)^{0.43}$

And from this you can see what the typical staff size is by:

Staff = Effort/Duration.

Having a formula for Duration at least gives us an estimate of staff so we don't choose to compress the schedule unrealistically with a huge staff.

14.9 Required Reading on EAF

- Did you know that we have no idea how to calculate this EAF?
- We don't know whether the above 3 productivity factors (programming language strength, programmer quality, other programming tools) should be multiplied together to give EAF? Or combined in a linear fashion?
- Can you imagine a multi-billion dollar industry like construction or aircraft manufacturing having no idea of the effect that various construction materials, personnel skills, and tools have on the productivity?

- How could they justify buying their staff extra power tools?
  - Why should software developers purchase CASE tools for their staff, if we don't have a good way of factoring the increase productivity into the effort estimations needed to bid on software development contracts?
  - Can we justify the increasing cost of university grads to software developers (maybe they should just hire cheaper BCIT grads)?

This is a horrible state of affairs. We are a **primitive** profession in comparison to other, better developed industries!

Wouldn't it be nice to have even just some simple linear estimates to help us compute EAF. e.g.

- effort per FP to write:
  - a user friendly or mil-spec user manual.
  - a commercial quality user manual
  - a quick and dirty user manual.
- effort per FP to do:
  - a mil-spec design
  - a commercial quality design
  - a rough design.
- effort per FP to do:
  - mil-spec commented and unit-tested code

- commercially commented and unit-tested code.

- effort per FP to do:
  - mil-spec integration and system test, vs.
  - commercial integration and system test.
- Number of pages of requirements spec, manual, architectural design document per Function Point for a:
  - mil-spec project
  - commercial project.

Builders, plumbers, and electricians have such tables of information for their industry. Why not software engineers?

14.9.1 Discussion

- I have shown you what function points are, and stated that the rather metricless software industry is starting to accept them as a reasonable, implementation-independent predictive metric of software product 'size'.
- I have mentioned that effort is considered to be a polynomial function of size. This has been postulated for a number of years, and in fact is embodied in the COCOMO models for effort estimation.
- What has been missing is connecting the two. I have not seen published in any book or journal the suggested form that my effort

estimation takes (i.e. proportional to $(FP)^{1.3}$, even with an arguably different exponent!). Yet this is an obvious relationship, looking at historical data, if FP is assumed to be proportional to code size.

The intermediate COCOMO model discussed next does have an adjustment factor that takes into account some product factors and some implementation factors. But it merges them all together. **It's extremely important to keep:**

- **the prediction of FP, and**
- **of Effort (given the FP) separate**.

Why?

- The idea is that we should use the linear FP, and CAF corrections, to estimate FP product size/complexity. We will then be able to say *whether two **different** products are the same 'size'*.
  - Two quite different products (e.g. a spreadsheet and a database) should, if they were the same FP size, take the same amount of effort to produce (given the same EAF).

- Similarly, we should separately use the polynomial and EAF equations to estimate the implementation effort.
  - We will then be able to predict the *difference in effort needed to develop the **same** product using two different implementation efficiencies* (different programmers/languages/tools).
  - It might even be found in future that certain developers who handle large projects well, only need an exponent of 1.25!
  - And without a separate (from CAF) examination of EAF, it is impossible to understand the effect of new implementation languages and tools.

Finally, it's important to realize that we cannot nail either of these down accurately without the other, because we rarely develop the same product twice to see the influence of doing it with different staff/language/tools (who's got an extra $1M to try it another way?). We need the same staff to develop different products, and those products which take the same amount of effort should be deemed to have the same FP. This will allow us to fine-tune the weights and constants in the FP calculation.

Once that's nailed down, over the normal course of the software development business, we will develop different products which happen to have

the same FP. Often these will be developed by different staff, in different languages, and with different tools. We will thus be able to evaluate the productivity of different implementation aspects, and thus develop a way to estimate the EAF. In the future you may see a set of questions regarding implementation aspects that you will rate between 1 and 5, and combine somehow into an EAF!

14.10 COCOMO Cost Model

Reference: [Boehm 81].

Barry Boehm published several effort and duration prediction equations. Was based on data gathered from a number of types of projects, and various sizes of each type.

His predictive function start with a guess of the number of lines of code (LOC).

His basic model is this:

Effort $E = a\,(KLOC)^b$ - in person-months.

Duration $D = c(E)^d$ - in months.

Typical staff size $S = E/D$ - in persons.

The coefficients a, b, c, and d, are given for different classes of projects as follows:

Project Class	**a**	**b**	**c**	**d**
Organic (general application)	2.4 (3.2)	1.05	2.5	.38
Semi-detached (moderately spec'ed special utilities)	3.0 (3.0)	1.12	2.5	.35
Embedded – Rigorously spec'ed system software.	3.6 (2.8)	1.2	2.5	.32

Example:

Given a semi-detached (utility) project of 10,000 lines:

Effort $E = 3\ (KLOC)^{1.12}$

$= 3\ (10)^{1.12} = 3 \times 13.2$

$= 39.5$ person-months

Duration $D = 2.5(E)^{0.35}$

$= 2.5(39.5)^{0.35}$

$= 2.5 \times 3.6 = 9$ months

Staff $S = E/D$

$= 39.5$ p-m/9 months $= 4.4$ persons.

Point out common errors for this example on final exam questions.

- Used 10,000 KLOC rather than 10.
- Got concept of person-months mixed up.
  - Like energy behind a hydro dam in MWatt-hours, effort is a product unit: it's number of persons working for a number of months.

14.10.1 Intermediate COCOMO Model

Boehm tried to produce a better model that took into account complexity adjustment factors and effort adjustment factors.

- And unfortunately mushed them all together.

Using table on next page, he multiplied (rather than added) a number of subfactors together in what I call the Boehm Effort Adjustment Factor (BEAF).

He then, (using the parenthized values in above table), computed:

$E = a(KLOC)^b * BEAF$

Boehm asked estimators to choose a value for each of the items below:

Complexity factors:

- Product attributes:
  - Reliability required (.75 – 1.4)
  - Database size (.94 -1.16)
  - Product complexity (.7 – 1.65)
- Computer attributes:
  - Performance demands (1 -1.66
  - Constrained RAM memory (1 -1.56)
  - Virtual machine volatility (.87 – 1.3)
  - Computer turn-around time (.87 – 1.15)

Effort Factors:

- Personnel Attributes:
  - Analyst expertise (1.46 - .71)
  - Language experience (1.14 - .95)
  - Programmer expertise (1.42 - .70)
  - Domain familiarity (1.29 - .82)
  - Machine+OS experience (1.21 - .90)
- Process Attributes:
  - Modern programming practices (1.24 - .82)
  - SW development tools (1.24 - .83)
  - Tight schedule (1.23 – 1.10)

Remember, if you increase the staff over the suggested size, add more time for interpersonal communications, training, and more effort.

Boehm in 1981 stated:

"Today, a software cost estimation model is doing well if it can estimate 70% of the time to within 20%. And on it's own turf (i.e. on the kinds of software domains and projects it has been calibrated on). This is not as precise as we'd like. But it is accurate enough to provide a good deal of helps in software engineering economic analysis and decision making."

I.e. Not going 50 – 100% over budget!