COCOMO II

1. Introduction

COCOMO 2 is tuned to modern software life cycles. The original COCOMO model has been very successful, but it doesn't apply to newer software development practices as well as it does to traditional practices. COCOMO II targets the software projects of the 1990s and 2000s, and will continue to evolve over the next few years.

The primary objectives of the COCOMO 2 effort are:

- To develop a software cost and schedule estimation model tuned to the life cycle practices of the 1990's and 2000's.
- To develop software cost database and tool support capabilities for continuous model improvement.
- To provide a quantitative analytic framework, and set of tools and techniques for evaluating the effects of software technology improvements on software life cycle costs and schedules.

COCOMO 2 is really three different models:

- The Application Composition Model

  Suitable for projects built with modern GUI-builder tools. Based on new Object Points.

- The Early Design Model

  You can use this model to get rough estimates of a project's cost and duration before you've determined it's entire architecture. It uses a small set of new Cost Drivers, and new estimating equations. Based on Unadjusted Function Points or KSLOC.

- The Post-Architecture Model

  This is the most detailed COCOMO 2 model. You'll use it after you've developed your project's overall architecture. It has new cost drivers, new line counting rules, and new equations.

2. Application Composition : Object points

Object Point estimation is a relatively new software sizing approach, but it is well-matched to the practices in the Applications Composition sector. It is also a good match to associated prototyping efforts, based on the use of a rapid-composition Integrated Computer Aided Software Environment (ICASE) providing graphic user interface builders, software development tools, and large, composable infrastructure and applications components. In these areas, it has compared well to Function Point estimation on a nontrivial (but still limited) set of applications.

Baseline Object Point Estimation Procedure

Step 1: Assess Object-Counts: estimate the number of screens, reports, and 3GL components that will comprise this application. Assume the standard definitions of these objects in your ICASE environment.

Step 2: Classify each object instance into simple, medium and difficult complexity levels depending on values of characteristic dimensions. Use the following scheme:

For Screens				For Reports			
Number of Views contained	# and source of data tables			Number of Sections contained	# and source of data tables		
	Total < 4 (< 2 srvr < 3 clnt)	Total < 8 (2/3 srvr 3-5 clnt)	Total 8+ (> 3 srvr > 5 clnt)		Total < 4 (< 2 srvr < 3 clnt)	Total < 8 (2/3 srvr 3-5 clnt)	Total 8+ (> 3 srvr > 5 clnt)
< 3	simple	simple	medium	0 or 1	simple	simple	medium
3 - 7	simple	medium	difficult	2 or 3	simple	medium	difficult
> 8	medium	difficult	difficult	4 +	medium	difficult	difficult

Step 3: Weigh the number in each cell using the following scheme. The weights reflect the relative effort required to implement an instance of that complexity level.:

Object Type	Complexity-Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Component			10

Step 4: Determine Object-Points: add all the weighted object instances to get one number, the Object-Point count.

Step 5: Estimate percentage of reuse you expect to be achieved in this project. Compute the New Object Points to be developed, *NOP = (Object-Points) (100 - %reuse)/ 100.*

Step 6: Determine a productivity rate, PROD = NOP / person-month, from the following scheme

Developers' experience and capability	Very Low	Low	Nominal	High	Very High
ICASE maturity and capability	Very Low	Low	Nominal	High	Very High
PROD	4	7	13	25	50

Step 7: Compute the estimated person-months: PM = NOP / PROD.

Definitions of terms in Figure are as follows:

• NOP: New Object Points (Object Point count adjusted for reuse)

• srvr: number of server (mainframe or equivalent) data tables used in conjunction with the SCREEN or REPORT.

• clnt: number of client (personal workstation) data tables used in conjunction with the SCREEN or REPORT.

• %reuse: the percentage of screens, reports, and 3GL modules reused from previous applications, pro-rated by degree of reuse.

3. Function Count Procedure

Step 1: Determine function counts by type. The unadjusted function counts should be counted by a lead technical person based on information in the software requirements and design documents. The number of each of the five user function types should be counted (Internal Logical File* (ILF), External Interface File (EIF), External Input (EI), External Output (EO), and External Inquiry (EQ)).

Step 2: Determine complexity-level function counts. Classify each function count into Low, Average and High complexity levels depending on the number of data element types contained and the number of file types referenced. Use the following scheme:

For ILF and EIF				For EO and EQ				For EI			
Record Elements	Data Elements			File Types	Data Elements			File Types	Data Elements		
	1 - 19	20 - 50	51+		1 - 5	6 - 19	20+		1 - 4	5 - 15	16+
1	Low	Low	Avg	0 or 1	Low	Low	Avg	0 or 1	Low	Low	Avg
2 - 5	Low	Avg	High	2 - 3	Low	Avg	High	2 - 3	Low	Avg	High
6+	Avg	High	High	4+	Avg	High	High	3+	Avg	High	High

Step 3: Apply complexity weights. Weight the number in each cell using the following scheme. The weights reflect the relative value of the function to the user.

Function Type	Complexity-Weight		
	Low	Average	High
Internal Logical Files	7	10	15
External Interfaces Files	5	7	10
External Inputs	3	4	6
External Outputs	4	5	7
External Inquiries	3	4	6

Step 4: Compute Unadjusted Function Points. Add all the weighted functions counts to get one number, the Unadjusted Function Points.

* Note: The word *file* refers to a logically related group of data and not the physical implementation of those groups of data

4. Converting Function Points to Lines of Code

To determine the nominal person months for the Early Design model, the unadjusted function points have to be converted to source lines of code in the implementation language (assembly, higher order language, fourth-generation language, etc.) in order to assess the relative conciseness of implementation per function point.

Language	SLOC / UFP
Ada	71
AI Shell	49
APL	32
Assembly	320
Assembly (Macro)	213
ANSI/Quick/Turbo Basic	64
Basic - Compiled	91
Basic - Interpreted	128
C	128
C++	29
ANSI Cobol 85	91
Fortan 77	105
Forth	64
Jovial	105
Lisp	64
Modula 2	80
Pascal	91
Prolog	64
Report Generator	80
Spreadsheet	6

5. Development Effort Estimates

In COCOMO II effort is expressed as Person Months (PM). person month is the amount of time one person spends working on the software development project for one month.

$$PM_{nominal} = A \times (Size)^{B}$$

The inputs are the *Size* of software development, a constant, *A*, and a scale factor, *B*. The size is in units of thousands of source lines of code (KSLOC).
The constant, *A*, is used to capture the multiplicative effects on effort with projects of increasing size.

The scale (or exponential) factor, *B*, accounts for the relative economies or diseconomies of scale encountered for software projects of different sizes.
If *B < 1.0*, the project exhibits economies of scale. If the product's size is doubled, the project effort is less than doubled. The project's productivity increases as the product size is increased. Some project economies of scale can be achieved via project-specific tools (e.g., simulations, testbeds) but in general these are difficult to achieve. For small projects, fixed start-up costs such as tool tailoring and setup of standards and administrative reports are often a source of economies of scale.
If *B = 1.0*, the economies and diseconomies of scale are in balance. This linear model is often used for cost estimation of small projects. It is used for the COCOMO 2 *Applications Composition* model.
If *B > 1.0*, the project exhibits diseconomies of scale. This is generally due to two main factors: growth of interpersonal communications overhead and growth of large-system integration overhead. Larger projects will have more personnel, and thus more interpersonal communications paths consuming overhead. Integrating a small product as part of a larger product requires not only the effort to develop the small product, but also the additional overhead effort to design, maintain, integrate, and test its interfaces with the remainder of the product.

A project's numerical ratings *W* are summed across all of the factors, and used to determine a scale exponent *B* via the following formula:

$$B = 1.01 + 0.01 \Sigma W_i$$

Rating Scheme for the COCOMO 2.0 Scale Factors

Scale Factors (W_i)	Very Low (5)	Low (4)	Nominal (3)	High (2)	Very High (1)	Extra High (0)
Precedentedness	thoroughly unprecedented	largely unprecedented	somewhat unprecedented	generally familiar	largely familiar	throughly familiar
Development Flexibility	rigorous	occasional relaxation	some relaxation	general conformity	some conformity	general goals
Architecture / risk resolution*	little (20%)	some (40%)	often (60%)	generally (75%)	mostly (90%)	full (100%)
Team cohesion	very difficult interactions	some difficult interactions	basically cooperative interactions	largely cooperative	highly cooperative	seamless interactions
Process maturity†	Weighted average of "Yes" answers to CMM Maturity Questionnaire					

The form of the Process Maturity scale is being resolved in coordination with the SEI. The intent is to produce a process maturity rating as a weighted average of the project's percentage compliance levels to the 18 Key Process Areas in Version 1.1 of the Capability Maturity Model-based [Paulk et al. 1993] rather than to use the previous 1-to-5 maturity levels. The weights to be applied to the Key Process Areas are still being determined.

6. Cost Factors: Effort-Multiplier Cost Drivers

COCOMO 2.0 uses a set of effort multipliers to adjust the nominal person-month estimate obtained from the project's size and exponent drivers:

$$PM_{adjusted} = PM_{nominal} \times \left(\prod_i EM_i \right)$$

Early Design and Post-Architecture Cost Drivers

Early Design Cost Driver	Counterpart Combined Post-Arch. Cost Driver
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PERS	ACAP, PCAP, PCON
PREX	AEXP, PEXP, LTEX
FCIL	TOOL, SITE
SCED	SCED

ACAP Analyst Capability
AEXP Applications Experience
CPLX Product Complexity
DATA Database Size
DOCU Documentation to match lifecycle needs
FCIL Facilities
LTEX Language and Tool Experience
PCAP Programmer Capability
PCON Personnel Continuity
PDIF Platform Difficulty
PERS Personnel Capability
PEXP Platform Experience
PREX Personnel Experience
PVOL Platform Volatility
RCPX Product Reliability and Complexity
RELY Required Software Reliability
RUSE Required Reusability
SCED Required Development Schedule
STOR Main Storage Constraint
TIME Execution Time Constraint
TOOL Use of Software Tools

7. Development Schedule Estimates

The initial baseline schedule equation for all three COCOMO 2.0 models is:

$$TDEV = [3.0 \times (PM)^{(0.33 + 0.2 \times (B - 1.01))}] \times \frac{SCEDPercentage}{100}$$

where *TDEV* is the calendar time in months from the determination of its requirements baseline to the completion of an acceptance activity certifying that the product satisfies its requirements. PM is the estimated person-months excluding the SCED effort multiplier, and *SCEDPercentage* is the schedule compression / expansion percentage in the SCED cost driver rating. Future versions of COCOMO 2.0 will have a more extensive schedule estimation model, reflecting the different classes of process model a project can use; the effects of reusable and COTS software; and the effects of applications composition capabilities.