COCOMO II Model Definition Manual

Acknowledgments

COCOMO II is an effort to update the well-known COCOMO (Constructive Cost Model) software cost estimation model originally published in Software Engineering Economics by Dr. Barry Boehm in 1981. It focuses on issues such as non-sequential and rapid-development process models; reuse-driven approaches involving commercial-off-the-shelf (COTS) packages, reengineering, applications composition, and application generation capabilities; object-oriented approaches supported by distributed middleware; software process maturity effects and process-driven quality estimation. The COCOMO II research effort is being led by the **Director of the Center for Software Engineering at USC**, **Dr. Barry Boehm,** and the other researchers (in alphabetic order) are listed below.

Chris Abts	**Graduate Research Assistant**
Brad Clark	**Graduate Research Assistant**
Sunita Devnani-Chulani	**Graduate Research Assistant**
Ellis Horowitz	**Chair, Computer Science Department, USC**
Ray Madachy	**Adjunct Assistant Professor**
Don Reifer	**Visiting Associate**
Rick Selby	**Professor, UCI**
Bert Steece	**Deputy Dean of Faculty, Marshall School of Business, USC**

This work is being supported both financially and technically by the COCOMO II Program Affiliates: Aerospace, Air Force Cost Analysis Agency, Allied Signal, AT&T, Bellcore, EDS, Raytheon E-Systems, GDE Systems, Hughes, IDA, JPL, Litton, Lockheed Martin, Loral, MCC, MDAC, Motorola, Northrop Grumman, Rational, Rockwell, SAIC, SEI, SPC, Sun, TI, TRW, USAF Rome Lab, US Army Research Labs, Xerox.

The successive versions of the tool based on the COCOMO model have been developed as part of a Graduate Level Course Project by several student development teams led by **Dr. Ellis Horowitz**. The current version, USC COCOMO II.1998.0, has been developed by **Jongmoon Baik.**

Overall Model Definition

The four main elements of the COCOMO II strategy are:

- Preserve the openness of the original COCOMO;
- Key the structure of COCOMO II to the future software marketplace sectors described earlier;
- Key the inputs and outputs of the COCOMO II submodels to the level of information available;
- Enable the COCOMO II submodels to be tailored to a project's particular process strategy.

COCOMO II follows the openness principles used in the original COCOMO. Thus, all of its relationships and algorithms will be publicly available. Also, all of its interfaces are designed to be public, well-defined, and parametrized, so that complementary preprocessors (analogy, case-based, or other size estimation models), post-processors (project planning and control tools, project dynamics models, risk analyzers), and higher level packages (project management packages, product negotiation aids), can be combined straightforwardly with COCOMO II.

To support the software marketplace sectors above, COCOMO II provides a family of increasingly detailed software cost estimation models, each tuned to the sectors' needs and type of information available to support software cost estimation.

COCOMO II Models for the Software Marketplace Sectors

The COCOMO II capability for estimation of Application Generator, System Integration, or Infrastructure developments is based on two increasingly detailed estimation models for subsequent portions of the life cycle, *Early Design* and *Post-Architecture*.

COCOMO II Model Rationale and Elaboration

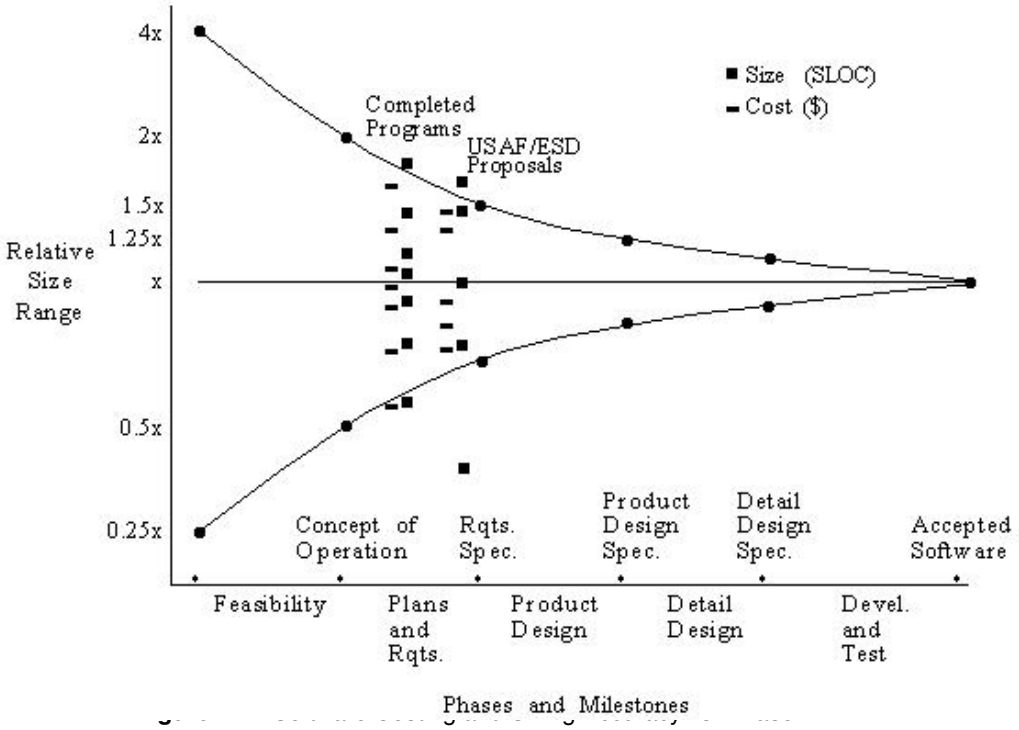
The rationale for providing this tailorable mix of models rests on three primary premises.

First, unlike the initial COCOMO situation in the late 1970's, in which there was a single, preferred software life cycle model, current and future software projects will be tailoring their processes to their particular process drivers. These process drivers include COTS or reusable software availability; degree of understanding of architectures and requirements; market window or other schedule constraints; size; and required reliability (see [Boehm 1989, pp. 436-37] for an example of such tailoring guidelines).

Second, the granularity of the software cost estimation model used needs to be consistent with the granularity of the information available to support software cost estimation. In the early stages of a software project, very little may be known about the size of the product to be developed, the nature of the target platform, the nature of the personnel to be involved in the project, or the detailed specifics of the process to be used.

Figure I-1, extended from [Boehm 1981, p. 311], indicates the effect of project uncertainties on the accuracy of software size and cost estimates. In the very early stages, one may not know the specific nature of the product to be developed to better than a factor of 4. As the life cycle proceeds, and product decisions are made, the nature of the products and its consequent size are better known, and the nature of the process and its consequent cost drivers[1] are better known. The earlier "completed programs" size and effort data points in Figure I-1 are the actual sizes and efforts of seven software products built to an imprecisely-defined specification [Boehm et. al. 1984][2]. The later "USAF/ESD proposals" data points are from five proposals submitted to the U.S. Air Force Electronic Systems Division in response to a fairly thorough specification [Devenny 1976].

Third, given the situation in premises 1 and 2, COCOMO II enables projects to furnish coarse-grained cost driver information in the early project stages, and increasingly fine-grained information in later stages. Consequently, COCOMO II



does not produce point estimates of software cost and effort, but rather range estimates tied to the degree of definition of the estimation inputs. The uncertainty ranges in Figure I-1 are used as starting points for these estimation ranges.

With respect to process strategy, Application Generator, System Integration, and Infrastructure software projects will involve a mix of three major process models, The appropriate models will depend on the project marketplace drivers and degree of product understanding.

The Early Design model involves exploration of alternative software/system architectures and concepts of operation. At this stage, not enough is generally known to support fine-grain cost estimation. The corresponding COCOMO II capability involves the use of function points and a course-grained set of 7 cost drivers (e.g. two cost drivers for Personnel Capability and Personnel Experience in place of the 6 COCOMO II Post-Architecture model cost drivers covering various aspects of personnel capability, continuity, and experience).

The Post-Architecture model involves the actual development and maintenance of a software product. This stage proceeds most cost-effectively if a software life-cycle architecture has been developed; validated with respect to the system's mission, concept of operation, and risk; and established as the framework for the product. The corresponding COCOMO II model has about the same granularity as the previous COCOMO and Ada COCOMO models. It uses source instructions and / or function points for sizing, with modifiers for reuse and software breakage; a set of 17 multiplicative cost drivers; and a set of 5 factors determining the project's scaling exponent. These factors replace the development modes (Organic,

1 A cost driver refers to a particular characteristic of the software development that has the effect of increasing or decreasing the amount of development effort, e.g. required product reliability, execution time constraints, project team application experience.

2 These seven projects implemented the same algorithmic version of the Intermediate COCOMO cost model, but with the use of different interpretations of the other product specifications: produce a "friendly user interface" with a "single-user file system."

Semidetached, or Embedded) in the original COCOMO model, and refine the four exponent-scaling factors in Ada COCOMO.

To summarize, COCOMO II provides the following three-stage series of models for estimation of Application Generator, System Integration, and Infrastructure software projects:

1. The earliest phases or spiral cycles will generally involve prototyping, using the Application Composition model capabilities. The COCOMO II Application Composition model supports these phases, and any other prototyping activities occurring later in the life cycle.
2. The next phases or spiral cycles will generally involve exploration of architectural alternatives or incremental development strategies. To support these activities, COCOMO II provides an early estimation model called the Early Design model. This level of detail in this model is consistent with the general level of information available and the general level of estimation accuracy needed at this stage.
3. Once the project is ready to develop and sustain a fielded system, it should have a life- cycle architecture, which provides more accurate information on cost driver inputs, and enables more accurate cost estimates. To support this stage, COCOMO II provides the Post-Architecture model.

The above should be considered as current working hypotheses about the most effective forms for COCOMO II. They will be subject to revision based on subsequent data analysis. Data analysis should also enable the further calibration of the relationships between object points, function points, and source lines of code for various languages and composition systems, enabling flexibility in the choice of sizing parameters.

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. This number is exclusive of holidays and vacations but accounts for weekend time off. The number of person months is different from the time it will take the project to complete; this is called the development schedule. For example, a project may be estimated to require 50 PM of effort but have a schedule of 11 months.

Equation I-1 is the base model for the Early Design and Post-Architecture cost estimation models. 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). This is derived from estimating the size of software modules that will constitute the application program. It can also be estimated from unadjusted function points (UFP), converted to SLOC then divided by one thousand. Procedures for counting SLOC or UFP are explained in the chapters on the Post- Architecture and Early Design models respectively.

The scale (or exponential) factor, *B*, accounts for the relative economies or diseconomies of scale encountered for software projects of different sizes [Banker et al 1994a]. The constant, *A*, is used to capture the multiplicative effects on effort with projects of increasing size. The nominal effort for a given size project and expressed as person months (PM) is given by Equation I-1.

(EQ I-1)

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

Software Economies and Diseconomies of Scale

Software cost estimation models often have an exponential factor to account for the relative economies or diseconomies of scale encountered in different size software projects. The exponent, *B*, in Equation I-1 is used to capture these effects.

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 II *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.

See [Banker et al 1994a] for a further discussion of software economies and diseconomies of scale.

Previous Approaches

The data analysis on the original COCOMO indicated that its projects exhibited net diseconomies of scale. The projects factored into three classes or modes of software development (Organic, Semidetached, and Embedded), whose exponents B were 1.05, 1.12, and 1.20, respectively. The distinguishing factors of these modes were basically environmental: Embedded-mode projects were more unprecedented, requiring more communication overhead and complex integration; and less flexible, requiring more communications overhead and extra effort to resolve issues within tight schedule, budget, interface, and performance constraints.

The scaling model in Ada COCOMO continued to exhibit diseconomies of scale, but recognized that a good deal of the diseconomy could be reduced via management controllables. Communications overhead and integration overhead could be reduced significantly by early risk and error elimination; by using thorough, validated architectural specifications; and by stabilizing requirements. These practices were combined into an Ada process model [Boehm and Royce 1989, Royce 1990]. The project's use of these practices, and an Ada process model experience or maturity factor, were used in Ada COCOMO to determine the scale factor B.

Ada COCOMO applied this approach to only one of the COCOMO development modes, the Embedded mode. Rather than a single exponent $B = 1.20$ for this mode, Ada COCOMO enabled B to vary from 1.04 to 1.24, depending on the project's progress in reducing diseconomies of scale via early risk elimination, solid architecture, stable requirements, and Ada process maturity.

COCOMO II combines the COCOMO and Ada COCOMO scaling approaches into a single rating-driven model. It is similar to that of Ada COCOMO in having additive factors applied to a base exponent B. It includes the Ada COCOMO factors, but combines the architecture and risk factors into a single factor, and replaces the Ada process maturity factor with a Software Engineering Institute (SEI) process maturity factor (The exact form of this factor is still being worked out with the SEI). The scaling model also adds two factors, precedentedness and flexibility, to account for the mode effects in original COCOMO, and adds a Team Cohesiveness factor to account for the diseconomy-of-scale effects on software projects whose developers, customers, and users have difficulty in synchronizing their efforts. It does not include the Ada COCOMO Requirements Volatility factor, which is now covered by increasing the effective product size via the Breakage factor.

Scaling Drivers

Equation I-2 defines the exponent, B, used in Equation I-1. Table I-1 provides the rating levels for the COCOMO II scale drivers. The selection of scale drivers is based on the rationale that they are a significant source of exponential variation on a project's effort or productivity variation. Each scale driver has a range of rating levels, from Very Low to Extra High. Each rating level has a weight, W, and the specific value of the weight is called a scale factor. A project's scale factors, W_i, are summed across all of the factors, and used to determine a scale exponent, B, via the following formula:

(EQ I-2)

$$B = 0.91 + 0.01 \times \sum W_i$$

For example, if scale factors with an Extra High rating are each assigned a weight of (0), then a 100 KSLOC project with Extra High ratings for all factors will have $\sum W_i = 0$, $B = 1.01$, and a relative effort $E = 100^{1.01} = 105$ *PM*. If scale factors with Very Low rating are each assigned a weight of (5), then a project with Very Low (5) ratings for all factors will have $\sum W_i = 25$, $B = 1.26$, and a relative effort $E = 331$ *PM*. This represents a large variation, but the increase involved in a one-unit change in one of the factors is only about 4.7%.

Table I-1: Scale Factors for COCOMO II Early Design and Post-Architecture Models

Scale Factors (W_i)	Very Low	Low	Nominal	High	Very High	Extra High
PREC	thoroughly unprecedented	largely unprecedented	somewhat unprecedented	generally familiar	largely familiar	throughly familiar
FLEX	rigorous	occasional relaxation	some relaxation	general conformity	some conformity	general goals
RESL[3]	little (20%)	some (40%)	often (60%)	generally (75%)	mostly (90%)	full (100%)
TEAM	very difficult interactions	some difficult interactions	basically cooperative interactions	largely cooperative	highly cooperative	seamless interactions
PMAT	Weighted average of "Yes" answers to CMM Maturity Questionnaire					

Precedentedness (PREC) and Development Flexibility (FLEX)

These two scale factors largely capture the differences between the Organic, Semidetached and Embedded modes of the original COCOMO model [Boehm 1981]. Table I-2 reorganizes [Boehm 1981, Table 6.3] to map its project features onto the Precedentedness and Development Flexibility scales. This table can be used as a more in depth explanation for the PREC and FLEX rating scales given in Table I-1.

Table I-2: Scale Factors Related to COCOMO Development Modes

Feature	Very Low	Nominal / High	Extra High
Precedentedness			
Organizational understanding of product objectives	General	Considerable	Thorough
Experience in working with related software systems	Moderate	Considerable	Extensive
Concurrent development of associated new hardware and operational procedures	Extensive	Moderate	Some
Need for innovative data processing architectures, algorithms	Considerable	Some	Minimal
Development Flexibility			
Need for software conformance with pre-established requirements	Full	Considerable	Basic
Need for software conformance with external interface specifications	Full	Considerable	Basic

3 % significant module interfaces specified,% significant risks eliminated.

Premium on early completion	High	Medium	Low

Architecture / Risk Resolution (RESL)

This factor combines two of the scale factors in Ada COCOMO, “Design Thoroughness by Product Design Review (PDR)” and “Risk Elimination by PDR” [Boehm and Royce 1989; Figures 4 and 5]. Table I-3 consolidates the Ada COCOMO ratings to form a more comprehensive definition for the COCOMO II RESL rating levels. The RESL rating is the subjective weighted average of the listed characteristics.

Table I-3: RESL Rating Components

Characteristic	Very Low	Low	Nominal	High	Very High	Extra High
Risk Management Plan identifies all critical risk items, establishes milestones for resolving them by PDR.	None	Little	Some	Generally	Mostly	Fully
Schedule, budget, and internal milestones through PDR compatible with Risk Management Plan	None	Little	Some	Generally	Mostly	Fully
Percent of development schedule devoted to establishing architecture, given general product objectives	5	10	17	25	33	40
Percent of required top software architects available to project	20	40	60	80	100	120
Tool support available for resolving risk items, developing and verifying architectural specs	None	Little	Some	Good	Strong	Full
Level of uncertainty in Key architecture drivers: mission, user interface, COTS, hardware, technology, performance.	Extreme	Significant	Consider-able	Some	Little	Very Little
Number and criticality of risk items	> 10 Critical	5-10 Critical	2-4 Critical	1 Critical	> 5Non-Critical	< 5 Non-Critical

Team Cohesion (TEAM)

The Team Cohesion scale factor accounts for the sources of project turbulence and entropy due to difficulties in synchronizing the project's stakeholders: users, customers, developers, maintainers, interfacers, others. These difficulties may arise from differences in stakeholder objectives and cultures; difficulties in reconciling objectives; and stakeholder's lack of experience and familiarity in operating as a team. Table I-4 provides a detailed definition for the overall TEAM rating levels. The final rating is the subjective weighted average of the listed characteristics.

Table I-4 : TEAM Rating Components

Characteristic	Very Low	Low	Nominal	High	Very High	Extra HIgh
Consistency of stakeholder objectives and cultures	Little	Some	Basic	Consider-able	Strong	Full
Ability, willingness of stakeholders to accommodate other stakeholders' objectives	Little	Some	Basic	Consider-able	Strong	Full
Experience of stakeholders in operating as a team	None	Little	Little	Basic	Consider-able	Extensive
Stakeholder teambuilding to achieve shared vision and commitments	None	Little	Little	Basic	Consider-able	Extensive

Process Maturity (PMAT)

The procedure for determining PMAT is organized around the Software Engineering Institute's Capability Maturity Model (CMM). The time period for rating Process Maturity is the time the project starts. There are two ways of rating Process Maturity. The first captures the result of an organized evaluation based on the CMM.

Overall Maturity Level

p CMM Level 1 (lower half)
p CMM Level 1 (upper half)
p CMM Level 2
p CMM Level 3
p CMM Level 4
p CMM Level 5

Key Process Areas

The second is organized around the 18 Key Process Areas (KPAs) in the SEI Capability Maturity Model [Paulk et al. 1993, 1993a]. The procedure for determining PMAT is to decide the percentage of compliance for each of the KPAs. If the project has undergone a recent CMM Assessment then the percentage compliance for the overall KPA (based on KPA Key Practice compliance assessment data) is used. If an assessment has not been done then the levels of compliance to the KPA's goals are used (with the Likert scale below) to set the level of compliance. The goal-based level of compliance is determined by a judgement-based averaging across the goals for each Key Process Area. If more information is needed on the KPA goals, they are listed in Appendix C of this document.

Table I-5

Key Process Areas	Almost Always (>90%)	Often (60-90%)	About Half (40-60%)	Occasion -ally (10-40%)	Rarely If Ever (<10%)	Does Not Apply	Don't Know
Requirements Management	☐	☐	☐	☐	☐	☐	☐
Software Project Planning	☐	☐	☐	☐	☐	☐	☐
Software Project Tracking and Oversight	☐	☐	☐	☐	☐	☐	☐
Software Subcontract Management	☐	☐	☐	☐	☐	☐	☐
Software Quality Assurance	☐	☐	☐	☐	☐	☐	☐
Software Configuration Management	☐	☐	☐	☐	☐	☐	☐
Organization Process Focus	☐	☐	☐	☐	☐	☐	☐
Organization Process Definition	☐	☐	☐	☐	☐	☐	☐
Training Program	☐	☐	☐	☐	☐	☐	☐
Integrated Software Management	☐	☐	☐	☐	☐	☐	☐
Software Product Engineering	☐	☐	☐	☐	☐	☐	☐
Intergroup Coordination	☐	☐	☐	☐	☐	☐	☐
Peer Reviews	☐	☐	☐	☐	☐	☐	☐
Quantitative Process Management	☐	☐	☐	☐	☐	☐	☐
Software Quality Management	☐	☐	☐	☐	☐	☐	☐
Defect Prevention	☐	☐	☐	☐	☐	☐	☐
Technology Change Management	☐	☐	☐	☐	☐	☐	☐
Process Change Management	☐	☐	☐	☐	☐	☐	☐

- Check Almost Always when the goals are consistently achieved and are well established in standard operating procedures (over 90% of the time).
- Check Frequently when the goals are achieved relatively often, but sometimes are omitted under difficult circumstances (about 60 to 90% of the time).
- Check About Half when the goals are achieved about half of the time (about 40 to 60% of the time).
- Check Occasionally when the goals are sometimes achieved, but less often (about 10 to 40% of the time).
- Check Rarely If Ever when the goals are rarely if ever achieved (less than 10% of the time).
- Check Does Not Apply when you have the required knowledge about your project or organization and the KPA, but you feel the KPA does not apply to your circumstances.
- Check Don't Know when you are uncertain about how to respond for the KPA.

After the level of KPA compliance is determined each compliance level is weighted and a PMAT factor is calculated, as in Equation I-3. Initially, all KPAs will be equally weighted.

(EQ I-3)

$$5 - \left[\sum_{i=1}^{18} \left(\frac{KPA\%_i}{100} \times \frac{5}{18} \right) \right]$$

Adjusting Nominal Effort

Cost drivers are used to capture characteristics of the software development that affect the effort to complete the project. Cost drivers that have a multiplicative effect on predicting effort are called Effort Multipliers (EM). Each EM has a rating level that expresses the impact of the multiplier on development effort, PM. These rating can range from Extra Low to Extra High. For the purposes of quantitative analysis, each rating level of each EM has a weight associated with it. The nominal or average weight for an EM is 1.0. If a rating level causes more software development effort, then its corresponding EM weight is above 1.0. Conversely, if the rating level reduces the effort then the corresponding EM weight is less than 1.0.

The selection of effort-multipliers is based on a strong rationale that they would independently explain a significant source of project effort or productivity variation.

Early Design Model

This Early Design model is used in the early stages of a software project when very little may be known about the size of the product to be developed, the nature of the target platform, the nature of the personnel to be involved in the project, or the detailed specifics of the process to be used. This model could be employed in either Application Generator, System Integration, or Infrastructure development sectors.

The Early Design model adjusts the nominal effort using 7 EMs, Equation I-4. Each multiplier has 7 possible weights. The cost drivers for this model are explained in the later.

(EQ I-4)

$$PM_{adjusted} = PM_{nominal} \times \left(\prod_{i=1}^{7} EM_i \right)$$

Post-Architecture Model

The Post-Architecture model is the most detailed estimation model and it is intended to be used when a software life-cycle architecture has been developed. This model is used in the development and maintenance of software products in the Application Generators, System Integration, or Infrastructure sectors.

The Post-Architecture model adjusts nominal effort using 17 effort multipliers. The larger number of multipliers takes advantage of the greater knowledge available later in the development stage. The Post-Architecture effort multipliers are explained later.

(EQ I-5)

$$PM_{adjusted} = PM_{nominal} \times \left(\prod_{i=1}^{17} EM_i \right)$$

Development Schedule Estimation

COCOMO II provides a simple schedule estimation capability similar to those in COCOMO and Ada COCOMO. The initial baseline schedule equation for all three COCOMO II stages is:

(EQ I-6)

$$TDEV = \left[3.67 \times (\overline{PM})^{(0.28 + 0.2 \times (B - 1.01))}\right] \times \frac{\text{SCED\%}}{100}$$

where *TDEV* is the calendar time in months from the determination of a product's 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, *B* is the sum of project scale factors (discussed in the next chapter) and SCED% is the compression / expansion percentage in the SCED effort multiplier in Table I-1.

As COCOMO II evolves, it 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.

Using COCOMO II

Determining Size

Lines of Code

In COCOMO II, the logical source statement has been chosen as the standard line of code. Defining a line of code is difficult due to conceptual differences involved in accounting for executable statements and data declarations in different languages. The goal is to measure the amount of intellectual work put into program development, but difficulties arise when trying to define consistent measures across different languages. To minimize these problems, the Software Engineering Institute (SEI) definition checklist for a logical source statement is used in defining the line of code measure. The Software Engineering Institute (SEI) has developed this checklist as part of a system of definition checklists, report forms and supplemental forms to support measurement definitions [Park 1992, Goethert et al. 1992].

Figure II-1 shows a portion of the definition checklist as it is being applied to support the development of the COCOMO II model. Each checkmark in the "Includes" column identifies a particular statement type or attribute included in the definition, and vice-versa for the excludes. Other sections in the definition clarify statement attributes for usage, delivery, functionality, replications and development status. There are also clarifications for language specific statements for ADA, C, C++, CMS-2, COBOL, FORTRAN, JOVIAL and Pascal.

Figure II-1: Definition Checklist for Source Statements Counts

Definition name: __Logical Source Statements___Date:_____

_____(basic definition)_____Originator:_COCOMO II____

Measurement unit:	Physical source lines						
	Logical source statements			4			
Statement type	Definition	4	Data Array			Includes	Excludes
When a line or statement contains more than one type, classify it as the type with the highest precedence.							
1 Executable	Order of precedence				1	4	
2 Nonexecutable							
3 Declarations					2	4	
4 Compiler directives					3	4	
5 Comments							
6 On their own lines					4		4
7 On lines with source code					5		4
8 Banners and non-blank spacers					6		4
9 Blank (empty) comments					7		4
10 Blank lines					8		4
11							
12							
How produced	Definition	4	Data array			Includes	Excludes
1 Programmed						4	
2 Generated with source code generators							4
3 Converted with automated translators						4	
4 Copied or reused without change						4	
5 Modified						4	
6 Removed							4
7							
8							
Origin	Definition	4	Data array			Includes	Excludes
1 New work: no prior existence						4	
2 Prior work: taken or adapted from							
3 A previous version, build, or release						4	
4 Commercial, off-the-shelf software (COTS), other than libraries							4
5 Government furnished software (GFS), other than reuse libraries							4
6 Another product							4
7 A vendor-supplied language support library (unmodified)							4
8 A vendor-supplied operating system or utility (unmodified)							4
9 A local or modified language support library or operating system							4
10 Other commercial library							4
11 A reuse library (software designed for reuse)						4	
12 Other software component or library						4	
13							
14							

Some changes were made to the line-of-code definition that depart from the default definition provided in [Park 1992]. These changes eliminate categories of software which are generally small sources of project effort. Not included in the definition are commercial-off-the-shelf software (COTS), government furnished software (GFS), other products, language support libraries and operating systems, or other commercial libraries. Code generated with source code generators is not included though measurements will be taken with and without generated code to support analysis.

The "COCOMO II line-of-code definition" is calculated directly by the Amadeus automated metrics collection tool [Amadeus 1994] [Selby et al. 1991], which is being used to ensure uniformly collected data in the COCOMO II data

collection and analysis project. We have developed a set of Amadeus measurement templates that support the COCOMO II data definitions for use by the organizations collecting data, in order to facilitate standard definitions and consistent data across participating sites.

To support further data analysis, Amadeus will automatically collect additional measures including total source lines, comments, executable statements, declarations, structure, component interfaces, nesting, and others. The tool will provide various size measures, including some of the object sizing metrics in [Chidamber and Kemerer 1994], and the COCOMO sizing formulation will adapt as further data is collected and analyzed.

Function Points

The function point cost estimation approach is based on the amount of functionality in a software project and a set of individual project factors [Behrens 1983] [Kunkler 1985] [IFPUG 1994]. Function points are useful estimators since they are based on information that is available early in the project life cycle. A brief summary of function points and their calculation in support of COCOMO II is as follows.

Function points measure a software project by quantifying the information processing functionality associated with major external data or control input, output, or file types. Five user function types should be identified as defined in Table II-1.

Table II-1: User Function Types

External Input (Inputs)	Count each unique user data or user control input type that (i) enters the external boundary of the software system being measured and (ii) adds or changes data in a logical internal file.
External Output (Outputs)	Count each unique user data or control output type that leaves the external boundary of the software system being measured.
Internal Logical File (Files)	Count each major logical group of user data or control information in the software system as a logical internal file type. Include each logical file (e.g., each logical group of data) that is generated, used, or maintained by the software system.
External Interface Files (Interfaces)	Files passed or shared between software systems should be counted as external interface file types within each system.
External Inquiry (Queries)	Count each unique input-output combination, where an input causes and generates an immediate output, as an external inquiry type.

Each instance of these function types is then classified by complexity level. The complexity levels determine a set of weights, which are applied to their corresponding function counts to determine the Unadjusted Function Points quantity. This is the Function Point sizing metric used by COCOMO II. The usual Function Point procedure involves assessing the degree of influence (DI) of fourteen application characteristics on the software project determined according to a rating scale of 0.0 to

0.05 for each characteristic. The 14 ratings are added together, and added to a base level of 0.65 to produce a general characteristics adjustment factor that ranges from 0.65 to 1.35.

Each of these fourteen characteristics, such as distributed functions, performance, and reusability, thus have a maximum of 5% contribution to estimated effort. This is inconsistent with COCOMO experience; thus COCOMO II uses Unadjusted Function Points for sizing, and applies its reuse factors, cost driver effort multipliers, and exponent scale factors to this sizing quantity.

Counting Procedure for Unadjusted Function Points

The COCOMO II procedure for determining Unadjusted Function Points is described here. This procedure is used in both the Early Design and the Post-Architecture models.

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 File4 (ILF), External Interface File (EIF), External Input (EI), External Output (EO), and External Inquiry (EQ)).
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:

Table II-2

For ILF and EIF				For EO and EQ				For EI			
Record Element s	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

1. 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.

Table II-3

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

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

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

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. COCOMO II does this for both the Early Design and Post-Architecture models by using tables such as those found in [Jones 1991] to translate Unadjusted Function Points into equivalent SLOC.

Table II-4 : Converting Function Points to Lines of Code

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

Breakage

COCOMO II uses a breakage percentage, BRAK, to adjust the effective size of the product. Breakage reflects the requirements volatility in a project. It is the percentage of code thrown away due to requirements volatility. For example, a project which delivers 100,000 instructions but discards the equivalent of an additional 20,000 instructions has a BRAK value of 20. This would be used to adjust the project's effective size to 120,000 instructions for a COCOMO II estimation. The BRAK factor is not used in the *Applications Composition* model, where a certain degree of product iteration is expected, and included in the data calibration.

Adjusting for Reuse

COCOMO adjusts for the reuse by modifying the size of the module or project. The model treats reuse with function points and source lines of code the same in either the Early Design model or the Post-Architecture model.

Nonlinear Reuse Effects

Analysis in [Selby 1988] of reuse costs across nearly 3000 reused modules in the NASA Software Engineering Laboratory indicates that the reuse cost function is nonlinear in two significant ways (see Figure II-2):

- It does not go through the origin. There is generally a cost of about 5% for assessing, selecting, and assimilating the reusable component.
- Small modifications generate disproportionately large costs. This is primarily due to two factors: the cost of understanding the software to be modified, and the relative cost of interface checking.

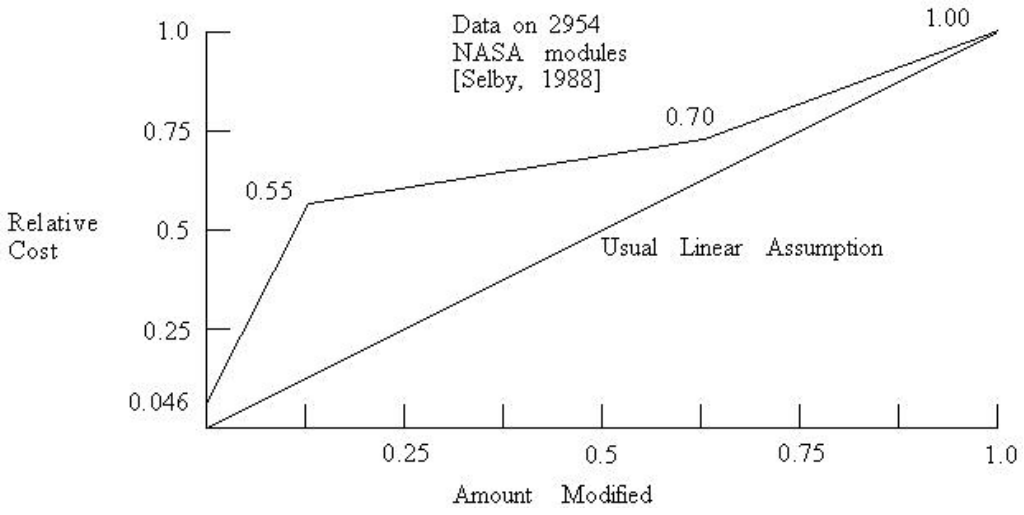


Figure II-2 . Nonlinear Reuse Effects

[Parikh and Zvegintzov 1983] contains data indicating that 47% of the effort in software maintenance involves understanding the software to be modified. Thus, as soon as one goes from unmodified (black-box) reuse to modified-software (white-box)

reuse, one encounters this software understanding penalty. Also, [Gerlich and Denskat 1994] shows that, if one modifies *k* out of *m* software modules, the number *N* of module interface checks required is $N = k * (m-k) + k * (k-1)/2$.
Figure II-3 shows this relation between the number of modules modified *k* and the resulting number of module interface checks required.

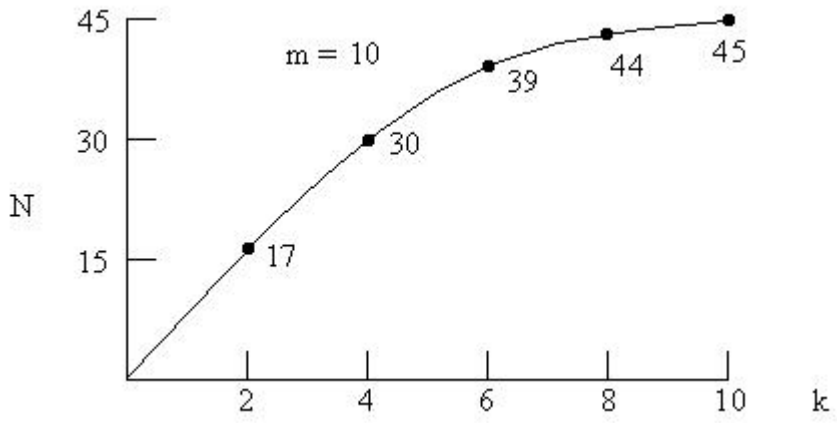


Figure II-3 . Number of Module Interface Checks vs. Fraction Modified

The shape of this curve is similar for other values of *m*. It indicates that there are nonlinear effects involved in the module interface checking which occurs during the design, code, integration, and test of modified software.

The size of both the software understanding penalty and the module interface checking penalty can be reduced by good software stucturing. Modular, hierarchical structuring can reduce the number of interfaces which need checking [Gerlich and Denskat 1994], and software which is well structured, explained, and related to its mission will be easier to understand. COCOMO II reflects this in its allocation of estimated effort for modifying reusable software.

A Reuse Model

The COCOMO II treatment of software reuse uses a nonlinear estimation model, Equation II-1. This involves estimating the amount of software to be adapted, ASLOC, and three degree- of-modification parameters: the percentage of design modified (DM), the percentage of code modified (CM), and the percentage of modification to the original integration effort required for integrating the reused software (IM).
The *Software Understanding* increment (SU) is obtained from Table II-5. SU is expressed quantitatively as a percentage. If the software is rated very high on structure, applications clarity, and self-descriptiveness, the software understanding and interface checking penalty is 10%. If the software is rated very low on these factors, the penalty is 50%. SU is determined by taking the subjective average of the three categories.

Table II-5: Rating Scale for Software Understanding Increment SU

	Very Low	Low	Nom	High	Very High
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately low cohesion, high coupling.	Reasonably well-structured; some weak areas.	High cohesion, low coupling.	Strong modularity, information hiding in data / control structures.
Application Clarity	No match between program and application world views.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application world-views.
Self- Descriptiveness	Obscure code; docu-	Some code com-	Moderate level of	Good code com-	Self-descriptive

	mentation missing, obscure or obsolete	mentary and headers; some useful documentation.	code commentary, headers, documentations.	mentary and headers; useful documentation; some weak areas.	code; documentation up-to-date, well-organized, with design rationale.
SU Increment to ESLOC	50	40	30	20	10

I

The other nonlinear reuse increment deals with the degree of *Assessment and Assimilation* (AA) needed to determine whether a fully-reused software module is appropriate to the application, and to integrate its description into the overall product description. Table II-6 provides the rating scale and values for the assessment and assimilation increment. AA is a percentage.

Table II-6 : Rating Scale for Assessment and Assimilation Increment (AA)

AA Increment	Level of AA Effort
0	None
2	Basic module search and documentation
4	Some module Test and Evaluation (T&E), documentation
6	Considerable module T&E, documentation
8	Extensive module T&E, documentation

The amount of effort required to modify existing software is a function not only of the amount of modification (AAF) and understandability of the existing software (SU), but also of the programmer's relative unfamiliarity with the software (UNFM). The UNFM parameter is applied multiplicatively to the software understanding effort increment. If the programmer works with the software every day, the 0.0 multiplier for UNFM will add no software understanding increment. If the programmer has never seen the software before, the 1.0 multiplier will add the full software understanding effort increment. The rating of UNFM is in Table II-7.

Table II-7: Rating Scale for Programmer Unfamiliarity (UNFM)

UNFM Increment	Level of Unfamiliarity
0.0	Completely familiar
0.2	Mostly familiar
0.4	Somewhat familiar
0.6	Considerably familiar
0.8	Mostly unfamiliar
1.0	Completely unfamiliar

(EQ II-1)

$$AAF = 0.4(DM) + 0.3(CM) + 0.3(IM)$$

$$ESLOC = \frac{ASLOC[AA + AAF(1 + 0.02(SU)(UNFM))]}{100}, \quad AAF \leq 0.5$$

$$ESLOC = \frac{ASLOC[AA + AAF + (SU)(UNFM)]}{100}, \quad AAF > 0.5$$

Equation II-1 is used to determine an equivalent number of new instructions, equivalent source lines of code (ESLOC). ESLOC is divided by one thousand to derive KESLOC which is used as the COCOMO size parameter. The calculation of

ESLOC is based on an intermediate quantity, the Adaptation Adjustment Factor (AAF). The adaptation quantities, DM, CM, IM are used to calculate AAF where :

- DM: Percent Design Modified. The percentage of the adapted software's design which is modified in order to adapt it to the new objectives and environment. (This is necessarily a subjective quantity.)
- CM: Percent Code Modified. The percentage of the adapted software's code which is modified in order to adapt it to the new objectives and environment.
- IM: Percent of Integration Required for Modified Software. The percentage of effort required to integrate the adapted software into an overall product and to test the resulting product as compared to the normal amount of integration and test effort for software of comparable size.

If there is no DM or CM (the component is being used unmodified) then there is no need for SU. If the code is being modified then SU applies.

Adjusting for Re-engineering or Conversion

The COCOMO II reuse model needs additional refinement to estimate the costs of software re-engineering and conversion. The major difference in re-engineering and conversion is the efficiency of automated tools for software restructuring. These can lead to very high values for the percentage of code modified (CM in the COCOMO II reuse model), but with very little corresponding effort. For example, in the NIST re-engineering case study [Ruhl and Gunn 1991], 80% of the code (13,131 COBOL source statements) was re-engineered by automatic translation, and the actual re-engineering effort, 35 person months, was a factor of over 4 lower than the COCOMO estimate of 152 person months.

The COCOMO II re-engineering and conversion estimation approach involves estimation of an additional parameter, AT, the percentage of the code that is re-engineered by automatic translation. Based on an analysis of the project data above, the productivity for automated translation is 2400 source statements / person month. This value could vary with different technologies and will be designated in the COCOMO II model as ATPROD. In the NIST case study *ATPROD = 2400*. Equation II-2 shows how automated translation affects the estimated nominal effort, *PM*.

(EQ II-2)

$$PM_{nominal} = A \times (Size)^{B} + \left\lceil \frac{\mathrm{ASLOC}\left(\frac{\mathrm{AT}}{100}\right)}{ATPROD} \right\rceil$$

The NIST case study also provides useful guidance on estimating the AT factor, which is a strong function of the difference between the boundary conditions (e.g., use of COTS packages, change from batch to interactive operation) of the old code and the re-engineered code. The NIST data on percentage of automated translation (from an original batch processing application without COTS utilities) are given in Table II-8 [Ruhl and Gunn 1991].

Table II-8: Variation in Percentage of Automated Re-engineering

Re-engineering Target	AT (% automated translation)
Batch processing	96%
Batch with SORT	90%
Batch with DBMS	88%
Batch, SORT, DBMS	82%
Interactive	50%

Applications Maintenance

COCOMO II uses the reuse model for maintenance when the amount of added or changed base source code is less than or equal to 20% or the new code being developed. Base code is source code that already exists and is being changed for use in the current project. For maintenance projects that involve more than 20% change in the existing base code (relative to new code being developed) COCOMO II uses maintenance size. An initial maintenance size is obtained in one to two ways, Equation II-3 or Equation II-5. Equation II-3 is used when the base code size is known and the percentage of change to the base code is known.

(EQ II-3)

$$(\text{Size})_M = [(\text{Base Code Size}) \cdot \text{MCF}] \cdot \text{MAF}$$

The percentage of change to the base code is called the Maintenance Change Factor (MCF). The MCF is similar to the Annual Change Traffic in COCOMO 81, except that maintenance periods other than a year can be used. Conceptually the MCF represents the ratio in Equation II-4:

(EQ II-4)

$$\text{MCF} = \frac{\text{Size Added} + \text{Size Modified}}{\text{Base Code Size}}$$

Equation II-5 is used when the fraction of code added or modified to the existing base code during the maintenance period is known. Deleted code is not counted.

(EQ II-5)

$$(\text{Size})_M = (\text{Size Added} + \text{Size Modified}) \cdot \text{MAF}$$

The size can refer to thousands of source lines of code (KSLOC), Function Points, or Object Points. When using Function Points or Object Points, it is better to estimate MCF in terms of the fraction of the overall application being changed, rather than the fraction of inputs, outputs, screens, reports, etc. touched by the changes. Our experience indicates that counting the items touched can lead to significant over estimates, as relatively small changes can touch a relatively large number of items.

The initial maintenance size estimate (described above) is adjusted with a Maintenance Adjustment Factor (MAF), Equation II-6. COCOMO 81 used different multipliers for the effects of Required Reliability (RELY) and Modern Programming Practices (MODP) on maintenance versus development effort. COCOMO II instead used the Software Understanding (SU) and Programmer Unfamiliarity (UNFM) factors from its reuse model to model the effects of well or poorly structured/understandable software on maintenance effort.

(EQ II-6)

$$\text{MAF} = 1 + \left(\frac{\text{SU}}{100} \cdot \text{UNFM}\right)$$

The resulting maintenance effort estimation formula is the same as the COCOMO II Post- Architecture development model:

(EQII-7)

$$PM_M = A \cdot (Size_M)^B \cdot \prod_{i=1}^{17} EM_i$$

The COCOMO II approach to estimating either the maintenance activity duration, T_M, or the average maintenance staffing level, FSP_M, is via the relationship:

(EQ II-8)

$$PM_M = T_M \cdot FSP_M$$

Most maintenance is done as a level of effort activity. This relationship can estimate the level of effort, FSP_M, given T_M (as in annual maintenance estimates, where T_M = 12 months), or vice-versa (given a fixed maintenance staff level, FSP_M, determine the necessary time, T_M, to complete the effort).

Effort Multipliers

Early Design

The Early Design model uses KSLOC for size. Unadjusted function points are converted to the equivalent SLOC and then to KSLOC. The application of project scale factors is the same for Early Design and the Post-Architecture models. In the Early Design model a reduced set of cost drivers are used. The Early Design cost drivers are obtained by combining the Post-Architecture model cost drivers from Table II-9. Whenever an assessment of a cost driver is between the rating levels always round to the Nominal rating, e.g. if a cost driver rating is between Very Low and Low, then select Low.

Table II-9: Early Design and Post-Architecture Effort Multipliers

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

Overall Approach: Personnel Capability (PERS) Example

The following approach is used for mapping the full set of Post-Architecture cost drivers and rating scales onto their Early Design model counterparts. It involves the use and combination of numerical equivalents of the rating levels. Specifically, a Very Low Post-Architecture cost driver rating corresponds to a numerical rating of 1, Low is 2, Nominal is 3, High is 4, Very High is 5, and Extra High is 6. For the combined Early Design cost drivers, the numerical values of the contributing Post-Architecture cost drivers, Table II-9, are summed, and the resulting totals are allocated to an expanded

Early Design model rating scale going from Extra Low to Extra High. The Early Design model rating scales always have a Nominal total equal to the sum of the Nominal ratings of its contributing Post-Architecture elements.

An example will illustrate this approach. The Early Design PERS cost driver combines the Post-Architecture cost drivers analyst capability (ACAP), programmer capability (PCAP), and personnel continuity (PCON). Each of these has a rating scale from Very Low (=1) to Very High (=5). Adding up their numerical ratings produces values ranging from 3 to 15. These are laid out on a scale, and the Early Design PERS rating levels assigned to them, as shown in Table II-16.

Table II-10: PERS Rating Levels

	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
Sum of ACAP, PCAP, PCON Ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Combined ACAP and PCAP Percentile	20%	39%	45%	55%	65%	75%	85%
Annual Personnel Turnover	45%	30%	20%	12%	9%	5%	4%

The Nominal PERS rating of 9 corresponds to the sum (3 + 3 + 3) of the Nominal ratings for ACAP, PCAP, and PCON, and its corresponding effort multiplier is 1.0. Note, however that the Nominal PERS rating of 9 can result from a number of other combinations, e.g. 1 + 3 + 5 = 9 for ACAP = Very Low, PCAP = Nominal, and PCON = Very High.

The rating scales and effort multipliers for PCAP and the other Early Design cost drivers maintain consistent relationships with their Post-Architecture counterparts. For example, the PERS Extra Low rating levels (20% combined ACAP and PCAP percentile; 45% personnel turnover) represent averages of the ACAP, PCAP, and PCON rating levels adding up to 3 or 4.

Maintaining these consistency relationships between the Early Design and Post-Architecture rating levels ensures consistency of Early Design and Post-Architecture cost estimates. It also enables the rating scales for the individual Post-Architecture cost drivers, Table II-16, to be used as detailed backups for the top-level Early Design rating scales given below.

Product Reliability and Complexity (RCPX)

This Early Design cost driver combines the four Post-Architecture cost drivers Required Software Reliability (RELY), Database size (DATA), Product complexity (CPLX), and Documentation match to life-cycle needs (DOCU). Unlike the PERS components, the RCPX components have rating scales with differing width. RELY and DOCU range from Very Low to Very High; DATA ranges from Low to Very High, and CPLX ranges from Very Low to Extra High. The numerical sum of their ratings thus ranges from 5 (VL, L, VL, VL) to 21 (VH, VH, EH, VH).

Table II-16 assigns RCPX ratings across this range, and associates appropriate rating scales to each of the RCPX ratings from Extra Low to Extra High. As with PERS, the Post- Architecture RELY, DATA CPLX, and DOCU rating scales in Table II-16 provide detailed backup for interpreting the Early Design RCPX rating levels.

Table II-11: RCPX Rating Levels

	Extra	Very	Low	Nomina l	High	Very	Extra

	Low	Low				High	High
Sum of RELY, DATA, CPLX, DOCU Ratings	5, 6	7, 8	9 - 11	12	13 - 15	16 - 18	19 - 21
Emphasis on reliability, documentation	Very little	Little	Some	Basic	Strong	Very Strong	Extreme
Product complexity	Very simple	Simple	Some	Moderate	Complex	Very complex	Extremely complex
Database size	Small	Small	Small	Moderate	Large	Very Large	Very Large

Required Reuse (RUSE)

This Early Design model cost driver is the same as its Post-Architecture counterpart, which is covered in the chapter on the Post-Architecture model. A summary of its rating levels is given below and in Table II-16.

Table II-12: RUSE Rating Level Summary

	Very Low	Low	Nominal	High	Very High	Extra High
RUSE		None	across project	across program	across product line	across multiple product lines

Platform Difficulty (PDIF)

This Early Design cost driver combines the three Post- Architecture cost drivers execution time (TIME), main storage constraint (STOR), and platform volatility (PVOL). TIME and STOR range from Nominal to Extra High; PVOL ranges from Low to Very High. The numerical sum of their ratings thus ranges from 8 (N, N, L) to 17 (EH, EH, VH).

Table II-16 assigns PDIF ratings across this range, and associates the appropriate rating scales to each of the PDIF rating levels. The Post-Architecture rating scales in Table II-16 provide additional backup definition for the PDIF ratings levels.

Table II-13: PDIF Rating Levels

	Low	Nominal	High	Very High	Extra High
Sum of TIME, STOR, and PVOL ratings	8	9	10 - 12	13 - 15	16, 17
Time and storage constraint	□ 50%	□ 50%	65%	80%	90%
Platform volatility	Very stable	Stable	Somewhat volatile	Volatile	Highly volatile

Personnel Experience (PREX)

This Early Design cost driver combines the three Post-Architecture cost drivers application experience (AEXP), platform experience (PEXP), and language and tool experience (LTEX). Each of these range from Very Low to Very High; as with PERS, the numerical sum of their ratings ranges from 3 to 15.

Table II-16 assigns PREX ratings across this range, and associates appropriate effort multipliers and rating scales to each of the rating levels.

Table II-14: PREX Rating Levels

	Extra Low	Very Low	Low	Nomina l	High	Very High	Extra High
Sum of AEXP, PEXP, and LTEX ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Applications, Platform, Language and Tool Experience	≤ 3 mo.	5 months	9 months	1 year	2 years	4 years	6 years

Facilities (FCIL)

This Early Design cost driver combines the two Post-Architecture cost drivers: use of software tools (TOOL) and multisite development (SITE). TOOL ranges from Very Low to Very High; SITE ranges from Very Low to Extra High. Thus, the numerical sum of their ratings ranges from 2 (VL, VL) to 11 (VH, EH).

Table II-16 assigns FCIL ratings across this range, and associates appropriate rating scales to each of the FCIL rating levels. The individual Post-Architecture TOOL and SITE rating scales in Table II-16 again provide additional backup definition for the FCIL rating levels.

FCIL Rating Levels

	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
Sum of TOOL and SITE ratings	2	3	4, 5	6	7, 8	9, 10	11
TOOL support	Minimal	Some	Simple CASE tool collection	Basic life-cycle tools	Good; moderately integrated	Strong; moderately integrated	Strong; well integrated
Multisite conditions	Weak support of complex multisite development	Some support of complex M/S devel.	Some support of moderately complex M/S devel.	Basic support of moderately complex M/S devel.	Strong support of moderately complex M/S devel.	Strong support of simple M/ S devel.	Very strong support of collocated or simple M/S devel.

Schedule (SCED)

The Early Design cost driver is the same as its Post-Architecture counterpart. A summary of its rating levels is given in Table II-16 below.

SCED Rating Level Summary

	Very Low	Low	Nominal	High	Very High	Extra High
SCED	75% of nominal	85%	100%	130%	160%	

Post-Architecture

These are the 17 effort multipliers used in COCOMO II Post-Architecture model to adjust the nominal effort, Person Months, to reflect the software product under development. They are grouped into four categories: product, platform, personnel, and project. Table II-16 lists the different cost drivers with their rating criterion (found at the end of this section). Whenever an assessment of a cost driver is between the rating levels always round to the Nominal rating, e.g. if a cost driver rating is between High and Very High, then select High. The counterpart 7 effort multipliers for the Early Design model are discussed in the chapter explaining that model

Product Factors

Required Software Reliability (RELY)

This is the measure of the extent to which the software must perform its intended function over a period of time. If the effect of a software failure is only slight inconvenience then RELY is low. If a failure would risk human life then RELY is very high.

	Very Low	Low	Nominal	High	Very High	Extra High
RELY	slight inconvenience	low, easily recoverable losses	moderate, easily recoverable losses	high financial loss	risk to human life	

Data Base Size (DATA)

This measure attempts to capture the affect large data requirements have on product development. The rating is determined by calculating D/P. The reason the size of the database is important to consider it because of the effort required to generate the test data that will be used to exercise the program.

(EQ II-9)

$$\frac{D}{P} = \frac{DataBaseSize(Bytes)}{ProgramSize(SLOC)}$$

DATA is rated as low if D/P is less than 10 and it is very high if it is greater than 1000.

	Very Low	Low	Nominal	High	Very High	Extra High
DATA		DB bytes/ Pgm SLOC < 10	10 ≤ D/P < 100	100 ≤ D/P < 1000	D/P ≥ 1000	

Product Complexity (CPLX)

Table II-15 (found at the end of this section) provides the new COCOMO II CPLX rating scale. Complexity is divided into five areas: control operations, computational operations, device-dependent operations, data management operations, and user interface management operations. Select the area or combination of areas that characterize the product or a sub-system of the product. The complexity rating is the subjective weighted average of these areas.

Table II-15: Module Complexity Ratings versus Type of Module

	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations	User Interface Managemen t Operations
Very Low	Straight-line code with a few non-nested structured programming operators: DOs, CASEs, IFTHENELSEs. Simple module composition via procedure calls or simple scripts.	Evaluation of simple expressions: e.g., *A=B+C*(D-E)*	Simple read, write statements with simple formats.	Simple arrays in main memory. Simple COTS-DB queries, updates.	Simple input forms, report generators.
Low	Straightforward nesting of structured programming operators. Mostly simple predicates	Evaluation of moderate-level expressions: e.g., *D=SQRT(B**2-4.*A*C)*	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level.	Single file subsetting with no data structure changes, no edits, no intermediate files. Moderately complex COTS-DB queries, updates.	Use of simple graphic user interface (GUI) builders.
Nominal	Mostly simple nesting. Some intermodule control. Decision tables. Simple callbacks or message passing, including middleware-supported distributed processing	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. Complex COTS-DB queries, updates.	Simple use of widget set.
High	Highly nested structured programming operators with many compound predicates. Queue and stack control. Homogeneous, distributed processing. Single processor soft	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns.	Operations at physical I/O level (physical storage address translations; seeks, reads, etc.). Optimized I/O overlap.	Simple triggers activated by data stream contents. Complex data restructuring.	Widget set development and extension. Simple voice I/O, multimedia.

	real-time control.				
Very High	Reentrant and recursive coding. Fixed-priority interrupt handling. Task synchronization, complex callbacks, heterogeneous distributed processing. Single-processor hard real-time control.	Difficult but structured numerical analysis: near-singular matrix equations, partial differential equations. Simple parallelization.	Routines for interrupt diagnosis, servicing, masking. Communication line handling. Performance-intensive embedded systems.	Distributed database coordination. Complex triggers. Search optimization.	Moderately complex 2D/ 3D, dynamic graphics, multimedia.
Extra High	Multiple resource scheduling with dynamically changing priorities. Microcode-level control. Distributed hard real-time control.	Difficult and unstructured numerical analysis: highly accurate analysis of noisy, stochastic data. Complex parallelization.	Device timing-dependent coding, micro-programmed operations. Performance-critical embedded systems.	Highly coupled, dynamic relational and object structures. Natural language data management.	Complex multimedia, virtual reality.

Required Reusability (RUSE)

This cost driver accounts for the additional effort needed to construct components intended for reuse on the current or future projects. This effort is consumed with creating more generic design of software, more elaborate documentation, and more extensive testing to ensure components are ready for use in other applications.

	Very Low	Low	Nominal	High	Very High	Extra High
RUSE		none	across project	across program	across product line	across multiple product lines

Documentation match to life-cycle needs (DOCU)

Several software cost models have a cost driver for the level of required documentation. In COCOMO II, the rating scale for the DOCU cost driver is evaluated in terms of the suitability of the project's documentation to its life-cycle needs. The rating scale goes from Very Low (many life-cycle needs uncovered) to Very High (very excessive for life-cycle needs).

	Very Low	Low	Nominal	High	Very High	Extra High
DOCU	Many life-cycle needs uncovered	Some life-cycle needs uncovered.	Right-sized to life-cycle needs	Excessive for life-cycle needs	Very excessive for life-cycle needs	

Platform Factors

The platform refers to the target-machine complex of hardware and infrastructure software (previously called the virtual machine). The factors have been revised to reflect this as described in this section. Some additional platform factors were considered, such as distribution, parallelism, embeddedness, and real-time operations. These considerations have been accommodated by the expansion of the Module Complexity ratings in Equation II-15.

Execution Time Constraint (TIME)

This is a measure of the execution time constraint imposed upon a software system. The rating is expressed in terms of the percentage of available execution time expected to be used by the system or subsystem consuming the execution time resource. The rating ranges from nominal, less than 50% of the execution time resource used, to extra high, 95% of the execution time resource is consumed.

	Very Low	Low	Nominal	High	Very High	Extra High
TIME			≤ 50% use of available execution time	70%	85%	95%

Main Storage Constraint (STOR)

This rating represents the degree of main storage constraint imposed on a software system or subsystem. Given the remarkable increase in available processor execution time and main storage, one can question whether these constraint variables are still relevant. However, many applications continue to expand to consume whatever resources are available, making these cost drivers still relevant. The rating ranges from nominal, less that 50%, to extra high, 95%.

	Very Low	Low	Nominal	High	Very High	Extra High
STOR			≤ 50% use of available storage	70%	85%	95%

Platform Volatility (PVOL)

"Platform" is used here to mean the complex of hardware and software (OS, DBMS, etc.) the software product calls on to perform its tasks. If the software to be developed is an operating system then the platform is the computer hardware. If a database management system is to be developed then the platform is the hardware and the operating system. If a network text browser is to be developed then the platform is the network, computer hardware, the operating system, and the distributed information repositories. The platform includes any compilers or assemblers supporting the development of the software system. This rating ranges from low, where there is a major change every 12 months, to very high, where there is a major change every two weeks.

	Very Low	Low	Nominal	High	Very High	Extra High
PVOL		major change every 12 mo.; minor change every 1 mo.	major: 6 mo.; minor: 2 wk.	major: 2 mo.; minor: 1 wk.	major: 2 wk.; minor: 2 days	

Personnel Factors

Analyst Capability (ACAP)

Analysts are personnel that work on requirements, high level design and detailed design. The major attributes that should be considered in this rating are Analysis and Design ability, efficiency and thoroughness, and the ability to

communicate and cooperate. The rating should not consider the level of experience of the analyst; that is rated with AEXP. Analysts that fall in the 15th percentile are rated very low and those that fall in the 95th percentile are rated as very high.

	Very Low	Low	Nominal	High	Very High	Extra High
ACAP	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile	

Programmer Capability (PCAP)

Current trends continue to emphasize the importance of highly capable analysts. However the increasing role of complex COTS packages, and the significant productivity leverage associated with programmers' ability to deal with these COTS packages, indicates a trend toward higher importance of programmer capability as well.

Evaluation should be based on the capability of the programmers as a team rather than as individuals. Major factors which should be considered in the rating are ability, efficiency and thoroughness, and the ability to communicate and cooperate. The experience of the programmer should not be considered here; it is rated with AEXP. A very low rated programmer team is in the 15th percentile and a very high rated programmer team is in the 95th percentile.

	Very Low	Low	Nominal	High	Very High	Extra High
PCAP	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile	

Applications Experience (AEXP)

This rating is dependent on the level of applications experience of the project team developing the software system or subsystem. The ratings are defined in terms of the project team's equivalent level of experience with this type of application. A very low rating is for application experience of less than 2 months. A very high rating is for experience of 6 years or more.

	Very Low	Low	Nominal	High	Very High	Extra High
AEXP	2 months	6 months	1 year	3 years	6 years	

Platform Experience (PEXP)

The Post-Architecture model broadens the productivity influence of PEXP, recognizing the importance of understanding the use of more powerful platforms, including more graphic user interface, database, networking, and distributed middleware capabilities.

	Very Low	Low	Nominal	High	Very High	Extra High
PEXP	2 months	6 months	1 year	3 years	6 year	

Language and Tool Experience (LTEX)

This is a measure of the level of programming language and software tool experience of the project team developing the software system or subsystem. Software development includes the use of tools that perform requirements and design representation and analysis, configuration management, document extraction, library management, program style and formatting, consistency checking, etc. In addition to experience in programming with a specific language the supporting tool set also effects development time. A low rating given for experience of less than 2 months. A very high rating is given for experience of 6 or more years.

	Very Low	Low	Nominal	High	Very High	Extra High
LTEX	2 months	6 months	1 year	3 years	6 year	

Personnel Continuity (PCON)

The rating scale for PCON is in terms of the project's annual personnel turnover: from 3%, very high, to 48%, very low.

	Very Low	Low	Nominal	High	Very High	Extra High
PCON	48% / year	24% / year	12% / year	6% / year	3% / year	

Project Factors

Use of Software Tools (TOOL)

Software tools have improved significantly since the 1970's projects used to calibrate COCOMO. The tool rating ranges from simple edit and code, very low, to integrated lifecycle management tools, very high.

	Very Low	Low	Nominal	High	Very High	Extra High
TOOL	edit, code, debug	simple, fron tend, back end CASE, little integra tion	basic lifecy cle tools, moderately integrated	strong, mature life cycle tools, moderately integrated	strong, mature, pro active lifecy cle tools, well inte grated with processes, methods, reuse	

Multisite Development (SITE)

Given the increasing frequency of multisite developments, and indications that multisite development effects are significant, the SITE cost driver has been added in COCOMO II. Determining its cost driver rating involves the assessment and averaging of two factors: site collocation (from fully collocated to international distribution) and communication support (from surface mail and some phone access to full interactive multimedia).

	Very Low	Low	Nominal	High	Very High	Extra High
SITE: Communica tions	Some phone, mail	Individual phone, FAX	Narrowband email	Wideband electronic communica tion.	Wideband elect. comm, occasional video conf.	Interactive multimedia

Required Development Schedule (SCED)

This rating measures the schedule constraint imposed on the project team developing the software. The ratings are defined in terms of the percentage of schedule stretch-out or acceleration with respect to a nominal schedule for a project requiring a given amount of effort. Accelerated schedules tend to produce more effort in the later phases of development because more issues are left to be determined due to lack of time to resolve them earlier. A schedule compress of 74% is rated very low. A stretch-out of a schedule produces more effort in the earlier phases of development where there is more time for thorough planning, specification and validation. A stretch-out of 160% is rated very high.

	Very Low	Low	Nominal	High	Very High	Extra High
SCED	75% of nom inal	85%	100%	130%	160%	

Table II-16: Post-Architecture Cost Driver Rating Level Summary

	Very Low	Low	Nominal	High	Very High	Extra High
RELY	slight inconve-nience	low, easily recoverable losses	Moderate, eas-ily recoverable losses	high financial loss	risk to human life	
DATA		DB bytes/Pgm SLOC < 10	10 ≤ D/P < 100	100 ≤ D/P < 1000	D/P ≥ 1000	
CPLX	see Table II-15					
RUSE		none	Across project	across program	across product line	across multi-ple product lines
DOCU	Many life-cycle needs uncovered	Some life-cycle needs uncovered.	Right-sized to life-cycle needs	Excessive for life-cycle needs	Very exces sive for life-cycle needs	
TIME			50% use of available exe cution time	70%	85%	95%
STOR			50% use of available stor age	70%	85%	95%
PVOL		major change every 12 mo.; minor change every 1 mo.	major: 6 mo.; minor: 2 wk.	major: 2 mo.; minor: 1 wk.	major: 2 wk.; minor: 2 days	
ACAP	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile	
PCAP	15th percentile	35th percentile	55th percentile	75th percentile	90th percentile	
PCON	48% / year	24% / year	12% / year	6% / year	3% / year	
AEXP	≤ 2 months	6 months	1 year	3 years	6 years	
PEXP	≤ 2 months	6 months	1 year	3 years	6 year	
LTEX	≤ 2 months	6 months	1 year	3 years	6 year	
TOOL	edit, code,	simple, fron	basic lifecycle	strong, mature	strong, mature,	

	debug	tend, backend CASE, little integration	tools, moder ately inte grated	lifecycle tools, moderately integrated	proactive life cycle tools, well inte grated with processes, methods, reuse	
SITE: Colloc ation	International	Multi-city and Multi-com pany	Multi-city or Multi-com pany	Same city or metro. area	Same building or complex	Fully collo cated
SITE: Comm unicati ons	Some phone, mail	Individual phone, FAX	Narrowband email	Wideband electronic communica tion.	Wideband elect. comm, occasional video conf.	Interactive multimedia
SCED	75% of nomi nal	85%	100%	130%	160%	