COCOMO II Model Definition Manual

Acknowledgments

This work has been supported both financially and technically by the COCOMO II Program Affiliates: Aerospace, Air Force Cost Analysis Agency, Allied Signal, AT&T, Bellcore, EDS, E-Systems, GDE Systems, Hughes, IDA, 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.

Graduate Assistants: Chris Abts, Brad Clark, Sunita Devnani-Chulani

The COCOMO II project is being led by Dr. Barry Boehm

Table of Contents

Chapter 1: Future Software Practices Marketplace

"We are becoming a software company," is an increasingly-repeated phrase in organizations as diverse as finance, transportation, aerospace, electronics, and manufacturing firms. Competitive advantage is increasingly dependent on the development of smart, tailorable products and services, and on the ability to develop and adapt these products and services more rapidly than competitors' adaptation times.

Dramatic reductions in computer hardware platform costs, and the prevalence of commodity software solutions have indirectly put downward pressure on systems development costs. This situation makes cost-benefit calculations even more important in selecting the correct components for construction and life cycle evolution of a system, and in convincing skeptical financial management of the business case for software investments. It also highlights the need for concurrent product and process determination, and for the ability to conduct trade-off analyses among software and system life cycle costs, cycle times, functions, performance, and qualities.

Concurrently, a new generation of software processes and products is changing the way organizations develop software. These new approaches-evolutionary, risk-driven, and collaborative software processes; fourth generation languages and application generators; commercial off-the-shelf (COTS) and reuse-driven software approaches; fast-track software development approaches; software process maturity initiatives-lead to significant benefits in terms of improved software quality and reduced software cost, risk, and cycle time.

However, although some of the existing software cost models have initiatives addressing aspects of these issues, these new approaches have not been strongly matched to date by complementary new models for estimating software costs and schedules. This makes it difficult for organizations to conduct effective planning, analysis, and control of projects using the new approaches.

These concerns have led to the formulation of a new version of the Constructive Cost Model (COCOMO) for software effort, cost, and schedule estimation. The original COCOMO [Boehm 1981] and its specialized Ada COCOMO successor [Boehm and Royce 1989] were reasonably well-matched to the classes of software project that they modeled: largely custom, build-to-specification software [Miyazaki and Mori 1985, Boehm 1985, Goudy 1987]. Although Ada COCOMO added a capability for estimating the costs and schedules for incremental software development, COCOMO encountered increasing difficulty in estimating the costs of business software [Kemerer 1987, Ruhl and Gunn 1991], of object-oriented software [Pfleeger 1991], of software created via spiral or evolutionary development models, or of software developed largely via commercial-off-the-shelf (COTS) applications-composition capabilities.

1.1 Objectives

The initial definition of COCOMO II and its rationale are described in this paper. The definition will be refined as additional data are collected and analyzed. The primary objectives of the COCOMO II 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.

These objectives support the primary needs expressed by software cost estimation users in a recent Software Engineering Institute survey [Park et al. 1994]. In priority order, these needs were for support of project planning and scheduling, project staffing, estimates-to-complete, project preparation, replanning and rescheduling, project tracking, contract negotiation, proposal evaluation, resource leveling, concept exploration, design evaluation, and bid/no-bid decisions. For each of these needs, COCOMO II will provide more up-to-date support than the original COCOMO and Ada COCOMO predecessors.

1.2 Future Marketplace Model

Figure1 summarizes the model of the future software practices marketplace that we are using to guide the development of COCOMO II. It includes a large upper "end-user programming" sector with roughly 55 million practitioners in the U.S. by the year 2005; a lower "infrastructure" sector with roughly 0.75 million practitioners; and three intermediate sectors, involving the development of applications generators and composition aids (0.6 million practitioners), the development of systems by applications composition (0.7 million), and system integration of large-scale and/or embedded software systems (0.7 million)[1]

.

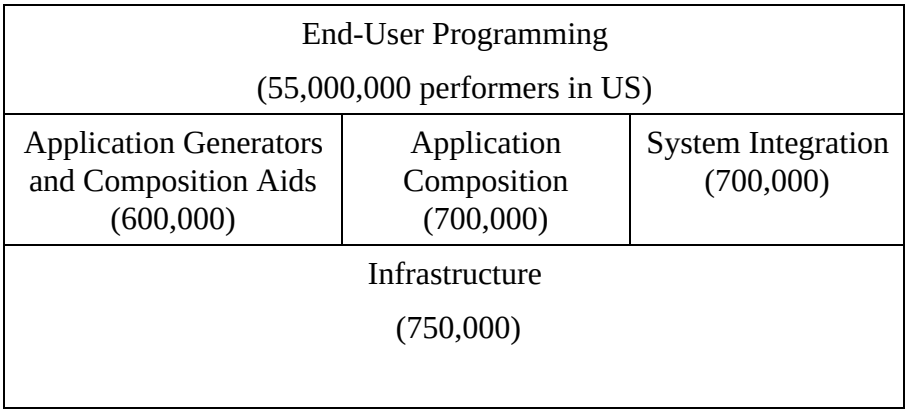

Figure 1: Future Software Practices Marketplace Model

End-User Programming will be driven by increasing computer literacy and competitive pressures for rapid, flexible, and user-driven information processing solutions. These trends will push the software marketplace toward having users develop most information processing applications themselves via application generators. Some example application generators are spreadsheets, extended query systems, and simple, specialized planning or inventory systems. They enable users to determine their desired information processing application via domain-familiar options, parameters, or simple rules. Every enterprise from Fortune 100 companies to small businesses and the U.S. Department of Defense will be involved in this sector.

Typical Infrastructure sector products will be in the areas of operating systems, database management systems, user interface management systems, and networking systems. Increasingly, the Infrastructure sector will address "middleware" solutions for such generic problems as distributed processing and transaction processing. Representative firms in the Infrastructure sector are Microsoft, NeXT, Oracle, SyBase, Novell, and the major computer vendors.

In contrast to end-user programmers, who will generally know a good deal about their applications domain and relatively little about computer science, the infrastructure developers will generally know a good deal about computer science and relatively little about applications. Their product lines will have many reusable components, but the pace of technology (new processor, memory, communications, display, and multimedia technology) will require them to build many components and capabilities from scratch.

Performers in the three intermediate sectors in Figure 1 will need to know a good deal about computer science-intensive Infrastructure software and also one or more applications domains. Creating this talent pool is a major national challenge.

[1] These figures are judgment-based extensions of the Bureau of Labor Statistics moderate-growth labor distribution scenario for the year 2005 [CSTB 1993; Silvestri and Lukaseiwicz 1991]. The 55 million End-User programming figure was obtained by applying judgment based extrapolations of the 1989 Bureau of the Census data on computer usage fractions by occupation [Kominski 1991] to generate end-user programming fractions by occupation category. These were then applied to the 2005 occupation-category populations (e.g., 10% of the 25M people in "Service Occupations"; 40% of the 17M people in "Marketing and Sales Occupations"). The 2005 total of 2.75 M software practitioners was obtained by applying a factor of 1.6 to the number of people traditionally identified as "Systems Analysts and Computer Scientists"

The Application Generators sector will create largely prepackaged capabilities for user programming. Typical firms operating in this sector are Microsoft, Lotus, Novell, Borland, and vendors of computer-aided planning, engineering, manufacturing, and financial analysis systems. Their product lines will have many reusable components, but also will require a good deal of new-capability development from scratch. Application Composition Aids will be developed both by the firms above and by software product-line investments of firms in the Application Composition sector.

The Application Composition sector deals with applications which are too diversified to be handled by prepackaged solutions, but which are sufficiently simple to be rapidly composable from interoperable components. Typical components will be graphic user interface (GUI) builders, database or object managers, middleware for distributed processing or transaction processing, hypermedia handlers, smart data finders, and domain-specific components such as financial, medical, or industrial process control packages.

Most large firms will have groups to compose such applications, but a great many specialized software firms will provide composed applications on contract. These range from large, versatile firms such as Andersen Consulting and EDS, to small firms specializing in such specialty areas as decision support or transaction processing, or in such applications domains as finance or manufacturing.

The Systems Integration sector deals with large scale, highly embedded, or unprecedented systems. Portions of these systems can be developed with Application Composition capabilities, but their demands generally require a significant amount of up-front systems engineering and custom software development. Aerospace firms operate within this sector, as do major system integration firms such as EDS and Andersen Consulting, large firms developing software-intensive products and services (telecommunications, automotive, financial, and electronic products firms), and firms developing large-scale corporate information systems or manufacturing support systems.

Chapter 2: COCOMO II Strategy and Rationale

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 above;
- 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.

2.1 COCOMO II Models for the Software Marketplace Sectors

The End-User Programming sector from Figure 1 does not need a COCOMO II model. Its applications are typically developed in hours to days, so a simple activity-based estimate will generally be sufficient.

The COCOMO II model for the Application Composition sector is based on Object Points. Object Points are a count of the screens, reports and third-generation-language modules developed in the application, each weighted by a three-level (simple, medium, difficult) complexity factor [Banker et al. 1994, Kauffman and Kumar 1993]. This is commensurate with the level of information generally known about an Application Composition product during its planning stages, and the corresponding level of accuracy needed for its software cost estimates (such applications are generally developed by a small team in a few weeks to months).

The COCOMO II capability for estimation of Application Generator, System Integration, or Infrastructure developments is based on a tailorable mix of the *Application Composition* model (for early prototyping efforts) and two increasingly detailed estimation models for subsequent portions of the life cycle, *Early Design* and *Post-Architecture*.

2.2 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 2, 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[2] are better known. The earlier "completed programs" size and effort data points in Figure 2 are the actual sizes and efforts of seven software products built to an imprecisely-defined specification [Boehm et al. 1984][3]. 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].

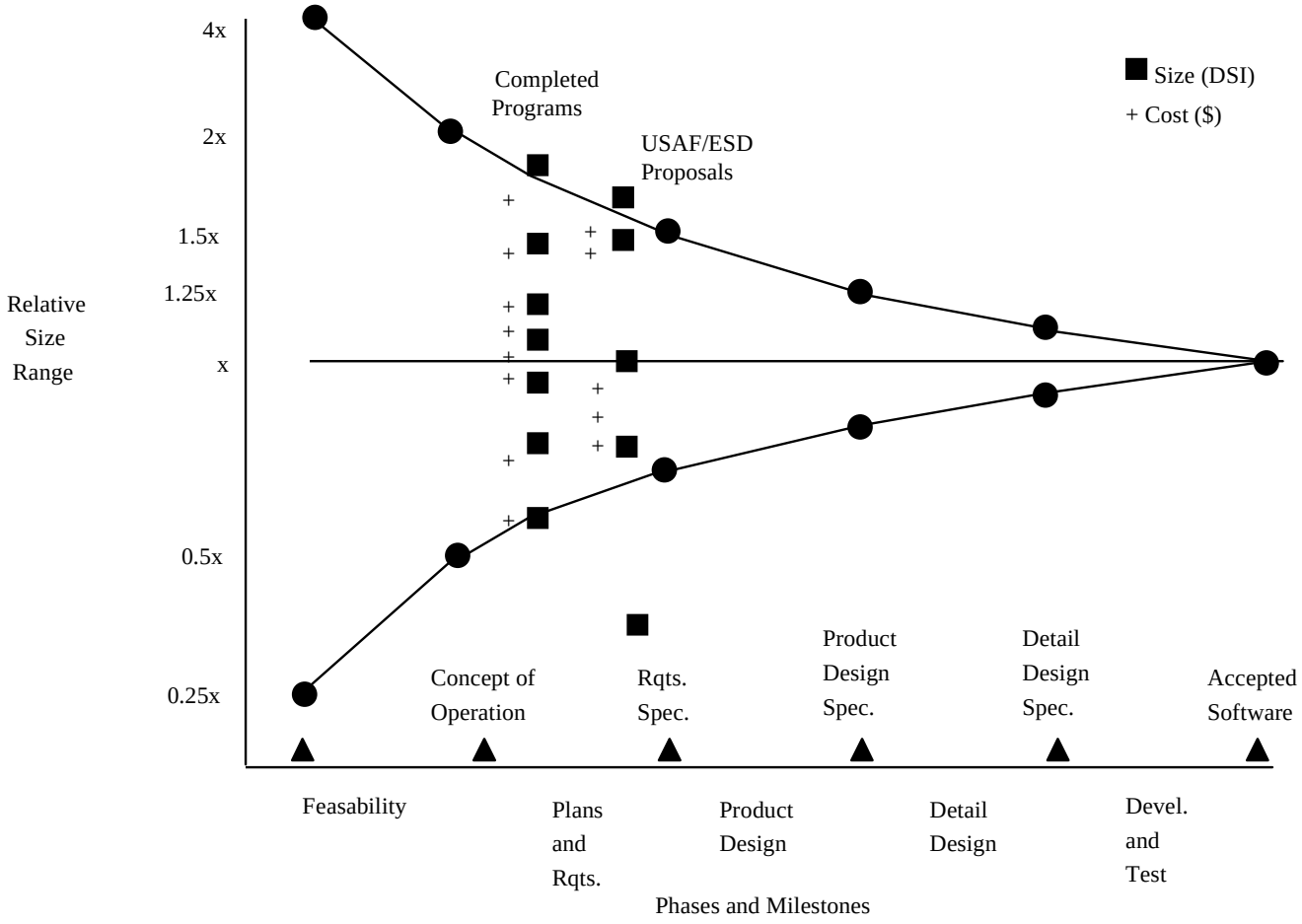

Figure 2: Software Costing and Sizing Accuracy vs. Phase

[2] 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.

[3] 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."

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 2 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 Application Composition model involves prototyping efforts to resolve potential high-risk issues such as user interfaces, software/system interaction, performance, or technology maturity. The costs of this type of effort are best estimated by the Applications Composition model.

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, 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.

2.3 Development Effort Estimates

In COCOMO II effort is expressed as Person Months (PM). All effort equations are presented in Appendix A. A 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.

2.3.1 Nominal Person Months

Equation 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]. This factor is discussed in the chapter on Software Economies and Diseconomies of Scale.

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 1.

$$PM_{NOMINAL} = A \times (Size)^{B} \qquad \textbf{EQ 1.}$$

2.3.2 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.

2.3.3 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 3):

- 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.

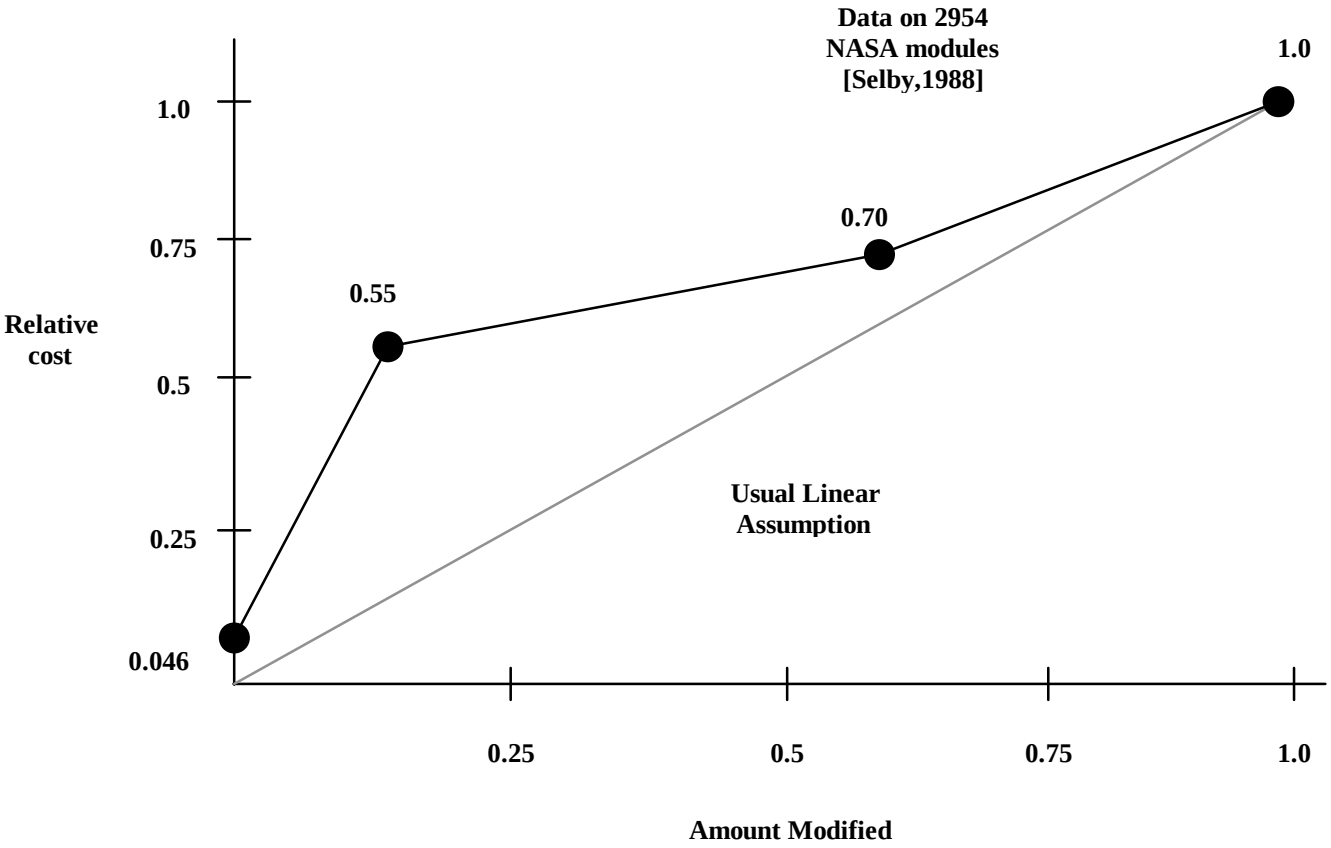

Figure 3: Nonlinear Reuse Effects

[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 module the number *N* of module interface checks required is $N = k * (m-k) + k * (k-1)/2$. Figure 4 shows this relation between the number of modules modified *k* and the resulting number of module interface checks required. 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 structuring. 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 2. This involves estimating the amount of software to be adapted, ASLOC, and three

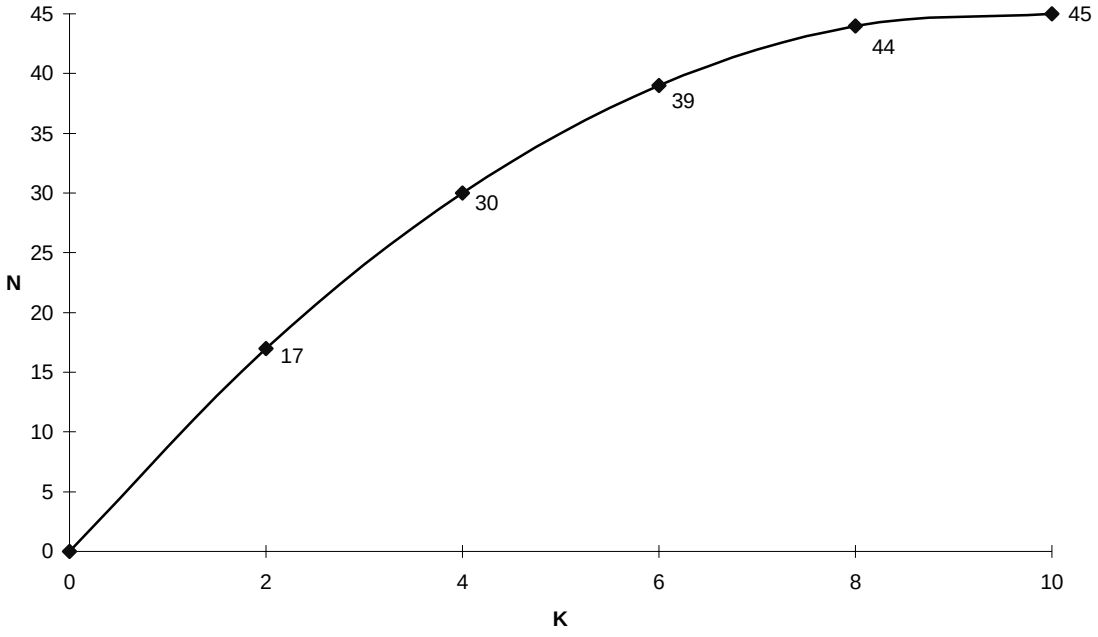

Figure 4: Number of Module Interface Checks vs. Fraction Modified

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 1. 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.

	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	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; documentation missing, obscure or obsolete	Some code commentary and headers; some useful documentation.	Moderate level of code commentary, headers, documentations.	Good code commentary and headers; useful documentation; some weak areas.	Self-descriptive code; documentation up-to-date, well-organized, with design rationale.
SU Increment to ESLOC	50	40	30	20	10

Table 1: Rating Scale for Software Understanding Increment SU

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 2 provides the rating scale and values for the assessment and assimilation increment. AA is a percentage.

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

Table 2: Rating Scale for Assessment and Assimilation Increment (AA)

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 3.

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

Table 3: Rating Scale for Programmer Unfamiliarity (UNFM)

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

$$ESLOC = \frac{ASLOC[AA + AAF(1 + 0.02(SU)(UNFM))]}{100}, AAF \le 0.5 \qquad \textbf{EQ 2.}$$

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

Equation 2 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.

2.3.4 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 3 shows how automated translation affects the estimated nominal effort, *PM*.

$$PM_{nominal} = A \times (Size)^{B} + \left[\frac{ASLOC\left(\frac{AT}{100}\right)}{ATPROD} \right] \qquad \textbf{EQ 3.}$$

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 4 [Ruhl and Gunn 1991].

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

Batch, SORT, DBMS	82%
Interactive	50%

Table 4: Variation in Percentage of Automated Re-engineering

2.3.5 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 4 or Equation 6. Equation 4 is used when the base code size is known and the percentage of change to the base code is known.

$$(Size)_M = [(BaseCodeSize) \times MCF] \times MAF \qquad \textbf{EQ 4.}$$

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 5:

$$MCF = \frac{SizeAdded + SizeModified}{BaseCodeSize} \qquad \textbf{EQ 5.}$$

Equation 6 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.

$$(Size)_M = (SizeAdded + SizeModified) \times MAF \qquad \textbf{EQ 6.}$$

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 7. 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.

$$MAF = 1 + \left(\frac{SU}{100} \times UNFM\right) \qquad \textbf{EQ 7.}$$

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

$$PM_M = A \times \left(Size_M\right)^B \times \prod_{i=1}^{17} EM_i \qquad \textbf{EQ 8.}$$

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:

$$PM_M = T_M \times FSP_M \qquad \textbf{EQ 9.}$$

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).

2.3.6 Adjusting Person Months

Cost drivers are used to capture characteristics of the software development that affect the effort to complete the project. Cost drivers have a rating level that expresses the impact of the driver 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 cost driver has a weight associated with it. The weight is called an effort multiplier (EM). The average EM assigned to a cost driver is 1.0 and the rating level associated with that weight is called Nominal. If a rating level causes more software development effort, then its corresponding EM is above 1.0. Conversely, if the rating level reduces the effort then the corresponding EM 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.

The EMs are used to *adjust* the nominal person month effort. There are 7 effort-multipliers for the Early Design model and 17 effort-multipliers for the Post-Architecture model. Each set is explained with their models in later chapters. The full equations are presented in Appendix A.

$$PM_{adjusted} = PM_{nominal} \times \left(\prod_{i} EM_i \right) \qquad \textbf{EQ 10.}$$

2.4 Development Schedule Estimates

The initial version of 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:

$$TDEV = \left[3.0 \times \overline{PM^{(0.33+0.2\times(B-1.01))}} \right] \times \frac{SCED\%}{100} \qquad \textbf{EQ 11.}$$

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 21.

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.

2.4.1 Output Ranges

A number of COCOMO users have expressed a preference for estimate ranges rather than point estimates as COCOMO outputs. The three-stage COCOMO II model enables the estimation of likely ranges of output estimates, using the costing and sizing accuracy relationships in Figure 2. Once the most likely effort estimate *E* is calculated from the chosen Application Composition, Early Design, or Post-Architecture model, a set of optimistic and pessimistic estimates, representing roughly one standard deviation around the most likely estimate, are calculated as follows:

Stage	Optimistic Estimate	Pessimistic Estimate
1	0.50 E	2.0 E
2	0.67 E	1.5 E
3	0.80 E	1.25 E

Table 5: Output Range Estimates

The effort range values can be used in the schedule equation, Equation 11, to determine schedule range values.

Chapter 3: Software Economies and Diseconomies of Scale

3.1 Approach

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 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.

3.1.1 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.

3.2 Scaling Drivers

Equation 12 defines the exponent, *B*, used in Equation 1. Table 21 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:

$$B = 1.01 + 0.01 \times \sum W_i \qquad \textit{EQ 12.}$$

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 $E = 1001.01 = 105$ *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%.

Scale Factors (*Wi*)	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[a]	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					

Table 6: Scale Factors for COCOMO II Early Design and Post-Architecture Models

[a] % significant module interfaces specified, % significant risks eliminated.

3.2.1 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 7 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 21.

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
Premium on early completion	High	Medium	Low

Table 7: Scale Factors Related to COCOMO Development Modes

3.2.2 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 8 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. (*Explain the Ada COCOMO ratings)*

3.2.3 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 9 provides a detailed definition for the overall TEAM rating levels. The final rating is the subjective weighted average of the listed characteristics.

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	Considerable	Some	Little	Very Little
Number and criticality of risk items	> 10 Critical	5-10 Critical	2-4 Critical	1 Critical	> 5 Non-Critical	< 5 Non-Critical

Table 8: RESL Rating Components

Table 9: TEAM Rating Components

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

3.2.4 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

❒ CMM Level 1 (lower half)

❒ CMM Level 1 (upper half)

❒ CMM Level 2

❒ CMM Level 3

❒ CMM Level 4

❒ 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 B of this document.

Key Process Areas	Almost Always (>90%)	Frequently (60-90%)	About Half (40-60%)	Occasionally (10-40%)	Rarely If Ever (<10%)	Does Not Apply	Don't Know
1 Requirements Management	❒	❒	❒	❒	❒	❒	❒
2 Software Project Planning	❒	❒	❒	❒	❒	❒	❒
3 Software Project Tracking and Oversight	❒	❒	❒	❒	❒	❒	❒
4 Software Subcontract Management	❒	❒	❒	❒	❒	❒	❒
5 Software Quality Assurance	❒	❒	❒	❒	❒	❒	❒
6 Software Configuration Management	❒	❒	❒	❒	❒	❒	❒
7 Organization Process Focus	❒	❒	❒	❒	❒	❒	❒
8 Organization Process Definition	❒	❒	❒	❒	❒	❒	❒
9 Training Program	❒	❒	❒	❒	❒	❒	❒

10 Integrated Software Management	❒	❒	❒	❒	❒	❒	❒
11 Software Product Engineering	❒	❒	❒	❒	❒	❒	❒
12 Intergroup Coordination	❒	❒	❒	❒	❒	❒	❒
13 Peer Reviews	❒	❒	❒	❒	❒	❒	❒
14 Quantitative Process Management	❒	❒	❒	❒	❒	❒	❒
15 Software Quality Management	❒	❒	❒	❒	❒	❒	❒
16 Defect Prevention	❒	❒	❒	❒	❒	❒	❒
17 Technology Change Management	❒	❒	❒	❒	❒	❒	❒
18 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 13. Initially, all KPAs will be equally weighted.

$$5 - \left[\sum_{i=1}^{18} \left(\frac{KPA\%_i}{100} \times \frac{5}{18} \right) \right] \qquad \textbf{EQ 13.}$$

Chapter 4: The Application Composition Model

This model address applications that are too diversified to be created quickly in a domain specific tool such as a spreadsheet yet are well enough known to be composed from interoperable components. Examples of these components-based systems are graphic user interface (GUI) builders, database or object managers, middleware for distributed processing or transaction processing, hypermedia handlers, smart data finders, and domain-specific components such as financial, medical, or industrial process control packages.

4.1 Approach

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.

The [Banker et al. 1991] comparative study of Object Point vs. Function Point estimation analyzed a sample of 19 investment banking software projects from a single organization, developed using ICASE applications composition capabilities, and ranging from 4.7 to 71.9 person-months of effort. The study found that the Object Points approach explained 73% of the variance (R2) in person-months adjusted for reuse, as compared to 76% for Function Points.

A subsequent statistically-designed experiment [Kaufman and Kumar 1993] involved four experienced project managers using Object Points and Function Points to estimate the effort required on two completed projects (3.5 and 6 actual person-months), based on project descriptions of the type available at the beginning of such projects. The experiment found that Object Points and Function Points produced comparably accurate results (slightly more accurate with Object Points, but not statistically significant). From a usage standpoint, the average time to produce an Object Point estimate was about 47% of the corresponding average time for Function Point estimates. Also, the managers considered the Object Point method easier to use (both of these results were statistically significant).

Thus, although these results are not yet broadly-based, their match to Applications Composition software development appears promising enough to justify selecting Object Points as the starting point for the COCOMO II Applications Composition estimation model.

4.2 Object Point Counting Procedure

The COCOMO II Object Point procedure for estimating the effort involved in Applications Composition and prototyping projects is a synthesis of the procedure in Appendix B.3 of [Kauffman and Kumar 1993] and the productivity data from the 19 project data points in [Banker et al. 1994].

Definitions of the terms 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.

The productivity rates are based on an analysis of the year-1 and year-2 project data in [Banker et al. 1991]. In year-1, the CASE tool was itself under construction and the developers were new to its use. The average productivity of NOP/person-month in the twelve year-1 projects is associated with the Low levels of developer and ICASE maturity and capability. In the seven year-2 projects, both the CASE tool and the developers' capabilities were considerably more mature. The average

productivity was 25 NOP/person-month, corresponding with the High levels of developer and ICASE maturity.

As another definitional point, note that the use of the term "object" in "Object Points" defines screens, reports, and 3GL modules as objects. This may or may not have any relationship to other definitions of "objects", such as those possessing features such as class affiliation, inheritance, encapsulation, message passing, and so forth. Counting rules for "objects" of that nature, when used in languages such as C++, will be discussed in the chapter on the Post Architecture model.

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.

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

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

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

5. Estimate percentage of reuse you expect to be achieved in this project. Compute the New Object Points to be developed, Equation 14..

$$NOP = \frac{(Object\ Points) \times (100 - \%Reuse)}{100} \qquad \textbf{EQ 14.}$$

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

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

7. Compute the estimated person-months:

$$PM = \frac{NOP}{PROD} \qquad \textbf{EQ 15.}$$

Chapter 5: The Early Design Model

This section covers the Early Design model using Unadjusted Function Points (UFP) as the sizing metric. This 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. For discussion of these marketplace sectors see Chapter 1.

5.1 Counting with 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 10.

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.

Table 10: User Function Types

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.

5.2 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 File[4] (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:

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

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	7	10	15
External Interfaces	5	7	10
External Inputs	3	4	6
External Outputs	4	5	7
External Inquiries	3	4	6

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

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

5.3 Converting Function Points to Lines of Code

To determine the nominal person months given in Equation 1 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.

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

Table 11: Converting Function Points to Lines of Code

5.4 Cost Drivers

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 and was described in Chapter 3l. 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 21. 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. The effort equation is the same as given in Equation 10. See Appendix A for comprehensive equation.

5.4.1 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 12,

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

Table 12: Early Design and Post-Architecture Effort Multipliers

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 21.

	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%

Table 13: PERS Rating Levels

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 21, to be used as detailed backups for the top-level Early Design rating scales given below.

5.4.2 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 21 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 21 provide detailed backup for interpreting the Early Design RCPX rating levels.

	Extra Low	Very Low	Low	Nominal	High	Very High	Extra 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

Table 14: RCPX Rating Levels

5.4.3 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 21.

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

Table 15: RUSE Rating Level Summary

5.4.4 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 21 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 21 provide additional backup definition for the PDIF ratings 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

Table 16: PDIF Rating Levels

5.4.5 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 21 assigns PREX ratings across this range, and associates appropriate effort multipliers and rating scales to each of the rating levels.

	Extra Low	Very Low	Low	Nominal	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

Table 17: PREX Rating Levels

5.4.6 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 21 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 21 again provide additional backup definition for the FCIL rating levels.

5.4.7 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 21 below.

	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	Basic life-cycle tools	Good; moderatel	Strong; moderatel	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 moderatel y complex M/S devel.	Strong support of moderatel y complex M/S devel.	Strong support of simple M/S devel.	Very strong support of collocated or simple M/S devel.

Table 18: FCIL Rating Levels

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

Table 19: SCED Rating Level Summary

Chapter 6: The Post-Architecture Model

This model is the most detailed 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, see Figure 1.

6.1 Lines of Code Counting Rules

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 5 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. The complete checklist is in Appendix B.

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.

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	✓			
Statement Type	Definition ✓	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 Æ		1	✓	
2. Nonexecutable			2	✓	
3. Declarations					
4. Compiler directives			3		✓
5. Comments			4		✓
6. On their own lines					
7. On lines with source code			5		✓
8. Banners and nonblank spacers			6		✓
9. Blank (empty) comments			7		✓
10. Blank lines			8		✓
How produced	Definition ✓	Data Array		Includes	Excludes
1. Programmed				✓	
2. Generated with source code generators					✓
3. Converted with automated translators				✓	
4. Copied or reused without change				✓	
5. Modified				✓	
6. Removed					✓
Origin	Definition ✓	Data Array		Includes	Excludes
1. New work: no prior existence				✓	
2. Prior work: taken or adapted from				✓	
3. A previous version, build, or release					✓
4. Commercial, off-the-shelf software (COTS), other than libraries					✓
5. Government furnished software (GFS), other than reuse libraries					✓
6. Another product					✓
7. A vendor-supplied language support library (unmodified)					✓
8. A vendor-supplied operating system or utility (unmodified)					✓
9. A local or modified language support library or operating system					✓
10. Other commercial library					✓
11. A reuse library (software designed for reuse)				✓	
12. Other software component or library				✓	

Figure 5: Definition Checklist

6.2 Function Points

For the Post-Architecture model function point estimation, the calculations proceed by converting Unadjusted Function Points to KSLOC as discussed in the chapter on the Early Design model. COCOMO II allows some components to be sized using function points, and others (which function points may not describe well, such as real-time or scientific computations) in SLOC. All size is express in KSLOC and this is used as shown in Eqation 10. Appendix A has the master equation for the Post-Architecture model.

6.3 Cost Drivers

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. Figure 21 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

6.3.1 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.

$$\frac{D}{P} = \frac{DataBaseSize(Bytes)}{\{\Pr ogramSize(SLOC)} \qquad \textit{EQ 16.}$$

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 20 (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.

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	

6.3.2 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 20.

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	

6.3.3 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	

6.3.4 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, frontend, backend CASE, little integration	basic lifecycle tools, moderately integrated	strong, mature lifecycle tools, moderately integrated	strong, mature, proactive lifecycle tools, well integrated 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: Communications	Some phone, mail	Individual phone, FAX	Narrowband email	Wideband electronic communication.	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 nominal	85%	100%	130%	160%	

	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations	User Interface Management 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 real-time control.	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.
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.

Table 20: Module Complexity Ratings versus Type of Module

	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		DB bytes/ Pgm SLOC < 10	10 ≤ D/P < 100	100 ≤ D/P < 1000	D/P ≥ 1000	
CPLX	see Table 20					
RUSE		none	across project	across program	across product line	across multiple 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 excessive for life-cycle needs	
TIME			≤ 50% use of available execution time	70%	85%	95%
STOR			≤ 50% use of available storage	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, debug	simple, frontend, backend CASE, little integration	basic lifecycle tools, moderately integrated	strong, mature lifecycle tools, moderately integrated	strong, mature, proactive lifecycle tools, well integrated with processes, methods, reuse	
SITE: Collocation	International	Multi-city and Multi-company	Multi-city or Multi-company	Same city or metro. area	Same building or complex	Fully collocated
SITE: Communications	Some phone, mail	Individual phone, FAX	Narrowband email	Wideband electronic communication.	Wideband elect. comm, occasional video conf.	Interactive multimedia
SCED	75% of nominal	85%	100%	130%	160%	

Table 21: Post-Architecture Cost Driver Rating Level Summary

Chapter 7: References

Amadeus (1994), *Amadeus Measurement System User's Guide*, Version 2.3a, Amadeus Software Research, Inc., Irvine, California, July 1994.

Banker, R., R. Kauffman and R. Kumar (1994), "An Empirical Test of Object-Based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment," *Journal of Management Information Systems* (to appear, 1994).

Banker, R., H. Chang and C. Kemerer (1994a), "Evidence on Economies of Scale in Software Development," Information and Software Technology (to appear, 1994).

Behrens, C. (1983), "Measuring the Productivity of Computer Systems Development Activities with Function Points," *IEEE Transactions on Software Engineering*, November 1983.

Boehm, B. (1981), *Software Engineering Economics*, Prentice Hall.

Boehm, B. (1983), "The Hardware/Software Cost Ratio: Is It a Myth?" *Computer* 16(3), March 1983, pp. 78-80.

Boehm, B. (1985), "COCOMO: Answering the Most Frequent Questions," In *Proceedings, First COCOMO Users' Group Meeting*, Wang Institute, Tyngsboro, MA, May 1985.

Boehm, B. (1989), *Software Risk Management*, IEEE Computer Society Press, Los Alamitos, CA.

Boehm, B., T. Gray, and T. Seewaldt (1984), "Prototyping vs. Specifying: A Multi-Project Experiment," *IEEE Transactions on Software Engineering*, May 1984, pp. 133-145.

Boehm, B., and W. Royce (1989), "Ada COCOMO and the Ada Process Model," *Proceedings, Fifth COCOMO Users' Group Meeting*, Software Engineering Institute, Pittsburgh, PA, November 1989.

Chidamber, S. and C. Kemerer (1994), "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, (to appear 1994).

Computer Science and Telecommunications Board (CSTB) National Research Council (1993), *Computing Professionals: Changing Needs for the 1990's*, National Academy Press, Washington DC, 1993.

Devenny, T. (1976). "An Exploratory Study of Software Cost Estimating at the Electronic Systems Division," Thesis No. GSM/SM/765-4, Air Force Institute of Technology, Dayton, OH.

Gerlich, R., and U. Denskat (1994), "A Cost Estimation Model for Maintenance and High Reuse," *Proceedings, ESCOM 1994*, Ivrea, Italy.

Goethert, W., E. Bailey, M. Busby (1992), "Software Effort and Schedule Measurement: A Framework for Counting Staff Hours and Reporting Schedule Information." CMU/SEI-92-TR-21, Software Engineering Institute, Pittsburgh, PA.

Goudy, R. (1987), "COCOMO-Based Personnel Requirements Model," *Proceedings, Third COCOMO Users' Group Meeting*, Software Engineering Institute, Pittsburgh, PA, November 1987.

IFPUG (1994), *IFPUG Function Point Counting Practices: Manual Release 4.0*, International Function Point Users' Group, Westerville, OH.

Jones, C. (1991), Applied Software Measurement, Assuring Productivity and Quality, McGraw-Hill, New York, N.Y.

Kauffman, R., and R. Kumar (1993), "Modeling Estimation Expertise in Object Based ICASE Environments," Stern School of Business Report, New York University, January 1993.

Kemerer, C. (1987), "An Empirical Validation of Software Cost Estimation Models," *Communications of the ACM*, May 1987, pp. 416-429.

Kominski, R. (1991), *Computer Use in the United States: 1989*, Current Population Reports, Series P-23, No. 171, U.S. Bureau of the Census, Washington, D.C., February 1991.

Kunkler, J. (1983), "A Cooperative Industry Study on Software Development/Maintenance Productivity," Xerox Corporation, Xerox Square --- XRX2 52A, Rochester, NY 14644, Third Report, March 1985.

Miyazaki, Y., and K. Mori (1985), "COCOMO Evaluation and Tailoring," *Proceedings, ICSE 8*, IEEE-ACM-BCS, London, August 1985, pp. 292-299.

Parikh, G., and N. Zvegintzov (1983). "The World of Software Maintenance," *Tutorial on Software Maintenance*, IEEE Computer Society Press, pp. 1-3.

Park R. (1992), "Software Size Measurement: A Framework for Counting Source Statements." CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, PA.

Park R, W. Goethert, J. Webb (1994), "Software Cost and Schedule Estimating: A Process Improvement Initiative", CMU/SEI-94-SR-03, Software Engineering Institute, Pittsburgh, PA.

Paulk, M., B. Curtis, M. Chrissis, and C. Weber (1993), "Capability Maturity Model for Software, Version 1.1", CMU-SEI-93-TR-24, Software Engineering Institute, Pittsburgh PA 15213, Feb. 1993.

Paulk, M., C. Weber, S. Garcia, M. Chrissis, and M. Bush (1993a), "Capability Maturity Model for Software, Version 1.1", CMU-SEI-93-TR-25, Software Engineering Institute, Pittsburgh PA 15213, Feb. 1993

Pfleeger, S. (1991), "Model of Software Effort and Productivity," *Information and Software Technology* 33 (3), April 1991, pp. 224-231.

Royce, W. (1990), "TRW's Ada Process Model for Incremental Development of Large Software Systems," *Proceedings, ICSE 12*, Nice, France, March 1990.

Ruhl, M., and M. Gunn (1991), "Software Reengineering: A Case Study and Lessons Learned," NIST Special Publication 500-193, Washington, DC, September 1991.

Selby, R. (1988), "Empirically Analyzing Software Reuse in a Production Environment," In Software Reuse: Emerging Technology, W. Tracz (Ed.), IEEE Computer Society Press, 1988., pp. 176-189.

Selby, R., A. Porter, D. Schmidt and J. Berney (1991), "Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development," *Proceedings of the Thirteenth International Conference on Software Engineering (ICSE 13)*, Austin, TX, May 13-16, 1991, pp. 288-298.

Silvestri, G. and J. Lukaseiwicz (1991), "Occupational Employment Projections," Monthly Labor Review 114(11), November 1991, pp. 64-94.

Chapter 8: Glossary and Index

(Index part to still be done)

3GL	Third Generation Language
AA	Percentage of reuse effort due to assessment and assimilation
AAF	Adaptation Adjustment Factor
AAM	Adaptation Adjustment Multiplier
ACAP	Analyst Capability
ACT	Annual Change Traffic
AEXP	Applications Experience
ASLOC	Adapted Source Lines of Code
AT	Automated Translation
BRAK	Breakage. The amount of controlled change allowed in a software development before requirements are "frozen."
CASE	Computer Aided Software Engineering
CM	Percentage of code modified during reuse
CMM	Capability Maturity Model
COCOMO	Constructive Cost Model
Cost Drivers	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.
COTS	Commercial Off The Shelf
CPLX	Product Complexity
CSTB	Computer Science and Telecommunications Board
DATA	Database Size
DBMS	Database Management System
DI	Degree of Influence
DM	Percentage of design modified during reuse
DOCU	Documentation to match lifecycle needs
EDS	Electronic Data Systems
ESLOC	Equivalent Source Lines of Code
FCIL	Facilities
FP	Function Points
GFS	Government Furnished Software

GUI	Graphical User Interface
ICASE	Integrated Computer Aided Software Environment
IM	Percentage of integration redone during reuse
KASLOC	Thousands of Adapted Source Lines of Code
KESLOC	Thousands of Equivalent Source Lines of Code
KSLOC	Thousands of Source Lines of Code
LEXP	Programming Language Experience
LTEX	Language and Tool Experience
MODP	Modern Programming Practices
NIST	National Institute of Standards and Technology
NOP	New Object Points
OS	Operating Systems
PCAP	Programmer Capability
PCON	Personnel continuity
PDIF	Platform Difficulty
PERS	Personnel Capability
PEXP	Platform Experience
PL	Product Line
PM	Person Months. A person month is the amount of time one person spends working on the software development project for one month.
PREX	Personnel Experience
PROD	Productivity rate
PVOL	Platform Volatility
RCPX	Product Reliability and Complexity
RELY	Required Software Reliability
RUSE	Required Reusability
RVOL	Requirements Volatility
SCED	Required Development Schedule
SECU	Classified Security Application
SEI	Software Engineering Institute
SITE	Multi-site operation
SLOC	Source Lines of Code
STOR	Main Storage Constraint
SU	Percentage of reuse effort due to software understanding
T&E	Test and Evaluation

TIME	Execution Time Constraint
TOOL	Use of Software Tools
TURN	Computer Turnaround Time
UNFM	Programmer Unfamiliarity
USAF/ESD	U.S. Air Force Electronic Systems Division
VEXP	Virtual Machine Experience
VIRT	Virtual Machine Volatility
VMVH	Virtual Machine Volatility: Host
VMVT	Virtual Machine Volatility: Target

Appendix A: Master Equations

These are the different models. They are presented here in a unified form to show the relationships between the different factors used in the equations. These models are for *whole project data* only. The Early Design and Post-Architecture models differ from the presented models when performing component-level estimation. It is assumed that multiplication and division have precedence over addition and subtraction in the equations. The main body of this manual explains when each model is used and defines the different factors.

9. Application Composition

New Object Points are determined by:

$$NOP = \frac{(ObjectPoints) - (100 - \%Reuse)}{100} \qquad \textbf{EQ 17.}$$

A productivity rate, PROD, is estimated from a subjective average of developer's experience and the ICASE maturity/capability:

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

Estimate effort with:

$$PM = \frac{NOP}{PROD} \qquad \textbf{EQ 18.}$$

10. Early Design

Estimate effort with:Estimate effort with:

$$PM = A \times \left[Size' \right]^{B} \times \prod_{i=1}^{7} EM_i + PM_M$$

where

$$PM_M = \frac{ASLOC\left(\frac{AT}{100}\right)}{ATPROD}$$

$$B = 1.01 + 0.01 \times \sum_{j=1}^{5} SF_j$$

$$Size' = Size \times \left(1 + \frac{BRAK}{100}\right) \qquad \textbf{EQ 19.}$$

$$Size = KNSLOC + KASLOC \times \left(\frac{100 - AJ}{100}\right) \times (AAM)$$

$$AAM = \begin{cases} \dfrac{AA + AAF \times (1 + 0/02(SU)(UNFM)}{100}, AAF \le 0.05 & or \\ \dfrac{AA + AAF + (SU) \times (UNFM)}{100}, AAF > 0.05 \end{cases}$$

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

Symbol	Description
A	Constant, provisionally set to 2.5
AA	Assessment and assimilation
ADAPT	Percentage of components adapted (represents the effort required in understanding software)
AT	Percentage of components that are automatically translated
ATPROD	Automatic translation productivity
BRAK	Breakage: Percentage of code thrown away due to requirements volatility
CM	Percentage of code modified
DM	Percentage of design modified
EM	Effort Multipliers: RCPX, RUSE, PDIF, PERS, PREX, FCIL, SCED
IM	Percentage of integration and test modified
KASLOC	Size of the adapted component expressed in thousands of adapted source lines of code
KNSLOC	Size of component expressed in thousands of new source lines of code
PM	Person Months of estimated effort
SF	Scale Factors: PREC, FLEX, RESL, TEAM, PMAT
SU	Software understanding (zero if DM = 0 and CM = 0)
UNFM	Programmer Unfamiliarity with Software

11. Post-Architecture

Estimate effort with:

$$PM = A \times \left[Size' \right]^{B} \times \prod_{i=1}^{17} EM_i + PM_M$$

where

$$PM_M = \frac{ASLOC\left(\frac{AT}{100}\right)}{ATPROD}$$

$$B = 1.01 + 0.01 \times \sum_{j=1}^{5} SF_j$$

$$Size' = Size \times \left(1 + \frac{BRAK}{100}\right) \qquad \textbf{EQ 20.}$$

$$Size = KNSLOC + KASLOC \times \left(\frac{100 - AJ}{100}\right) \times (AAM)$$

$$AAM = \begin{cases} \frac{AA + AAF \times (1 + 0/02(SU)(UNFM)}{100}, AAF \le 0.05 \\ \text{or} \\ \frac{AA + AAF + (SU) \times (UNFM)}{100}, AAF > 0.05 \end{cases}$$

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

Symbol	Description
A	Constant, provisionally set to 2.5
AA	Assessment and assimilation
ADAPT	Percentage of components adapted (represents the effort required in understanding software)
AT	Percentage of components that are automatically translated
ATPROD	Automatic translation productivity
BRAK	Breakage: Percentage of code thrown away due to requirements volatility
CM	Percentage of code modified
DM	Percentage of design modified
EM	Effort Multipliers: RELY, DATA, CPLX, RUSE, DOCU, TIME, STOR, PVOL, ACAP, PCAP,
IM	Percentage of integration and test modified
KASLOC	Size of the adapted component expressed in thousands of adapted source lines of code
KNSLOC	Size of component expressed in thousands of new source lines of code
PM	Person Months of estimated effort
SF	Scale Factors: PREC, FLEX, RESL, TEAM, PMAT
SU	Software understanding (zero if DM = 0 and CM = 0)
UNFM	Programmer Unfamiliarity with Software

12. Schedule Estimation

Determine time to develop (TDEV) with an estimated effort, PM, that excludes the effect of the SCED effort multiplier:

$$TDEV = \left[A \times \overline{PM^{(0.33+0.2\times(B-1.01))}} \right] \times \frac{SCED\%}{100}$$

where ***EQ 21.***

$$B = 1.01 + 0.01 \sum_{j=1}^{5} SF_j$$

Symbol	Description
A	Constant, Provisionally set to 3.0
SCED%	The compression / expansion percentage in the SCED effort multiplier
PM	Person Months of estimated effort from Early Design or Post-Architecture models (excluding the
SF	Scale Factors: PREC, FLEX, RESL, TEAM, PMAT
TDEV	Time to develop

Appendix B: Logical Lines of Source Code Counting Rules[5]

What is a line of source code? This checklist, adopted from the Software Engineering Institute, attempts to define a *logical* line of source code. The intent is to define a logical line of code while not becoming too language specific for use in collection data to validate the COCOMO II model.

13. Statement type	Includes	Excludes
When a line or statement contains more than one type, classify it as the type with the highest precedence. Order of precedence is in ascending order.		
1. Executable	✓	
2. Non-executable:		
3. Declarations	✓	
4. Compiler directives	✓	
5. Comments:		
6. On their own lines		✓
7. On lines with source code		✓
8. Banners and non-blank spacers		✓
9. Blank (empty) comments		✓
10. Blank lines		✓

14. How produced	Includes	Excludes
1. Programmed	✓	
2. Generated with source code generators		✓
3. Converted with automated translators	✓	
4. Copied or reused without change	✓	
5. Modified	✓	
6. Removed		✓

15. Origin	Includes	Excludes
1. New work: no prior existence	✓	
2. Prior work: taken or adapted from:		
3. A previous version, build, or release	✓	
4. Commercial, off-the-shelf software (COTS), other than libraries	✓	

[5] Park R. (1992), "Software Size Measurement: A Framework for Counting Source Statements." CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, PA.

5.	Government furnished software (GFS), other than reuse libraries	✓	
6.	Another product	✓	
7.	A vendor-supplied language support library (unmodified)	✓	
8.	A vendor-supplied operating system or utility (unmodified)	✓	
9.	A local or modified language support library or operating system	✓	
10.	Other commercial library	✓	
11.	A reuse library (software designed for reuse)	✓	
12.	Other software component or library	✓	

16. Usage

	Includes	Excludes
1. In or as part of the primary product	✓	
2. External to or in support of the primary product		✓

17. Delivery

	Includes	Excludes
1. Delivered:		
2. Delivered as source	✓	
3. Delivered in compiled or executable form, but not as source		✓
4. Not delivered:		
5. Under configuration control		✓
6. Not under configuration control		✓

18. Functionality

	Includes	Excludes
1. Operative	✓	
2. Inoperative (dead, bypassed, unused, unreferenced, or unaccessible):		
3. Functional (intentional dead code, reactivated for special purposes)	✓	
4. Nonfunctional (unintentionally present)		✓

19. Replications

	Includes	Excludes
1. Master source statements (originals)	✓	
2. Physical replicates of master statements, stored in the master code	✓	
3. Copies inserted, instantiated, or expanded when compiling or linking	✓	
4. Postproduction replicates-as in distributed, redundant, or reparameterized systems		✓

20. Development status

	Includes	Excludes
Each statement has one and only one status, usually that of its parent unit.		
1. Estimated or planned		✓
2. Designed		✓
3. Coded		✓
4. Unit tests completed		✓
5. Integrated into components		✓
6. Test readiness review completed	✓	
7. Software (CSCI) tests completed		✓
8. System tests completed	✓	

21. Language

	Includes	Excludes
List each source language on a separate line.		
1. Separate totals for each language	✓	

22. Clarifications (general)

	Includes	Excludes
1. Nulls, continues, and no-ops	✓	
2. Empty statements, e.g. ";;" and lone semicolons on separate lines		✓
3. Statements that instantiate generics	✓	
4. Begin...end and {...} pairs used as executable statements	✓	
5. Begin...end and {...} pairs that delimit (sub)program bodies	✓	
6. Logical expressions used as test conditions		✓
7. Expression evaluations used as subprograms arguments		✓
8. End symbols that terminate executable statements		✓
9. End symbols that terminate declarations or (sub)program bodies		✓
10. Then, else, and otherwise symbols		✓
11. Elseif statements	✓	
12. Keywords like procedure division, interface, and implementation	✓	
13. Labels (branching destinations) on lines by themselves		✓

23. Clarifications (language specific)

	Includes	Excludes
23.1 Ada		
1. End symbols that terminate declarations or (sub)program bodies		✓
2. Block statements, e.g. begin...end	✓	
3. With and use clauses	✓	
4. When (the keyword preceding executable statements)		✓
5. Exception (the keyword, used as a frame header)	✓	
6. Pragmas	✓	
23.2 Assembly		
1. Macro calls	✓	
2. Macro expansions		✓
23.3 C and C++		
1. Null statement, e.g. ";" by itself to indicate an empty body		✓
2. Expression statements (expressions terminated by semicolons)	✓	
3. Expression separated by semicolons, as in a "for" statement	✓	
4. Block statements, e.g. {...} with no terminating semicolon	✓	
5. ";", ";" or ";" on a line by itself when part of a declaration		✓
6. ";" or ";" on a line by itself when part of an executable statement		✓
7. Conditionally compiled statements (#if, #ifdef, #ifndef)	✓	
8. Preprocessor statements other than #if, #ifdef, and #ifndef	✓	
23.4 CMS-2		
1. Keywords like SYS-PROC and SYS-DD	✓	
23.5 COBOL		
1. "PROCEDURE DIVISION", "END DECLARATIVES", etc.	✓	
23.6 FORTRAN		
1. END statements	✓	
2. Format statements	✓	
3. Entry statements	✓	
23.7 JOVIAL		
1.		

23.8 PASCAL

1. Executable statements not terminated by semicolons	✓
2. Keywords like INTERFACE and IMPLEMENTATION	✓
3. FORWARD declarations	✓

24. Summary of Statement Types

24.1 Executable statements

Executable statements cause runtime actions. They may be simple statements such as assignments, goto's, procedure calls, macro calls, returns, breaks, exits, stops, continues, nulls, no-ops, empty statements, and FORTRAN's END. Or they may be structured or compound statements, such as conditional statements, repetitive statements, and "with" statements. Languages like Ada, C, C++, and Pascal have block statements [begin...end and {...}] that are classified as executable when used where other executable statements would be permitted. C and C++ define expressions as executable statements when they terminate with a semicolon, and C++ has a <declaration> statement that is executable.

24.2 Declarations

Declarations are nonexecutable program elements that affect an assembler's or compiler's interpretation of other program elements They are used to name, define, and initialize; to specify internal and external interfaces; to assign ranges for bounds checking; and to identify and bound modules and sections of code. Examples include declarations of names, numbers, constants, objects, types, subtypes, programs, subprograms, tasks, exceptions, packages, generics, macros, and deferred constants. Declarations also include renaming declarations, use clauses, and declarations that instantiate generics. Mandatory begin...end and {...} symbols that delimit bodies of programs and subprograms are integral parts of program and subprogram declarations. Language superstructure elements that establish boundaries for different sections of source code are also declarations. Examples include terms such as PROCEDURE DIVISION, DATA DIVISION, DECLARATIVES, END DECLARATIVES, INTERFACE, IMPLEMENTATION, SYS-PROC and SYS-DD. Declarations, in general, are never required by language specifications to initiate runtime actions, although some languages permit compilers to implement them that way.

24.3 Compiler Directives

Compiler directives instruct compilers, preprocessors, or translators (but not runtime systems) to perform special actions. Some, such as Ada's pragma and COBOL's COPY, REPLACE, and USE, are integral parts of the source language. In other languages like C and C++, special symbols like # are used along with standardized keywords to direct preprocessor or compiler actions. Still other languages rely on nonstandardized methods supplied by compiler vendors. In these languages, directives are often designated by special symbols such as #, $, and {$}.

This page left intentionally blank.

Appendix C: COCOMO II Process Maturity

For the authoritative source on Key Process Areas in the Capability Maturity Model see: Paulk, M.C., C.V. Weber, S.M. Garcia, M.B. Chrissis, and M. Bush, "Key Practices of the Capability Maturity Model, Version 1.1," CMU/SEI-93-TR-25, Software Engineering Institute, Pittsburgh, Pa., February 1993 ("Appendix C: Abridged Version of the Key Practices" is especially convenient in this reference). This document is available via FTP from ftp.sei.cmu.edu.

The source of this textual information was taken from: "Maturity Questionnaire", Zubrow, D.; Hayes, W.; Siegel, J.; Goldenson, D., CMU/SEI-94-SR-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, June 1994.

25. Requirements Management

The purpose of Requirements Management is to establish a common understanding between the customer and the software project of the customer's requirements that will be addressed by the software project. Requirements Management involves establishing and maintaining an agreement with the customer on the requirements for the software project. The agreement covers both the technical and nontechnical (e.g., delivery dates) requirements. The agreement forms the basis for estimating, planning, performing, and tracking the software project's activities throughout the software life cycle. Whenever the system requirements **allocated** to software are changed, the affected **software plans**, **work products**, and activities are adjusted to remain consistent with the updated requirements.

Definitions

allocated requirements (system requirements allocated to software) - The subset of the system requirements that are to be implemented in the software components of the system. The allocated requirements are a primary input to the software development plan. Software requirements analysis elaborates and refines the allocated requirements and results in software requirements that are documented.

software plans - The collection of plans, both formal and informal, used to express how software development and/or maintenance activities will be performed. Examples of plans that could be included: software development plan, software quality assurance plan, software configuration management plan, software test plan, risk management plan, and process improvement plan.

software work product - Any artifact created as part of defining, maintaining, or using a software process, including process descriptions, plans, procedures, computer programs, and associated documentation, which may or may not be intended for delivery to a customer or end user.

Goals

1. System requirements allocated to software are controlled to establish a baseline for software engineering and management use.
2. Software plans, products, and activities are kept consistent with the system requirements allocated to software.

26. Software Project Planning

The purpose of Software Project Planning is to establish reasonable **plans** for performing the software engineering activities and for managing the software project. Software Project Planning involves developing estimates for the work to be performed, establishing the necessary **commitments**, and defining the **plan** to perform the work.

Definitions

commitment - A pact that is freely assumed, visible, and expected to be kept by all parties.

software plans - The collection of plans, both formal and informal, used to express how software development and/or maintenance activities will be performed. Examples of plans that could be included: software development plan, software quality assurance plan, software configuration management plan, software test plan, risk management plan, and process improvement plan.

Goals

1. Software estimates are documented for use in planning and tracking the software project.
2. Software project activities and commitments are planned and documented.
3. Affected groups and individuals agree to their commitments related to the software project.

27. Software Project Tracking and Oversight

The purpose of Software Project Tracking and Oversight is to provide adequate visibility into actual progress so that management can take corrective actions when the software project's performance deviates significantly from the **software plans**. Corrective actions may include revising the software development plan to reflect the actual accomplishments and replanning the remaining work or taking actions to improve the performance. Software Project Tracking and Oversight involves tracking and reviewing the software accomplishments and results against documented estimates, **commitments**, and **plans**, and adjusting these plans based on the actual accomplishments and results.

Definitions

commitment - A pact that is freely assumed, visible, and expected to be kept by all parties.

software plans - The collection of plans, both formal and informal, used to express how software development and/or maintenance activities will be performed. Examples of plans that could be included: software development plan, software quality assurance plan, software configuration management plan, software test plan, risk management plan, and process improvement plan.

software work product - Any artifact created as part of defining, maintaining, or using a software process, including process descriptions, plans, procedures, computer programs, and associated documentation, which may or may not be intended for delivery to a customer or end user.

Goals

1. Actual results and performances are tracked against the software plans.
2. Corrective actions are taken and managed to closure when actual results and performance deviate significantly from the software plans.
3. Changes to software commitments are agreed to by the affected groups and individuals.

28. Software Subcontract Management

The purpose of Software Subcontract Management is to select qualified software subcontractors and manage them effectively. Software Subcontract Management involves selecting a software subcontractor, establishing commitments with the subcontractor, and tracking and reviewing the subcontractor's performance and results. These practices cover the management of a software (only) subcontract, as well as the management of the software component of a subcontract that includes software, hardware, and possibly other system components.

Definitions

documented procedure - A written description of a course of action to be taken to perform a given task [IEEE-STD-610 Glossary]

event-driven review/activity - A review or activity that is performed based on the occurrence of an event within the project (e.g., a formal review or the completion of a life-cycle stage).

periodic review/activity - A review/activity that occurs at a specified regular time interval, rather than at the completion of major events.

Goals

1. The prime contractor selects qualified software subcontractors.
2. The prime contractor and the software subcontractor agree to their commitments to each other.
3. The prime contractor and the software subcontractor maintain ongoing communications.
4. The prime contractor tracks the software subcontractor's actual results and performance against its commitments.

29. Software Quality Assurance

The purpose of Software Quality Assurance (SQA) is to provide management with appropriate visibility into the process being used by the software project and of the products being built. Software Quality Assurance involves reviewing and auditing the software products and activities to verify that they comply with the applicable procedures and standards and providing the software project and other appropriate managers with the results of these reviews and audits.

Definitions

audit - An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria. [IEEE-STD-610 Glossary]

periodic review/activity - A review/activity that occurs at a specified regular time interval, rather than at the completion of major events.

policy - A guiding principle, typically established by senior management, which is adopted by an organization or project to influence and determine decisions.

procedure - A written description of a course of action to be taken to perform a given task.[IEEE-STD-610 Glossary]

software quality assurance (SQA) - (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that a software work product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which software work products are developed and/or maintained.

standard - Mandatory requirements employed and enforced to prescribe a disciplined, uniform approach to software development.

Goals

1. Software quality assurance activities are planned.
2. Adherence of software products and activities to the applicable standards, procedures, and requirements is verified objectively.
3. Affected groups and individuals are informed of software quality assurance activities and results.
4. Noncompliance issues that cannot be resolved within the software project are addressed by senior management.

30. Software Configuration Management

The purpose of Software Configuration Management (SCM) is to establish and maintain the integrity of the products of the software project throughout the project's software life cycle. Software Configuration Management involves identifying the configuration of the software (i.e., selected software work products and their descriptions) at given points in time, systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the software life cycle. The work products placed under software configuration management include the software products that are delivered to the customer and the items that are identified with or required to create these software products.

Definitions

configuration item - An aggregation of hardware, software, or both, that is designated for configuration management and treated as a single entity in the configuration management process. [IEEE-STD-610 Glossary]

documented procedure - A written description of a course of action to be taken to perform a given task. [IEEE-STD-610 Glossary]

software baseline - A set of configuration items (software documents and software components) that has been formally reviewed and agreed upon, that thereafter serves as the basis for future development, and that can be changed only through formal change control procedures

software work product - Any artifact created as part of defining, maintaining, or using a software process, including process descriptions, plans, procedures, computer programs, and associated documentation, which may or may not be intended for delivery to a customer or end user.

Goals

1. Software configuration management activities are planned.
2. Selected software work products are identified, controlled, and available.
3. Changes to identified software work products are controlled.
4. Affected groups and individuals are informed of the status and content of software baselines.

31. Organization Process Focus

The purpose of Organization Process Focus is to establish the organizational responsibility for software process activities that improve the organization's overall software process capability. Organization Process Focus involves developing and maintaining an understanding of the organization's and projects' software processes and coordinating the activities to assess, develop, maintain, and improve these processes. The organization provides long-term commitments and resources to coordinate the development and maintenance of the software processes across current and future software projects via a group such as a software engineering process group. This group is responsible for the organization's software process activities.

Definitions

periodic review/activity - A review/activity that occurs at a specified regular time interval, rather than at the completion of major events.

software process - A set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (e.g., project plans, design documents, code, test cases, and user manuals)

software process assessment - An appraisal by a trained team of software professionals to determine the state of an organization's current software process, to determine the high-priority software process-related issues facing an organization, and to obtain the organizational support for software process improvement.

software engineering process group (SEPG) - A group of specialists who facilitate the definition, maintenance and improvement of the software process used by the organization. In the key practices, this group is generically referred to as "the group responsible for the organization's software process activities."

Goals

1. Software process development and improvement activities are coordinated across the organization.
2. The strengths and weaknesses of the software processes used are identified relative to a process standard.
3. Organization-level process development and improvement activities are planned.

32. Organization Process Definition

The purpose of Organization Process Definition is to develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for cumulative, long- term benefits to the organization. Organization Process Definition involves developing and maintaining the organization's standard software process, along with related process assets, such as descriptions of software life cycles, process tailoring guidelines and criteria, the organization's software process database, and a library of software process-related documentation.

Definitions

audit - An independent examination of a work product or set of work products to assess compliance with specifications, standards, contractual agreements, or other criteria.

organization's standard software process - The operational definition of the basic process that guides the establishment of a common software process across the software projects in an organization. It describes the fundamental software process elements that each software project is expected to incorporate into its defined software process. It also describes the relationships (e.g., ordering and interfaces) between these software process elements.

software quality assurance (SQA) - (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that a software work product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which software work products are developed and/or maintained.

Goals

1. A standard software process for the organization is developed and maintained.
2. Information related to the use of the organization's standard software process by the software projects is collected, reviewed, and made available.

33. Training Program

The purpose of the Training Program key process area is to develop the skills and knowledge of individuals so they can perform their roles effectively and efficiently. Training Program involves first identifying the training needed by the organization, projects, and individuals, then developing or procuring training to address the identified needs. Some skills are effectively and efficiently imparted through informal vehicles (e.g., on- the-job training and informal mentoring), whereas other skills need more formal training vehicles (e.g., classroom training and guided self-study) to be effectively and efficiently imparted. The appropriate vehicles are selected and used.

Definitions

periodic review/activity - A review/activity that occurs at a specified regular time interval, rather than at the completion of major events.

software engineering group (SEG) - The collection of individuals (both managers and technical staff) who have responsibility for software development and maintenance activities (i.e., requirements analysis, design, code, and test) for a project. Groups performing software related work, such as the software quality assurance group, the software configuration management group, and the software engineering process group, are not included in the software engineering group.

Goals

1. Training activities are planned.
2. Training for developing the skills and knowledge needed to perform software management and technical roles is provided.
3. Individuals in the software engineering group and software-related groups receive the training necessary to perform their roles.

34. Integrated Software Management

The purpose of Integrated Software Management is to integrate the software engineering and management activities into a coherent, defined software process that is tailored from the organization's standard software process and related process assets. Integrated Software Management involves developing the project's defined software process and managing the software project using this defined software process. The project's defined software process is tailored from the organization's standard software process to address the specific characteristics of the project. The software development plan is based on the project's defined software process and describes how the activities of the project's defined software process will be implemented and managed.

Definitions

organization's standard software process - The operational definition of the basic process that guides the establishment of a common software process across the software projects in an organization. It describes the fundamental software process elements that each software project is expected to incorporate into its defined software process. It also describes the relationships (e.g., ordering and interfaces) between these software process elements.

policy - A guiding principle, typically established by senior management, which is adopted by an organization or project to influence and determine decisions.

project's defined software process - The operational definition of the software process used by a project. The project's defined software process is a well-characterized and understood software process, described in terms of software standards, procedures, tools, and methods. It is developed by tailoring the organization's standard software process to fit the specific characteristics of the project.

tailoring - To modify a process, standard, or procedure to better match process or product requirements.

Goals

1. The project's defined software process is a tailored version of the organization's standard software process.
2. The project is planned and managed according to the project's defined software process.

35. Software Product Engineering

The purpose of Software Product Engineering is to consistently perform a well-defined engineering process that integrates all the software engineering activities to produce and support correct, consistent software products effectively and efficiently. Software Product Engineering involves performing the engineering tasks to build and maintain the software using the project's defined software process and appropriate methods and tools. The software engineering tasks include analyzing the system requirements allocated to software, developing the software architecture, designing the software, implementing the software in the code, and testing the software to verify that it satisfies the specified requirements

Definitions

project's defined software process - The operational definition of the software process used by a project. The project's defined software process is a well-characterized and understood software process, described in terms of software standards, procedures, tools, and methods. It is developed by tailoring the organization's standard software process to fit the specific characteristics of the project.

organization's standard software process - The operational definition of the basic process that guides the establishment of a common software process across the software projects in an organization. It describes the fundamental software process elements that each software project is expected to incorporate into its defined software process. It also describes the relationships (e.g., ordering and interfaces) between these software process elements.

software work product - Any artifact created as part of defining, maintaining, or using a software process, including process descriptions, plans, procedures, computer programs, and associated documentation, which may or may not be intended for delivery to a customer or end user.

Goals

1. The software engineering tasks are defined, integrated, and consistently performed to produce the software.
2. Software work products are kept consistent with each other.

36. Intergroup Coordination

The purpose of Intergroup Coordination is to establish a means for the software engineering group to participate actively with the other engineering groups so the project is better able to satisfy the customer's needs effectively and efficiently. Intergroup Coordination involves the software engineering group's participation with other project engineering groups to address system-level requirements, objectives, process, and issues. Representatives of the project's engineering groups participate in establishing the system-level requirements, objectives, and plans by working with the customer and end users, as appropriate. These requirements, objectives, and plans become the basis for all engineering activities.

Definitions

software engineering group (SEG) - The collection of individuals (both managers and technical staff) who have responsibility for software development and maintenance activities (i.e., requirements analysis, design, code, and test) for a project. Groups performing software related work, such as the software quality assurance group, the software configuration management group, and the software engineering process group, are not included in the software engineering group.

commitment - A pact that is freely assumed, visible, and expected to be kept by all parties.

Goals

1. The customer's requirements are agreed to by all affected groups.
2. The commitments between the engineering groups are agreed to by the affected groups.
3. The engineering groups identify, track, and resolve intergroup issues.

37. Peer Reviews

The purpose of Peer Reviews is to remove defects from the software work products early and efficiently. An important corollary effect is to develop a better understanding of the software work products and of defects that might be prevented. Peer Reviews involve a methodical examination of software work products by the producers' peers to identify defects and areas where changes are needed. The specific products that will undergo a peer review are identified in the project's defined software process and scheduled as part of the software project planning activities.

Definitions

peer review - A review of a software work product, following defined procedures, by peers of the producers of the product for the purpose of identifying defects and improvements.

software work product - Any artifact created as part of defining, maintaining, or using a software process, including process descriptions, plans, procedures, computer programs, and associated documentation, which may or may not be intended for delivery to a customer or end user

Goals

1. Peer review activities are planned.
2. Defects in the software work products are identified and removed.

38. Quantitative Process Management

The purpose of Quantitative Process Management is to control the process performance of the software project quantitatively. Quantitative Process Management involves taking measurements of the process performance, analyzing these measurements, and making adjustments to maintain process performance within acceptable limits. When the process performance is stabilized within acceptable limits, the project's defined software process, the associated measurements, and the acceptable limits for the measurements are established as a baseline and used to control process performance quantitatively.

Definitions

organization's standard software process - The operational definition of the basic process that guides the establishment of a common software process across the software projects in an organization. It describes the fundamental software process elements that each software project is expected to incorporate into its defined software process. It also describes the relationships (e.g., ordering and interfaces) between these software process elements.

process capability - The range of expected results that can be achieved by following a process.

process performance - A measure of the actual results achieved by following a process.

project's defined software process - The operational definition of the software process used by a project. The project's defined software process is a well-characterized and understood software process, described in terms of software standards, procedures, tools, and methods. It is developed by tailoring the organization's standard software process to fit the specific characteristics of the project.

Goals

1. The quantitative process management activities are planned.
2. The process performance of the project's defined software process is controlled quantitatively.
3. The process capability of the organization's standard software process is known in quantitative terms.

39. Software Quality Management

Software Quality Management involves defining quality goals for the software products, establishing plans to achieve these goals, and monitoring and adjusting the software plans, software work products, activities, and quality goals to satisfy the needs and desires of the customer and end user. Quantitative product quality goals are established based on the needs of the organization, customer, and end user for high-quality products. So that these goals may be achieved, the organization establishes strategies and plans, and the project specifically adjusts its defined software process, to accomplish the quality goals.

Definitions

software quality goal - Quantitative quality objectives defined for software work product.

Goals

1. The project's software quality management activities are planned.

2. Measurable goals for software product quality and their priorities are defined.
3. Actual progress toward achieving the quality goals for the software products is quantified and managed.

40. Defect Prevention

Defect Prevention involves analyzing defects that were encountered in the past and taking specific actions to prevent the occurrence of those types of defects in the future. The defects may have been identified on other projects as well as in earlier stages or tasks of the current project. Trends are analyzed to track the types of defects that have been encountered and to identify defects that are likely to recur. Both the project and the organization take specific actions to prevent recurrence of the defects.

Definitions

causal analysis meeting - A meeting, conducted after completing a specific task, to analyze defects uncovered during the performance of that task.

common cause (of a defect) - A cause of a defect that is inherently part of a process or system. Common causes affect every outcome of the process and everyone working in the process.

Goals

1. Defect prevention activities are planned.
2. Common causes of defects are sought out and identified.
3. Common causes of defects are prioritized and systematically eliminated.

41. Technology Change Management

Technology Change Management involves identifying, selecting, and evaluating new technologies, and incorporating effective technologies into the organization. The objective is to improve software quality, increase productivity, and decrease the cycle time for product development. The organization establishes a group (such as a software engineering process group or a technology support group) that works with the software projects to introduce and evaluate new technologies and manage changes to existing technologies. Particular emphasis is placed on technology changes that are likely to improve the capability of the organization's standard software process. Pilot efforts are performed to assess new and unproven technologies before they are incorporated into normal practice. With appropriate sponsorship of the organization's management, the selected technologies are incorporated into the organization's standard software process and current projects, as appropriate.

Definitions

documented procedure - A written description of a course of action to be taken to perform a given task. [IEEE-STD-610 Glossary]

organization's standard software process - The operational definition of the basic process that guides the establishment of a common software process across the software projects in an organization. It describes the fundamental software process elements that each software project is expected to incorporate into its defined software process. It also describes the relationships (e.g., ordering and interfaces) between these software process elements.

Goals

1. Incorporation of technology changes are planned.
2. New technologies are evaluated to determine their effect on quality and productivity.
3. Appropriate new technologies are transferred into normal practice across the organization.

42. Process Change Management

Process Change Management involves defining process improvement goals and, with senior management sponsorship, proactively and systematically identifying, evaluating, and implementing improvements to the organization's standard software process and the projects' defined software processes on a continuous basis. Training and incentive programs are established to enable and encourage everyone in the organization to participate in process improvement activities. Improvement opportunities are identified and evaluated for potential payback to the organization. Pilot efforts are performed to assess process changes before they are incorporated into normal practice. When software process improvements are approved for normal practice, the organization's standard software process and the projects' defined software processes are revised as appropriate.

Definitions

documented procedure - A written description of a course of action to be taken to perform a given task.[IEEE-STD-610 Glossary]

organization's standard software process - The operational definition of the basic process that guides the establishment of a common software process across the software projects in an organization. It describes the fundamental software process elements that each software project is expected to incorporate into its defined software process. It also describes the relationships (e.g., ordering and interfaces) between these software process elements.

project's defined software process - The operational definition of the software process used by a project. The project's defined software process is a well-characterized and understood software process, described in terms of software standards, procedures, tools, and methods. It is developed by tailoring the organization's standard software process to fit the specific characteristics of the project.

Goals

1. Continuous process improvement is planned.
2. Participation in the organization's software process improvement activities is organization wide.
3. The organization's standard software process and the projects' defined software processes are improved continuously.

This page left intentionally blank.

Appendix D: Values for COCOMO II.1997

This Appendix has the values for the COCOMO II.1997 model that is a result of calibration on 83 projects.

W(i)	Very Low	Low	Nominal	High	Very High	Extra High
Precedentedness	4.05	3.24	2.43	1.62	0.81	0.00
Development Flexibility	6.07	4.86	3.64	2.43	1.21	0.00
Architecture / Risk Resolution	4.22	3.38	2.53	1.69	0.84	0.00
Team Cohesion	4.94	3.95	2.97	1.98	0.99	0.00
Process Maturity	4.54	3.64	2.73	1.82	0.91	0.00

Table 1: Scale Factors

Cost Driver	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
RELY	0.75	0.88	1.00	1.15	1.39	
DATA		0.93	1.00	1.09	1.19	
CPLX	0.75	0.88	1.00	1.15	1.30	1.66
RUSE		0.91	1.00	1.14	1.29	1.49
DOCU	0.89	0.95	1.00	1.06	1.13	
TIME			1.00	1.11	1.31	1.67
STOR			1.00	1.06	1.21	1.57
PVOL		0.87	1.00	1.15	1.30	
ACAP	1.50	1.22	1.00	0.83	0.67	
PCAP	1.37	1.16	1.00	0.87	0.74	
PCON	1.24	1.10	1.00	0.92	0.84	
AEXP	1.22	1.10	1.00	0.89	0.81	
PEXP	1.25	1.12	1.00	0.88	0.81	
LTEX	1.22	1.10	1.00	0.91	0.84	
TOOL	1.24	1.12	1.00	0.86	0.72	
SITE	1.25	1.10	1.00	0.92	0.84	0.78
SCED	1.29	1.10	1.00	1.00	1.00	

Table 2: Cost Drivers

Multiplicative Constant for Effort = 2.45

Multiplicative Constant for Schedule = 2.66