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.

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. Lukasieicz (1991), "Occupational Employment Projections," Monthly Labor Review 114(11), November 1991, pp. 64-94.

SPR (1993), "Checkpoint User's Guide for the Evaluator", Software Productivity Research, Inc., Burlington, MA., 1993.

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.

Boehm, B., B. Clark, E. Horowitz, C. Westland, R. Madachy, R. Selby (1995), "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," to appear in Annals of Software Engineering Special Volume on Software Process and Product Measurement, J.D. Arthur and S.M. Henry, Eds., J.C. Baltzer AG, Science Publishers, Amsterdam, The Netherlands. Available from the Center for Software Engineering, University of Southern California.

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.

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,

7. CONCLUSIONS

Software development trends towards reuse, reengineering, commercial off-the shelf (COTS) packages, object orientation, applications composition capabilities, non-sequential process models, rapid development approaches, and distributed middleware capabilities require new approaches to software cost estimation.

The wide variety of current and future software processes, and the variability of information available to support software cost estimation, require a family of models to achieve effective cost estimates.

The baseline COCOMO 2.0 family of software cost estimation models presented here provides a tailorable cost estimation capability well matched to the major current and likely future software process trends.

The baseline COCOMO 2.0 model effectively addresses its objectives of openness, parsimony, and continuity from previous COCOMO models. It is currently serving as the framework for an extensive data collection and analysis effort to further refine and calibrate its estimation capabilities.

8. ACKNOWLEDGMENTS

This work has been supported both financially and technically by the COCOMO 2.0 Program Affiliates: Aerospace, AT&T Bell Labs, Bellcore, DISA, EDS, E-Systems, Hewlett-Packard, Hughes, IDA, IDE, JPL, Litton Data Systems, Lockheed, Loral, MDAC, Motorola, Northrop, Rational, Rockwell, SAIC, SEI, SPC, Sun, TASC, Teledyne, TI, TRW, USAF Rome Lab, US Army Research Lab, Xerox.

9. 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.

6.3.4 PEXP - Platform Experience

COCOMO 2.0 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;

6.3.5 LTEX - Language and Tool Experience

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.

6.3.6 PCON - Personnel Continuity

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

6.4 Project Factors

6.4.1 TOOL - Use of Software Tools

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.

6.4.2 SITE - Multisite Development

Given the increasing frequency of multisite developments, and indications that multisite development effects are significant, the SITE cost driver has been added in COCOMO 2.0. 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).

6.4.3 SCED - Required Development Schedule

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.

Table 6: Module Complexity Ratings versus Type of Module

Rating	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations	User Interface Management Operations
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 multi-media, virtual reality.

veloped 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.

6.3 Personnel Factors

6.3.1 ACAP - Analyst Capability

Analysts are personnel that work on requirements, high level design and detailed design. The major attributes that should be considered in this rating are 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. Analyst that fall in the 15th percentile are rated very low and those that fall in the 95th percentile are rated as very high.

6.3.2 PCAP - Programmer Capability

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.

6.3.3 AEXP - Applications Experience

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.

Table 6: Module Complexity Ratings versus Type of Module

Rating	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.

6.1.3 CPLX - Product Complexity

Table 6 provides the new COCOMO 2.0 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 ratings range from very low to extra high.

6.1.4 RUSE - Required Reusability

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.

6.1.5 DOCU - Documentation match to life-cycle needs

Several software cost models have a cost driver for the level of required documentation. In COCOMO 2.0, 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).

6.2 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 operation, but these considerations have been accommodated by the expansion of the Module Complexity ratings in Table 6.

6.2.1 TIME - Execution Time Constraint

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.

6.2.2 STOR - Main Storage Constraint

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%.

6.2.3 PVOL - Platform Volatility

"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 de-

Table 5: Effort Multipliers Cost Driver Ratings for the Post-Architecture Model

	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 6					
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%	

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.

5.3 Adjusting Effort for Development Characteristics

Effort multipliers (EM) capture characteristics of the software development that affect the effort to complete the project. These multipliers are weighted and their product is used to adjust the nominal person month effort. The nominal weight assigned to each multiplier is 1.0. If a rating level has a detrimental effect on effort, then its corresponding multiplier is above 1.0. Conversely, if the rating level reduces the effort then the corresponding multiplier is less than 1.0. The effort multipliers are discussed in the next section.

$$PM_{adjusted} = PM_{nominal} \times \left(\prod_i EM_i \right) \quad \text{EQ 4.}$$

6. Effort-Multiplier Cost Drivers

These are the effort 17 multipliers used in COCOMO 2.0 Post-Architecture model to adjust the nominal effort, Person Months, to reflect the software product under development. They are grouped into four categories: product, platform, personnel, and project. Table 4 lists the different cost drivers with their rating criterion. The counterpart 7 cost drivers for the Early Design Model are defined in [Boehm et al. 1995]

6.1 Product Factors

6.1.1 RELY- Required Software Reliability

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.

6.1.2 DATA - Data Base Size

This measure attempts to capture the effect of large data requirements on product development. The rating is determined by calculating D/P.

$$\frac{D}{P} = \frac{DataBaseSize\,(Bytes)}{ProgramSize\,(SLOC)} \quad \text{EQ 5.}$$

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

months (PM) is given by EQ 2.

$$PM_{nominal} = A \times (Size)^{B} \qquad \text{EQ 2.}$$

5.2 Exponent Scale Factors

Table 4 provides the rating levels for the COCOMO 2.0 exponent scale factors. A project's numerical ratings W_i are summed across all of the factors, and used to determine a scale exponent B via the following formula:

$$B = 1.01 + 0.01 \Sigma W_i \qquad \text{EQ 3.}$$

Thus, a 100 KSLOC project with Extra High (0) ratings for all factors will have $W_i = 0$, $B = 1.01$, and a relative effort $E = 100^{1.01} = 105$ *PM*. A project with Very Low (5) ratings for all factors will have $W_i = 25$, $B = 1.26$, 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%. Thus, this approach avoids the 40% swings involved in choosing a development mode for a 100 KSLOC product in the original COCOMO.

Table 4: Rating Scheme for the COCOMO 2.0 Scale Factors

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

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

† The form of the Process Maturity scale is being resolved in coordination with the SEI. The intent is to produce a process maturity rating as a weighted average of the project's responses to the most recent Capability Maturity Model-based Maturity Questionnaire, rather than to use the previous 1-to-5 maturity levels. The weights to be applied to the Maturity Questionnaire questions are still being determined·

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 startup costs such as tailoring and setup of standards and administrative reports are a source of economies of scale.

If $B = 1.0$, the economies and diseconomies of scale are in balance. This linear model is often used for cost estimation of small projects. It is used for the COCOMO 2.0 Applications Composition model.

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

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

The COCOMO 2.0 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, an effort estimator for automated translation is 2400 source statements / person month; the normal COCOMO 2.0 reuse model is used for the remainder of the re-engineered software.

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 3.

Table 3: Variation in Percentage of Automated Re-engineering [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%

4.2.4 Applications Maintenance

The original COCOMO used Annual Change Traffic (ACT), the percentage of code modified and added to the software product per year, as the primary measure for sizing a software maintenance activity. This has caused some difficulties, primarily the restriction to annual increment and a set of inconsistencies with the reuse model. COCOMO 2.0 remedies these difficulties by applying the reuse model to maintenance as well.

5. COCOMO 2.0 COST MODELING

5.1 Modeling Effort

This software cost estimation model has an exponential factor to account for the relative economies or diseconomies of scale encountered as a software project increases its size. This factor is represented as the exponent *B.* A constant is used to capture the linear effects on effort with projects of increasing size. The nominal effort for a given size project and expressed as person

This involves estimating the amount of software to be adapted, ASLOC, and three degree-of-modification parameters: the percentage of design modification (DM); the percentage of code modification (CM), and the percentage of the original integration effort required for integrating the reused software (IM). The *Software Understanding* increment (SU) is obtained from Table 1. As indicated in Table 1, if the software is rated very high on structure, applications clarity, and self-descriptiveness, the software understanding and interface checking penalty is only 10%. If the software is rated very low on these factors, the penalty is 50%.

Table 1: Rating Scale for Software Understanding Increment SU

	Very Low	**Low**	**Nom**	**High**	**Very High**
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately low cohesion, high coupling.	Reasonably well-structured; some weak areas.	High cohesion, low coupling.	Strong modularity, information hiding in data / control structures.
Application Clarity	No match between program and application world views.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application world-views.
Self-Descriptiveness	Obscure code; documentation mission, 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 AAF	50	40	30	20	10

The other nonlinear reuse increment deals with the degree of assessment and assimilation needed to determine whether even 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). For software conversion, this factor extends the Conversion Planning Increment in [Boehm 1981, p. 558].

4.2.3 RE-ENGINEERING AND CONVERSION COST ESTIMATION

The COCOMO 2.0 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 2.0 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.

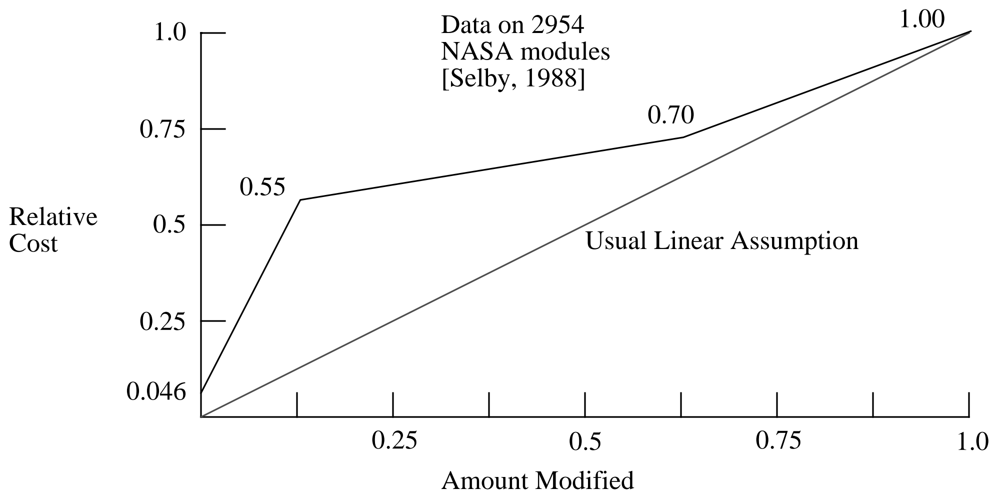

Figure 3. Nonlinear Reuse Effects

1)/2.

Figure 4 shows this relation between the number of modules modified *k* and the resulting number of module interface checks required.

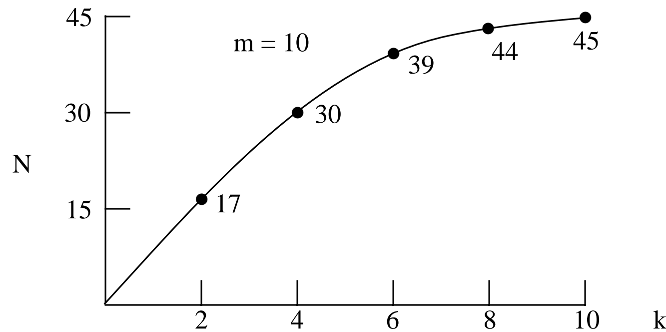

Figure 4. Number of Module Interface Checks vs. Fraction Modified

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

The size of both the software understanding penalty and the module interface checking penalty can be reduced by good software stucturing. Modular, hierarchical structuring can reduce the number of interfaces which need checking [Gerlich and Denskat 1994], and software which is well structured, explained, and related to its mission will be easier to understand. COCOMO 2.0 reflects this in its allocation of estimated effort for modifying reusable software.

The reuse equation for equivalent new software (ESLOC) to be developed is:

$$ESLOC = ASLOC \times \left(\frac{AA + SU}{100} + 0.4 \times DM + 0.3 \times CM + 0.3 \times IM \right) \qquad \textit{EQ 1.}$$

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 2.0 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 2.0 data collection and analysis project. We have developed a set of Amadeus measurement templates that support the COCOMO 2.0 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 formulation will adapt according to the analysis results.

4.2 Adjusting Software Development Size

4.2.1 Breakage

COCOMO 2.0 replaces the COCOMO Requirements Volatility effort multiplier and the Ada COCOMO Requirements Volatility exponent driver by a breakage percentage, BRAK, used to adjust the effective size of the product. Consider a project which delivers 100,000 instructions but discards the equivalent of an additional 20,000 instructions. This project would have a BRAK value of 20, which would be used to adjust its effective size to 120,000 instructions for COCOMO 2.0 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.

4.2.2 Effects from Reuse

The COCOMO 2.0 model uses a nonlinear estimation model for estimating size in reusing software products. 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.

[Parikh and Zvegintzov 1983] contains data indicating that 47% of the effort in software maintenance involves understanding the software to be modified. Thus, as soon as one goes from unmodified (black-box) reuse to modified-software (white-box) reuse, one encounters this software understanding penalty. Also, [Gerlich and Denskat 1994] shows that, if one modifies k out of m software modules, the number N of module interface checks required is $N = k * (m-k) + k * (k-$

Definition Checklist for Source Statements Counts

Definition name: __Logical Source Statements___Date:________________

________________(basic definition)__________Originator:_COCOMO 2.0____

Measurement unit:	**Physical source lines**				
	Logical source statements	4			
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 →		1	✔	
2 Nonexecutable					
3 Declarations			2	✔	
4 Compiler directives			3	✔	
5 Comments					
6 On their own lines			4		✔
7 On lines with source code			5		✔
8 Banners and non-blank spacers			6		✔
9 Blank (empty) comments			7		✔
10 Blank lines			8		✔
11					
12					
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					✔
7					
8					
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				✔	
13					
14					

Figure 2. Definition Checklist

tion points and a coarse-grained set of 7 cost drivers (e.g., two cost drivers for Personnel Capability and Personnel Experience in place of the 6 Post-Architecture model cost drivers covering various aspects of personnel capability, continuity, and experience). This level of detail is consistent with the general level of information available and the general level of estimation accuracy needed at this time in the development.

The *Post-Architecture Model* involves the actual development and maintenance of a software product. The product should have a life-cycle architecture, which provides more accurate information on cost driver inputs, and enables more accurate cost estimates. This architecture can be validated with respect to the system's mission, concept of operation, and risk; and established as the framework for the product. 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.

The above models should be considered as current working hypotheses about the most effective forms for COCOMO 2.0. 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.

4. Sizing Software Development

As has been discussed, the COCOMO 2.0 model uses three different metrics for sizing a project: Object Points, Unadjusted Function Points, and Source Lines of Code (SLOC). Object Points are defined as in Section 3.1.1 above. Unadjusted Function Points involve using the standard [IFPUG 1994] sizing approach involving a linear combination of inputs, outputs, files, interfaces, and queries; but not using the 14 application characteristics such as distributed functions, performance, and reuse. Instead, COCOMO 2.0 accounts for such factors via its normal set of cost drivers and relationships. The remainder of this Section discusses the definition and rationale for SLOC and how size is adjusted in dealing with software breakage, reuse, re-engineering, and maintenance.

4.1 Source Lines of Code

In COCOMO 2.0, 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 2 shows a portion of the definition checklist as it is being applied to support the development of the COCOMO 2.0 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.

- Preserve the openness of the original COCOMO;
- Key the structure of COCOMO 2.0 to the future software marketplace sectors described above;
- Key the inputs and outputs of the COCOMO 2.0 submodels to the level of information available;
- Enable the COCOMO 2.0 submodels to be tailored to a project's particular process strategy.

COCOMO 2.0 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 2.0.

To support the software marketplace sectors above, COCOMO 2.0 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.

3.1 COCOMO 2.0 Models for the Software Marketplace Sectors

The *User Programming* sector does not need a COCOMO 2.0 model. Its applications are typically developed in hours to days, so a simple activity-based estimate will generally be sufficient.

3.1.1 Application Composition

The COCOMO 2.0 estimation model for this sector, called the *Application Composition Model*, 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).

3.1.2 Application Generator, System Integration, or Infrastructure

Estimations for these three sectors are based on a tailorable mix of the Application Composition Model and two increasingly detailed estimation models, called the Early Design Model and the Post-Architecture Model, for subsequent portions of the life cycle.

The earliest life cycle phases or spiral cycles will generally involve prototyping, using Application Composition capabilities. The *Application Composition Model* supports these phases. It also supports other prototyping activities occurring later in the life cycle such as efforts to resolve potential high-risk issues such as user interfaces, software/system interaction, performance, or technology maturity.

The *Early Design Model* involves exploration of alternative software/system architectures and concepts of operation. At this point the development, not enough is generally known to support fine-grain cost estimation. The corresponding COCOMO 2.0 capability involves the use of func-

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 middle 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.

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.

3. COCOMO 2.0 STRATEGY AND RATIONALE

The four main elements of the COCOMO 2.0 strategy are:

ject Points, Function Points, and Source Lines of Code (SLOC), with new adjustment models for reuse and re-engineering. Section 5 discusses the new exponent-driver approach to modeling relative project diseconomies of scale. Section 6 summarizes the revisions to one of COCOMO's effort-multiplier cost driver sets. Section 7 presents the resulting conclusions based on COCOMO 2.0's current state. Further details on the definition of COCOMO 2.0 are provided in [Boehm et al. 1995]

2. FUTURE SOFTWARE PRACTICES MARKETPLACE MODEL

Figure 1 summarizes the model of the future software practices marketplace that we are using to guide the development of COCOMO 2.0. 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)*.

<table>
<tr><td colspan="3">End-User Programming
(55M performers in US)</td></tr>
<tr><td>Application Generators
and Composition Aids
(0.6M)</td><td>Application
Composition
(0.7M)</td><td>System
Integration
(0.7M)</td></tr>
<tr><td colspan="3">Infrastructure
(0.75M)</td></tr>
</table>

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

* These figures are judgement-based extensions of the Bureau of Labor Statistics moderate-growth labor distribution scenario for the year 2005 [CSTB 1993; Silvestri and Lukasieicz 1991]. The 55 million End-User programming figure was obtained by applying judgement 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" (0.829M in 2005) and "Computer Programmers (0.882M). The expansion factor of 1.6 to cover software personnel with other job titles is based on the results of a 1983 survey on this topic [Boehm 1983].The 2005 distribution of the 2.75 M software developers is a judgement-based extrapolation of current trends.

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 the authors to formulate 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.

1.2 COCOMO 2.0 Objectives

The initial definition of COCOMO 2.0 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 2.0 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.

1.3 Topics Addressed

Section 2 describes the future software marketplace model being used to guide the development of COCOMO 2.0. Section 3 presents the overall COCOMO 2.0 strategy and its rationale. Section 4 summarizes a COCOMO 2.0 software sizing approach, involving a tailorable mix of Ob-

Presenters: Barry W. Boehm and Bradford K. Clark

Title: An Overview of the COCOMO 2.0 Software Cost Model

Authors: Barry Boehm, Bradford Clark, Ellis Horowitz, Ray Madachy, Richard Selby, Chris Westland

Track: Track 6

Day: Thursday

Keywords: Cost Modeling, Cost Estimation, COCOMO, Reuse

AN OVERVIEW OF THE COCOMO 2.0 SOFTWARE COST MODEL

Abstract

Current software cost estimation models, such as the 1981 Constructive Cost Model (COCOMO) for software cost estimation and its 1987 Ada COCOMO update, have been experiencing increasing difficulties in estimating the costs of software developed to new life cycle processes and capabilities. These include non-sequential and rapid-development process models; reuse-driven approaches involving commercial off the shelf (COTS) packages, reengineering, applications composition, and applications generation capabilities; object-oriented approaches supported by distributed middleware; and software process maturity initiatives.

This paper provides an overview of the baseline COCOMO 2.0 model tailored to these new forms of software development, including rationales for the model decisions. The major new modeling capabilities of COCOMO 2.0 are a tailorable family of software sizing models, involving Object Points, Function Points, and Source Lines of Code; nonlinear models for software reuse and reengineering; an exponent-driver approach for modeling relative software diseconomies of scale; and several additions, deletions, and updates to previous COCOMO effort-multiplier cost drivers. This model is serving as a framework for an extensive current data collection and analysis effort to further refine and calibrate the model's estimation capabilities.

1. INTRODUCTION

1.1 MOTIVATION

"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 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