E-COCOMO: The Extended COst Constructive MOdel for Cleanroom Software Engineering

Hitesh KUMAR SHARMA
University of Petroleum and Energy Studies, India
hkshitesh@gmail.com

Mistakes create rework. Rework takes time and increases costs. The traditional software engineering methodology defines the ratio of Design:Code:Test as 40:20:40. As we can easily see that 40% time and efforts are used in testing phase in traditional approach, that means we have to perform rework again if we found some bugs in testing phase. This rework is being performed after Design and code phase. This rework will increase the cost exponentially. The cleanroom software engineering methodology controls the exponential growth in cost by removing this rework. It says that "do the work correct in first attempt and move to next phase after getting the proof of correctness".
This new approach minimized the rework and reduces the cost in the exponential ratio. Due to the removal of testing phase, the COCOMO (COst COnstructive MOdel) used for the traditional engineering is not directly applicable in cleanroom software engineering. The traditional cost drivers used for traditional COCOMO needs to be revised. We have proposed the Extended version of COCOMO (i.e. E-COCOMO) in which we have incorporated some new cost drivers. This paper explains the proposed E-COCOMO and the detailed description of proposed new cost driver.
Keywords: *Cleanroom Software Engineering, COCOMO, Effort Estimation, Cost Drivers, SDLC.*

1 Introduction

Harlan Mills and his colleagues from IBM developed the CSE (Cleanroom Software Engineering) methodology in the early 1980s. They were part of IBM's Federal Defense System where software failures could mean millions of dollars and most importantly, human lives. In This software methodology they used the same analogy as used in cleanroom fabrication of semiconductors. Instead of trying to clean dirt off the semiconductor wafers after production, the object is to prevent the dirt from getting into the production environment in the first place. The reason for this is that defect prevention is more cost effective than defect removal. Therefore, in software development, the CSE methodology eliminates or avoids as many defects as possible before software execution using controlled and measurable statistics. Because of that reason they start the cleanroom software Development methodology for software development.

The **Constructive Cost Model** (**COCOMO**) is an algorithmic software cost estimation model developed by Barry Boehm. The model uses a basic regression formula, with parameters that are derived from historical project data and current project characteristics. COCOMO was first published in 1981 Barry W. Boehm's Book *Software engineering economic* as a model for estimating effort, cost, and schedule for software projects. It drew on a study of 63 projects at TRW Aerospace where Barry Boehm was Director of Software Research and Technology in 1981. The study examined projects ranging in size from 2,000 to 100,000 lines of code, and programming languages ranging from assembly to PL/I. These projects were based on the waterfall model of software

development which was the prevalent software development process in 1981.

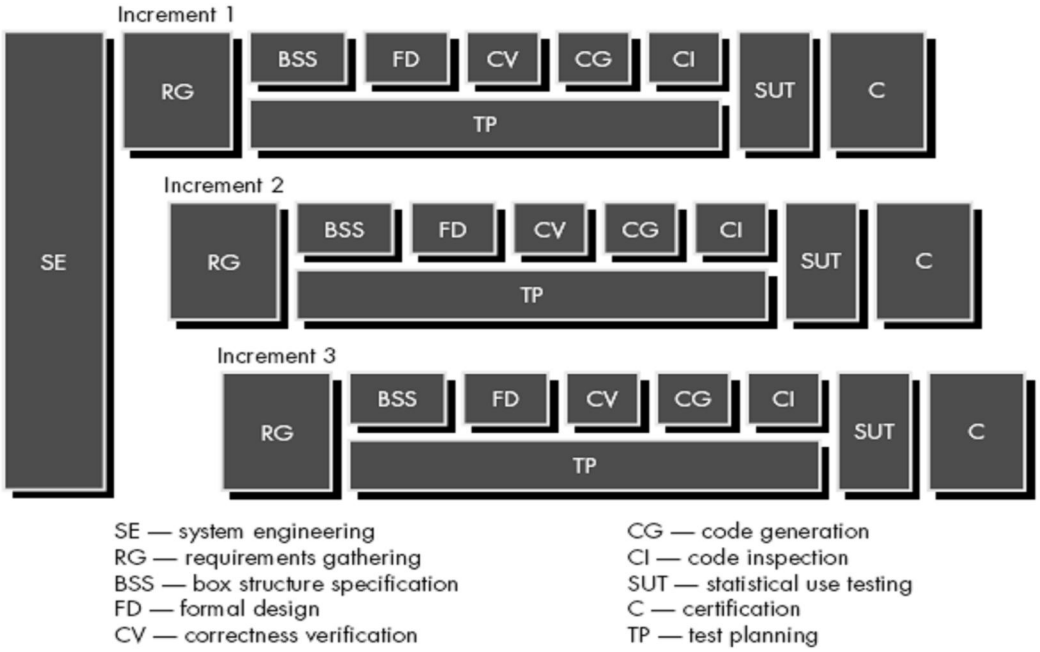
2. Cleanroom Software Engineering (CSE)

The cleanroom approach makes use of a specialized version of the incremental software model. A "pipeline of software increments" is developed by small independent software engineering teams. As each increment is certified, it is integrated in the whole. Hence, functionality of the system grows with time. The sequence of cleanroom tasks for each increment is illustrated in Figure 1. Overall system or product requirements are developed using the system engineering methods. Once functionality has been assigned to the software element of the system, the pipeline of cleanroom increments is initiated. The following tasks occur in CSE:

Increment planning. A project plan that adopts the incremental strategy is developed. The functionality of each increment, its projected size, and a cleanroom development schedule are created. Special care must be taken to ensure that certified increments will be integrated in a timely manner.

Requirements gathering. Using traditional techniques, a more-detailed description of customer-level requirements (for each increment) is developed.

Box structure specification. A specification method that makes use of box structures is used to describe the functional specification. Box structures "isolate and separate the creative definition of behavior, data, and procedures at each level of refinement."



Formal design. Using the box structure approach, cleanroom design is a natural and seamless extension of specification. Although it is possible to make a clear distinction between the two activities, specifications (called *black boxes*) are iteratively refined (within an increment) to become analogous to architectural and component-level designs (called *state boxes* and *clear boxes,* respectively).

Correctness verification. The cleanroom team conducts a series of rigorous

correctness verification activities on the design and then the code. Verification begins with the highest-level box structure (specification) and moves toward design detail and code. The first level of correctness verification occurs by applying a set of "correctness questions". If these do not demonstrate that the specification is correct, more formal (mathematical) methods for verification are used.

Code generation, inspection, and verification. The box structure specifications, represented in a specialized language, are translated into the appropriate programming language. Standard walkthrough or inspection techniques are then used to ensure semantic conformance of the code and box structures and syntactic correctness of the code. Then correctness verification is conducted for the source code.

Statistical test planning. The projected usage of the software is analyzed and a suite of test cases that exercise a "probability distribution" of usage are planned and designed Referring to Figure 1, this cleanroom activity is conducted in parallel with specification, verification, and code generation.

Statistical use testing. Recalling that exhaustive testing of computer software is impossible, it is always necessary to design a finite number of test cases. Statistical use techniques execute a series of tests derived from a statistical sample (the probability distribution noted earlier) of all possible program executions by all users from a targeted population.

Certification. Once verification, inspection, and usage testing have been completed (and all errors are corrected), the increment is certified as ready for integration. Like other software process models discussed elsewhere in this book, the cleanroom process relies heavily on the need to produce high-quality analysis and design models. As we will see later in this chapter, box structure notation is simply another way for a software engineer to represent requirements and design. The real distinction of the cleanroom approach is that formal verification is applied to engineering models.

Dyer alludes to the differences of the cleanroom approach when he defines the process:

"*Cleanroom represents the first practical attempt at putting the software development process under statistical quality control with a well-defined strategy for continuous process improvement. To reach this goal, a cleanroom unique life cycle was defined which focused on mathematics based software engineering for correct software designs and on statistics-based software testing for certification of software reliability."*

Cleanroom software engineering differs from the conventional and object-oriented views because:

- It makes explicit use of statistical quality control.
- It verifies design specification using a mathematically based proof of correctness.
- It relies heavily on statistical use testing to uncover high-impact errors.

Obviously, the cleanroom approach applies most, if not all, of the basic software engineering principles and concept. Good analysis and design procedures are essential if high quality is to result. But cleanroom engineering diverges from conventional software practices by deemphasizing (some would say, eliminating) the role of unit testing and debugging and dramatically reducing (or eliminating) the amount of testing performed by the developer of the software. In conventional software development, errors are accepted as a fact of life. Because errors are deemed to be inevitable, each program module should be unit tested (to uncover errors) and then

debugged (to remove errors). When the software is finally released, field use uncovers still more defects and another test and debug cycle begins. The rework associated with these activities is costly and time consuming. Worse, it can be degenerative error correction can (inadvertently) lead to the introduction of still more errors. In cleanroom software engineering, unit testing and debugging are replaced by correctness verification and statistically based testing. These activities, coupled with the record keeping necessary for continuous improvement, make the cleanroom approach unique.

3. Formal specification

Formal methods allow a software engineer to create a specification that is more complete, consistent, and unambiguous than those produced using conventional or object oriented methods. Set theory and logic notation are used to create a clear statement of facts (requirements). This mathematical specification can then be analyzed to prove correctness and consistency. Because the specification is created using mathematical notation, it is inherently less ambiguous than informal modes of representation. A specially trained software engineer creates a formal specification. In safety-critical or mission critical systems, failure can have a high price. Lives may be lost or severe economic consequences can arise when computer software fails. In such situations, it is essential that errors are uncovered before software is put into operation. Formal methods reduce specification errors dramatically and, as a consequence, serve as the basis for software that has very few errors once the customer begins using it. The first step in the application of formal methods is to define the data invariant, state, and operations for a system function. The data invariant is a condition that is true throughout the execution of a function that contains a collection of data, The state is the stored data that a function accesses and alters; and operations are actions that take place in a system as it reads or writes data to a state. An operation is associated with two conditions: a precondition and a post condition. The notation and heuristics of sets and constructive specification set operators, logic operators, and sequences form the basis of formal methods. A specification represented in a formal language such as Z or VDM is produced when formal methods are applied.

4. COCOMO (COnstructive COst MOdel)

Boehm's COCOMO model is one of the mostly used model commercially. The first version of the model delivered in 1981 and COCOMO II is available now. COCOMO'81 is derived from the analysis of 63 software projects in 1981. Boehm proposed three levels of the model :

- Basic COCOMO
- Intermediate COCOMO
- Detailed COCOMO

4.1 Basic COCOMO

Basic COCOMO computes software development effort (and cost) as a function of program size. Program size is expressed in estimated thousands of lines of code (KLOC). COCOMO applies to three classes of software projects:

Table 1. Classes of Projects

Project Class	Project Size	Nature of Project	Deadline	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Not tight	Simple/Familiar/ In-house
Semi-Detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Tight	Complex

Formula for Basic COCOMO

$$E = a_b(KLOC)^{b_b}$$

$$D = c_b(E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table 2:

Table 2. Coefficients a_b, b_b, c_b and d_b values

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Basic COCOMO is good for quick estimate of software costs. However it does not account for differences in hardware constraints, personnel quality and experience, use of modern tools and techniques, and so on.

4.2 Intermediate COCOMO

Intermediate COCOMO computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel and project attributes. This extension considers a set of four "cost drivers", each with a number of subsidiary attributes:

- Product attributes
  - Required software reliability
  - Size of application database
  - Complexity of the product
- Hardware attributes
  - Run-time performance constraints

  - Memory constraints
  - Volatility of the virtual machine environment
  - Required turn about time
- Personnel attributes
  - Analyst capability
  - Software engineering capability
  - Applications experience
  - Virtual machine experience
  - Programming language experience
- Project attributes
  - Use of software tools
  - Application of software engineering methods
  - Required development schedule

Each of the 15 attributes receives a rating on a six-point scale that ranges from "very low" to "extra high" (in importance or value). An effort multiplier from the table below applies to the rating. The product of all effort multipliers results in an *effort adjustment factor (EAF)*. Typical values for EAF range from 0.9 to 1.4.

The Intermediate COCOMO formula now takes the form:

$$E = a_i (KLOC)^{b_i} * EAF$$
$$D = c_i (E)^{d_i}$$

where E is the effort applied in person-months, **KLoC** is the estimated number of thousands of delivered lines of code for the project, and **EAF** is the factor calculated above. The coefficient $\mathbf{a_i}$ and the exponent $\mathbf{b_i}$ are given in the next table.

Table 3. Coefficients a_i, b_i, c_i and d_i values

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

The Development time **D** calculation uses **E** in the same way as in the Basic COCOMO.

4.3 Detailed COCOMO

Detailed COCOMO is defined in Barry Boehm's book "Software Engineering Economics in 1981". Detailed COCOMO incorporates all characteristics of the Intermediate COCOMO version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process. Detailed COCOMO offers a means for processing all the project characteristics to construct a software estimate. The detailed model introduces two more capabilities:

The formula for detailed COCOMO is:

$$E_p = \mu_p E$$
$$D_p = \tau_p D$$

5. E-COCOMO (Extended COst COnstructive MOdel)

As we have discussed in intermediate COCOMO that there are 15 cost driver factors in traditional software engineering to calculate EAF. But as we are moving towards Cleanroom methodology in software development we need some new cost drivers which will be incorporated due to the inclusion of BSS and Formal Specification. The drivers should be added to the personal attribute category because the humans involve in CSE process should have the knowledge of these new included components. Due to the need to include

some new cost driver we found to add one new cost driver in Intermediate COCOMO that is "**Formal Method Knowledge Capability(FMKC)**". It specify the knowledge experience of Formal Method and Formal Specification Language like 'Z' Specification language. Formal Method knowledge it must require for the cleanroom Development Mythology. Formal Methods used in developing computer systems are mathematically used techniques for describing system properties. The four phases used in the detailed COCOMO model are: requirements planning and product design (RPD), detailed design (DD), code and unit test (CUT), and integration and test (IT) that is based on Waterfall model if cleanroom development mythology used then these phase will changed. We proposed to use Four phase in Detailed COCOMO model are: Increment planning and Requirement gathering (IPRG), Box structure specification and Formal Design (BSSFD), Correctness verification and code generation(CVCG),Statistical Test planning and Use Testing(STPUT).

Table 4. Table for multiplying factors for EAF
(The values for new cost driver "FMKC" is highlighted)

Cost Drivers	**Ratings**					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
RELY	0.75	0.88	1	1.15	1.4	
DATA		0.94	1	1.08	1.16	
CPLX	0.7	0.85	1	1.15	1.3	1.65
Hardware attributes						
TURN			1	1.11	1.3	1.66
VIRT			1	1.06	1.21	1.56
STOR		0.87	1	1.15	1.3	
TIME		0.87	1	1.07	1.15	
Personnel attributes						
ACAP	1.46	1.19	1	0.86	0.71	
LEXP	1.29	1.13	1	0.91	0.82	
VEXP	1.42	1.17	1	0.86	0.7	
PCAP	1.21	1.1	1	0.9		
AEXP	1.14	1.07	1	0.95		
FMKC	1.43	1.18	1	0.86	0.7	-
Project attributes						
MODP	1.24	1.1	1	0.91	0.82	
SCED	1.24	1.1	1	0.91	0.83	
TOOL	1.23	1.08	1	1.04	1.1	

The values of the coefficient (i.e. Effort coefficient μ_p and Time Coefficient $\tau_{p)}$ used it will also change in Detailed COCOMO. The modified values have been shown in the following tables.

Table 5. Table for E-COOCMO μ_p used for cleanroom engineering phases

Mode & code size	IRPG	BSSFD	CVCG	STPUT
Organic small	0.15	0.65	0.17	0.03
Organic medium	0.15	0.64	0.17	0.04
Semidetached medium	0.16	0.64	0.16	0.04
Semidetached large	016	0.63	0.15	0.06
Embedded large	0.18	0.62	0.14	0.06
Embedded extra large	0.18	0.61	0.14	0.07

Table 6. Table for E-COOCMO τ_p used for cleanroom engineering phases

Mode & code size	IRPG	BSSFD	CVCG	STPUT
Organic small	0.14	0.66	0.17	0.03
Organic Medium	0.14	0.65	0.17	0.04
Semidetached Medium	0.15	0.65	0.16	0.04
Semidetached Large	015	0.64	0.15	0.06
Embedded Large	0.17	0.63	0.14	0.06
Embedded extra large	0.17	0.62	0.14	0.07

Conclusion & future work

The software industries are adopting the new methodologies and leaving the traditional methodology far behind. Due to this transition the metrics and measurement based on the traditional methodology should also change. The old methods for effort and time calculation cannot apply on new development methodologies. To keep this transition in mind we have defined some new parameter those should be included in traditional COCOMO to calculate the effort and time for a software project. We have given a new name to this new version of COCOMO as E-COCOMO (i.e. Extended COst COnstructive MOdel). This model can be used to calculate effort and time for the projects those are adapting cleanroom software engineering methodology.

In future the work will be extended for other development methodologies (i.e Agile Development, Object Oriented Development, Component Based Engineering etc.). These methodologies cannot use traditional COCOMO to calculate the efforts and time. Some enhancement is need in traditional COCOMO to calculate exact results.

References

[1] M. Wolak , Taking the Art out of Software Development. An In-Depth Review of Cleanroom software Engineering by Chaelynne.

[2] Linger, R.C., "Cleanroom Process Model," IEEE Software. March 1994, pp. 50–58.

[3] Hevner, A.R. and H.D. Mills, “Box Structure Methods for System Development with Objects,” IBM Systems Journal, vol. 31, no.2, February 1993, pp. 232–251.

[4] Linger, R.M. and H.D. Mills, “A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility,” Proc. COMPSAC ’88, Chicago, October 1988.

[5] Poore, J.H. and H.D. Mills, "Bringing Software Under Statistical Quality Control," Quality Progress, November 1988, pp. 52–55.
[6] Dyer, M., The Cleanroom Approach to Quality Software Development, Wiley, 1992.
[7] Harlan D. Mills, Michael Dyer and Richard C. Linger, "Cleanroom Software Engineering", IEEE software, September 1987.
[8] Robert Oshana and Frank P. Clyle "Implementing cleanroom Software engineering into a mature CMM-based software organization" Proceedings of the 1997 International Conference on Software Engineering, Boston United States, pp: 572-573, May 1997.
[9] Richard C. Linger "Cleanroom Software engineering for zero-defect software",Proceedings of the 15th international conference on software engineering, Baltimore.
[10] Boehm, B.W. (1981). Software Engineering Economics. Prentice Hall.
[11] K. K. Agarwal. "Software Engineering".
[12] R. Pressman , "A practioner approach for software engineering". Fifth Edition.
[13] Mills, H.D., M. Dyer, and R. Linger, "Cleanroom Software Engineering," IEEE Software, vol. 4, no. 5, September 1987, pp. 19–24.
[14] Wohlin, C. and P. Runeson, "Certification of Software Components," IEEE Trans. Software Engineering, vol. SE-20, no. 6, June 1994, pp. 494–499.
[15] Hausler, P.A., R. Linger, and C. Trammel, "Adopting Cleanroom Software Engineering with a Phased Approach," IBM Systems Journal, vol. 33, no.1, January 1994, pp. 89–109.

Hitesh KUMAR SHARMA is an Assistant Professor in University of Petroleum & Energy Studies, Dehradun. He has published 8 research papers in National Journals and 5 research papers in International Journal. Currently He is pursuing his Ph.D. in the area of database tuning**.**