COCOMO based on Clean room Development Methodology

Mahesh Bahadur Singh, Asst.Professor, SIEM, Mathura, 09794489194, maheshcolab@gmail.com
Hradesh Kumar, Asst.Professor, SIEM, Mathura, 09761314696, hradesh.pachauri@gmail.com
Anendra Kumar, Asst.Professor, SIEM, Mathura, 09389364346, anendrasachan@gmail.com

Abstract

The typical software lifecycle is about 40% design, 20% code, and 40% unit testing. The Clean room lifecycle is 80% design and 20% code and no unit test. And effort estimation using the COCOMO (constructive cost model) is based on the typical software lifecycle. In detail COCOMO, Plan/requirement 6% to 8% (Effort); 10% to 40% (Time) Product Design 16% to 18% (Effort); 19% to 38% (Time) Programming 48% to 68% (Effort); 24% to 64% (Time) Integration/Test 16% to 34% (Effort); 24% to 64% (Time) is based on typical software development lifecycle model like Waterfall. When Clean room development methodology is used then these effort will be changed and some cost driver are also changed. We attempt to rectify this by promising a COCOMO-style effort model for Clean room software development methodology, thereby providing new insights into the economic impact of incremental approaches to software projects.

1. Introduction

Harlan Mills and his colleagues from IBM developed the CSE 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. This software methodology follows the same analogy as Clean room 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 start the Clean room 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. Clean room Development Methodology

2.1 Specification

Clean room software engineering is based on a set of formal specifications describing the external behavior of the system. Instead of developing software quickly so we can rush to debug it, we spend more time upfront preventing errors from being put in.

A strict stepwise refinement and verification process is done using a box-structured approach that allows accurate definition of the required user functions and system object architecture. The approach attempts to increase productivity by allowing the product to be incrementally developed. We can say, for example, that 50% of the product is 100% complete instead of 100% being 50% complete. Clean room implements this approach using an object-based technology of box structures called black, state, and clear boxes.

A black box is a specification of external behavior of the function for all possible circumstances. The state box is derived from the black box and represents the external behavior plus the data needed to implement that behavior. The final structure is the clear box, which shows the external behavior, the data needed to implement that behavior, the procedures required to implement that behavior and, finally, any new lower level black boxes that are required to implement the function.

2.2 Correctness Verification

Verification reviews are held by the team to formally or informally verify the software using a set of correctness proofs that are standard in a structured programming environment.

Correctness verification is done before the software is ever executed, so that the developers are not permitted to get into a "debugging" mode of operation.

2.3 Statistical Usage Based Testing and Certification

Clean room uses a formal statistical approach to testing that can indicate the quality of the software and stopping criteria as well. This approach differs from the traditional approach, in which testers assume there are errors in the software and set out to find as many as possible, with the faulty assumption that if enough errors are found and fixed, the software quality would improve. Certification (the Clean room term for testing) is performed to certify the software reliability, not test the software in the classical sense. The methods used are statistical quality control (SQC) and statistical usage testing.

Clean room certification and, specifically, statistical usage based testing, does not measure quality in defects per line of code. Instead, quality is measured in sigma units.

Clean room certification can reduce time to market because you are not spending time testing and fixing when you need not be. When you reach the quality level desired, you can ship, even if complete path coverage is not achieved. Clean room certification can tell you when testing is complete and the product can be released.

2.4 Incremental Development

Using Clean room methodology, software products are developed in a series of functional increments that sum up to the final deliverable product. These increments represent operational user functions. The most stable requirements are
implemented first, with further steps of requirements resolution performed with each successive increment. The integration of these increments is done top down.

Clean room incremental development produces a pipeline of increments that represent end-mend user function, execute in the system environment, and accumulate into the final product. Increments are developed and certified in a top down manner, with successive increments replacing stubs in earlier increments and adding function as the system grows to completion. Incremental development allows intellectual control over complex systems by dividing development into manageable parts within the top down paradigm. Once an initial understanding of the system requirement specifications is achieved, increments are defined based on several criteria, such as size, complexity, component reuse, team skills, etc. Increment quality certification provides objective management feedback to sustain or improve the development process as necessary to achieve quality objectives.

2.5 Box Structured Analysis and Design

Box structures are based on an underlying mathematical foundation that permits scale-up of analysis and design to systems of arbitrary size and complexity. Box structures model systems and system parts in three increasingly detailed and behaviorally equivalent forms:

- The **black box** gives an external description of the behavior of a system or system part in terms of a mathematical function from stimulus histories to responses. The black box is the most abstract description of system behavior and can be considered as a requirements specification for a system or system part.

- The **state box** includes a designed state and an internal black box that transforms the stimulus and an initial state into the response and a new state. The required state is designed from an analysis of stimulus histories and responses for the system or system part.

- The **clear box** replaces the internal black box of the state box with the designed sequential or concurrent usage of other black boxes as subsystems. These new black boxes are in turn expanded at the next level of the system box structure usage hierarchy into state box and clear box forms.

The **box structure usage hierarchy** prescribes a rigorous Mework for top down incremental development. Box structure design guidance is provided by the principles of referential transparency, transaction closure, state migration, and common services.

2.6 Benefits

The goal of Clean room software development is to be able to develop software with zero failures in the field. The process, described in the next section, leads to some benefits that can themselves justify the use of Clean room software development over other methodologies.

The biggest benefit of the Clean room process is that it reduces the amount of errors and failures found during testing. This results in reducing the development cycle time by avoiding the need for rework so prevalent in most software development.

Another benefit that results from the Clean room process is that due to the detailed specifications and models created for the product, the product itself can lead a longer life.
3. **Clean room Development Team**

Clean room software engineering is a theory-based, team-oriented, incremental process for developing and certifying very high quality software systems. Clean room combines formal methods of software development with statistical quality certification to produce systems that are zero defect with high probability.

Four team functions are performed in Clean room software development. These functions may be folded into a single team for small projects, or allocated within a team of teams for large projects:

3.1 **Specification Team** - The Clean room specification team elicits, analyzes, and represents customer requirements. System specifications in formal black boxes are produced along with usage specifications for testing purposes. The Clean room specification includes the incremental development plan for the system.

3.2 **Development Team** - The Clean room development team carries out incremental analysis and design activities to produce a formal system design in box structures. Designs are verified to be correct through mental proofs of correctness in team reviews using the techniques of function-theoretic verification.

3.3 **Certification Team** - The Clean room certification team develops test cases based on the usage specification of the system. Formal statistical testing methods produce scientific measures of software quality.

3.4 **Documentation Team** - Documentation is produced simultaneously with system development and certification. The evolving documentation is validated for quality as corresponding increments are certified.

4. **Formal Specification**

Problem specifications are essential for designing, validating, documenting, communicating, reengineering, and reusing solutions. Formality helps in obtaining high-quality specifications within such processes; it also provides the basis for their automated support. The act of formalization in itself has been widely experienced to raise many questions and detect serious problems in original informal formulations. Besides, the semantics of the formalism being used provides precise rules of interpretation that allow many of the problems with natural language to be overcome. A language with rich structuring facilities may also produce better structured specifications. As the major payoff, formal specifications may be manipulated by automated tools for a wide variety of purposes:

- to derive premises or logical consequences of the specification, for user Confirmation.
- to confirm that an operational specification satisfies more abstract specifications, or to generate behavior.
- to generate concrete scenarios illustrating desired or undesired features about the specification or, conversely, to infer the specification inductively from such scenarios
- to check specific forms of specification consistency/completeness efficiently
- to generate high-level exceptions and conflict preconditions that may make the specification unsatisfiable
• to support formal reuse of components through specification matching. Formal specifications can also be generated from program code as a basis for reverse engineering and software evolution.

4.1 Formal Specification

- The translation of non-mathematical description (diagrams, table, natural language) into a formal specification language
- It represents a concise description of high-level behavior and properties of a system
- Well-defined language semantics support formal deduction about the specification

4.2 Type of Formal Specifications

- Model Oriented: Construct a model of the system behavior using mathematical objects like sets, sequences etc.
  - Statecharts, SCR, VDM, Z
  - Petri Nets, CCS, CSP, Automata theoretical models
- Property Oriented: Use a set of necessary properties to describe system behavior, such as axioms, rules etc.
  - Algebraic semantics
  - Temporal logic models

4.2.1 Formal Proofs

- Proof is an essential part of specification
- Proofs are constructed as a series of small steps, each of which is justified using a small set of rules
- Proofs can be done manually, but usually constructed with some automated assistance

4.2.2 Model Checking

- A technique relies on building a finite model of a system and checking that a desired property holds in that model

- Two general approaches
  - temporal model checking
  - automaton model checking
- Use model checkers
  - SMV

4.2.3 Abstraction

- Representation of the program using a smaller model
- Allows you to focus on the most important central properties and characteristics
- Getting the right level of abstraction is very important in a specification

4.3 Benefits of Formal Specifications

- Higher level of rigor leads to better problem understanding
- Defects are uncovered that would be missed using traditional specification methods
- Allows earlier defect identification
- Formal specification language semantics allow checks for self-consistency
- Enables the use of formal proofs to establish fundamental system properties and invariants

4.4 Limitations to Formal Methods

- Requires a sound mathematical knowledge of the developer
- Different aspects of a design may be represented by different formal specification methods
- Useful for consistency checks, but formal methods cannot guarantee the completeness of a specifications
- For the majority of systems Does not offer significant cost or quality advantages over others

5. COCOMO'81  (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, intermediate, detailed.

5.1 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.

COCOMO applies to three classes of software projects:

- Organic projects - "small" teams with "good" experience working with "less than rigid" requirements
- Semi-detached projects - "medium" teams with mixed experience working with a mix of rigid and less than rigid requirements
- Embedded projects - developed within a set of "tight" constraints (hardware, software, operational, ...)

The basic COCOMO equations take the form

\[
\text{Effort Applied} = a_b(KLOC)^{1.05} \quad \text{man-months}
\]
\[
\text{Development Time} = c_b(\text{Effort Applied})^{0.38} \quad \text{months}
\]
\[
\text{People required} = \frac{\text{Effort Applied}}{\text{Development Time}} \quad \text{count}
\]

The coefficients \(a_b, b, c, d\) are given in the following table.

<table>
<thead>
<tr>
<th>Software project</th>
<th>(a_b)</th>
<th>(b)</th>
<th>(c)</th>
<th>(d)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic</td>
<td>2.4</td>
<td>1.05</td>
<td>2.5</td>
<td>0.38</td>
</tr>
<tr>
<td>Semi-detached</td>
<td>3.0</td>
<td>1.12</td>
<td>2.5</td>
<td>0.35</td>
</tr>
<tr>
<td>Embedded</td>
<td>3.6</td>
<td>1.20</td>
<td>2.5</td>
<td>0.32</td>
</tr>
</tbody>
</table>

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.

5.2 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:

\[ E = a_i (KLoC)^{b_i} \cdot EAF \]

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 \( a_i \) and the exponent \( b_i \) are given in the next table.

<table>
<thead>
<tr>
<th>Software project</th>
<th>( a_i )</th>
<th>( b_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic</td>
<td>3.2</td>
<td>1.05</td>
</tr>
<tr>
<td>Semi-detached</td>
<td>3.0</td>
<td>1.12</td>
</tr>
<tr>
<td>Embedded</td>
<td>2.8</td>
<td>1.20</td>
</tr>
</tbody>
</table>

The Development time \( D \) calculation uses \( E \) in the same way as in the Basic COCOMO.

5.3 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:

1. Phase-sensitive effort multiplier: Some phases (design, programming, integration/test) are more affected than others by factors defined by the cost drivers. The detailed model provides a set of phase sensitive effort multipliers for each cost driver. This helps in determining the manpower allocation for each phase of the project.

2. Three-level product hierarchy: Three product levels are defined. These are module, subsystem and system levels. The rating of the cost drivers are done at appropriate level; that is, the level at which it is most susceptible to variation.

5.3.1. Plan/requirement: The requirement for the project is analyzed, plan is made and a full product specification is made. This phase consumes 6%-8% of the effort and 10%-40% of the development time. The percentage depends not only in the mode (organic, semi-detached, embedded) but also on the size of the project.

5.3.2. Product Design: The second phase deals with the product architecture and the specification of the subsystem. It requires 16%-18% effort and can take up 19%-38% of the development time.

5.3.3. Programming: The next phase is divided into sub phase. The first is the detailed design and second is the code/unit testing. This phase takes up 48%-68% of the effort and 24%-64% of the development time.

5.3.4. Integration/Test: This phase mainly consist of integrating all the tested modules and doing an integrated test. This phase requires 16%-34% of effort and 24%-64% development time.
6. Changes of COCOMO due to Clean room Development Methodology


Formal Method knowledge is must required for the Clean room Development Methodology. Formal Method used in developing computer systems are mathematically ased 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 Clean room development methodology used then these phase will changed.

I 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-Multiplier value for Effort Calculation

<table>
<thead>
<tr>
<th>Cost Driver</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personnel</td>
<td></td>
</tr>
<tr>
<td>Attribut e</td>
<td></td>
</tr>
<tr>
<td>FMKC</td>
<td></td>
</tr>
<tr>
<td>Value Low</td>
<td>Low</td>
</tr>
<tr>
<td>High</td>
<td>Very High</td>
</tr>
<tr>
<td>Extra High</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mode &amp; code size</th>
<th>IRP G</th>
<th>BSSFD</th>
<th>CVCG</th>
<th>STPUT T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic small</td>
<td>0.15</td>
<td>0.65</td>
<td>0.17</td>
<td>0.03</td>
</tr>
<tr>
<td>Organic medium</td>
<td>0.15</td>
<td>0.64</td>
<td>0.17</td>
<td>0.04</td>
</tr>
<tr>
<td>Semidetached medium</td>
<td>0.16</td>
<td>0.64</td>
<td>0.16</td>
<td>0.04</td>
</tr>
<tr>
<td>Semidetached large</td>
<td>0.16</td>
<td>0.63</td>
<td>0.15</td>
<td>0.06</td>
</tr>
<tr>
<td>Embedded large</td>
<td>0.18</td>
<td>0.62</td>
<td>0.14</td>
<td>0.06</td>
</tr>
<tr>
<td>Embedded extra large</td>
<td>0.18</td>
<td>0.61</td>
<td>0.14</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table-Effort and schedule fractions occurring in each phase of the lifecycle

<table>
<thead>
<tr>
<th>Mode &amp; code size</th>
<th>IRP G</th>
<th>BSSFD</th>
<th>CVCG</th>
<th>STPUT T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organic small</td>
<td>0.14</td>
<td>0.66</td>
<td>0.17</td>
<td>0.03</td>
</tr>
<tr>
<td>Organic medium</td>
<td>0.14</td>
<td>0.65</td>
<td>0.17</td>
<td>0.04</td>
</tr>
<tr>
<td>Semidetached medium</td>
<td>0.15</td>
<td>0.65</td>
<td>0.16</td>
<td>0.04</td>
</tr>
<tr>
<td>Semidetached large</td>
<td>0.15</td>
<td>0.64</td>
<td>0.15</td>
<td>0.06</td>
</tr>
<tr>
<td>Embedded large</td>
<td>0.17</td>
<td>0.63</td>
<td>0.14</td>
<td>0.06</td>
</tr>
<tr>
<td>Embedded extra large</td>
<td>0.17</td>
<td>0.62</td>
<td>0.14</td>
<td>0.07</td>
</tr>
</tbody>
</table>
7. Acknowledgements

It is a pleasure to acknowledge the excellent comments of Mr. Hitesh Sharma, Asst.Professor, ITM, Gurgaon Kirtesh Jailia and the references in the preparation of this paper.

References

- Harish Ananthpadmanabhan, Chetan Kale, Mujtaba Khambatti, Ying Jin, Shaun Taufiq Usman, Shu Zhang Arizona State University, Clean room Software Development
- Taking the Art out of Software Development An In-Depth Review of Clean room software Engineering by Chaelynne M. Wolak
- Achieving Software Quality Through Clean room Software Engineering by Richard C. Linger Clean room Software Technology Center IBM Corporation
- Clean room Software Engineering by Bruce R. Maxim, UM-Dearborn
- K. K. Agarwal. “Software Engineering”