Software Maintenance

Software Maintenance

Maintenance means restoring something to it’s original condition. It covers a wide range of activites including correcting code and design errors, updating documentation and test data and upgrading user support.

Many activities classified as maintenance are actually enhancement. Enhancement means adding, modifying or redeveloping the code to support changes in the specification. It is necessary to keep up with changing user needs and operational evolvement.

Note: It stands for all the modification and updation done after delivery of software product

Reasons for modifications / Characteristics

Market Conditions -

Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification. Amazon website launched, them created mobile App

Client Requirements -

Over the time, customer may ask for new features or functions in the software.

Example Started with 5 modules in Amazon added new module of return of material

Host Modifications -
If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability. Example Created in JAVA then switched to python

Organization Changes -
If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise. Example introduced for electric material and mobile phones

Categories / Type of Maintenance

- **Corrective maintenance**

This refer to modifications initiated by defects in the software.

- **Adaptive maintenance**

It includes modifying the software to match changes in the ever changing environment.

- **Perfective maintenance**

It means improving processing efficiency or performance, or restructuring the software to improve changeability. This may include enhancement of existing system functionality, improvement in computational efficiency etc.

- Preventive Maintenance -

  It involves changes made to a system to reduce the chance of future system failure.

  There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflect deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible. This work may be named as preventive maintenance.

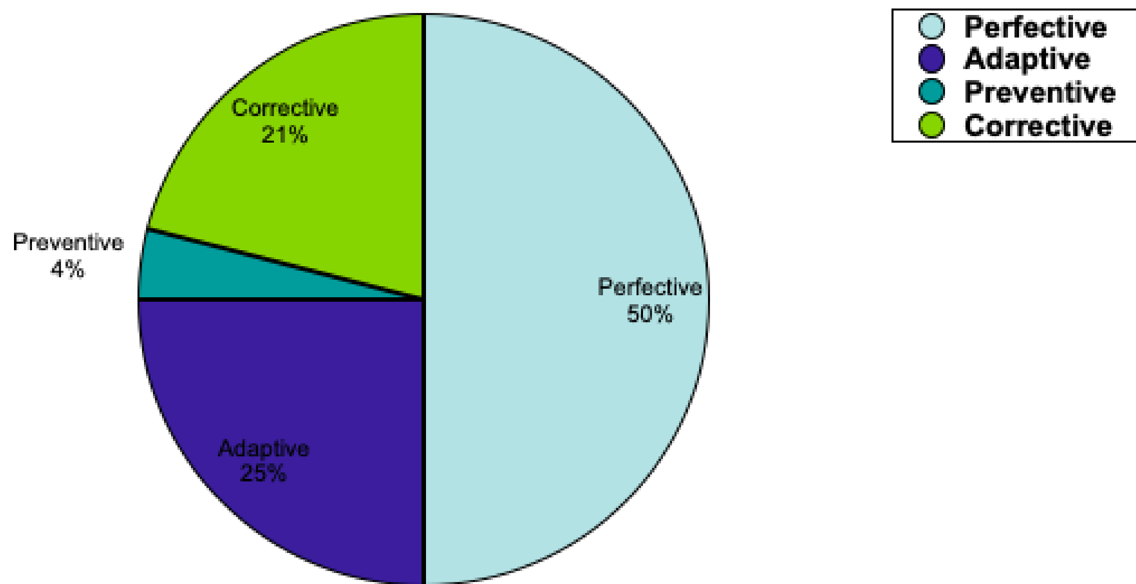


Fig : Distribution of maintenance effort

Problems During Maintenance

- Often the program is written by another person or group of persons.
- Often the program is changed by person who did not understand it clearly.
- Program listings are not structured.
- High staff turnover.
- Information gap.
- Systems are not designed for change.

Maintenance is Manageable

A common misconception about maintenance is that it is not manageable.

Report of survey conducted by Lientz & Swanson gives some interesting observations:

1	Emergency debugging	12.4%
2	Routine debugging	9.3%
3	Data environment adaptation	17.3%
4	Changes in hardware and OS	6.2%
5	Enhancements for users	41.8%
6	Documentation Improvement	5.5%
7	Code efficiency improvement	4.0%
8	Others	3.5%

Kinds of maintenance requests

1	New reports	40.8%
2	Add data in existing reports	27.1%
3	Reformed reports	10%
4	Condense reports	5.6%
5	Consolidate reports	6.4%
6	Others	10.1%

Solutions to Maintenance Problems

- Budget and effort reallocation
- Complete replacement of the system
- Maintenance of existing system

The Maintenance Process

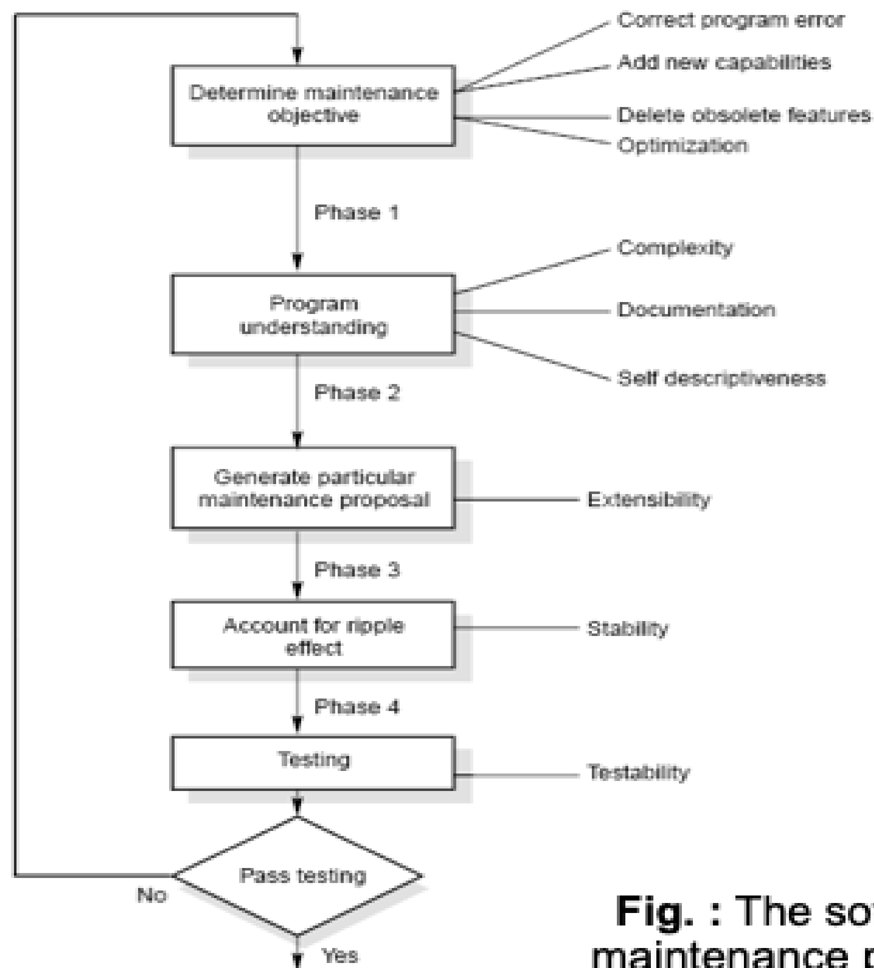


Fig. : The software maintenance process

- Program Understanding

The first phase consists of analyzing the program in order to understand.

- Generating Particular Maintenance Proposal

The second phase consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective.

- Ripple Effect

The third phase consists of accounting for all of the ripple effect as a consequence of program modifications.

- Modified Program Testing

The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before.

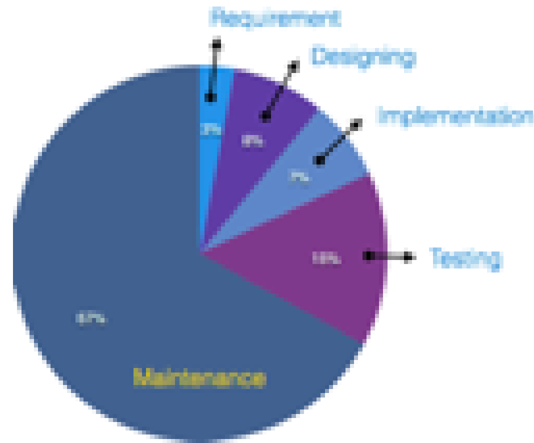
- Maintainability

Each of these four phases and their associated software quality attributes are critical to the maintenance process. All of these factors must be combined to form maintainability.

Cost of Maintenance

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process cycle.

On an average, the cost of software maintenance is more than 50% of all SDLC phases. There are various factors, which trigger maintenance cost go high, such as:



- Real-world factors affecting Maintenance Cost
- The standard age of any software is considered up to 10 to 15 years.
- Older software, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced software on modern hardware.
- As technology advances, it becomes costly to maintain old software.
- Most maintenance engineers are newbie and use trial and error method to rectify problem.
- Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- Changes are often left undocumented which may cause more conflicts in future.

- Estimation:
  - Process of reliably predicting the various parameters associated with making a product

    Example: effort, cost, time, quality

- Estimation:
  - Outcomes of the estimation process

Note: Accuracy and consistency of estimation plays a crucial role.

Traditional Approach to Software Estimating

- Identifying the principals of user functions
- Estimate the size of each Function (code)
- Convert size into code (productivity)
- Obtain productivity metrices
- Team size = Effort / Time
- Ancillary resources from manpower loading
- Multiply by the contigency factors

Software Reverse Engineering

Reverse Engineering

It is a process to achieve system specification by thoroughly analysing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analysing lower abstraction levels.

An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.

Software Re- engineering

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.

Re-Engineering Process

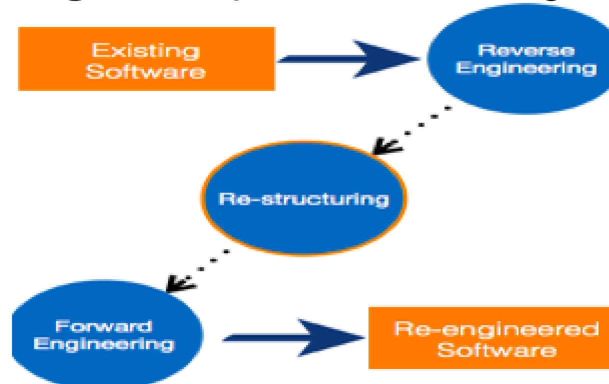
Decide what to re-engineer. Is it whole software or a part of it?

Perform Reverse Engineering, in order to obtain specifications of existing software.

Restructure Program if required. For example, changing function-oriented programs into object-oriented programs.

Re-structure data as required.

Apply Forward engineering concepts in order to get re-engineered software.

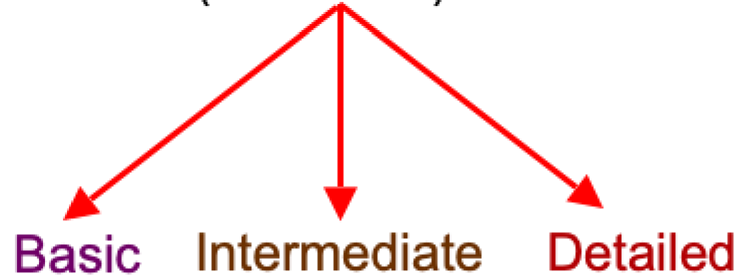


Constructive Cost Models (COCOMO),

- COCOMO Model. Cocomo (Constructive Cost Model) is a **regression model** based on LOC, i.e number of Lines of Code. It is a procedural cost estimate model for software projects and often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time and quality.
- The initial version was published in 1981 and was known as **COCOMO-81.** It was developed considering a waterfall process would be used and that all software will be developed from scratch. The COCOMO model is one of the most popular models cost estimating in software engineering domain.

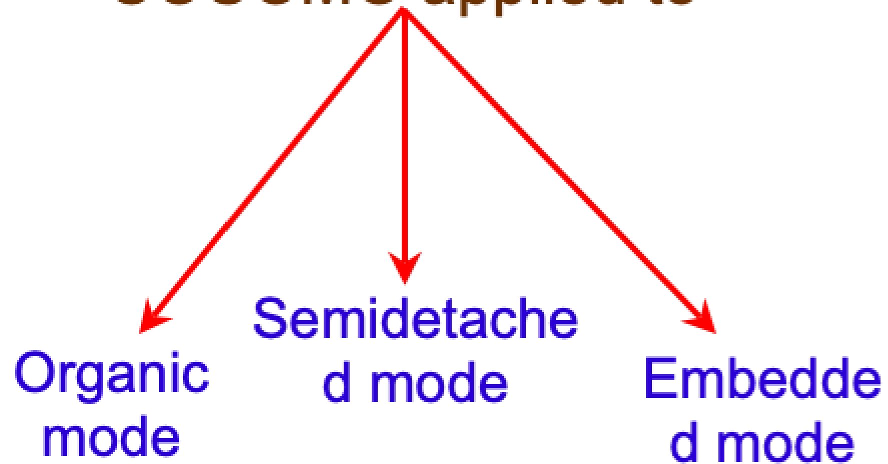
The Constructive Cost Model (COCOMO)

Constructive Cost model
(COCOMO)



Model proposed by

B. W. Boehm’s


COCOMO applied to
Organic mode
Semidetached mode
Embedded mode

Mode	***Project size***	***Nature of Project***	***Innovation***	***Deadline of the project***	***Development Environment***
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	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	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

The comparison of three COCOMO modes

Basic Model

Basic COCOMO model takes the form

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

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

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.

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

Table : Basic COCOMO coefficients

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size}(SS) = \frac{E}{D}\, Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity}(P) = \frac{KLOC}{E}\, KLOC / PM$$

Example:

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

Solution

The basic COCOMO equation take the form:

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

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

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ PM}$$

(ii) Semidetached mode

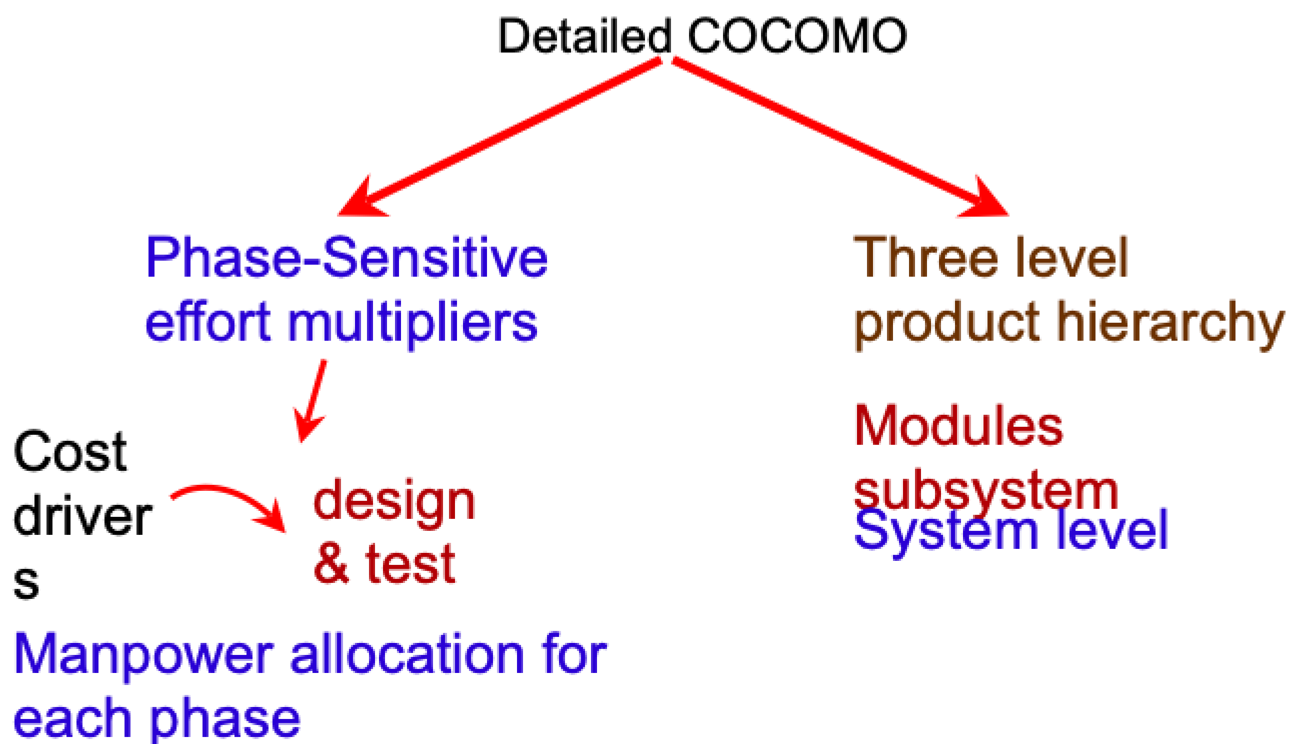
$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ PM}$$


Detailed COCOMO Model
Detailed COCOMO
Phase-Sensitive
effort multipliers
Three level
product hierarchy
Cost
driver
s
design
& test
Modules
subsystem
System level
Manpower allocation for
each phase

Development Phase

Plan / Requirements

EFFORT	:	6% to 8%
DEVELOPMENT TIME	:	10% to 40%

% depend on mode & size

Design

Effort	:	16% to 18%
Time	:	19% to 38%

Programming

Effort	:	48% to 68%
Time	:	24% to 64%

Integration & Test

Effort	:	16% to 34%
Time	:	18% to 34%

Principle of the effort estimate

Size equivalent

As the software might be partly developed from software already existing (that is, re-usable code), a full development is not always required. In such cases, the parts of design document (DD%), code (C%) and integration (I%) to be modified are estimated. Then, an adjustment factor, A, is calculated by means of the following equation.

$$A = 0.4\ DD + 0.3\ C + 0.3\ I$$

The size equivalent is obtained by

$$S\ (\text{equivalent}) = (S \times A) / 100$$

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Lifecycle Phase Values of μ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.06	0.16	0.26	0.42	0.16
Organic medium S≈32	0.06	0.16	0.24	0.38	0.22
Semidetached medium S≈32	0.07	0.17	0.25	0.33	0.25
Semidetached large S≈128	0.07	0.17	0.24	0.31	0.28
Embedded large S≈128	0.08	0.18	0.25	0.26	0.31
Embedded extra large S≈320	0.08	0.18	0.24	0.24	0.34

Lifecycle Phase Values of τ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.10	0.19	0.24	0.39	0.18
Organic medium S≈32	0.12	0.19	0.21	0.34	0.26
Semidetached medium S≈32	0.20	0.26	0.21	0.27	0.26
Semidetached large S≈128	0.22	0.27	0.19	0.25	0.29
Embedded large S≈128	0.36	0.36	0.18	0.18	0.28
Embedded extra large S≈320	0.40	0.38	0.16	0.16	0.30

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

Distribution of software life cycle:

1. Requirement and product design

   (a) Plans and requirements

   (b)System design

2. Detailed Design

   (a) Detailed design

3. Code & Unit test

   (a) Module code & test

4. Integrate and Test

   (a) Integrate & Test

COCOMO-II

The following categories of applications / projects are identified by COCOMO-II and are shown in fig. 4 shown below:

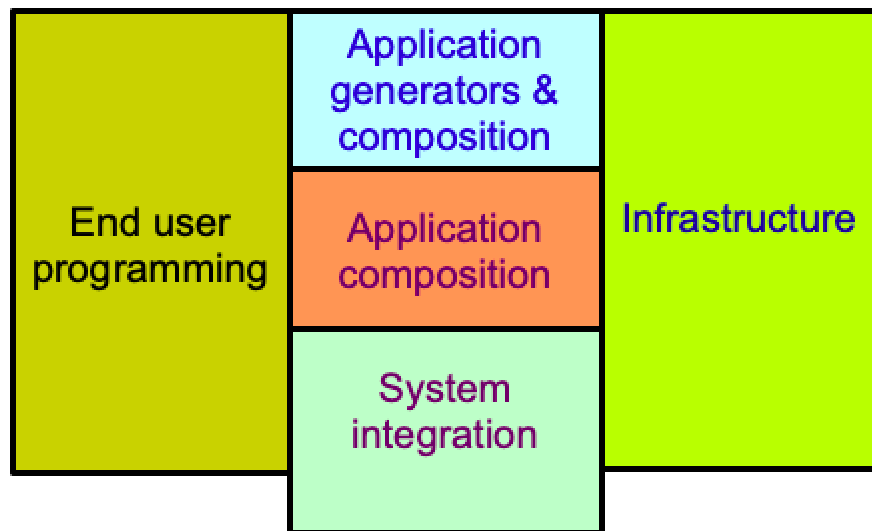


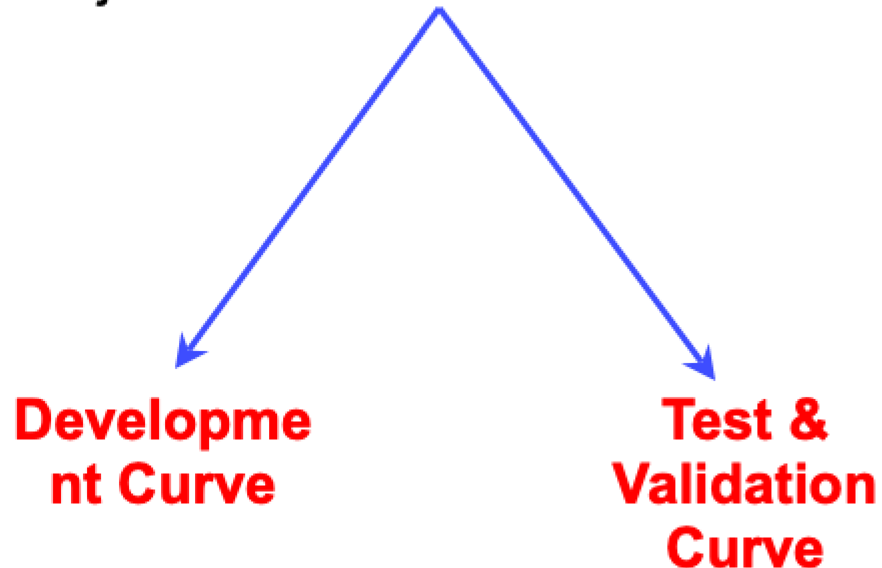
Fig. : Categories of applications / projects

Stage No	***Model Name***	***Application for the types of projects***	***Applications***
Stage I	Application composition estimation model	Application composition	In addition to application composition type of projects, this model is also used for prototyping (if any) stage of application generators, infrastructure & system integration.
Stage II	Early design estimation model	Application generators, infrastructure & system integration	Used in early design stage of a project, when less is known about the project.
Stage III	Post architecture estimation model	Application generators, infrastructure & system integration	Used after the completion of the detailed architecture of the project.

Table : Stages of COCOMO-II

Project life cycle

Project curve is the addition of two curves



Development Subcycle

All that has been discussed so far is related to project life cycle as represented by project curve

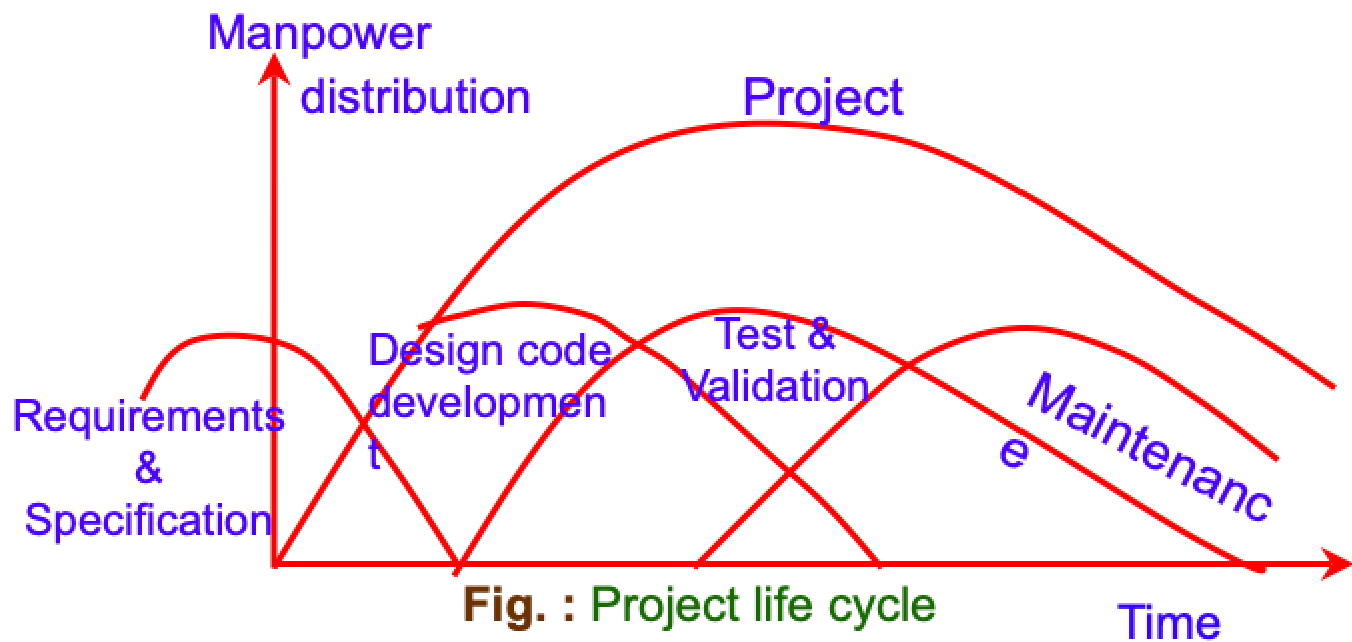


Fig. : Project life cycle

Note :

1. The COCOMO 1 model is a regression-based model that considers various historical programs software size and multipliers.The most important fundamental calculation is the use of effort equation to find the number of Person-Months required in developing a project.

1. The COCOMO 2 model in Software Engineering is tuned to modern software life cycles. COCOMO 1 model has been very successful. However, it doesn’t apply to newer software development practices as well as it does to traditional practices.This model targets modern software projects and will continue to evolve over the next few years.