Software Maintenance

Software maintenance is a part of the Software Development Life Cycle. Its primary goal is to modify and update software application after delivery to correct errors and to improve performance. Software is a model of the real world. When the real world changes, the software require alteration wherever possible.

Software Maintenance is an inclusive activity that includes error corrections, enhancement of capabilities, deletion of obsolete capabilities, and optimization.

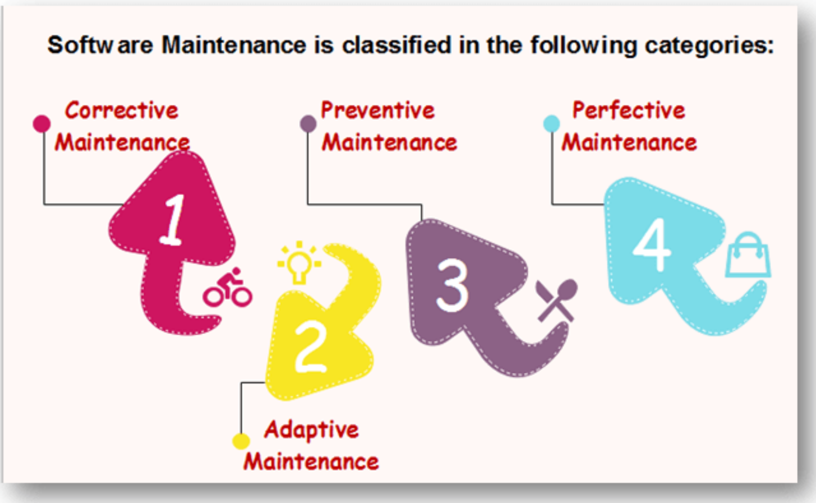
Need for Maintenance

Software Maintenance is needed for:-

- Correct errors
- Change in user requirement with time
- Changing hardware/software requirements
- To improve system efficiency
- To optimize the code to run faster
- To modify the components
- To reduce any unwanted side effects.

Thus the maintenance is required to ensure that the system continues to satisfy user requirements.

Types of Software Maintenance



1. Corrective Maintenance

Corrective maintenance aims to correct any remaining errors regardless of where they may cause specifications, design, coding, testing, and documentation, etc.

2. Adaptive Maintenance

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

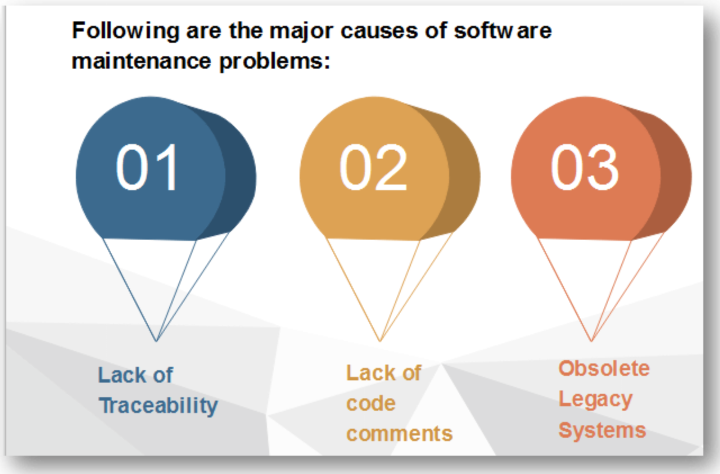
3. Preventive Maintenance

It is the process by which we prevent our system from being obsolete. It involves the concept of reengineering & reverse engineering in which an old system with old technology is re-engineered using new technology. This maintenance prevents the system from dying out.

4. Perfective Maintenance

It defines improving processing efficiency or performance or restricting the software to enhance changeability. This may contain enhancement of existing system functionality, improvement in computational efficiency, etc.

Causes of Software Maintenance Problems



Lack of Traceability

- Codes are rarely traceable to the requirements and design specifications.
- It makes it very difficult for a programmer to detect and correct a critical defect affecting customer operations.
- Like a detective, the programmer pores over the program looking for clues.
- Life Cycle documents are not always produced even as part of a development project.

Lack of Code Comments

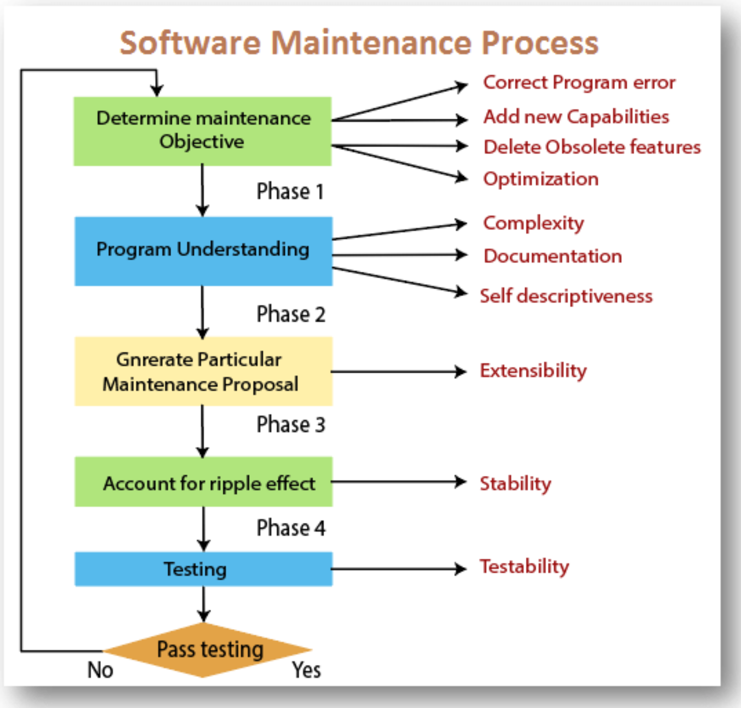
- Most of the software system codes lack adequate comments. Lesser comments may not be helpful in certain situations.

Obsolete Legacy Systems

- In most of the countries worldwide, the legacy system that provides the backbone of the nation's critical industries, e.g., telecommunications, medical, transportation utility services, were not designed with maintenance in mind.

- They were not expected to last for a quarter of a century or more!
- As a consequence, the code supporting these systems is devoid of traceability to the requirements, compliance to design and programming standards and often includes dead, extra and uncommented code, which all make the maintenance task next to the impossible.

Software Maintenance Process



Program Understanding

The first step consists of analyzing the program to understand.

Generating a Particular maintenance problem

The second phase consists of creating a particular maintenance proposal to accomplish the implementation of the maintenance goals.

Ripple Effect

The third step consists of accounting for all of the ripple effects as a consequence of program modifications.

Modified Program Testing

The fourth step consists of testing the modified program to ensure that the revised application has at least the same reliability level as prior.

Maintainability

Each of these four steps and their associated software quality attributes is critical to the maintenance process. All of these methods must be combined to form maintainability.

Reverse Engineering –

Reverse Engineering is processes of extracting knowledge or design information from anything man-made and reproducing it based on extracted information. It is also called back Engineering.

Software Reverse Engineering –

Software Reverse Engineering is the process of recovering the design and the requirements specification of a product from an analysis of it's code. Reverse Engineering is becoming important, since several existing software products, lack proper documentation, are highly unstructured, or their structure has degraded through a series of maintenance efforts.

Why Reverse Engineering?

- Providing proper system documentatiuon.
- Recovery of lost information.
- Assisting with maintenance.
- Facility of software reuse.
- Discovering unexpected flaws or faults.

Used of Software Reverse Engineering –

- Software Reverse Engineering is used in software design, reverse engineering enables the developer or programmer to add new features to the existing software with or without knowing the source code.
- Reverse engineering is also useful in software testing, it helps the testers to study the virus and other malware code .

Software Configuration Management

When we develop software, the product (software) undergoes many changes in their maintenance phase; we need to handle these changes effectively.

Several individuals (programs) works together to achieve these common goals. This individual produces several work product (SC Items) e.g., Intermediate version of modules or test data used during debugging, parts of the final product.

The elements that comprise all information produced as a part of the software process are collectively called a software configuration.

As software development progresses, the number of Software Configuration elements (SCI's) grow rapidly.

These are handled and controlled by SCM. This is where we require software configuration management.

A configuration of the product refers not only to the product's constituent but also to a particular version of the component.

Therefore, SCM is the discipline which

- Identify change
- Monitor and control change
- Ensure the proper implementation of change made to the item.
- Auditing and reporting on the change made.

Configuration Management (CM) is a technic of identifying, organizing, and controlling modification to software being built by a programming team.

The objective is to maximize productivity by minimizing mistakes (errors).

CM is used to essential due to the inventory management, library management, and updation management of the items essential for the project.

Why do we need Configuration Management?

Multiple people are working on software which is consistently updating. It may be a method where multiple version, branches, authors are involved in a software project, and the team is geographically distributed and works concurrently. It changes in user requirements, and policy, budget, schedules need to be accommodated.

Importance of SCM

It is practical in controlling and managing the access to various SCIs e.g., by preventing the two members of a team for checking out the same component for modification at the same time.

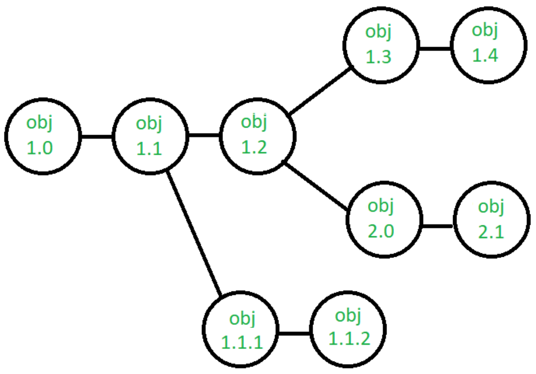
It provides the tool to ensure that changes are being properly implemented.

It has the capability of describing and storing the various constituent of software.

SCM is used in keeping a system in a consistent state by automatically producing derived version upon modification of the same component.

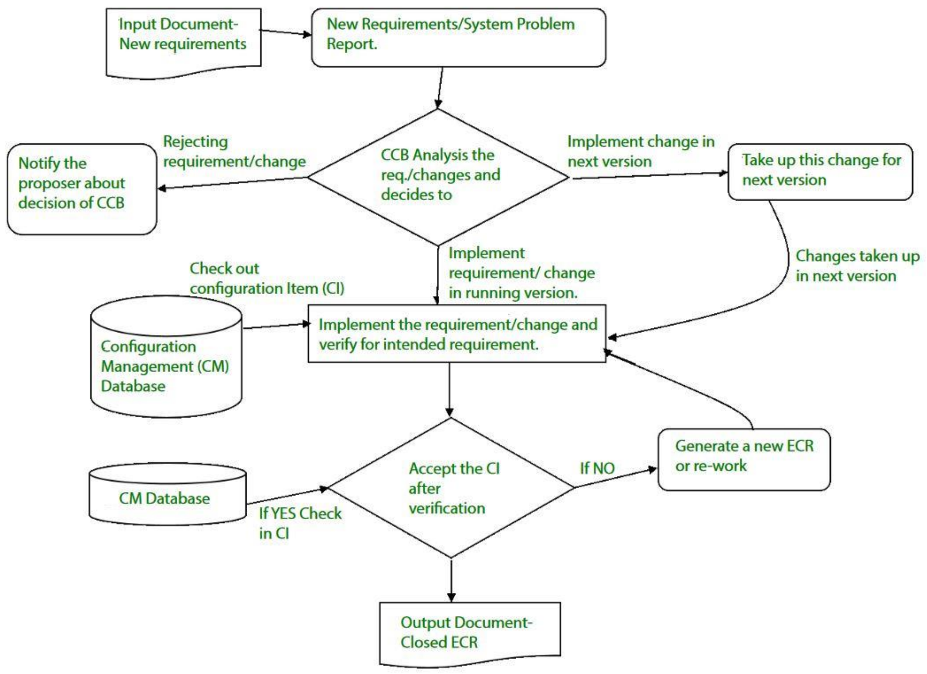
Processes involved in SCM –
Configuration management provides a disciplined environment for smooth control of work products. It involves the following activities:

1. **Identification and Establishment** – Identifying the configuration items from products that compose baselines at given points in time (a baseline is a set of mutually consistent Configuration Items, which has been formally reviewed and agreed upon, and serves as the basis of further development). Establishing relationship among items, creating a mechanism to manage multiple level of control and procedure for change management system.
2. **Version control** – Creating versions/specifications of the existing product to build new products from the help of SCM system. A description of version is given below:



Suppose after some changes, the version of configuration object changes from 1.0 to 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2, which is followed by a major update that is object 1.2. The development of object 1.0 continues through 1.3 and 1.4, but finally, a noteworthy change to the object results in a new evolutionary path, version 2.0. Both versions are currently supported.

3. **Change control** – Controlling changes to Configuration items (CI). The change control process is explained in Figure below:



Input Document- New requirements
New Requirements/System Problem Report.
Rejecting requirement/change
CCB Analysis the req./changes and decides to
Implement change in next version
Notify the proposer about decision of CCB
Take up this change for next version
Check out configuration Item (CI)
Implement requirement/ change in running version.
Changes taken up in next version
Configuration Management (CM) Database
Implement the requirement/change and verify for intended requirement.
Generate a new ECR or re-work
Accept the CI after verification
If NO
CM Database
If YES Check in CI
Output Document- Closed ECR

A change request (CR) is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a change report, which is used by a change control board (CCB) —a person or group who makes a final decision on the status and priority of the change. An engineering change Request (ECR) is generated for each approved change.

Also CCB notifies the developer in case the change is rejected with proper reason. The ECR describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is “checked out” of the project database, the change is made, and then the object is tested again. The object is then “checked in” to the database and appropriate version control mechanisms are used to create the next version of the software.

4. **Configuration auditing** – A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified. The audit confirms the completeness, correctness and consistency of items in the SCM system and track action items from the audit to closure.
5. **Reporting** – Providing accurate status and current configuration data to developers, tester, end users, customers and stakeholders through admin guides, user guides, FAQs, Release notes, Memos, Installation Guide, Configuration guide etc .

   SCM Tools –
   Different tools are available in market for SCM like: CFEngine, Bcfg2 server, Vagrant, SmartFrog, CLEAR CASETOOL (CC), SaltStack, CLEAR QUEST TOOL, Puppet, SVN- Subversion, Perforce, TortoiseSVN, IBM Rational team

concert, IBM Configuration management version management, Razor, Ansible, etc. There are many more in the list.
It is recommended that before selecting any configuration management tool, have a proper understanding of the features and select the tool which best suits your project needs and be clear with the benefits and drawbacks of each before you choose one to use.

Software Cost Estimation

For any new software project, it is necessary to know how much it will cost to develop and how much development time will it take. These estimates are needed before development is initiated, but how is this done? Several estimation procedures have been developed and are having the following attributes in common.

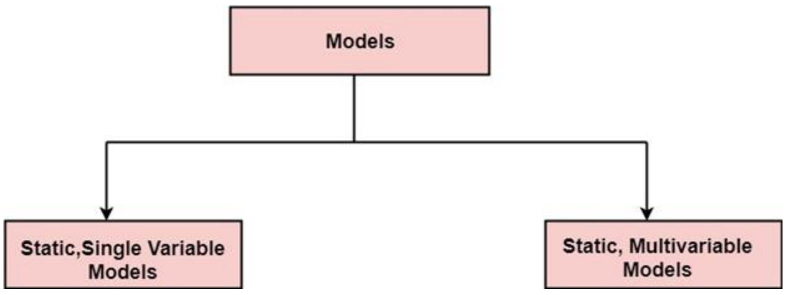
1. Project scope must be established in advanced.
2. Software metrics are used as a support from which evaluation is made.
3. The project is broken into small PCs which are estimated individually. To achieve true cost & schedule estimate, several option arise.
4. Delay estimation
5. Used symbol decomposition techniques to generate project cost and schedule estimates.
6. Acquire one or more automated estimation tools.

Uses of Cost Estimation

1. During the planning stage, one needs to choose how many engineers are required for the project and to develop a schedule.
2. In monitoring the project's progress, one needs to access whether the project is progressing according to the procedure and takes corrective action, if necessary.

Cost Estimation Models

A model may be static or dynamic. In a static model, a single variable is taken as a key element for calculating cost and time. In a dynamic model, all variable are interdependent, and there is no basic variable.



Static, Single Variable Models: When a model makes use of single variables to calculate desired values such as cost, time, efforts, etc. is said to be a single variable model. The most common equation is:

$$\mathbf{C=aL^b}$$

Where C = Costs
L= size
a and b are constants

The Software Engineering Laboratory established a model called SEL model, for estimating its software production. This model is an example of the static, single variable model.

$E=1.4L^{0.93}$
$DOC=30.4L^{0.90}$
$D=4.6L^{0.26}$

Where E= Efforts (Person Per Month)
DOC=Documentation (Number of Pages)
D = Duration (D, in months)
L = Number of Lines per code

Static, Multivariable Models: These models are based on method (1), they depend on several variables describing various aspects of the software development environment. In some model, several variables are needed to describe the software development process, and selected equation combined these variables to give the estimate of time & cost. These models are called multivariable models.

WALSTON and FELIX develop the models at IBM provide the following equation gives a relationship between lines of source code and effort:

$$\mathbf{E=5.2L^{0.91}}$$

In the same manner duration of development is given by

$$\mathbf{D=4.1L^{0.36}}$$

The productivity index uses 29 variables which are found to be highly correlated productivity as follows:

$$\mathbf{I = \sum_{i=1}^{29} W_i X_i}$$

Where $\mathbf{W_i}$ is the weight factor for the **ith**variable and $\mathbf{X_i=\{-1,0,+1\}}$ the estimator gives $\mathbf{X_i}$one of the values **-1, 0 or +1** depending on the variable decreases, has no effect or increases the productivity.

Example: Compare the Walston-Felix Model with the SEL model on a software development expected to involve 8 person-years of effort.

a. Calculate the number of lines of source code that can be produced.
b. Calculate the duration of the development.
c. Calculate the productivity in LOC/PY
d. Calculate the average manning

Solution:

The amount of manpower involved = 8PY=96persons-months

(a)Number of lines of source code can be obtained by reversing equation to give:

$$L = \left(\frac{E}{a}\right) 1/b$$

Then

L (SEL) = (96/1.4)1⁄0.93=94264 LOC
L (SEL) = (96/5.2)1⁄0.91=24632 LOC

(b)Duration in months can be calculated by means of equation

D (SEL) = 4.6 (L) 0.26
= 4.6 (94.264)0.26 = 15 months
D (W-F) = 4.1 $L^{0.36}$
= 4.1 (24.632)0.36 = 13 months

(c) Productivity is the lines of code produced per persons/month (year)

$$P\ (SEL) = \frac{94264}{8} = 11783\ \frac{LOC}{Person}\ -Years$$

$$P\ (Years) = \frac{24632}{8} = 3079\ \frac{LOC}{Person}\ -Years$$

(d)Average manning is the average number of persons required per month in the project

$$M\ (SEL) = \frac{96P-M}{15M} = 6.4Persons$$

$$M\ (W\text{-}F) = \frac{96P-M}{13M} = 7.4Persons$$

COCOMO Model

Boehm proposed COCOMO (Constructive Cost Estimation Model) in 1981.COCOMO is one of the most generally used software estimation models in the world. COCOMO predicts the efforts and schedule of a software product based on the size of the software.

The necessary steps in this model are:

1. Get an initial estimate of the development effort from evaluation of thousands of delivered lines of source code (KDLOC).
2. Determine a set of 15 multiplying factors from various attributes of the project.
3. Calculate the effort estimate by multiplying the initial estimate with all the multiplying factors i.e., multiply the values in step1 and step2.

The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single variable models, using KDLOC as the measure of the size. To determine the initial effort E_i in person-months the equation used is of the type is shown below

E_i=a*(KDLOC)b

The value of the constant a and b are depends on the project type.

In COCOMO, projects are categorized into three types:

1. Organic
2. Semidetached
3. Embedded

1.Organic: A development project can be treated of the organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar methods of projects. **Examples of this type of projects are simple business systems, simple inventory management systems, and data processing systems.**

2. Semidetached: A development project can be treated with semidetached type if the development consists of a mixture of experienced and inexperienced staff. Team members may have finite experience in related systems but may be unfamiliar with some aspects of the order being developed. **Example of Semidetached system includes developing a new operating system (OS), a Database Management System (DBMS), and complex inventory management system.**

3. Embedded: A development project is treated to be of an embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational method exist. **For Example:** ATM, Air Traffic control.

For three product categories, Bohem provides a different set of expression to predict effort (in a unit of person month)and development time from the size of estimation in KLOC(Kilo Line of code) efforts estimation takes into account the productivity loss due to holidays, weekly off, coffee breaks, etc.

According to Boehm, software cost estimation should be done through three stages:

1. Basic Model
2. Intermediate Model
3. Detailed Model

1. Basic COCOMO Model: The basic COCOMO model provide an accurate size of the project parameters. The following expressions give the basic COCOMO estimation model:

Effort=a_1*(KLOC) a_2 PM
Tdev=b_1*(efforts)b_2 Months

Where

KLOC is the estimated size of the software product indicate in Kilo Lines of Code,

a_1,a_2,b_1,b_2 are constants for each group of software products,

Tdev is the estimated time to develop the software, expressed in months,

Effort is the total effort required to develop the software product, expressed in **person months (PMs)**.

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic: Effort = 2.4(KLOC) 1.05 PM

Semi-detached: Effort = 3.0(KLOC) 1.12 PM

Embedded: Effort = 3.6(KLOC) 1.20 PM

Estimation of development time

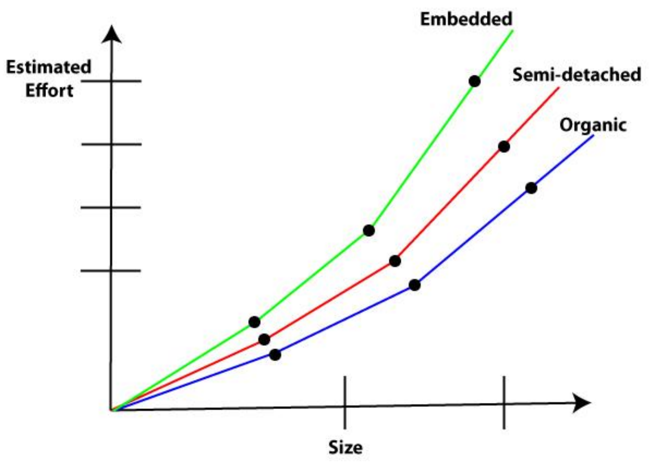
For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic: Tdev = 2.5(Effort) 0.38 Months

Semi-detached: Tdev = 2.5(Effort) 0.35 Months

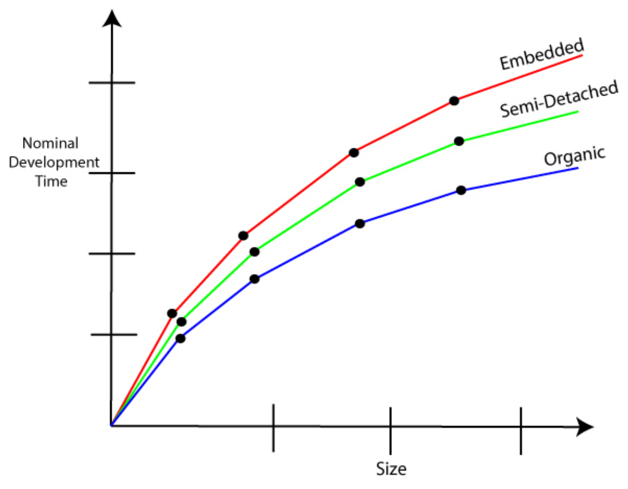
Embedded: Tdev = 2.5(Effort) 0.32 Months

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig shows a plot of estimated effort versus product size. From fig, we can observe that the effort is somewhat superliner in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.



Effort versus product size

The development time versus the product size in KLOC is plotted in fig. From fig it can be observed that the development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from fig, it can be observed that the development time is roughly the same for all three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.



Development time versus size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the COCOMO model are called a nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

Example1: Suppose a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three model i.e., organic, semi-detached & embedded.

Solution: The basic COCOMO equation takes the form:

Effort=a_1*(KLOC) a_2 PM
Tdev=b_1*(efforts)b_2 Months
Estimated Size of project= 400 KLOC

(i)Organic Mode

E = 2.4 * (400)1.05 = 1295.31 PM
D = 2.5 * (1295.31)0.38=38.07 PM

(ii)Semidetached Mode

E = 3.0 * (400)1.12=2462.79 PM
D = 2.5 * (2462.79)0.35=38.45 PM

(iii) Embedded Mode

E = 3.6 * (400)1.20 = 4772.81 PM
D = 2.5 * (4772.8)0.32 = 38 PM

Example2: A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the Effort, development time, average staff size, and productivity of the project.

Solution: The semidetached mode is the most appropriate mode, keeping in view the size, schedule and experience of development time.

Hence E=3.0(200)1.12=1133.12PM
D=2.5(1133.12)0.35=29.3PM

$$\text{Average Staff Size (SS)} = \frac{E}{D}\ \text{Persons}$$

$$= \frac{1133.12}{29.3} = 38.67\ \text{Persons}$$

$$\text{Productivity} = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765\ KLOC/PM$$

P = 176 LOC/PM

2. Intermediate Model: The basic Cocomo model considers that the effort is only a function of the number of lines of code and some constants calculated according to the various software systems. The intermediate COCOMO model recognizes these facts and refines the initial estimates obtained through the basic COCOMO model by using a set of 15 cost drivers based on various attributes of software engineering.

Classification of Cost Drivers and their attributes:

(i) Product attributes -

- Required software reliability extent
- Size of the application database
- The complexity of the product

Hardware attributes -

- Run-time performance constraints
- Memory constraints
- The volatility of the virtual machine environment
- Required turnabout time

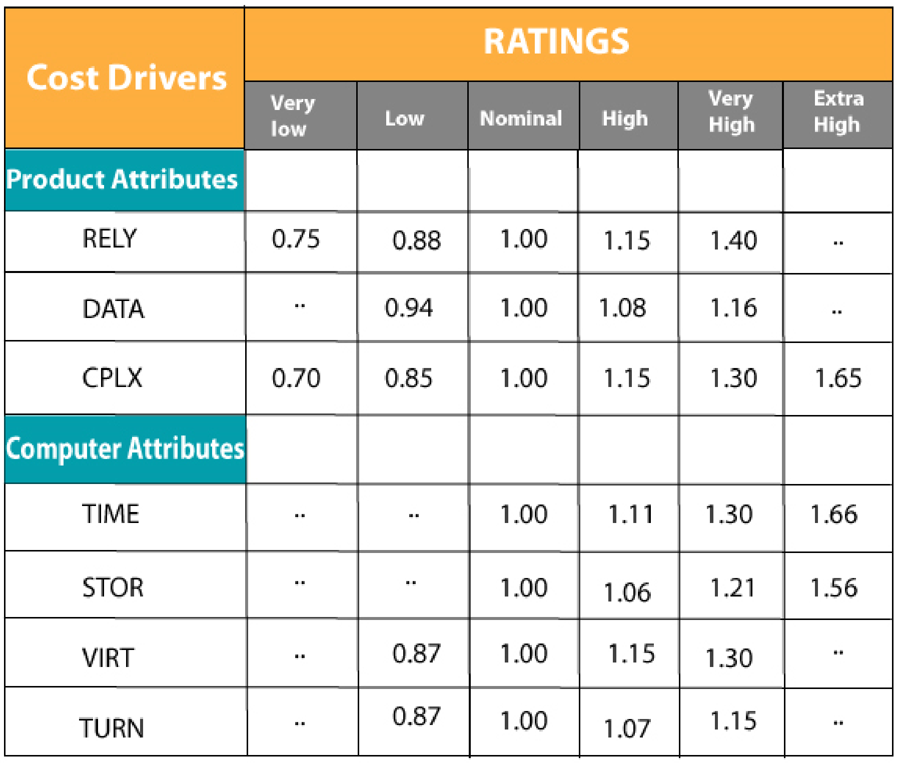
Personnel attributes -

- Analyst capability
- Software engineering capability
- Applications experience
- Virtual machine experience
- Programming language experience

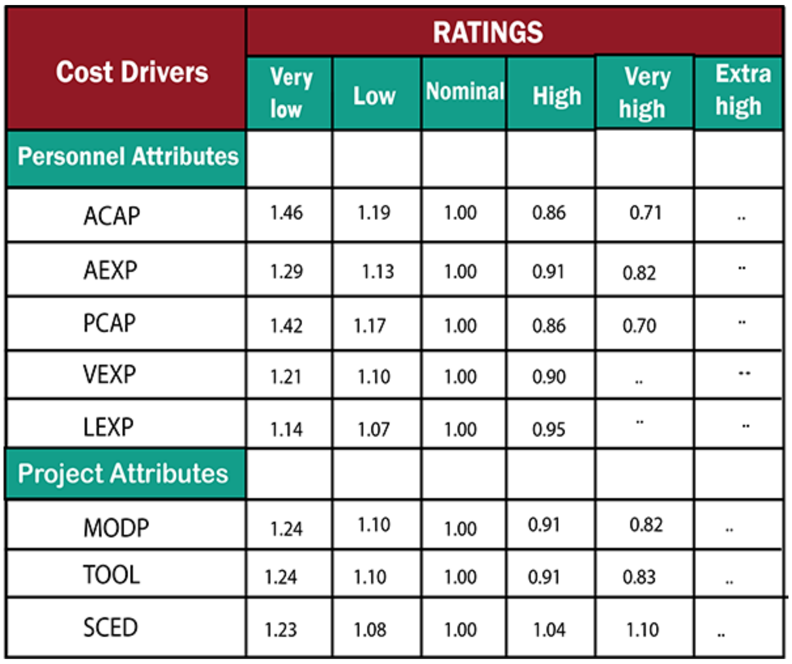
Project attributes -

- Use of software tools
- Application of software engineering methods
- Required development schedule

The cost drivers are divided into four categories:



Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	..
DATA	..	0.94	1.00	1.08	1.16	..
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	..	..	1.00	1.11	1.30	1.66
STOR	..	..	1.00	1.06	1.21	1.56
VIRT	..	0.87	1.00	1.15	1.30	..
TURN	..	0.87	1.00	1.07	1.15	..



Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	..
AEXP	1.29	1.13	1.00	0.91	0.82	..
PCAP	1.42	1.17	1.00	0.86	0.70	..
VEXP	1.21	1.10	1.00	0.90	..	..
LEXP	1.14	1.07	1.00	0.95	..	..
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	..
TOOL	1.24	1.10	1.00	0.91	0.83	..
SCED	1.23	1.08	1.00	1.04	1.10	..

Intermediate COCOMO equation:

$$E=a_i\ (KLOC)\ b_i*EAF$$
$$D=c_i\ (E)d_i$$

Coefficients for intermediate COCOMO

Project	**a_i**	**b_i**	**c_i**	**d_i**
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

3. Detailed COCOMO Model: Detailed COCOMO incorporates all qualities of the standard version with an assessment of the cost driver?s effect on each method of the software engineering process. The detailed model uses various effort multipliers for each cost driver property. In detailed cocomo, the whole software is differentiated into multiple modules, and then we apply COCOMO in various modules to estimate effort and then sum the effort.

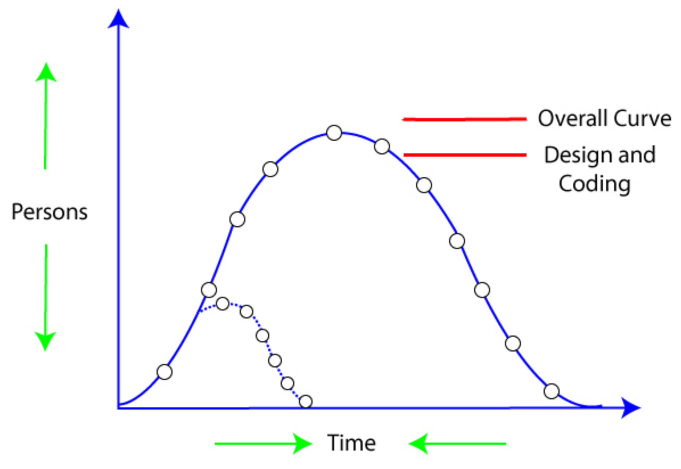
The Six phases of detailed COCOMO are:

1. Planning and requirements
2. System structure
3. Complete structure
4. Module code and test
5. Integration and test
6. Cost Constructive model

The effort is determined as a function of program estimate, and a set of cost drivers are given according to every phase of the software lifecycle.

Putnam Resource Allocation Model

The Lawrence Putnam model describes the time and effort requires finishing a software project of a specified size. Putnam makes a use of a so-called The Norden/Rayleigh Curve to estimate project effort, schedule & defect rate as shown in fig:



The Rayleigh manpower loading Curve

Putnam noticed that software staffing profiles followed the well known Rayleigh distribution. Putnam used his observation about productivity levels to derive the software equation:

$$L = C_k K^{1/3} t_d^{4/3}$$

The various terms of this expression are as follows:

K is the total effort expended (in PM) in product development, and L is the product estimate in **KLOC** .

t_d correlate to the time of system and integration testing. Therefore, **t_d** can be relatively considered as the time required for developing the product.

C_k Is the state of technology constant and reflects requirements that impede the development of the program.

Typical values of **C_k** = 2 for poor development environment

C_k= 8 for good software development environment

C_k = 11 for an excellent environment (in addition to following software engineering principles, automated tools and techniques are used).

The exact value of **C_k** for a specific task can be computed from the historical data of the organization developing it.

Putnam proposed that optimal staff develop on a project should follow the Rayleigh curve. Only a small number of engineers are required at the beginning of a plan to carry out planning and specification tasks. As the project progresses and more detailed work are necessary, the number of engineers reaches a peak. After implementation and unit testing, the number of project staff falls.

Effect of a Schedule change on Cost

Putnam derived the following expression:

$$L = C_k K^{1/3} t_d^{4/3}$$

Where, **K** is the total effort expended (in PM) in the product development

L is the product size in KLOC

t_d corresponds to the time of system and integration testing

C_k Is the state of technology constant and reflects constraints that impede the progress of the program

Now by using the above expression, it is obtained that,

$$K = L^3 / C_k^3 t_d^4$$

Or $$K = C / t_d^4$$

For the same product size, $C = L^3 / C_k^3$ is a constant.

Or $\frac{K_1}{K_2} = t_{d2}^4 / t_{d1}^4$

Or $K \propto 1/t_d^4$

Or, $\text{cost} \propto 1/t_d$

(As project development effort is equally proportional to project development cost)

From the above expression, it can be easily observed that when the schedule of a project is compressed, the required development effort as well as project development cost increases in proportion to the fourth power of the degree of compression. It means that a relatively small compression in delivery schedule can result in a substantial penalty of human effort as well as development cost.

For example, if the estimated development time is 1 year, then to develop the product in 6 months, the total effort required to develop the product (and hence the project cost) increases 16 times.

What is Risk?

"Tomorrow problems are today's risk." Hence, a clear definition of a "risk" is a problem that could cause some loss or threaten the progress of the project, but which has not happened yet.

These potential issues might harm cost, schedule or technical success of the project and the quality of our software device, or project team morale.

Risk Management is the system of identifying addressing and eliminating these problems before they can damage the project.

We need to differentiate risks, as potential issues, from the current problems of the project.

Different methods are required to address these two kinds of issues.

For example, staff storage, because we have not been able to select people with the right technical skills is a current problem, but the threat of our technical persons being hired away by the competition is a risk.

Risk Management

A software project can be concerned with a large variety of risks. In order to be adept to systematically identify the significant risks which might affect a software project, it is essential to classify risks into different classes. The project manager can then check which risks from each class are relevant to the project.

There are three main classifications of risks which can affect a software project:

1. Project risks
2. Technical risks
3. Business risks

1. Project risks: Project risks concern differ forms of budgetary, schedule, personnel, resource, and customer-related problems. A vital project risk is schedule slippage. Since the software is intangible, it is very tough to monitor and control a software project. It is very tough to control something which cannot be identified. For any manufacturing program, such as the manufacturing of cars, the plan executive can recognize the product taking shape.

2. Technical risks: Technical risks concern potential method, implementation, interfacing, testing, and maintenance issue. It also consists of an ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks appear due to the development team's insufficient knowledge about the project.

3. Business risks: This type of risks contain risks of building an excellent product that no one need, losing budgetary or personnel commitments, etc.

Other risk categories

1. **1. Known risks:** Those risks that can be uncovered after careful assessment of the project program, the business and technical environment in which the plan is being developed, and more reliable data sources (e.g., unrealistic delivery date)
2. **2. Predictable risks:** Those risks that are hypothesized from previous project experience (e.g., past turnover)
3. **3. Unpredictable risks:** Those risks that can and do occur, but are extremely tough to identify in advance.

Principle of Risk Management

1. **Global Perspective:** In this, we review the bigger system description, design, and implementation. We look at the chance and the impact the risk is going to have.
2. **Take a forward-looking view:** Consider the threat which may appear in the future and create future plans for directing the next events.
3. **Open Communication:** This is to allow the free flow of communications between the client and the team members so that they have certainty about the risks.
4. **Integrated management:** In this method risk management is made an integral part of project management.
5. **Continuous process:** In this phase, the risks are tracked continuously throughout the risk management paradigm.