COCOMO II: A Parametric Productivity Model

C**O**nstructive **CO**st **MO**del II (COCOMO II) is a model that allows one to estimate the cost, effort,and schedule when planning a new software development activity. COCOMO II is the latest majorextension to the original COCOMO (COCOMO 81) model published in 1981. It consists of threesubmodels, each one offering increased fidelity the further along one is in the project planning and design process. Listed in increasing fidelity, these sub models are called the Applications Composition,Early Design, and Post-architecture models.

COCOMO II can be used for the following major decision situations

- Making investment or other financial decisions involving a software development effort
- Setting project budgets and schedules as a basis for planning and control
- Deciding on or negotiating tradeoffs among software cost, schedule, functionality, performance or quality factors
- Making software cost and schedule risk management decisions
- Deciding which parts of a software system to develop, reuse, lease, or purchase
- Making legacy software inventory decisions: what parts to modify, phase out, outsource, etc
- Setting mixed investment strategies to improve organization's software capability, via reuse, tools, process maturity, outsourcing, etc
- Deciding how to implement a process improvement strategy, such as that provided in the SEI CMM

The original COCOMO model was first published by Dr. Barry Boehm in 1981, and reflected thesoftware development practices of the day. In the ensuing decade and a half, software development techniques changed dramatically. These changes included a move away from mainframe overnight batch processing to desktop-based real-time turnaround; a greatly increased emphasis on reusing existing software and building new systems using off-the-shelf software components; and spending as much effort to design and manage the software development process as was once spentcreating the software product.

These changes and others began to make applying the original COCOMO model problematic. Thesolution to the problem was to reinvent the model for the 1990s. After several years and the combined efforts of USC-CSSE, ISR at UC Irvine, and the COCOMO II Project Affiliate

Organizations, the result is COCOMO II, a revised cost estimation model reflecting the changes in professional software development practice that have come about since the 1970s. This new, improved COCOMO is now ready to assist professional software cost estimators for many years to come.

Staffing Pattern

Putnam was the first to study the problem of what should be a proper staffing pattern for software projects. He extended the classical work of Norden who had earlier investigated the staffing pattern of general research and development type of projects. In order to appreciate the staffing pattern desirable for software projects, we must understand both Norden's and Putnam's results.

Norden's Work

- Nordern studied the staffing patterns of several R&D projects.
- He found the staffing patterns of R&D projects to be very different from the manufacturing or sales type of work.
- Staffing pattern of R&D types of projects changes dynamically over time for efficient man power utilization.
- He concluded that staffing pattern for any R&D project can be approximated by the Rayleigh distribution curve.

Putnam's Work

- Putnam studied the problem of staffing of software projects.
- He found that staffing pattern for software development projects has characteristics very similar to R&D projects
- He adapted the Rayleigh-Norden curve to relate the no of delivered lines of code to the effort and the time required to develop the product.
- Initially less no of developers are needed.
- As the project progresses and more detailed work is performed, the number of developers increases and reaches a peak during product delivery
- After delivery, the no. of project staff falls consistently during product maintenance.

COSMIC full function points

- COSMIC FFP – Common Software Measurement International Consortium Full Function Point.
- COSMIC deals with decomposing the system architecture into a hierarchy of software layers.
- Unit is Cfsu(COSMIC functional size units).

A Data Movement moves one Data Group. A Data Group is a unique cohesive set of data (attributes) specifying an 'object of interest' (i.e. something that is 'of interest' tothe user). Each Data Movement is counted as one CFP (COSMIC function point).

COSMIC recognizes 4 (types of) Data Movements:

- Entry moves data from outside into the process
- Exit moves data from the process to the outside world
- Read moves data from persistent storage to the process
- Write moves data from the process to persistent storage.

Function Points

Function points were defined in 1979 in *Measuring Application Development Productivity* by Allan Albrecht at IBM. The functional user requirements of the software areidentified and each one is categorized into one of five types: outputs, inquiries, inputs, internal files, and external interfaces. Once the function is identified and categorized into a type, it is then assessed for complexity and assigned a number of function points. Each of these functional user requirements maps to an end-user business function, such as a data entry for an Input or a user query for an Inquiry. This distinction is important because it tends to make the functions measured in function points map easily into user-oriented requirements, but it also tends to hide internal functions (e.g. algorithms), which also require resources to implement.

There is currently no ISO recognized FSM Method that includes algorithmic complexity in the sizing result. Recently there have been different approaches proposed to deal with this perceived weakness, implemented in several commercial software products. The variations of the Albrecht-

based IFPUG method designed to make up for this (and **other weaknesses**) include:

- Early and easy function points – Adjusts for problem and data complexity with two questions that yield a somewhat subjective complexity measurement; simplifies measurement by eliminating the need to count data elements.
- Engineering function points – Elements (variable names) and operators (e.g., arithmetic, equality/inequality, Boolean) are counted. This variation highlights computational function. The intent is similar to that of the operator/operand-based Halstead complexity measures.
- Bang measure – Defines a function metric based on twelve primitive (simple) counts that affect or show Bang, defined as "the measure of true function to be delivered as perceived by the user." Bang measure may be helpful in evaluating a software unit's value in terms of how much useful function it provides, although there is little evidence in the literature of such application.

  The use of Bang measure could apply when re-engineering (either complete or piecewise) is being considered, as discussed in Maintenance of Operational Systems—An Overview.
- Feature points – Adds changes to improve applicability to systems with significant internal processing (e.g., operating systems, communications systems). This allows accounting for functions not readily perceivable by the user, but essential for proper operation.
- Weighted Micro Function Points – One of the newer models (2009) which adjusts function points using weights derived from program flow complexity, operand and operator vocabulary, object usage, and algorithm.

Benefits

The use of function points in favor of lines of code seek to address several additional issues:

- The risk of "inflation" of the created lines of code, and thus reducing the value of the measurement system, if developers are incentivized to be more productive. FP advocates refer to this as measuring the size of the solution instead of the size of the problem.
- Lines of Code (LOC) measures reward low level languages because more lines of code are needed to deliver a similar amount of functionality to a higher level language. C. Jones offersa method of correcting this in his work.
- LOC measures are not useful during early project phases where estimating the number of lines of code that will be delivered is challenging. However, Function Points can be derived from requirements and therefore are useful in methods such as estimation by proxy.

Effort and Cost estimation techniques

Step 1: Determine the Type of Count

There are three types of function point counts −

- Development Function Point Count
- Application Function Point Count
- Enhancement Function Point Count

Development Function Point Count

Function points can be counted at all phases of a development project from requirement to implementation stage. This type of count is associated with new development work and may include the prototypes, which may have been required as temporary solution, which supports the conversion effort. This type of count is called a baseline function point count.

Application Function Point Count

Application counts are calculated as the function points delivered, and exclude any conversion effort (prototypes or temporary solutions) and existing functionality that may have existed.

Enhancement Function Point Count

When changes are made to software after production, they are considered as enhancements. To size such enhancement projects, the Function Point Count gets Added, Changed or Deleted in the Application.

Step 2: Determine the Boundary of the Count

The boundary indicates the border between the application being measured and the external applications or the user domain.

To determine the boundary, understand −

- The purpose of the function point count
- Scope of the application being measured
- How and which applications maintain what data
- The business areas that support the applications

Step 3: Identify Each Elementary Process Required by the User

Compose and/or decompose the functional user requirements into the smallest unit of activity, which satisfies all of the following criteria −

- Is meaningful to the user.
- Constitutes a complete transaction.

- Is self-contained.
- Leaves the business of the application being counted in a consistent state.

For example, the Functional User Requirement − "Maintain Employee information" can be decomposed into smaller activities such as add employee, change employee, delete employee, and inquire about employee. Each unit of activity thus identified is an Elementary Process (EP).

Step 4: Determine the Unique Elementary Processes

Comparing two EPs already identified, count them as one EP (same EP) if they −

- Require the same set of DETs.
- Require the same set of FTRs.
- Require the same set of processing logic to complete the EP.

Do not split an EP with multiple forms of processing logic into multiple Eps.

For e.g., if you have identified 'Add Employee' as an EP, it should not be divided into two EPs to account for the fact that an employee may or may not have dependents. The EP is still 'Add Employee', and there is variation in the processing logic and DETs to account for dependents.

Step 5: Measure Data Functions

Classify each data function as either an ILF or an EIF.

A data function shall be classified as an −

- Internal Logical File (ILF), if it is maintained by the application being measured.
- External Interface File (EIF) if it is referenced, but not maintained by the application being measured.

ILFs and EIFs can contain business data, control data and rules based data. For example, telephone switching is made of all three types - business data, rule data and control data. Business data is the actual call. Rule data is how the call should be routed through the network, and control data is how the switches communicate with each other.

Consider the following documentation for counting ILFs and EIFs −

- Objectives and constraints for the proposed system.
- Documentation regarding the current system, if such a system exists.
- Documentation of the users' perceived objectives, problems and needs.
- Data models.

Step 5.1: Count the DETs for Each Data Function

Apply the following rules to count DETs for ILF/EIF −

- Count a DET for each unique user identifiable, non-repeated field maintained in or retrieved from the ILF or EIF through the execution of an EP.
- Count only those DETs being used by the application that is measured when two or more applications maintain and/or reference the same data function.
- Count a DET for each attribute required by the user to establish a relationship with another ILF or EIF.
- Review related attributes to determine if they are grouped and counted as a single DET or whether they are counted as multiple DETs. Grouping will depend on how the EPs use the attributes within the application.

Step 5.2: Count the RETs for Each Data Function

- Apply the following rules to count RETs for ILF/EIF −
- Count one RET for each data function.
- Count one additional RET for each of the following additional logical sub-groups of DETs.
- Associative entity with non-key attributes.
- Sub-type (other than the first sub-type).
- Attributive entity, in a relationship other than mandatory 1:1.

Step 5.3: Determine the Functional Complexity for Each Data Function

RETS	Data Element Types (DETs)		
	1-19	20-50	>50
1	L	L	A
2 to 5	L	A	H
>5	A	H	H

Functional Complexity: **L** = Low; **A** = Average; **H** = High

Step 5.4: Measure the Functional Size for Each Data Function

Functional Complexity	FP Count for ILF	FP Count for EIF
Low	7	5
Average	10	7
High	15	10

Step 6: Measure Transactional Functions

To measure transactional functions following are the necessary steps −

Step 6.1: Classify each Transactional Function

Transactional functions should be classified as an External Input, External Output or an External Inquiry.

External Input

External Input (EI) is an Elementary Process that processes data or control information that comes from outside the boundary. The primary intent of an EI is to maintain one or more ILFs and/or to alter the behavior of the system.

All of the following rules must be applied −

- The data or control information is received from outside the application boundary.
- At least one ILF is maintained if the data entering the boundary is not control information that alters the behavior of the system

For the identified EP, one of the three statements must apply −

- Processing logic is unique from processing logic performed by other EIs for the application.
- The set of data elements identified is different from the sets identified for other EIs in the application.
- ILFs or EIFs referenced are different from the files referenced by the other EIs in the application.

External Output

External Output (EO) is an Elementary Process that sends data or control information outside the application's boundary. EO includes additional processing beyond that of an external inquiry.

The primary intent of an EO is to present information to a user through processing logic other than or in addition to the retrieval of data or control information.

The processing logic must −

- Contain at least one mathematical formula or calculation.
- Create derived data.
- Maintain one or more ILFs.
- Alter the behavior of the system.

All of the following rules must be applied –

- Sends data or control information external to the application's boundary.

For the identified EP, one of the three statements must apply −

- Processing logic is unique from the processing logic performed by other EOs for the application.
- The set of data elements identified are different from other EOs in the application.
- ILFs or EIFs referenced are different from files referenced by other EOs in the application.

Additionally, one of the following rules must apply

The processing logic contains at least one mathematical formula or calculation.

- The processing logic maintains at least one ILF.
- The processing logic alters the behavior of the system.

External Inquiry

External Inquiry (EQ) is an Elementary Process that sends data or control information outside the boundary.

The primary intent of an EQ is to present information to the user through the retrieval of data or control information.

The processing logic contains no mathematical formula or calculations, and creates no derived data. No ILF is maintained during the processing, nor is the behavior of the system altered.

All of the following rules must be applied −

- Sends data or control information external to the application's boundary.

For the identified EP, one of the three statements must apply −

- Processing logic is unique from the processing logic performed by other EQs for the application.
- The set of data elements identified are different from other EQs in the application.
- The ILFs or EIFs referenced are different from the files referenced by other EQs in the application.

Additionally, all of the following rules must apply −

- The processing logic retrieves data or control information from an ILF or EIF.
- The processing logic does not contain mathematical formula or calculation.
- The processing logic does not alter the behavior of the system.
- The processing logic does not maintain an ILF.

Step 6.2: Count the DETs for Each Transactional Function

Apply the following Rules to count DETs for EIs −

- Review everything that crosses (enters and/or exits) the boundary
- Count one DET for each unique user identifiable, non-repeated attribute that crosses (enters and/or exits) the boundary during the processing of the transactional function.
- Count only one DET per transactional function for the ability to send an application response message, even if there are multiple messages.
- Count only one DET per transactional function for the ability to initiate action(s) even if there are multiple means to do so.

Do not count the following items as DETs −

- Attributes generated within the boundary by a transactional function and saved to an ILF without exiting the boundary.
- Literals such as report titles, screen or panel identifiers, column headings and attribute titles.
- Application generated stamps such as date and time attributes.
- Paging variables, page numbers and positioning information, for e.g., ‘Rows 37 to 54 of 211’.
- Navigation aids such as the ability to navigate within a list using “previous”, “next”, “first”, “last” and their graphical equivalents.

Apply the following rules to count DETs for EOs/EQs −

- Review everything that crosses (enters and/or exits) the boundary.

Count one DET for each unique user identifiable, non-repeated attribute that crosses (enters and/or exits) the boundary during the processing of the transactional function.

- Count only one DET per transactional function for the ability to send an application response message, even if there are multiple messages.
- Count only one DET per transactional function for the ability to initiate action(s) even if there are multiple means to do so.

Do not count the following items as DETs −

Attributes generated within the boundary without crossing the boundary.

- Literals such as report titles, screen or panel identifiers, column headings and attribute titles.
- Application generated stamps such as date and time attributes.
- Paging variables, page numbers and positioning information, for e.g., 'Rows 37 to 54 of 211'.
- Navigation aids such as the ability to navigate within a list using "previous", "next", "first", "last" and their graphical equivalents.

Step 6.3: Count the FTRs for Each Transactional Function

Apply the following rules to count FTRs for EIs −

- Count a FTR for each ILF maintained.
- Count a FTR for each ILF or EIF read during the processing of the EI.
- Count only one FTR for each ILF that is both maintained and read.

Apply the following rule to count FTRs for EO/EQs −

- Count a FTR for each ILF or EIF read during the processing of EP.

Additionally, apply the following rules to count FTRs for EOs −

- Count a FTR for each ILF maintained during the processing of EP.
- Count only one FTR for each ILF that is both maintained and read by EP.

Extreme Programming

One of the foremost Agile methodologies is called Extreme Programming (XP), which involves a high degree of participation between two parties in the software exchange: customers and developers. The former inspires further development by emphasizing the most useful features of a given software product through testimonials. The developers, in turn, base each successive set of software upgrades on this feedback while continuing to test new innovations every few weeks. XP has its share of pros and cons. On the upside, this Agile methodology involves a high level of collaboration and a minimum of up-front documentation. It's an efficient and persistent delivery model. However, the methodology also requires a great level of discipline, as well as plenty of involvement from people beyond the world of information technology. Furthermore, in order for the best results, advanced XP proficiency is vital on the part of every team member.

Extreme programming (**XP**) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development, it advocates frequent "releases" in short development cycles, which is intended to improve productivity and introduce checkpoints at which new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, code simplicity and clarity, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers. The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. As an example, code reviews are considered a beneficial practice; taken to the extreme, code can be reviewed *continuously*, i.e. the practice of pair programming.

Activities

XP describes four basic activities that are performed within the software development process:coding, testing, listening, and designing. Each of those activities is described below.

Coding

The advocates of XP argue that the only truly important product of the system development processis code – software instructions that a computer can interpret. Without code, there is no working product. Coding can also be used to figure out the most suitable solution. Coding can

also help to communicate thoughts about programming problems. A programmer dealing with a complex programming problem, or finding it hard to explain the solution to fellow programmers, might code it in a simplified manner and use the code to demonstrate what he or she means. Code, say the proponents of this position, is always clear and concise and cannot be interpreted in more than one way. Other programmers can give feedback on this code by also
coding their thoughts.

Testing

Extreme programming's approach is that if a little testing can eliminate a few flaws, a lot of testingcan eliminate many more flaws.

- Unit tests determine whether a given feature works as intended. Programmers write as many automated tests as they can think of that might "break" the code; if all tests run successfully, then the coding is complete. Every piece of code that is written is tested before moving on to the next feature.
- Acceptance tests verify that the requirements as understood by the programmers satisfy the customer's actual requirements.

System-wide integration testing was encouraged, initially, as a daily end-of-day activity, for early detection of incompatible interfaces, to reconnect before the separate sections diverged widely from coherent functionality. However, system-wide integration testing has been reduced, to weekly, or less often, depending on the stability of the overall interfaces in the system.

Listening

Programmers must listen to what the customers need the system to do, what "business logic" is needed. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved. Communication between the customer and programmer is further addressed in the planning game.

Designing

From the point of view of simplicity, of course one could say that system development doesn't need more than coding, testing and listening. If those activities are performed well, the result should always be a system that works. In practice, this will not work. One can come a long way without designing but at a given time one will get stuck. The system becomes too complex and the dependencies within the system cease to be clear. One can avoid this by creating a design

structurethat organizes the logic in the system. Good design will avoid lots of dependencies within a system;this means that changing one part of the system will not affect other parts of the system.

Advantages

Robustness

- Resilience
- Cost savings
- Lesser risks

Disadvantages

- It assumes constant involvement of customers
- Centered approach rather than design-centered approach
- Lack of proper documentation

Managing Interactive Processes

Booch suggests that there are two levels of development:

- The macro process
- The micro process

Macro process

- Establish core requirements (conceptualization).
- Develop a model of the desired behavior (analysis).
- Create an architecture (design).
- Evolve the implementation (evolution).
- Manage post delivery evolution (maintenance).

Micro process

- Identify the classes and objects at a given level of abstraction.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Specify the interface and then the implementation of these classes and objects

In principle, the micro process represents the daily activity of the individual developer, or of a small team of developers.

The macro process serves as the controlling framework of the micro process. It represents the activities of the entire development team on the scale of weeks to months at a time. The basic philosophy of the macro process is that of incremental development: the system as a whole is built up step by step, each successive version consisting of the previous ones plus a number of new functions.

Basics of Software estimation

Estimation techniques are of utmost importance in software development life cycle, where the time required to complete a particular task is estimated before a project begins. Estimation is the process of finding an estimate, or approximation, which is a value that can be used for some

purpose even if input data may be incomplete, uncertain, or unstable.

The four basic steps in software project estimation are:

1) Estimate the size of the development product. This generally ends up in either Lines of Code (LOC) or Function Points (FP), but there are other possible units of measure. A discussion of the pros & cons of each is discussed in some of the material referenced at the end of this report.

2) Estimate the effort in person-months or person-hours.

3) Estimate the schedule in calendar months.

4) Estimate the project cost in dollars (or local currency)

The major shortcomings of SLOC measure:

- No precise definition
- Difficult to estimate at start of a project
- Only a code measure
- Programmer-dependent
- Does not consider code complexity

Estimation is based on −

- Past Data/Past Experience
- Available Documents/Knowledge
- Assumptions
- Identified Risks

➢ Estimation need not be a one-time task in a project. It can take place during −

- Acquiring a Project.
- Planning the Project.
- Execution of the Project as the need arises.

➢ Project scope must be understood before the estimation process begins. It will be helpful to have historical Project Data.

- Project metrics can provide a historical perspective and valuable input for generation of quantitative estimates.

  Planning requires technical managers and the software team to make an initial commitment as it leads to responsibility and accountability.
  - Past experience can aid greatly.
  - Use at least two estimation techniques to arrive at the estimates and reconcile the resulting values. Refer Decomposition Techniques in the next section to learn about reconciling estimates.
  - Plans should be iterative and allow adjustments as time passes and more details are known.

General Project Estimation Approach

- The Project Estimation Approach that is widely used is **Decomposition Technique**. Decomposition techniques take a divide and conquer approach. Size, Effort and Cost estimation are performed in a stepwise manner by breaking down a Project into major Functions or related Software Engineering Activities.

  Step 1 − Understand the scope of the software to be built.

  Step 2 − Generate an estimate of the software size.
    - Start with the statement of scope.
    - Decompose the software into functions that can each be estimated individually.
    - Calculate the size of each function.
    - Derive effort and cost estimates by applying the size values to your baseline productivity metrics.
    - Combine function estimates to produce an overall estimate for the entire project.

  Step 3 − Generate an estimate of the effort and cost. You can arrive at the effort and cost estimates by breaking down a project into related software engineering activities.
    - Identify the sequence of activities that need to be performed for the project to be completed.
    - Divide activities into tasks that can be measured.
    - Estimate the effort (in person hours/days) required to complete each task.

- Combine effort estimates of tasks of activity to produce an estimate for the activity.
    - Obtain cost units (i.e., cost/unit effort) for each activity from the database.
    - Compute the total effort and cost for each activity.
    - Combine effort and cost estimates for each activity to produce an overall effort and cost estimate for the entire project.

Step 4 − Reconcile estimates: Compare the resulting values from Step 3 to those obtained from Step 2. If both sets of estimates agree, then your numbers are highly reliable. Otherwise, if widely divergent estimates occur conduct further investigation concerning whether −

- The scope of the project is not adequately understood or has been misinterpreted.
- The function and/or activity breakdown is not accurate.
- Historical data used for the estimation techniques is inappropriate for the application, or obsolete, or has been misapplied.

Step 5 − Determine the cause of divergence and then reconcile the estimates.

Estimation Accuracy

Accuracy is an indication of how close something is to reality. Whenever you generate an estimate, everyone wants to know how close the numbers are to reality. You will want every estimate to be as accurate as possible, given the data you have at the time you generate it. And of course you don't want to present an estimate in a way that inspires a false sense of confidence in the numbers.

- Important factors that affect the accuracy of estimates are −
    - The accuracy of all the estimate's input data.
    - The accuracy of any estimate calculation.
    - How closely the historical data or industry data used to calibrate the model matches the project you are estimating.
    - The predictability of your organization's software development process.
    - The stability of both the product requirements and the environment that supports the software engineering effort.

- Whether or not the actual project was carefully planned, monitored and controlled, and no major surprises occurred that caused unexpected delays.

➢ Following are some guidelines for achieving reliable estimates −

- Base estimates on similar projects that have already been completed.
- Use relatively simple decomposition techniques to generate project cost and effort estimates.
- Use one or more empirical estimation models for software cost and effort estimation.
- To ensure accuracy, you are always advised to estimate using at least two techniques and compare the results.

Estimation Issues

- Often, project managers resort to estimating schedules skipping to estimate size. This may be because of the timelines set by the top management or the marketing team. However, whatever the reason, if this is done, then at a later stage it would be difficult to estimate the schedules to accommodate the scope changes.
- While estimating, certain assumptions may be made. It is important to note all these assumptions in the estimation sheet, as some still do not document assumptions in estimation sheets.
- Even good estimates have inherent assumptions, risks, and uncertainty, and yet they are often treated as though they are accurate.
- The best way of expressing estimates is as a range of possible outcomes by saying, for example, that the project will take 5 to 7 months instead of stating it will be complete on a particular date or it will be complete in a fixed no. of months. Beware of committing to a range that is too narrow as that is equivalent to committing to a definite date.
- You could also include uncertainty as an accompanying probability value. For example, there is a 90% probability that the project will complete on or before a definite date.
- Organizations do not collect accurate project data. Since the accuracy of the estimates depend on the historical data, it would be an issue.
- For any project, there is a shortest possible schedule that will allow you to include the required functionality and produce quality output. If there is a schedule constraint by

- management and/or client, you could negotiate on the scope and functionality to be delivered.
- Agree with the client on handling scope creeps to avoid schedule overruns.
    - Failure in accommodating contingency in the final estimate causes issues. For e.g., meetings, organizational events.
    - Resource utilization should be considered as less than 80%. This is because the resources would be productive only for 80% of their time. If you assign resources at more than 80% utilization, there is bound to be slippages.

Estimation Guidelines

One should keep the following guidelines in mind while estimating a project −

- During estimation, ask other people's experiences. Also, put your own experiences at task.
- Assume resources will be productive for only 80 percent of their time. Hence, during estimation take the resource utilization as less than 80%.
- Resources working on multiple projects take longer to complete tasks because of the time lost switching between them.
- Always build in contingency for problem solving, meetings and other unexpected events.
- Allow enough time to do a proper project estimate. Rushed estimates are inaccurate, high-risk estimates. For large development projects, the estimation step should really be regarded as a mini project. Where possible, use documented data from your organization's similar past projects. It will result in the most accurate estimate. If your organization has not kept historical data, now is a good time to start collecting it.
- Use developer-based estimates, as the estimates prepared by people other than those who will do the work will be less accurate. Use several different people to estimate and use several different estimation techniques.
- Reconcile the estimates. Observe the convergence or spread among the estimates. Convergence means that you have got a good estimate. Wideband-Delphi technique can be used to gather and discuss estimates using a group of people, the intention being to produce an accurate, unbiased estimate.

- Re-estimate the project several times throughout its life cycle

Rapid Application Development

RAD model is Rapid Application Development model. It is a type of incremental model. In RAD model the components or functions are developed in parallel as if they were mini projects. The developments are time boxed, delivered and then assembled into a working prototype. This can quickly give the customer something to see and use and to provide feedback regarding the delivery and their requirements.

The phases in the rapid application development (RAD) model are:

Business modeling: The information flow is identified between various business functions.

Data modeling: Information gathered from business modeling is used to define data objects that are needed for the business.

Process modeling: Data objects defined in data modeling are converted to achieve the business information flow to achieve some specific business objective. Description are identified and created for CRUD of data objects.

Application generation: Automated tools are used to convert process models into code and the actual system.

Testing and turnover: Test new components and all the interfaces.

Advantages of the RAD model:

- Reduced development time.
- Increases reusability of components
- Quick initial reviews occur
- Encourages customer feedback
- Integration from very beginning solves a lot of integration issues.

Disadvantages of RAD model:

- Depends on strong team and individual performances for identifying business requirements.
- Only system that can be modularized can be built using RAD
- Requires highly skilled developers/designers.
- High dependency on modeling skills
- Inapplicable to cheaper projects as cost of modeling and automated code generation is very high.

Agile Methods

The Agile methodology derives from a namesake manifesto, which advanced ideas that were developed to counter the more convoluted methods that pervaded the software development world despite being notoriously inefficient and counterproductive. Promoting similar methods to Lean, the key principles of Agile are as follows:

- Satisfying customers is of foremost importance
- Develop projects with inspired contributors
- Interactions are best when done in person
- Software that works is a measure of progress
- Reflect and adapt on an ongoing basis

Additionally, the four core values of Agile are as follows:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

- Agile Project Management is one of the revolutionary methods introduced for the practice of project management. This is one of the latest project management strategies that is mainly applied to project management practice in software development. Therefore, it is best to relate agile project management to the software development process when understanding it.
- From the inception of software development as a business, there have been a number of processes following, such as the waterfall model. With the advancement of software development, technologies and business requirements, the traditional models are not robust enough to cater the demands.
- Therefore, more flexible software development models were required in order to address the agility of the requirements. As a result of this, the information technology community developed agile software development models.
- 'Agile' is an umbrella term used for identifying various models used for agile development, such as Scrum. Since agile development model is different from conventional models, agile project management is a specialized area in project management.
    - There are many differences in agile development model when compared to traditional models:

- The agile model emphasizes on the fact that entire team should be a tightly integrated unit. This includes the developers, quality assurance, project management, and the customer.
- Frequent communication is one of the key factors that makes this integration possible. Therefore, daily meetings are held in order to determine the day's work and dependencies.
- Deliveries are short-term. Usually a delivery cycle ranges from one week to four weeks. These are commonly known as sprints.
- Agile project teams follow open communication techniques and tools which enable the team members (including the customer) to express their views and feedback openly and quickly. These comments are then taken into consideration when shaping the requirements and implementation of the software.

Software Process and Process Models

A Software product development process usually starts when a request for the product is received from the customer.

- Starting from the inception stage:
  - A product undergoes a series of transformations through a few identifiable stages
  - Until it is fully developed and released to the customer.
- After release:
  - The product is used by the customer and during this time the product needs to be maintained for fixing bugs and enhancing functionalities. This stage is called Maintenance stage.
- This set of identifiable stages through which a product transits from inception to retirement form the life cycle of the product.
- Life cycle model (also called a process model) is a graphical or textual representationof its life cycle.

Choice of Process Models

The no. of inter related activities to create a final product can be organized in different ways and we can call these Process Models.

A software process model is a simplified representation of a software process. Each model represents a process from a specific perspective. These generic models are abstractions of the process that can be used to explain different approaches to the software development.

Any software process must include the following four activities:

1. **Software specification** (or requirements engineering): Define the main functionalities of the software and the constrains around them.
2. S**oftware design and implementation**: The software is to be designed and programmed.
3. **Software verification and validation**: The software must conforms to it's specification andmeets the customer needs.
4. **Software evolution** (software maintenance): The software is being modified to meetcustomer and market requirements changes.

The various Process Models are

Water Fall Model

Spiral Model Prototype

Model Incremental

Delivery

i) Water fall model

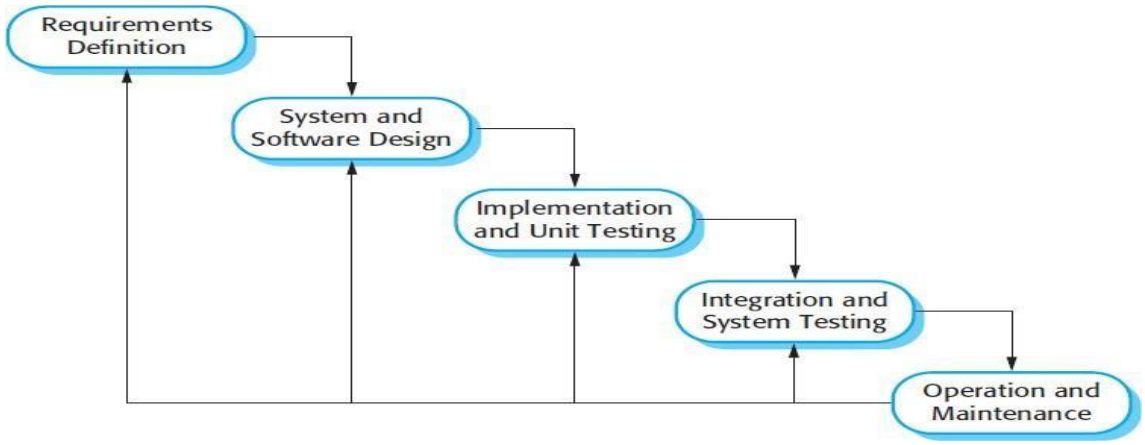
The waterfall model is a sequential approach, where each fundamental activity of a processrepresented as a separate phase, arranged in linear order.

In the waterfall model, you must plan and schedule all of the activities before starting working on them (plan-driven process).

Plan-driven process is a process where all the activities are planned first, and the progress is measured against the plan. While the agile process, planning is incremental and it's easier to changethe process to reflect requirement changes.

The phases of the waterfall model are:

- Requirements
- Design
- Implementation
- Testing
- Maintenance



In principle, the result of each phase is one or more documents that should be approved and the next phase shouldn't be started until the previous phase has completely been finished. In practice, however, these phases overlap and feed information to each other. For example, during design, problems with requirements can be identified, and during coding, some of the design problems can be found, etc.

The software process therefore is not a simple linear but involves feedback from one phase to another. So, documents produced in each phase may then have to be modified to reflect the changes made.

ii) Spiral Model

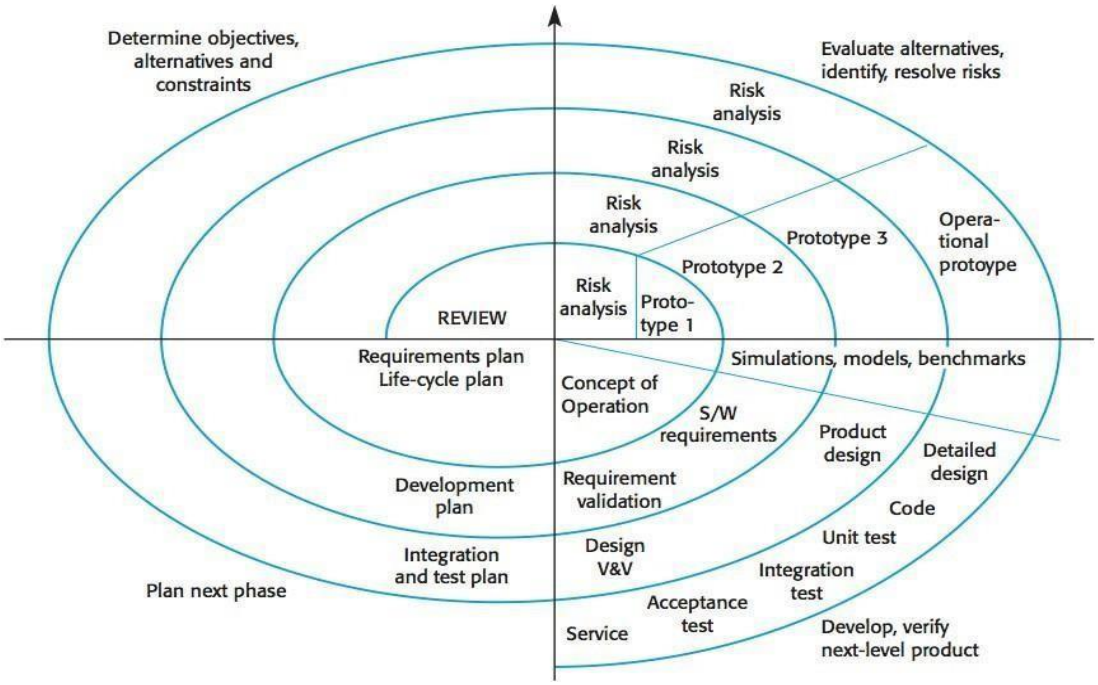
The spiral model is similar to the incremental model, with more emphasis placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spiralsin this model). The baseline spiral, starting in the planning phase, requirements is gathered and risk is assessed. Each subsequent spiral builds on the baseline spiral. It is one of the software development models like Waterfall, Agile, V-Model.

Planning Phase: Requirements are gathered during the planning phase. Requirements like 'BRS' that is 'Bussiness Requirement Specifications' and 'SRS' that is 'System Requirement specifications'.

Risk Analysis: In the **risk analysis phase**, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase. If any risk is found during the risk analysis then alternate solutions are suggested and implemented.

Engineering Phase: In this phase software is **developed**, along with **testing** at the end of the phase. Hence in this phase the development and testing is done.

Evaluation phase: This phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.



Advantages of Spiral model:

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.

Disadvantages of Spiral model:

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

iii) Software Prototyping

A prototype is a version of a system or part of the system that's developed quickly to checkthe customer's requirements or feasibility of some design decisions. So, a prototype is useful when a

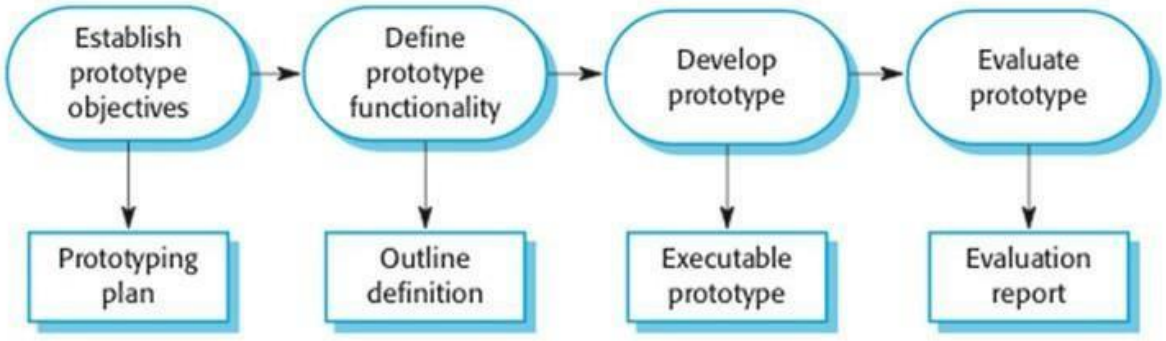
customer or developer is not sure of the requirements, or of algorithms, efficiency, business rules, response time, etc.

In prototyping, the client is involved throughout the development process, which increasesthe likelihood of client acceptance of the final implementation. While some prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.

A software prototype can be used:

[1] In the **requirements engineering**, a prototype can help with the elicitation and validation of system requirements. It allows the users to experiment with the system, and so, refine the requirements. They may get new ideas for requirements, and find areas of strength and weakness in the software. Furthermore, as the prototype is developed, it may reveal errors and in the requirements. The specification maybe then modified to reflect the changes.

[2] In the **system design**, a prototype can help to carry out deign experiments to check the feasibilityof a proposed design. For example, a database design may be prototype-d and tested to check it supports efficient data access for the most common user queries.



The phases of a prototype are:

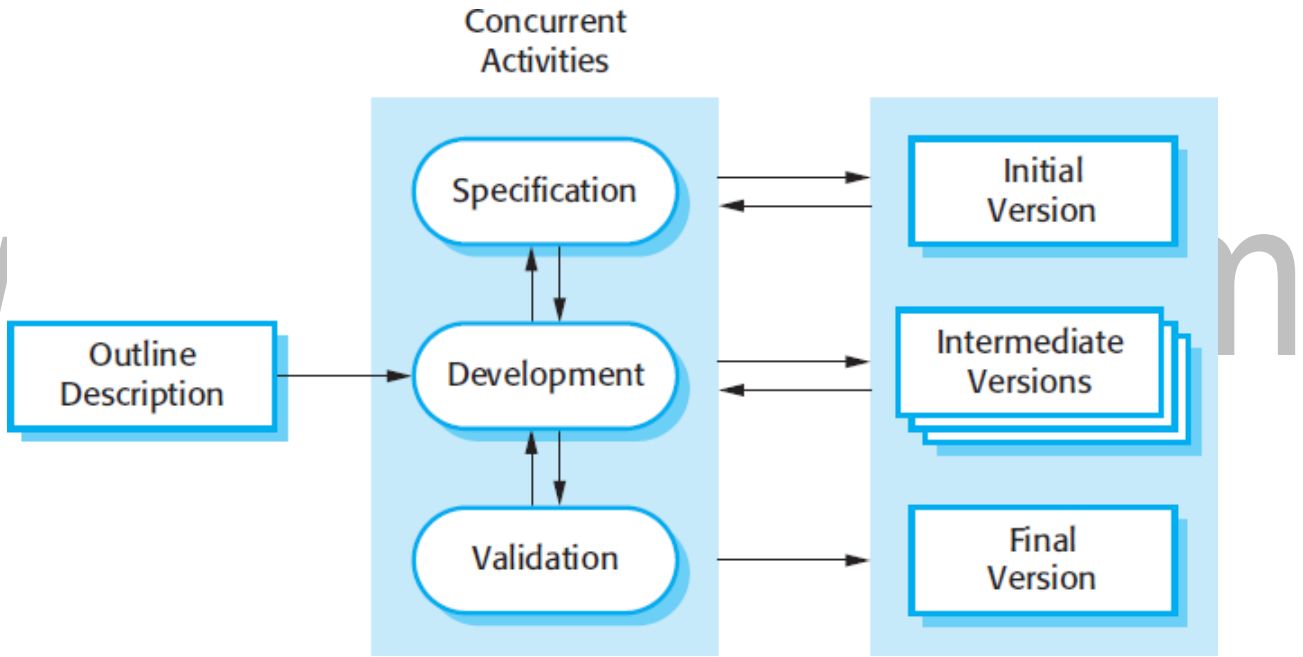
1. **Establish objectives**: The objectives of the prototype should be made explicit from the start ofthe process. Is it to validate system requirements, or demonstrate feasibility, etc.
2. **Define prototype functionality**: Decide what are the inputs and the expected output from a prototype. To reduce the prototyping costs and accelerate the delivery schedule, you may ignoresome functionality, such as response time and memory utilization unless they are relevant to the objective of the prototype.

4. **Evaluate the prototype**: Once the users are trained to use the prototype, they then discover requirements errors. Using the feedback both the specifications and the prototype can be improved. If changes are introduced, then a repeat of steps 3 and 4 may be needed.

Prototyping is not a standalone, complete development methodology, but rather an approachto be used in the context of a full methodology (such as incremental, spiral, etc).

iv) Incremental Delivery

Incremental development is based on the idea of developing an initial implementation, exposing this to user feedback, and evolving it through several versions until an acceptable system has been developed. The activities of a process are not separated but interleaved with feedback involved across those activities.



Each system increment reflects a piece of the functionality that is needed by the customer. Generally, the early increments of the system should include the most important or most urgently required functionality.

This means that the customer can evaluate the system at early stage in the development to see if it delivers what's required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental Vs Waterfall Model

Incremental software development is better than a waterfall approach for most business, e-commerce, and personal systems. By developing the software incrementally, it is cheaper and easier to make changes in the softwareas it is being developed.

Compared to the waterfall model, incremental development has three important benefits:

1. The **cost of accommodating changing** customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than that's required with waterfall model.
2. It's easier to get **customer feedback** on the work done during development than when the system is fully developed, tested, and delivered.
3. More **rapid delivery** of useful software is possible even if all the functionality hasn't been included. Customers are able to use and gain value from the software earlier than it's possiblewith the waterfall model.