**UNIT-1**

**SOFTWARE PROCESS AND PROJECT MANAGEMENT**

**Introduction to Software Engineering**

Software engineering is a discipline in which theories, methods and tools are applied to develop professional software. (Or)Software engineering is the systematic approach to develop and maintain a software product in a cost effective and efficient way.

**Characteristics of software**

* Software is engineered or developed, it is not manufactured in the classical sense.
* Software doesn’t wear out.
* Although the industry is moving toward component based assembly, most software continues to be custom built.

**Goals of Software Engineering**

Software Engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines. The goals of software engineering are:

* Software production which consists of developed programs and associated documentation.
* The software product should have the essential product attributes maintainability, dependability, efficiency and acceptability.
* It should also include suggestions for the process to be followed, the notations to be used, system models to be developed and rules governing these models and design guidelines.

The two fundamental types of software product are

* **Generic products:** These are stand alone systems developed by organizations and sold on open market to any customer who is able to buy them.
* **Customized products:** These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer.

**Generic process framework**

A process framework establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process.Each framework activity is populated by a set of software engineering actions-a collection of related tasks that produces a major software engineering work product. Each action is populated with individual work tasks that accomplish some part of the work implied by the action.

The following generic process framework is applicable to vast majority of software projects:

* **Communication:** This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.
* **Planning:** This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
* **Modeling:** The activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.
* **Construction:** This activity combines code generation and the testing that is required to uncover errors in the code.
* **Deployment:** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

The modeling activity is composed of two software engineering actions:

* **Analysis** encompasses a set of work tasks requirements gathering, elaboration, negotiation, specification and validation that lead to the creation of the analysis model or requirements specification.
* **Design** encompasses work tasks data design, architectural design, interface design and component-level design and create a design model or design specification.

Each software engineering action is represented by a number of different task sets-each a collection of software engineering work tasks, related work products, quality assurance points and project milestones. The task set that best accommodates the needs of the project and characteristics of the team is chosen. The framework described in the generic view of software engineering is completed by a number of umbrella activities. Typical activities in this category include:

* **Software Project Tracking and Control-**allows the software team to assess

progress against the project plan and take the necessary action to maintain schedule.

* **Risk Management**-assess the risks that may effect the outcome of the project or the

quality of the product.

* **Software Quality Assurance**-defines and conducts the activities required to ensure

software quality.

* **Formal Technical Reviews**-assesses software engineering work products in an effort

to uncover or remove errors before they are propagated to the next action or activity.

* **Measurement**-defines and collects process, project and product measures that assist the team in delivering software that meets customer needs.
* **Software Configuration Management**-manages the effects of change throughout

the software process.

* **Reusability Management**-defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
* **Work Product Preparation and Production**-encompasses the activities required to create work products such as models, documents, logs, forms and lists.

All process models can be categorized within the process framework discussed. But process models do differ fundamentally in:

* The overall flow of activities and tasks and the interdependencies among activities and tasks.
* The degree to which work tasks are defined within each framework activity.
* The degree to which work products are identified and required.
* The manner which quality assurance activities are applied.
* The manner in which project tracking and control activities are applied.
* The overall degree of detail and rigor with which the process is described.
* The degree to which customer and other stakeholders are involved within the project.
* The level of autonomy given to the software project team.
* The degree to which team organization and roles are prescribed.

**Prescriptive Process Models**

They are often referred to as “conventional” process models and prescribe a set of the following process elements such as

* Framework activities
* Software engineering actions
* Tasks
* Work products
* Quality assurance and
* Change control mechanisms for each project

Each process model prescribes a workflow and there are various process models that differ in their emphasis on different activities.

**WATERFALL MODEL**

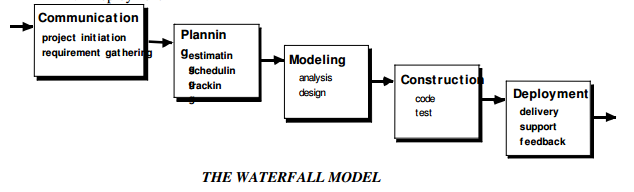
When the requirements of a problem are reasonably well understood-when work flows from communication through deployment in a reasonably linear fashion. This situation is encountered when

* Well defined adaptations or enhancements to an existing system must be made.
* In limited number of new development efforts, but only when requirements are well defined and reasonably stable.

The waterfall model, called the classic life cycle suggests a systematic, sequential approach

to software development.

* Begins with customer specification of requirements
* Progresses through planning
* Modeling
* Construction
* Deployment

****

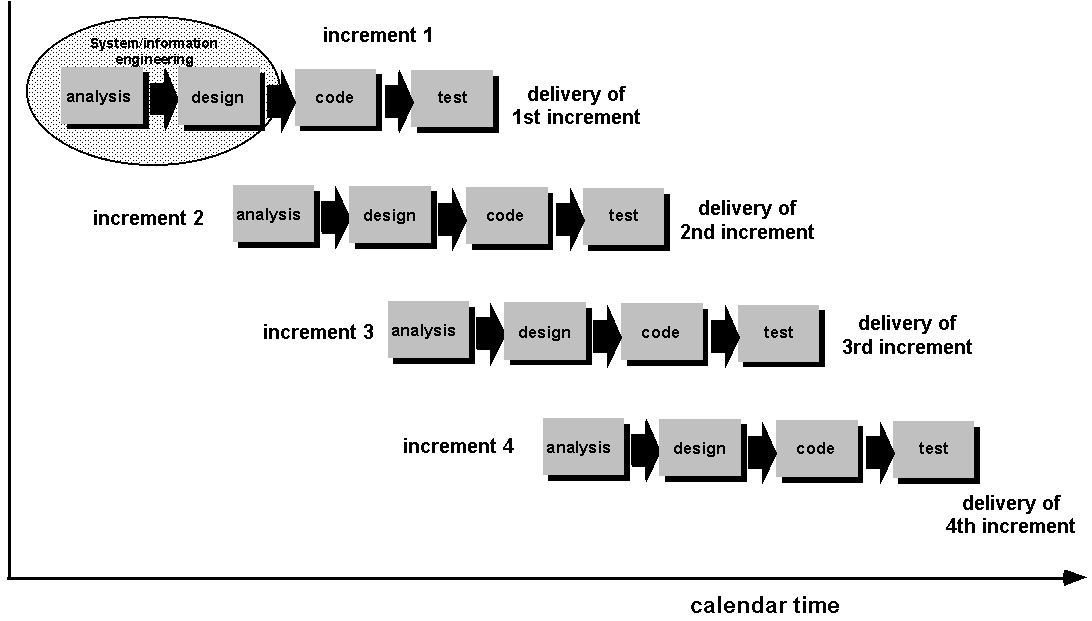
**Problems encountered when Waterfall Model is applied:**

* Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
* It is often difficult for the customer to state all the requirements explicitly. The waterfall model requires this and has difficulty accommodating uncertainty that exists at the beginning of many projects.
* The customer must have patience. A working version of the program will not be available until late in the project life-span. A major blunder, if undetected until the working program is reviewed can be disastrous.

Linear structure of the waterfall model leads to blocking states in which some project team members must wait for other members of the team to complete dependent tasks. Blocking states are more prevalent at the beginning and end of the linear sequential process.  **INCREMENTAL MODELS**

The incremental model applies linear sequences in a staggered fashion as calendar time progresses. Each linear sequence produces deliverable “increments” of the software in a manner that is similar to the increments produced by an evolutionary process flow.

* When an incremental model is used, the first increment is often a core product.
* The core product is used by the customer (or undergoes detailed evaluation).
* As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
* This process is repeated following the delivery of each increment, until the complete product is produced.



**Advantages of Incremental model:**

* Generates working software quickly and early during the software life cycle.
* This model is more flexible – less costly to change scope and requirements.
* It is easier to test and debug during a smaller iteration.
* In this model customer can respond to each built.
* Lowers initial delivery cost.
* Easier to manage risk because risky pieces are identified and handled during it’d iteration.

**Disadvantages of Incremental model:**

* Needs good planning and design.
* Needs a clear and complete definition of the whole system before it can be broken down and built incrementally.
* Total cost is higher than [waterfall](http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/).

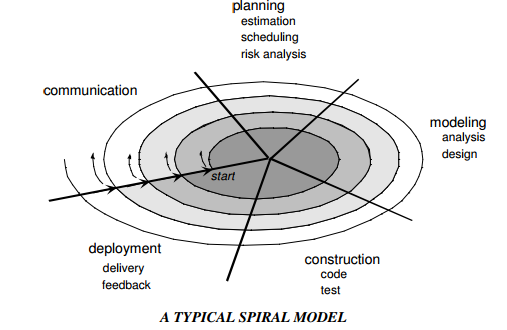
**SPIRAL MODEL**

The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It provide the potential for rapid development of increasingly more complex versions of the software.

Boehm definition:

The spiral development model is a risk-driven process model generator. It has two main distinguishing features :

* One is a cyclic approach for incrementally growing a system’s degree of definition and implementation while decreasing its degree of risk.
* The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

Using spiral model, software is developed in a series of evolutionary releases. During early iterations, the release is a paper model or prototype. During later iterations, more complex versions of the engineered system are produced. 

A spiral model is divided into a set of framework activities defined by the software engineering team. Each of the framework activities represents one segment of the spiral path as shown in the figure.

* As evolution begins the software team performs activities implied by the by the circuit around the spiral, in clockwise direction, beginning at the center.
* Risk is considered at each revolution made.
* Anchor point milestones-a combination of work products and conditions that are attained along the path of the spiral are noted for each evolutionary pass.
* First circuit around the spiral results in the development of product specifications.
* Subsequent passes develop a prototype, progressively lead to sophisticated version of the software.
* Cost and schedule are adjusted based on feedback derived from the customer after delivery.
* Unlike other process models that end when the software is delivered, the spiral model can be adapted throughout the life of the software.
* The first spiral represents a Concept Development Project which starts at core and continues for multiple iterations until concept development is implemented.
* The concept developed to an actual product, proceeds outward on the spiral and a New Product Development Project commences.
* The new product may evolve to represent Product Enhancement Project.

**Features of Spiral Model:**

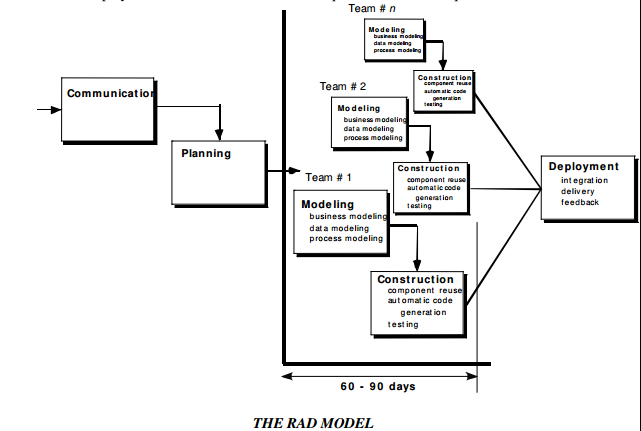
* Realistic approach to the development of large scale-systems and software.
* Because the software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level.
* Enables the developer to apply the prototyping approach at any stage in the evolution of the product.
* It maintains the systematic stepwise approach of classic life cycle but in incorporates it into an iterative framework that realistically reflects the real world.
* It demands direct consideration of technical risks at all stages of the project and if applied properly, reduce risks before they become problematic.

**RAD MODEL**

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a high-speed adaptation of the waterfall model, in which rapid development is achieved by using component-based construction approach. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a fully functional system within a very short time period.

Like other process models, RAD approach maps to the generic framework activities as

* Communication, works to understand the business problem and the information characteristics that the software must accommodate.
* Planning, is essential because multiple software teams work in parallel on different system functions.
* Modeling, encompasses three major phases-business modeling, data modeling and process modeling- and establishes design representations that serve as the basis for RAD’s construction activity.
* Construction, emphasizes the use of pre-existing software components and the application of automatic code generation.
* Deployment, establishes a basis for subsequent iterations, if required.



If a business application can be modularized in a way that enables each major function to be completed in less than three months, it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole. **RAD Drawbacks:**

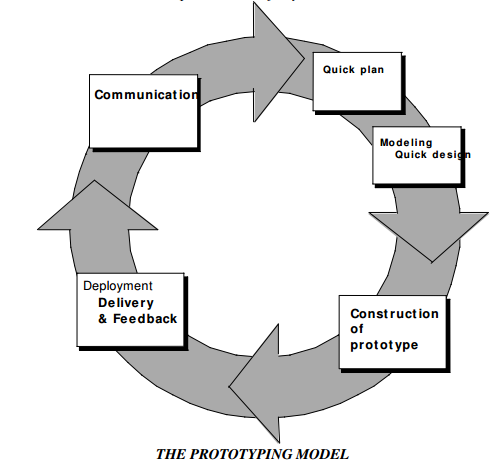
* For large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams.
* If developers and customers are not committed to the rapid fire activities, RAD project fails.
* If a system is not properly modularized, building the components for RAD will be problematic.
* If high performance is an issue and performance is to be achieved through tuning the interfaces to system components, the RAD approach may not work.
* Not appropriate when technical risks are high or when new technology is used.

**PROTOTYPING MODEL**

The customer usually defines a set of general objectives for software, but does not identify detailed input, processing or output requirements. The developer may be unsure of

* The efficiency of an algorithm
* The adaptability of an operating system
* The form that human-machine interaction should take

In these and many other situations, a prototyping paradigm may offer the best approach. Although prototyping can be used as a standalone process model, it is more often used as a technique that can be implemented within the context of any one of the other process models. The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when the requirements are fuzzy.



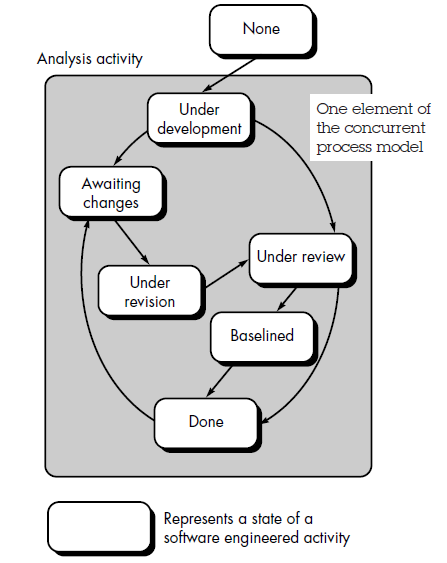
* The prototyping paradigm begins with communication. The software engineer and the customer meet and
* Define the overall objectives for the software
* Outline areas whether further definition is mandatory
* Identify whatever requirements are known
* The quick design leads to construction of a prototype.
* The prototype is deployed and then evaluated by customer/user. The feedback is used to refine requirements for the software.
* Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.
* Prototyping serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools that enable working programs to be generated quickly.
* The prototype can serve as the first system. Both customers and developers like the prototyping paradigm. Users get a feel for the actual system and developers get to build something immediately.

**Problems with Prototyping**

* The customer sees what appears to be a working version of the software, unaware that in the rush to get it working, the quality or maintainability is not considered. When informed that the software has to be rebuilt, demands for a few fixes to make it a working product.

The developer often makes implementation compromises to make the prototype work quickly. An inappropriate OS or programming language is used, but the developer forgets why these choices are inappropriate and gets comfortable with the system. The less-than-ideal choice has now become an integral part of the system.

**THE CONCURRENT DEVELOPMENT MODEL**



The Concurrent Development Model also called Concurrent Engineering, is represented as a

series of

* Framework activities
* Software engineering actions and tasks
* And their associated states

The figure provides a representation of one software engineering task within the modeling

activity for the concurrent process model. The activity-modeling-may be in any one of the states noted at a given time. Similarly other activities or tasks can be represented in an analogous manner. All activities exist concurrently but reside in different states.

For example

* Early in a project the communication activity has completed its first iteration and exists in the awaiting changes state.
* The modeling activity which exists in the none state while initial communication was completed, now makes a transition into the under development state.
* If customer indicates changes in the requirements, the modeling activity moves from the under development state into the awaiting changes state.

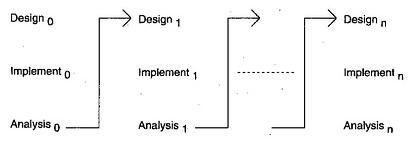
The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions or tasks. The concurrent

process model is applicable to all types of software development and provides an accurate

picture of the current state of a project. This model defines a network of activities. Events triggered at one point in the process network trigger transitions among the states.

**ITERATIVE MODELS**

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which can then be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software for each cycle of the model.



**Advantages of Iterative model:**

* In iterative model we can only create a high-level design of the application before we actually begin to build the product and define the design solution for the entire product. Later on we can design and built a skeleton version of that, and then evolved the design based on what had been built.
* In iterative model we are building and improving the product step by step. Hence we can track the defects at early stages. This avoids the downward flow of the defects.
* In iterative model we can get the reliable user feedback. When presenting sketches and blueprints of the product to users for their feedback, we are effectively asking them to imagine how the product will work.
* In iterative model less time is spent on documenting and more time is given for designing.

**Disadvantages of Iterative model:**

* Each phase of an iteration is rigid with no overlaps
* Costly system architecture or design issues may arise because not all requirements are gathered up front for the entire lifecycle

**SPECIALIZED PROCESS MODELS**

* Take on many of the characteristics of one or more of the conventional models
* Tend to be applied when a narrowly defined software engineering approach is chosen

**Examples:**

Component-Based Development

The Formal Methods Model

Aspect-Oriented Software Development

**Component-Based Development**

* Commercial off-the-shelf (COTS) software components can be used
* Components should have well-defined interfaces
* Incorporates many of the characteristics of the spiral model
* Evolutionary in nature
* Candidate components should be identified first
* Components can be designed as either conventional software modules or object-oriented classes or packages of classes.

**The Formal Methods Model**

* Encompasses a set of activities that leads to formal mathematical specification of computer software
* Have provision to apply a rigorous, mathematical notation
* Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily – not through ad hoc review, but through the application of mathematical analysis
* Offers the promise of defect-free software
* Clean room Software Engineering is a variation of the Formal Methods Model

**Critical Issues**

* The development of formal models is currently quite time-consuming and expensive
* Extensive training is required
* It is difficult to use the models as a communication mechanism for technically unsophisticated customers

**Aspect-Oriented Software Development (AOSD)**

* Certain “concerns” – customer required properties or areas of technical interest – span the entire s/w architecture
* Example “concerns”
* Security
* Fault Tolerance
* Task synchronization
* Memory Management
* When concerns cut across multiple system functions, features, and information, they are often referred to as crosscutting concerns
* Aspectual requirements define those crosscutting concerns that have impact across the s/w architecture
* AOSD or AOP (Aspect-Oriented Programming) provides a process and methodological approach for defining, specifying, designing, and constructing aspects – “mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern”
* A distinct aspect-oriented process has not yet matured
* It is likely that AOSD will adopt characteristics of both the spiral and concurrent process models.

**UNIFIED PROCESS MODEL**

The Unified Process recognizes the importance of customer communication and streamlined methods for describing the customer’s view of a system. It emphasizes the important role of software architecture and “helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse” [Jac99]. It suggests a process flow that is iterative andincremental, providing the evolutionary feel that is essential in modern software development

**Phases of Unified Process Model**

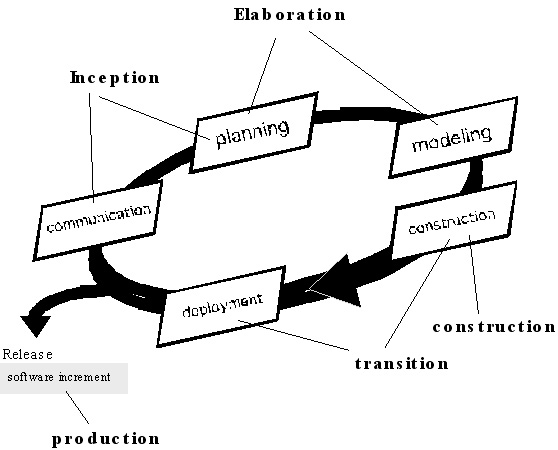
The **inception phase**of the UP encompasses both customer communication and planning activities. By collaborating with stakeholders, business requirements for the software are identified; a rough architecture for the system is proposed; and a plan for the iterative, incremental nature of the ensuing project is developed. Fundamental business requirements are described through a set of preliminary use cases that describe which features and functions each major class of users desires.

The **elaboration phase**encompasses the communication and modeling activities of the generic process model. Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model.

The **construction phase**of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.

The **transition phase**of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity. Software is given to end users for beta testing and user feedback reports both defects and necessary changes.

The **production phase**of the UP coincides with the deployment activity of the generic process. During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.



**ESTIMATION**

* S/W is the most expensive element of virtually all computer based systems
* The accuracy of a s/w project estimate is predicated on a number of things:
  + The degree to which the planner has properly estimated the size of the product to be built
  + The ability to translate the size estimate into human effort, calendar time, and dollars (required availability of past records)
  + The degree to which the project plan reflects the abilities of the s/w team
  + The stability of product requirements and the environment that supports the s/w engineering effort
* Sizing represents the project planner’s first major challenge
* Size refers to a quantifiable outcome of the s/w project (e.g. LOC and/or FP)

**Size-Oriented Metrics**

* LOC measure claim that LOC is an “artifact” of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists.
* The planner must estimate the LOC to be produced long before analysis and design has been completed.

**Function-Oriented Metrics**

* Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used function-oriented metric is the function point (FP). Computation of the function point is based on characteristics of the software’s information domain and complexity.
* The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach.
* Opponents claim that the method requires some “sleight of hand” in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be difficult to collect after the fact, and that FP has no direct physical meaning—it’s just a number.

**Reconciling LOC and FP Metrics**

The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation, an historical baseline of information must be established.

Within the context of process and project metrics, productivity and quality should be concerned which are the measures of software development “output” as a function of effort and time applied and measures of the “fitness for use” of the work products that are produced. For process improvement and project planning purposes, the interest is historical.

**Object-Oriented Metrics**

**Number of scenario scripts.** A scenario script is a detailed sequence of steps that describe the

interaction between the user and the application. Each script is organized into triplets of the form

{**initiator,** action, **participant**}

where **initiator** is the object that requests some service (that initiates a message), action is the result of the request, and **participant** is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

**Number of key classes.**Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

**Number of support classes.** Support classes are required to implement the system but are not immediately related to the problem domain.

**Average number of support classes per key class.** In general, key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified.

**Number of subsystems.** A subsystem is an aggregation of classes that support a function that is visible to the end user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

**Use-Case–Oriented Metrics**

Use cases are used widely as a method for describing customer-level or business domain requirements that imply software features and functions. It would seem reasonable to use the use case as a normalization measure similar to LOC or FP.

Like FP, the use case is defined early in the software process, allowing it to be used for estimation before significant modeling and construction activities are initiated. Use cases describe (indirectly, at least) user-visible functions and features that are basic requirements for a system. The use case is independent of programming language. In addition, the number of use cases is directly proportional to the size of the application in LOC and to the number of test cases that will have to be designed to fully exercise the application.

**WebApp Project Metrics**

**Number of static Web pages.** Web pages with static content (i.e., the end user has no control over the content displayed on the page) are the most common of all WebApp features.

**Number of dynamic Web pages.** Web pages with dynamic content (i.e., end-user actions or other external factors result in customized content displayed on the page) are essential in all e-commerce applications, search engines, financial applications, and many other WebApp categories.

**Number of internal page links.** Internal page links are pointers that provide a hyperlink to some other Web page within the WebApp.

**Number of persistent data objects.** One or more persistent data objects (e.g., a database or data file) may be accessed by a WebApp. As the number of persistent data objects grows, the complexity of the WebApp also grows and the effort to implement it increases proportionally.

**Number of external systems interfaced.** WebApps must often interface with “backroom” business applications. As the requirement for interfacing grows, system complexity and development effort also increase.

**Number of static content objects.** Static content objects encompass static text-based, graphical, video, animation, and audio information that are incorporated within the WebApp. Multiple content objects may appear on a single Web page.

**Number of dynamic content objects.** Dynamic content objects are generated based on end-user actions and encompass internally generated textbased, graphical, video, animation, and audio information that are incorporated within the WebApp. Multiple content objects may appear on a

single Web page.

**Number of executable functions.** An executable function (e.g., a script or applet) provides some computational service to the end user. As the number of executable functions increases, modeling and construction effort also increase.

A metric that reflects the degree of end-user customization that is required for the WebApp and correlate it to the effort expended on the project and/or the errors uncovered as reviews and testing are conducted can be defined as follows:

Nsp=number of static web pages

Ndp=number of dynamic web pages

Then,

Ndp

Customization index, C=---------------------------------

Ndp + Nsp

The value of C ranges from 0 to 1. As C grows larger, the level of WebApp customization becomes a significant technical issue.

**COst COnstructive Model(COCOMO)**

The Constructive Cost Model (COCOMO) is a software cost estimation model developed by Barry W. Boehm.

**Basic COCOMO**

Basic COCOMO computes software development effort (and cost) as a function of program size. Program size is expressed in estimated thousands of source lines of code ([SLOC](https://en.wikipedia.org/wiki/Source_lines_of_code), [KLOC](https://en.wikipedia.org/wiki/Source_lines_of_code#Related_terms)).

COCOMO applies to three classes of software projects:

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

The basic COCOMO equations take the form

**Effort Applied (E)** = ab(KLOC)bb **[**[**man-months**](https://en.wikipedia.org/wiki/Man-month)**]**

**Development Time (D)** = cb(Effort Applied)db **[months]**

**People required (P)** = Effort Applied / Development Time **[count]**

where, **KLOC** is the estimated number of delivered lines (expressed in thousands ) of code for project. The coefficients *ab*, *bb*, *cb* and *db* are given in the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Software project** | ***ab*** | ***bb*** | ***cb*** | ***db*** |
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semi-detached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

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

**Intermediate COCOMO**

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

* Product attributes
* Required software reliability
* Size of application database
* Complexity of the product
* Hardware attributes
* Run-time performance constraints
* Memory constraints
* Volatility of the virtual machine environment
* Required 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

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cost Drivers | **Ratings** | | | | | |
| Very Low | Low | Nominal | High | Very High | Extra High |
| **Product attributes** |  |  |  |  |  |  |
| Required software reliability | 0.75 | 0.88 | 1.00 | 1.15 | 1.40 |  |
| Size of application database |  | 0.94 | 1.00 | 1.08 | 1.16 |  |
| Complexity of the product | 0.70 | 0.85 | 1.00 | 1.15 | 1.30 | 1.65 |
| **Hardware attributes** |  |  |  |  |  |  |
| Run-time performance constraints |  |  | 1.00 | 1.11 | 1.30 | 1.66 |
| Memory constraints |  |  | 1.00 | 1.06 | 1.21 | 1.56 |
| Volatility of the virtual machine environment |  | 0.87 | 1.00 | 1.15 | 1.30 |  |
| Required turnabout time |  | 0.87 | 1.00 | 1.07 | 1.15 |  |
| **Personnel attributes** |  |  |  |  |  |  |
| Analyst capability | 1.46 | 1.19 | 1.00 | 0.86 | 0.71 |  |
| Applications experience | 1.29 | 1.13 | 1.00 | 0.91 | 0.82 |  |
| Software engineer capability | 1.42 | 1.17 | 1.00 | 0.86 | 0.70 |  |
| Virtual machine experience | 1.21 | 1.10 | 1.00 | 0.90 |  |  |
| Programming language experience | 1.14 | 1.07 | 1.00 | 0.95 |  |  |
| **Project attributes** |  |  |  |  |  |  |
| Application of software engineering methods | 1.24 | 1.10 | 1.00 | 0.91 | 0.82 |  |
| Use of software tools | 1.24 | 1.10 | 1.00 | 0.91 | 0.83 |  |
| Required development schedule | 1.23 | 1.08 | 1.00 | 1.04 | 1.10 |  |

The Intermediate Cocomo formula now takes the form:

**E=*ai* (KLoC)*(bi)*(EAF)**

where E is the effort applied in person-months, **KLoC** is the estimated number of thousands of delivered lines of code for the project, and **EAF** is the factor calculated above. The coefficient **ai** and the exponent **bi** are given in the next table.

|  |  |  |
| --- | --- | --- |
| **Software project** | **ai** | **bi** |
| Organic | 3.2 | 1.05 |
| Semi-detached | 3.0 | 1.12 |
| Embedded | 2.8 | 1.20 |

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

**Detailed COCOMO**

* Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.
* The detailed model uses different effort multipliers for each cost driver attribute. These **Phase Sensitive** effort multipliers are each to determine the amount of effort required to complete each phase. In detailed cocomo,the whole software is divided in different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort
* In detailed COCOMO, the effort is calculated as function of program size and a set of cost drivers given according to each phase of software life cycle.
* A Detailed project schedule is never static.

The Six phases of detailed COCOMO are:-

* plan and requirement.
* system design.
* detailed design.
* module code and test.
* integration and test.
* Cost Costructive Model

**COCOMO II**

Barry Boehm introduced COCOMO II (COst COnstructive MOdel) which is an hierarchy of estimation models that addresses the following areas:

* Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
* Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.
* Post-architecture-stage model. Used during the construction of the software.

Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

|  |  |  |  |
| --- | --- | --- | --- |
| **OBJECT TYPE** | **COMPLEXITY WEIGHT** | | |
| **SIMPLE** | **MEDIUM** | **DIFFICULT** |
| Screen | 1 | 2 | 3 |
| Report | 2 | 5 | 8 |
| 3GL component |  |  | 10 |

The object point is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application.

When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated and the object point count is adjusted:

NOP=(Object points)\*[(100-%reuse)/100]

where NOP is defined as new object points.

To derive an estimate of effort based on the computed NOP value, a “productivity rate” must be derived.

NOP

PROD=--------------------------

Person-month

for different levels of developer experience and development environment maturity.

Once the productivity rate has been determined, an estimate of project effort is computed using

NOP

Estimated effort=------------

PROD

In more advanced COCOMO II models, 12 a variety of scale factors, cost drivers, and adjustment procedures are required.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Developer’s experience/capability | Very low | Low | Nominal | High | Very high |
| Environment maturity/capability | Very low | Low | Nominal | High | Very high |
| PROD | 4 | 7 | 13 | 25 | 50 |

**PROJECT SCHEDULING**

Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of

software scheduling assumes that rough chronological bounds have been discussed but that the end date is set by the software engineering organization.

**Basic Principles**

* **Compartmentalization.** The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined.
* **Interdependency.** The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.
* **Time allocation.** Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.
* **Effort validation.** Every project has a defined number of people on the software team. As time allocation occurs, you must ensure that no more than the allocated number of people has been scheduled at any given time. For example, consider a project that has three assigned software engineers (e.g., three person-days are available per day of assigned effort4). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person-days of effort. More effort has been allocated than there are people to do the work.
* **Defined responsibilities.** Every task that is scheduled should be assigned to a specific team member.
* **Defined outcomes.** Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables.
* **Defined milestones.** Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality.

**The Relationship Between People and Effort**

*L,* is related to effort and development time by the equation:

L=P\*E1/3t4/3

Rearranging this software equation, we can arrive at an expression for development effort *E*:

L3

E=-------------------

P3 t4

**Effort Distribution**

A recommended distribution of effort across the software process is often referred to as the 40–20–40 rule. Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. 20 percent of effort is deemphasized in the coding.

Because of the effort applied to software design, code should follow with relatively little difficulty. A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort. The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical.

**Task Network**

**Task Set**

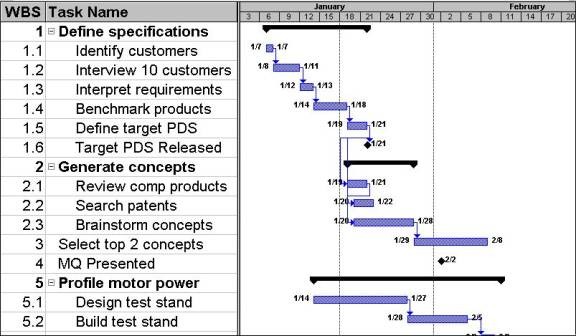
* A task set is the work breakdown structure for the project
* No single task set is appropriate for all projects and process models
  + It varies depending on the project type and the degree of rigor (based on influential factors) with which the team plans to work
* The task set should provide enough discipline to achieve high software quality
  + But it must not burden the project team with unnecessary work

**Purpose of a Task Network**

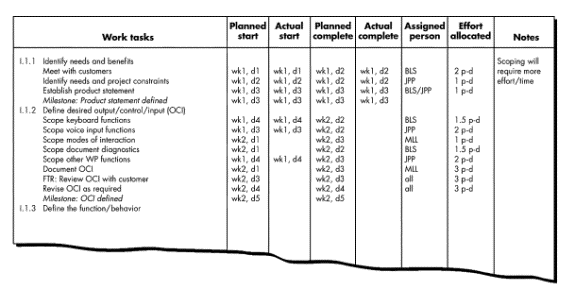
* Also called an activity network
* It is a graphic representation of the task flow for a project
* It depicts task length, sequence, concurrency, and dependency
* Points out inter-task dependencies to help the manager ensure continuous progress toward project completion
* The critical path
* A single path leading from start to finish in a task network
* It contains the sequence of tasks that must be completed on schedule if the project as a whole is to be completed on schedule
* It also determines the minimum duration of the project

**Timeline Chart**

* Also called a Gantt chart; invented by Henry Gantt, industrial engineer, 1917
* All project tasks are listed in the far left column
* The next few columns may list the following for each task: projected start date, projected stop date, projected duration, actual start date, actual stop date, actual duration, task inter-dependencies (i.e., predecessors)‏
* To the far right are columns representing dates on a calendar
* The length of a horizontal bar on the calendar indicates the duration of the task
* When multiple bars occur at the same time interval on the calendar, this implies task concurrency
* A diamond in the calendar area of a specific task indicates that the task is a milestone; a milestone has a time duration of zero

[](http://www.laynetworks.com/Software%20Engineering_2b.htm)

**Project Table**



**EARNED VALUE ANALYSIS**

The earned value system provides a common value scale for every [software project] task, regardless of the type of work being performed. The total hours to do the whole project are estimated, and every task is given an earned value based on its estimated percentage of the total.

To determine the earned value, the following steps are performed:

* The budgeted cost of work scheduled (BCWS) is determined for each work task represented in the schedule.
* The BCWS values for all work tasks are summed to derive the budget at completion (BAC). Hence,

BAC=∑(BCWSk) for all task k

* Next, the value for budgeted cost of work performed (BCWP) is computed. The value for BCWP is the sum of the BCWS values for all work tasks that have actually been completed by a point in time on the project schedule.

Given values for BCWS, BAC, and BCWP, important progress indicators can be computed:

BCWP

Schedule performance index, SPI = -------------

BCWS

Schedule Variance, SV = BCWP-BCWS

BCWS

Percent scheduled for completion=--------------------

BAC

BCWP

Percent complete=----------------

BAC

BCWP

Cost performance index,CPI=---------------

ACWP

Cost variance, CV=BCWP-ACWP

A CPI value close to 1.0 provides a strong indication that the project is within its defined budget. CV is an absolute indication of cost savings (against planned costs) or shortfall at a particular stage of a project. Like over-the-horizon radar, earned value analysis illuminates scheduling difficulties before they might otherwise be apparent. This enables you to take corrective action before a project crisis develops.

**RISK MANAGEMENT**

Risk concerns with failure happenings. Risk involves change, such as in changes in mind,

opinion, actions, or places.

* uncertainty—the risk may or may not happen;
* loss—if the risk becomes a reality, unwanted consequences or losses will occur

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems.

Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are

* building an excellent product or system that no one really wants (market risk),
* building a product that no longer fits into the overall business strategy for the company (strategic risk),
* building a product that the sales force doesn’t understand how to sell (sales risk),
* losing the support of senior management due to a change in focus or a change in people (management risk), and
* losing budgetary or personnel commitment (budget risks).

Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

**Risk Identification**

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories: generic risks and product-specific risks.

* Generic risks are a potential threat to every software project.
* Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

* Product size—risks associated with the overall size of the software to be built or modified.
* Business impact—risks associated with constraints imposed by management or the marketplace.
* Stakeholder characteristics—risks associated with the sophistication of the stakeholders and the developer’s ability to communicate with stakeholders in a timely manner.
* Process definition—risks associated with the degree to which the software process has been defined and is followed by the development organization.
* Development environment—risks associated with the availability and quality of the tools to be used to build the product.
* Technology to be built—risks associated with the complexity of the system to be built and the “newness” of the technology that is packaged by the system.
* Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

**Risk Projection**

Risk projection, also called risk estimation, attempts to rate each risk in two ways— the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps:

* Establish a scale that reflects the perceived likelihood of a risk.
* Delineate the consequences of the risk.
* Estimate the impact of the risk on the project and the product.
* Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

A risk table provides you with a simple technique for risk projection.

The overall risk exposure RE is determined using the following relationship:

RE =P \* C

**Risk Refinement**

One way to do this is to represent the risk in condition-transition-consequence (CTC) format. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

This general condition can be refined in the following manner:

**Subcondition 1.** Certain reusable components were developed by a third party with no knowledge of internal design standards.

**Subcondition 2.** The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

**Subcondition 3.** Certain reusable components have been implemented in a language that is not supported on the target environment.

**Risk Mitigation,Monitoring and Management**

An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

To mitigate this risk, a strategy has to be developed for reducing turnover. Among the possible steps to be taken are:

* Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
* Mitigate those causes that are under your control before the project starts.
* Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
* Organize project teams so that information about each development activity is widely dispersed.
* Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
* Conduct peer reviews of all work (so that more than one person is “up to speed”).
* Assign a backup staff member for every critical technologist.