SPM

SOFTWARE PROJECT MANAGEMENT

FALL SEMESTER, YEAR (VIth, 3rd)

FALL SESSION (2021-22)

(SPM)
MS. SHWETA TIWARI
Published: April 20, 2022

UNIT-2

SPM

SOFTWARE PROJECT MANAGEMENT

By SHWETA TIWARI

SOFTWARE PROJECT MANAGEMENT

UNIT II - PROJECT LIFE CYCLE AND EFFORT ESTIMATION

UNIT II - PROJECT LIFE CYCLE AND EFFORT ESTIMATION

2.1 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 representation of its life cycle.

2.2 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 constraints around them.

2. S**oftware design and implementation**: The software is to be designed and programmed.
3. **Software verification and validation**: The software must conform to its specification and meet the customer needs.
4. **Software evolution** (software maintenance): The software is being modified to meet customer and market requirements changes.

The various Process Models are Waterfall

Model

Spiral Model Prototype

Model Incremental

Delivery

2.2.1 Waterfall model

The waterfall model is a sequential approach, where each fundamental activity of a process is represented 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 change the process to reflect requirement changes.

The phases of the waterfall model are:

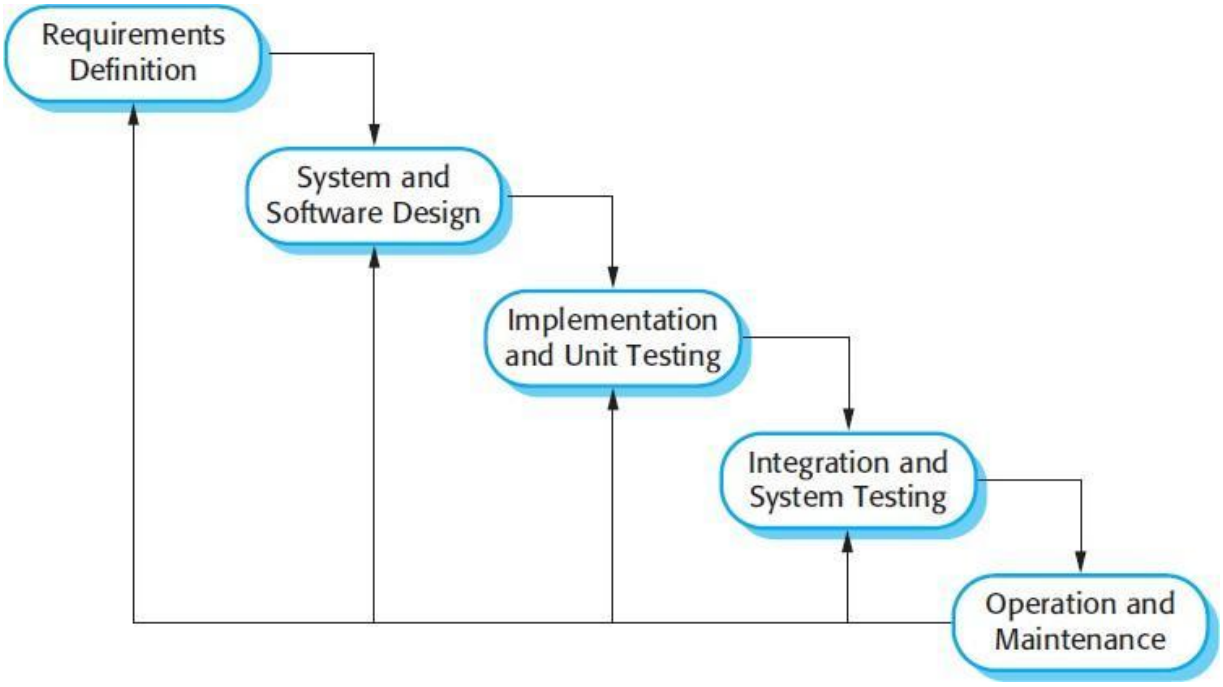
Requirements

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 process 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.

2.2.2 Spiral Model

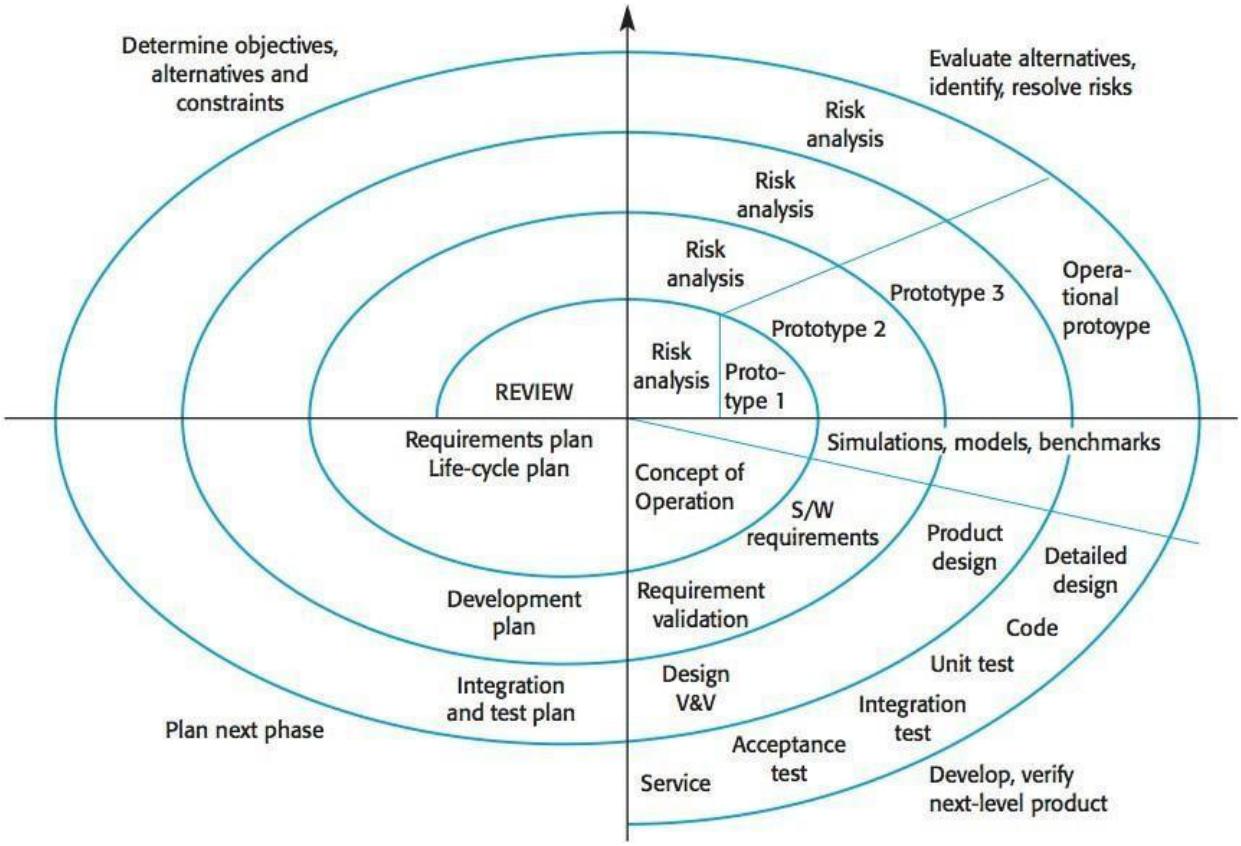
The spiral model is similar to the incremental model, with more emphasis placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Engineering and Evaluation. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are 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 'Business 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.

2.2.3 Software Prototyping

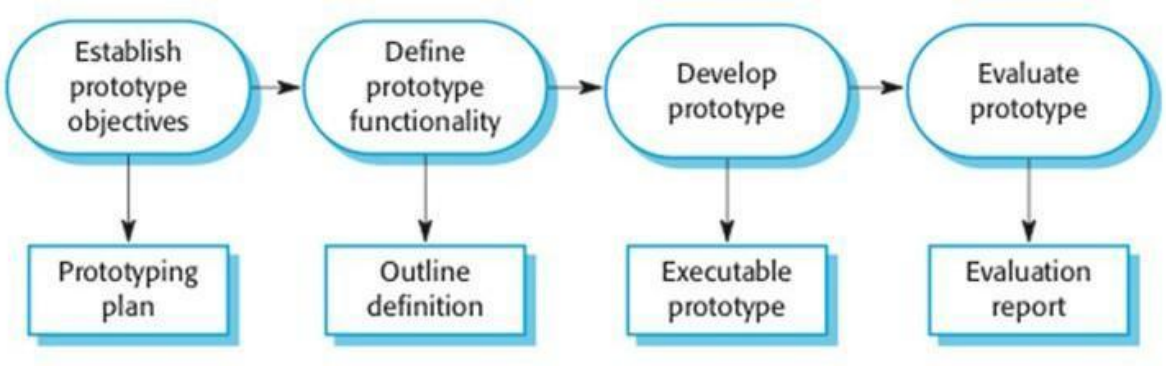
A prototype is a version of a system or part of the system that's developed quickly to check the customer's requirements or feasibility of some design decisions.So, a prototype is useful when a customer or developer is not sure of the requirements, or of algorithms, efficiency, business rules, response time, etc.

In prototyping, the client is involved throughout the development process, which increases the 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 in the requirements. The specification may then be modified to reflect the changes.

[2] In the **system design**, a prototype can help to carry out design experiments to check the feasibility of 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.



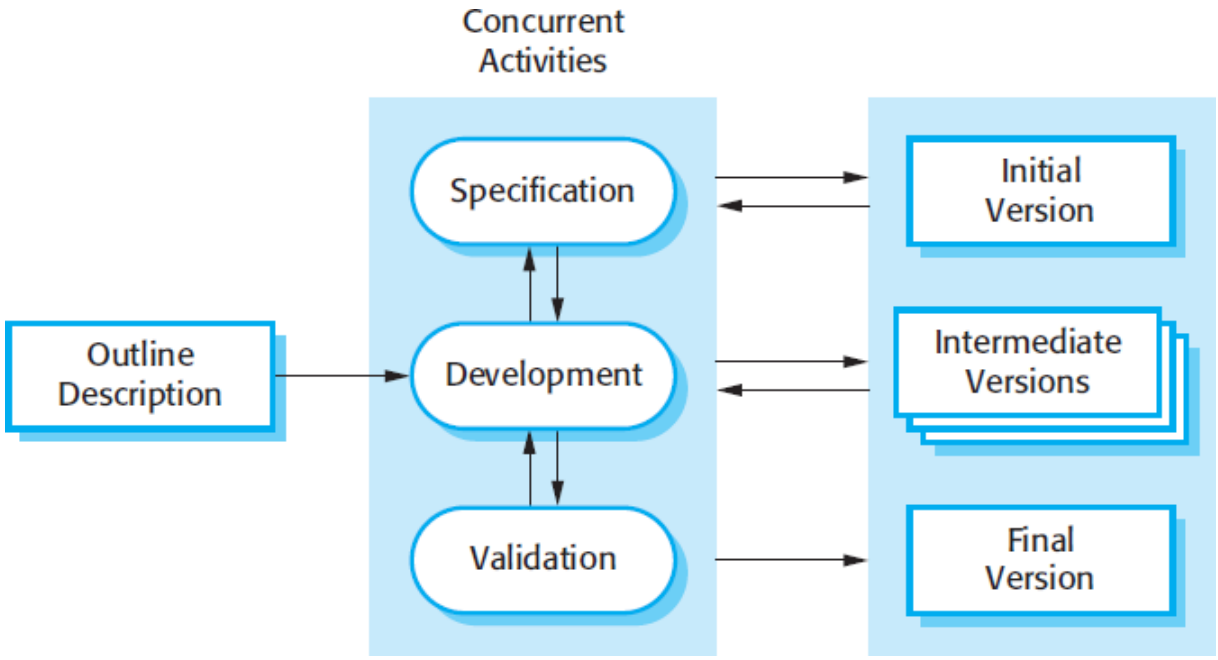
The phases of a prototype are:

1. **Establish objectives**: The objectives of the prototype should be made explicit from the start of the 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 ignore some functionality, such as response time and memory utilization unless they are relevant to the objective of the prototype.
3. **Develop the prototype**: The initial prototype is developed that includes only user interfaces.
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 approach to be used in the context of a full methodology (such as incremental, spiral, etc).

2.3 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 software as 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 required with the 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 possible with the waterfall model.

2.4 Rapid Application Development

The RAD model is the Rapid Application Development model. It is a type of incremental model. In the 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. Descriptions 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 the 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 the cost of modeling and automated code generation is very high.

2.5 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

Regardless of which Agile methodology a team adopts, the benefits cannot be fully realized without the commitment of everyone involved.

2.5.1 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 process is 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 testing can 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 structure that 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

2.6 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

the whole is built up step by step, each successive version consisting of the previous ones plus a number of new functions.

2.7 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

2.8 Effort and Cost estimation techniques

The Effort and Cost estimation techniques are

- *Algorithmic models*, which use effort drivers representing characteristics of the target system and the implementation environment to predict effort.

- *Expert judgment*, based on the advice of knowledgeable staff.

- *Analogy*, where a similar, completed project is identified and its actual effort is used as the basis of the estimate.

- *Parkinson*, where the staff effort available to do a project becomes the estimate.

- *Price to win*, where the estimate is a figure that seems sufficiently low to win a contract.

- *Top-down*, where an overall estimate for the whole project is broken down into the effort required for component tasks.

- *Bottom-up*, where component tasks are identified and sized and these individual estimates are aggregated.

2.9 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' to the 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 are identified 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 offers a 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.

2.10 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 major extension to the original COCOMO (COCOMO 81) model published in 1981. It consists of three submodels, each one offering increased fidelity the further along one is in the project planning and design process. Listed in increasing fidelity, these submodels 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 the software 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 spent creating the software product.

These changes and others began to make applying the original COCOMO model problematic. The solution 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.

2.11 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 patterns of R&D types of projects change dynamically over time for efficient man power utilization.

He concluded that staffing patterns 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 number 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.