UNIT II PROJECT LIFE CYCLE AND EFFORT ESTIMATION

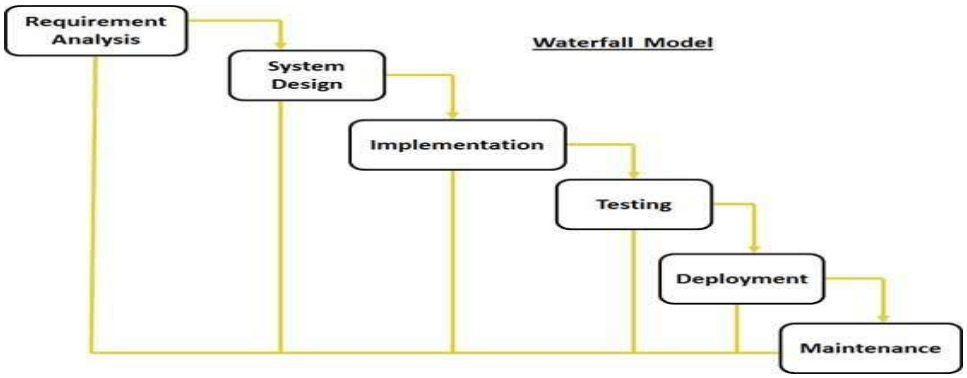
Software process and Process Models – Choice of Process models - mental delivery – Rapid Application development – Agile methods – Extreme Programming – SCRUM – Managing interactive processes – Basics of Software estimation – Effort and Cost estimation techniques – COSMIC Full function points - COCOMO II A Parametric Productivity Model - Staffing Pattern

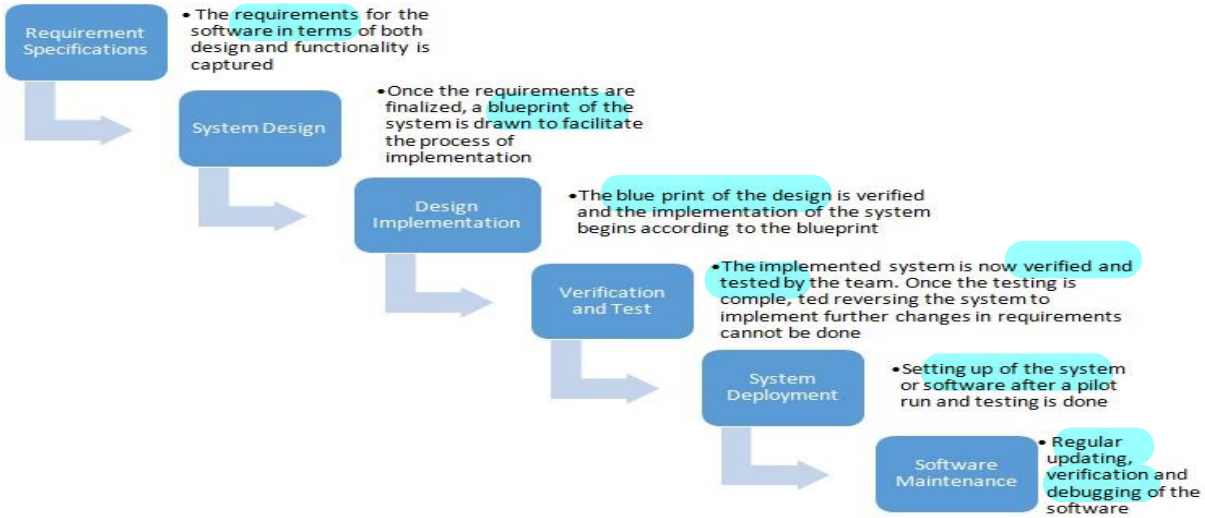
SOFTWARE PROCESS AND PROCESS MODELS :

- **Waterfall model**
- **V-Process model**
- **Spiral model**
- **Incremental Delivery**
- **Software prototype**

Waterfall model:

- **Waterfall Model** is a design process that is used in software development.
- It is a linear sequential model that is most effective when the problem statement is well defined and highly structured and all the requirements are known before the commencement of the project.
- All the essential activities that comprise the software development process are listed in a sequential manner.
- Development of software thus happens from one phase to another through a series of non-overlapping phases.
- Also known as one-shot or once-through model.
- Managers can review project progress to see whether the business case for the project is still valid.This is sometimes referred to as the stage-gate model.





Requirement Analysis & Definition:

- All requirements of the system which has to be developed are collected in this step.
- Like in other process models requirements are split up in functional requirements and constraints which the system has to fulfil.
- Requirements have to be collected by analysing the needs of the end user(s) and checking them for validity and the possibility to implement them.
- The aim is to generate a Requirements Specification Document which is used as an input for the next phase of the model.

System Design:

- The system has to be properly designed before any implementation is started.
- This involves an architectural design which defines and describes the main blocks and components of the system, their interfaces and interactions.
- By this the needed hardware is defined and the software is split up in its components.
- E.g. this involves the definition or selection of a computer platform, an operating system, other peripheral hardware, etc.
- The software components have to be defined to meet the end user requirements and to meet the need of possible scalability of the system.
- The aim of this phase is to generate a System Architecture Document this serves as an input for the software design phase of the development, but also as an input for hardware design or selection activities.
- Usually in this phase various documents are generated, one for each discipline, so that the software usually will receive a software architecture document.

Software Design:

- Based on the system architecture which defines the main software blocks the software design will break them further down into code modules.
- The interfaces and interactions of the modules are described, as well as their functional contents.
- All necessary system states like startup, shutdown, error conditions and diagnostic modes have to be considered and the activity and behaviour of the software has to be defined.
- The output of this phase is a Software Design Document which is the base of the following implementation work.

Coding:

- Based on the software design document the work is aiming to set up the defined modules or units and actual coding is started.
- The system is first developed in smaller portions called units.
- They are able to stand alone from an functional aspect and are integrated later on to form the complete software package.

Software Integration & Verification:

- Each unit is developed independently and can be tested for its functionality.
- This is the so called Unit Testing. It simply verifies if the modules or units to check if they meet their specifications.
- This involves functional tests at the interfaces of the modules, but also more detailed tests which consider the inner structure of the software modules.
- During integration the units which are developed and tested for their functionalities are brought together.
- The modules are integrated into a complete system and tested to check if all modules cooperate as expected.

System Validation:

- After successfully integration including the related tests the complete system has to be tested against its initial requirements.
- This will include the original hardware and environment, whereas the previous integration and testing phase may still be performed in a different environment or on a test bench.

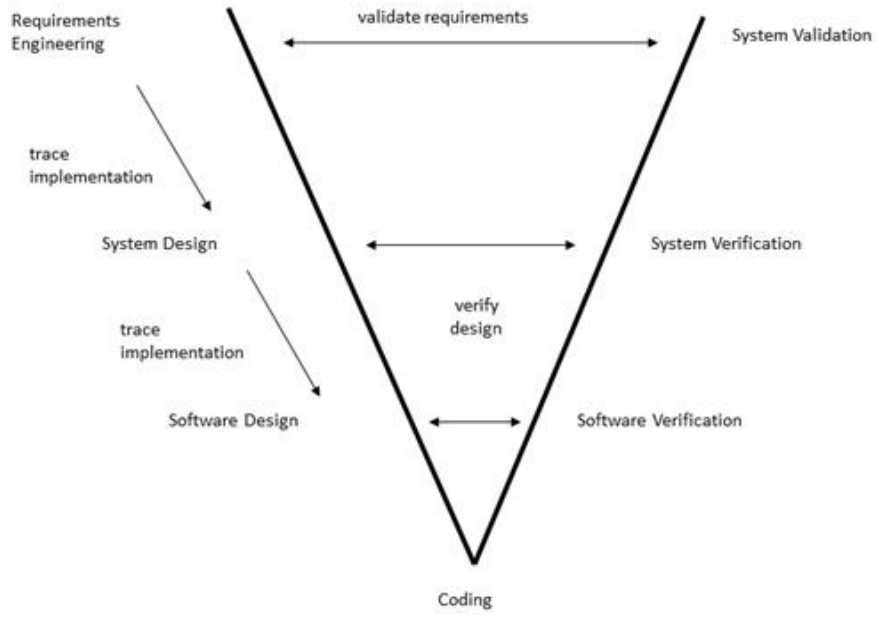
Operation & Maintenance:

- The system is handed over to the customer and will be used the first time by him.
- Naturally the customer will check if his requirements were implemented as expected but he will also validate if the correct requirements have been set up in the beginning.
- In case there are changes necessary it has to be fixed to make the system usable or to make it comply to the customer wishes. In most of the "Waterfall Model" descriptions this phase is extended to a never ending phase of "Operations & Maintenance".
- All the problems which did not arise during the previous phases will be solved in this last phase.

The weakness of the Waterfall Model is at hand:

- It is very important to gather all possible requirements during the first phase of requirements collection and analysis. If not all requirements are obtained at once the subsequent phases will suffer from it. Reality is that only a part of the requirements is known at the beginning and a certain percentage will be gathered during the complete development time.
- Iterations are only meant to happen within the same phase or at best from the start of the subsequent phase back to the previous phase. If the process is kept according to the school book this tends to shift the solution of problems into later phases which eventually results in a bad system design. Instead of solving the root causes the tendency is to patch problems with inadequate measures.
- There may be a very big "Maintenance" phase at the end. The process only allows for a single run through the waterfall. Eventually this could be only a first sample phase which means that the further development is squeezed into the last never ending maintenance phase and virtually run without a proper process.

V Process model:

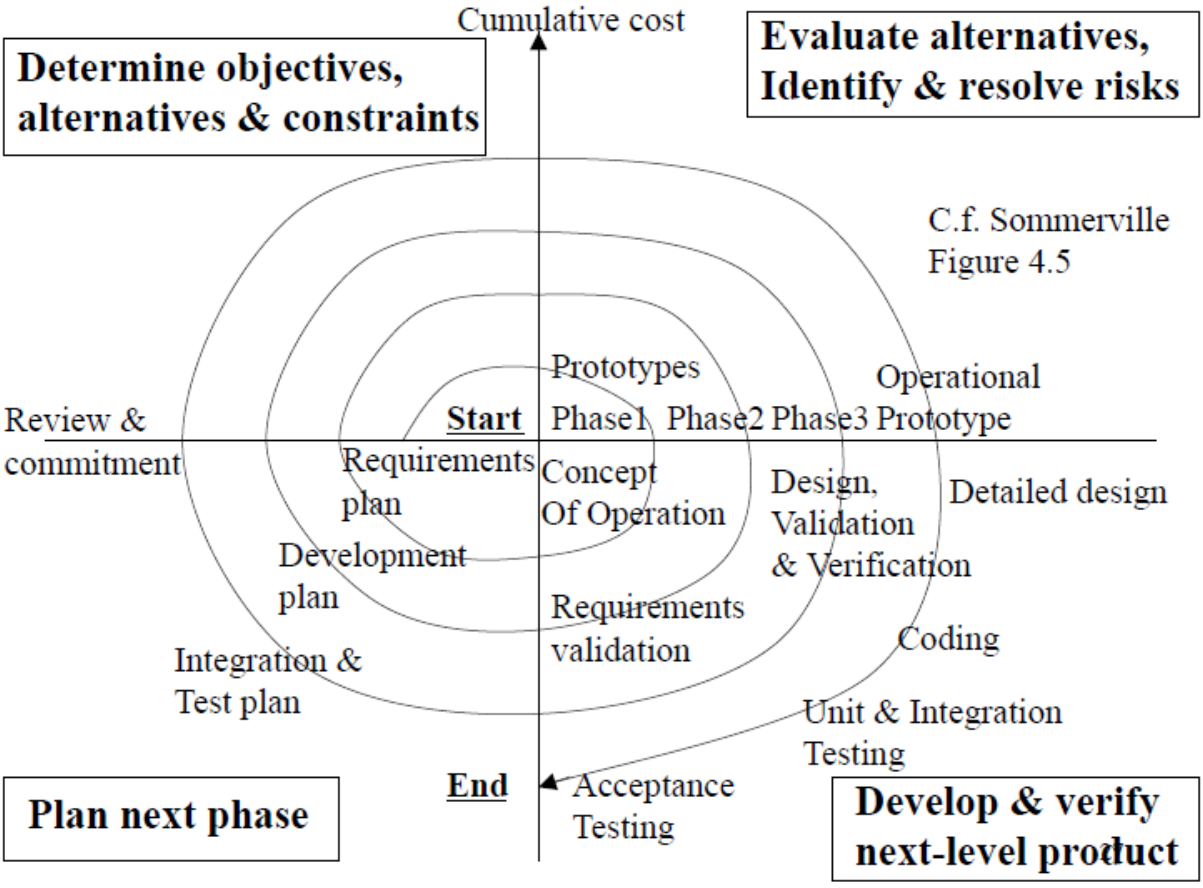


- V - Model is an extension of the waterfall model and is based on association of a testing phase for each corresponding development stage.
- This means that for every single phase in the development cycle there is a directly associated testing phase.

- This is a highly disciplined model and next phase starts only after completion of the previous phase

Spiral model:

- Extends waterfall model by adding iteration to explore/manage risk
- Project risk is a moving target.
- Each loop of the spiral is divided into four quadrants,indicating four stages in each phase.
  - In the first stage of a phase,one or more features of the product are analysed
  - Those features are identified and resolved through prototyping.
  - In the third stage identified features are implemented using waterfall model.
  - In the fourth and final stage,the developed increment is reviewed by the customer .
- In 1988 Boehm developed the spiral model as an iterative model which includes risk analysis and risk management.
- Key idea: on each iteration identify and solve the sub-problems with the highest risk.



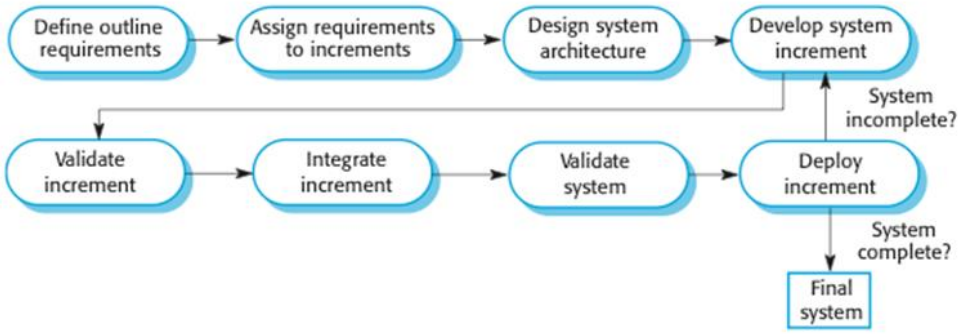
Advantages

1. Realism: the model accurately reflects the iterative nature of software development on projects with unclear requirements
2. Flexible: incoporates the advantages of the waterfall and evolutionary methods
3. Comprehensive model decreases risk
4. Good project visibility.

Disadvantages

1. Needs technical expertise in risk analysis and risk management to work well.
2. Model is poorly understood by nontechnical management, hence not so widely used
3. Complicated model, needs competent professional management. High administrative overhead.

Incremental Delivery:



- The incremental build model is a method of software development where the product is designed, implemented and tested incrementally (a little more is added each time) until the product is finished. It involves both development and maintenance. The product is defined as finished when it satisfies all of its requirements. This model combines the elements of the waterfall model with the iterative philosophy of prototyping.
- The product is decomposed into a number of components, each of which is designed and built separately (termed as builds). Each component is delivered to the client when it is complete. This allows partial utilization of the product and avoids a long development time. It also avoids a large initial capital outlay and subsequent long waiting period. This model of development also helps ease the traumatic effect of introducing a completely new system all at once.
- The incremental model applies the waterfall model incrementally.[1]
- The series of releases is referred to as "increments", with each increment providing more functionality to the customers. After the first increment, a core product is delivered, which can already be used by the customer. Based on customer feedback, a plan is developed for the next increments, and modifications are made accordingly. This process continues, with increments being delivered until the complete product is delivered. The incremental philosophy is also used in the agile process model (see agile modeling).[1]
- The Incremental model can be applied to DevOps. In DevOps it centers around the idea of minimizing risk and cost of a DevOps adoption whilst building the necessary in-house skillset and momentum.

Advantages

1. After each iteration, regression testing should be conducted. During this testing, faulty elements of the software can be quickly identified because few changes are made within any single iteration.
2. It is generally easier to test and debug than other methods of software development because relatively smaller changes are made during each iteration. This allows for more targeted and rigorous testing of each element within the overall product.
3. Customer can respond to features and review the product for any needed or useful changes.
4. Initial product delivery is faster and costs less.

Disadvantages

1. Resulting cost may exceed the cost of the organization.
2. As additional functionality is added to the product, problems may arise related to system architecture which were not evident in earlier prototypes.

Software prototyping :

- Software prototyping is the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed.
- The Software Prototyping refers to building software application prototypes which display the functionality of the product under development but may not actually hold the exact logic of the original software.
    - Prototype is a working model of software with some limited functionality.
    - The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation.
    - Prototyping is used to allow the users evaluate developer proposals and try them out before implementation.
    - It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

There is typically a four-step process for prototyping:

- **Identify initial requirements:** In this step, the software publisher decides what the software will be able to do. They publisher considers who the user will likely be and what the user will want from the product, then the publisher sends the project and specifications to a software designer or developer.
- **Develop initial prototype:** In step two, the developer will consider the requirements as proposed by the publisher and begin to put together a model of what the finished product might look like.
- **Review:** Once the prototype is developed, the publisher has a chance to see what the product might look like - how the developer has envisioned the publisher's specifications. In more advanced prototypes, the end consumer may have an opportunity to try out the product and offer suggestions for improvement. This is what we know of as beta testing.
- **Revise:** The final step in the process is to make revisions to the prototype based on the feedback of the publisher and/or beta testers.

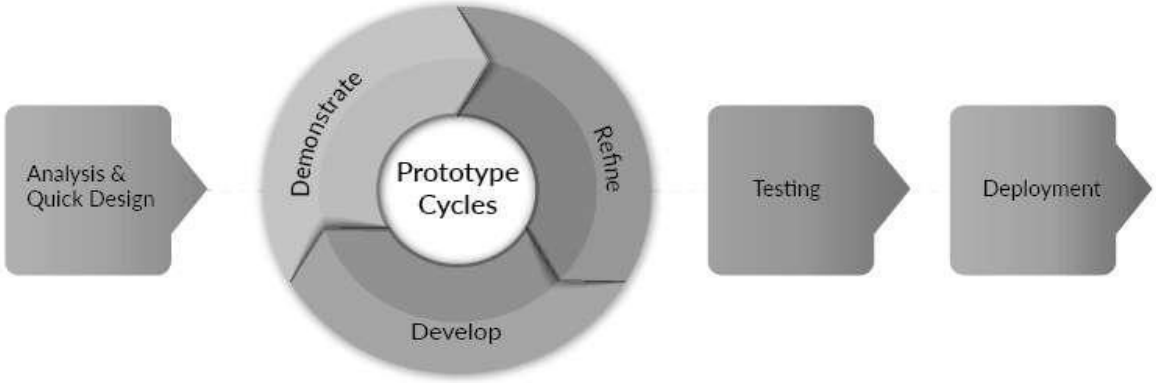
Models of Prototyping

There are two main models for prototypes:

- The **throwaway model** is designed to be thrown away once the review process has been completed. It is just a look at what the end product may look like, and it's typically not well defined and may only have a few of the publisher's requirements mapped out.
- The **evolutionary model** for prototyping is more complete and is incorporated into the final product. The revisions in step four are made directly to the prototype in order to get it to the final stage.

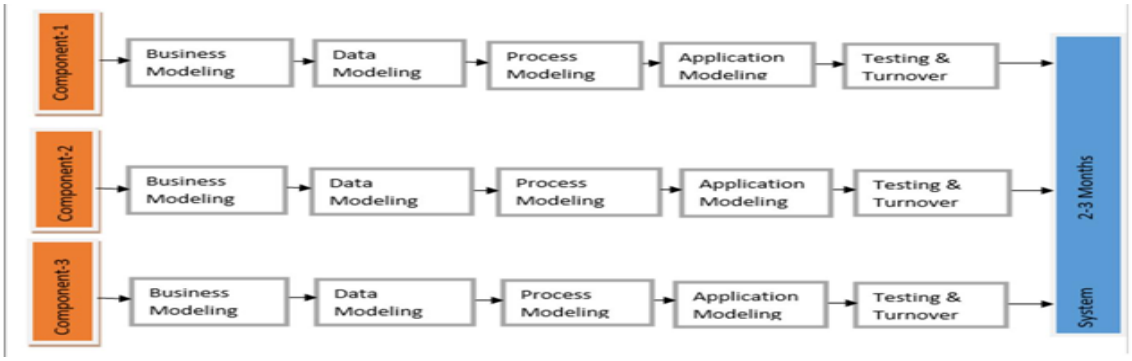
Rapid Application development:

- Rapid Application Development model relies on prototyping and rapid cycles of iterative development to speed up development and elicit early feedback from business users.
- After each iteration, developers can refine and validate the features with stakeholders.
- RAD model is also characterized by reiterative user testing and the re-use of software components. Hence, RAD has been instrumental in reducing the friction points in delivering successful enterprise applications.
- WaveMaker makes use of the RAD model to provide a Rapid Application Development platform to create web and mobile applications.
- The following diagram depicts WaveMaker RAD platform architecture, based on the MVC (Model-View-Controller) pattern. Open standards, easy customization and rapid prototyping are central to the platform.



Phases in RAD Model:

- Business Modeling
- Data Modeling
- Process Modeling
- Application Modeling
- Testing and Turnover



Business Modeling: In this phase of development business model should be designed based on the information available from different business activities. Before start the development there should be a complete picture of business process functionality.

Data Modeling: Once the business modeling phase over and all the business analysis completed, all the required and necessary data based on business analysis are identified in data modeling phase.

Process Modeling: All the data identified in data modeling phase are planned to process or implement the identified data to achieve the business functionality flow. In this phase all the data modification process is defined.

Application Modeling: In this phase application id developed and coding completed. With help of automation tools all data implemented and processed to work as real time. **Testing and turnover:** All the testing activates are performed to test the developed application.

Advantages of RAD Model:

- Fast application development and delivery.
- Lest testing activity required.
- Visualization of progress.
- Less resources required.
- Review by the client from the very beginning of development so very less chance to miss the requirements.
- Very flexible if any changes required.
- Cost effective.
- Good for small projects.

Disadvantages of RAD Model:

- High skilled resources required.
- On each development phase client's feedback required.
- Automated code generation is very costly.
- Difficult to manage.
- Not a good process for long term and big projects.
- Proper modularization of project required.

Agile method:

Agile software development methodology is a process for developing software (like other software development methodologies – Waterfall model, V-Model, Iterative model etc.)

In English, Agile means 'ability to move quickly and easily' and responding swiftly to change – this is a key aspect of Agile software development as well.

Overview:

- In traditional software development methodologies like Waterfall model, a project can take several months or years to complete and the customer may not get to see the end product until the completion of the project.
- At a high level, non-Agile projects allocate extensive periods of time for Requirements gathering, design, development, testing and UAT, before finally deploying the project.
- In contrast to this, Agile projects have Sprints or iterations which are shorter in duration (Sprints/iterations can vary from 2 weeks to 2 months) during which pre-determined features are developed and delivered.
- Agile projects can have one or more iterations and deliver the complete product at the end of the final iteration.

Example of Agile software development

Example: Google is working on project to come up with a competing product for MS Word, that provides all the features provided by MS Word and any other features requested by the marketing team. The final product needs to be ready in 10 months of time. Let us see how this project is executed in traditional and Agile methodologies.

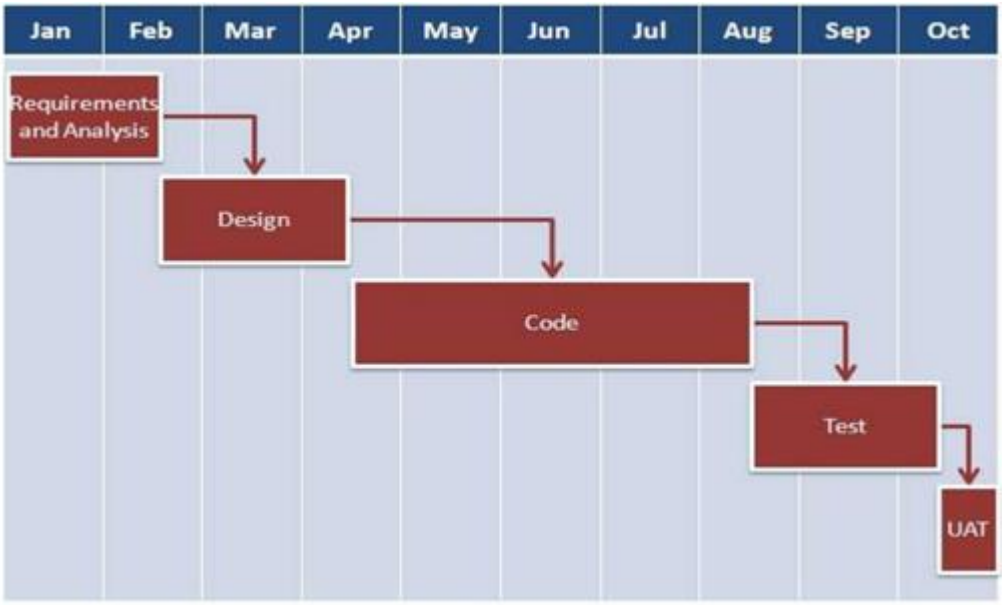
In traditional Waterfall model –

- At a high level, the project teams would spend 15% of their time on gathering requirements and analysis (1.5 months)
- 20% of their time on design (2 months)
- 40% on coding (4 months) and unit testing
- 20% on System and Integration testing (2 months).
- At the end of this cycle, the project may also have 2 weeks of User Acceptance testing by marketing teams.
- In this approach, the customer does not get to see the end product until the end of the project, when it becomes too late to make significant changes.

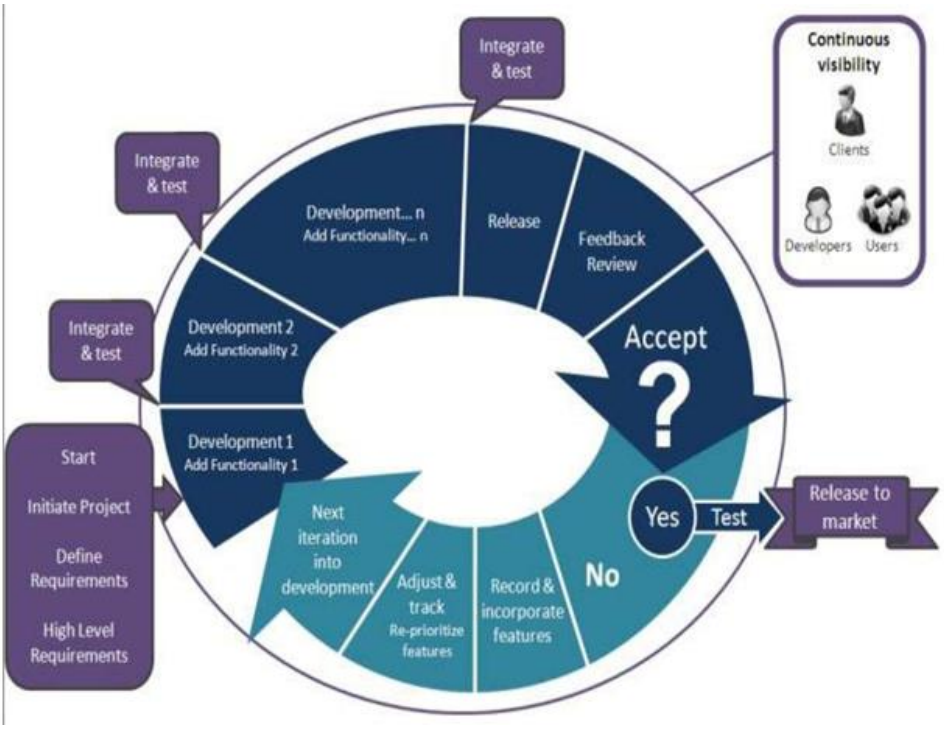
A**gile development** methodology –

- In the **Agile methodology**, each project is broken up into several 'Iterations'.
- All Iterations should be of the same time duration (between 2 to 8 weeks).
- At the end of each iteration, a working product should be delivered.
- In simple terms, in the Agile approach the project will be broken up into 10 releases (assuming each iteration is set to last 4 weeks).

- Rather than spending 1.5 months on requirements gathering, in Agile software development, the team will decide the basic core features that are required in the product and decide which of these features can be developed in the first iteration.
- Any remaining features that cannot be delivered in the first iteration will be taken up in the next iteration or subsequent iterations, based on priority.
- At the end of the first iterations, the team will deliver a working software with the features that were finalized for that iteration.

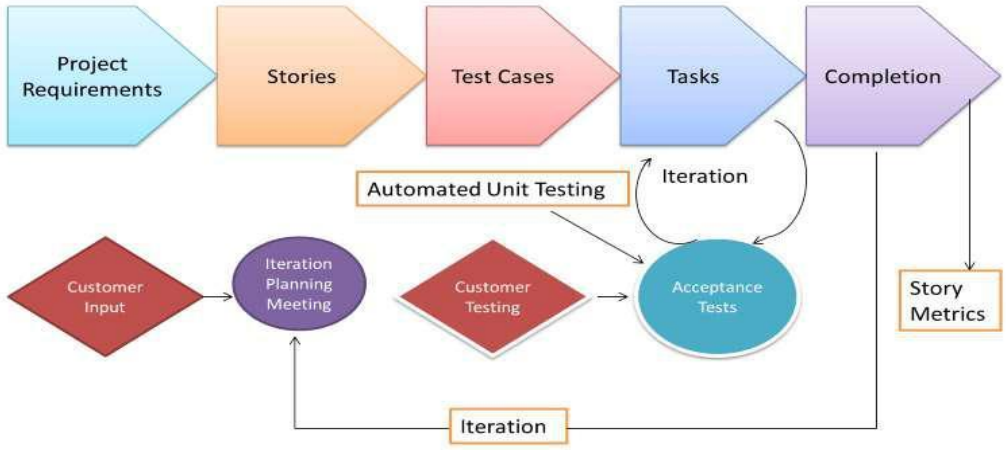


There will be 10 iterations and at the end of each iteration the customer is delivered a working software that is incrementally enhanced and updated with the features that were shortlisted for that iteration.



Extreme Programming:

Extreme Programming technique is very helpful when there is constantly changing demands or requirements from the customers or when they are not sure about the functionality of the system. It advocates frequent "releases" of the product in short development cycles, which inherently improves the productivity of the system and also introduces a checkpoint where any customer requirements can be easily implemented. The XP develops software keeping customer in the target.



Extreme Programming (XP)

Phases of eXtreme programming:

There are 6 phases available in Agile XP method, and those are explained as follows:

Planning

- Identification of stakeholders and sponsors
- Infrastructure Requirements
- Security related information and gathering
- Service Level Agreements and its conditions

Analysis

- Capturing of Stories in Parking lot
- Prioritize stories in Parking lot
- Scrubbing of stories for estimation
- Define Iteration SPAN(Time)
- Resource planning for both Development and QA teams

Design

- Break down of tasks
- Test Scenario preparation for each task
- Regression Automation Framework

Execution

- Coding
- Unit Testing
- Execution of Manual test scenarios
- Conversion of Manual to Automation regression test cases
- Mid Iteration review
- End of Iteration review

Wrapping

- Small Releases
- Regression Testing
- Demos and reviews
- Develop new stories based on the need

- Process Improvements based on end of iteration review comments

Closure

- Pilot Launch
- Training
- Production Launch
- SLA Guarantee assurance
- Review SOA strategy
- Production Support

There are two storyboards available to track the work on a daily basis, and those are listed below for reference.

Story Cardboard

This is a traditional way of collecting all the stories in a board in the form of stick notes to track daily XP activities. As this manual activity involves more effort and time, it is better to switch to an online form.

Online Storyboard

Online tool Storyboard can be used to store the stories. Several teams can use it for different purposes.

Roles

Customer

- Writes User Stories and specifies Functional Tests
- Sets priorities, explains stories
- May or may not be an end-user
- Has authority to decide questions about the stories

Programmer

- Estimates stories
- Defines Tasks from stories, and estimates
- Implements Stories and Unit Tests

Coach

- Watches everything, makes sure the project stays on course
- Helps with anything Tracker
- Monitors Programmers progress, takes action if things seem to be going off track.
- Actions include setting up a meeting with Customer, asking Coach or another Programmer to help

Tester

- Implements and runs Functional Tests (not Unit Tests!)
- Graphs results, and makes sure people know when test results decline. Doomsayer
- Ensures that everybody knows the risks involved
- Ensures that bad news isn't hidden, glossed over, or blown out of proportion

Manager

- Schedules meetings (e.g. Iteration Plan, Release Plan), makes sure the meeting process is followed, records results of meeting for future reporting, and passes to the Tracker
- Possibly responsible to the Gold Owner.
- Goes to meetings, brings back useful information

Gold Owner

- The person funding the project, which may or may not be the same as the Customer

SCRUM:

- SCRUM is an agile development method which concentrates specifically on how to manage tasks within a team based development environment. Basically, Scrum is derived from activity that occurs during a rugby match. Scrum believes in empowering the development team and advocates working in small teams (say- 7 to 9 members).
- It consists of three roles, and their responsibilities are explained as follows

Scrum Master

Master is responsible for setting up the team, sprint meeting and removes obstacles to progress

Product owner

The Product Owner creates product backlog, prioritizes the backlog and is responsible for the delivery of the functionality at each iteration

Scrum Team

Team manages its own work and organizes the work to complete the sprint or cycle

Product Backlog

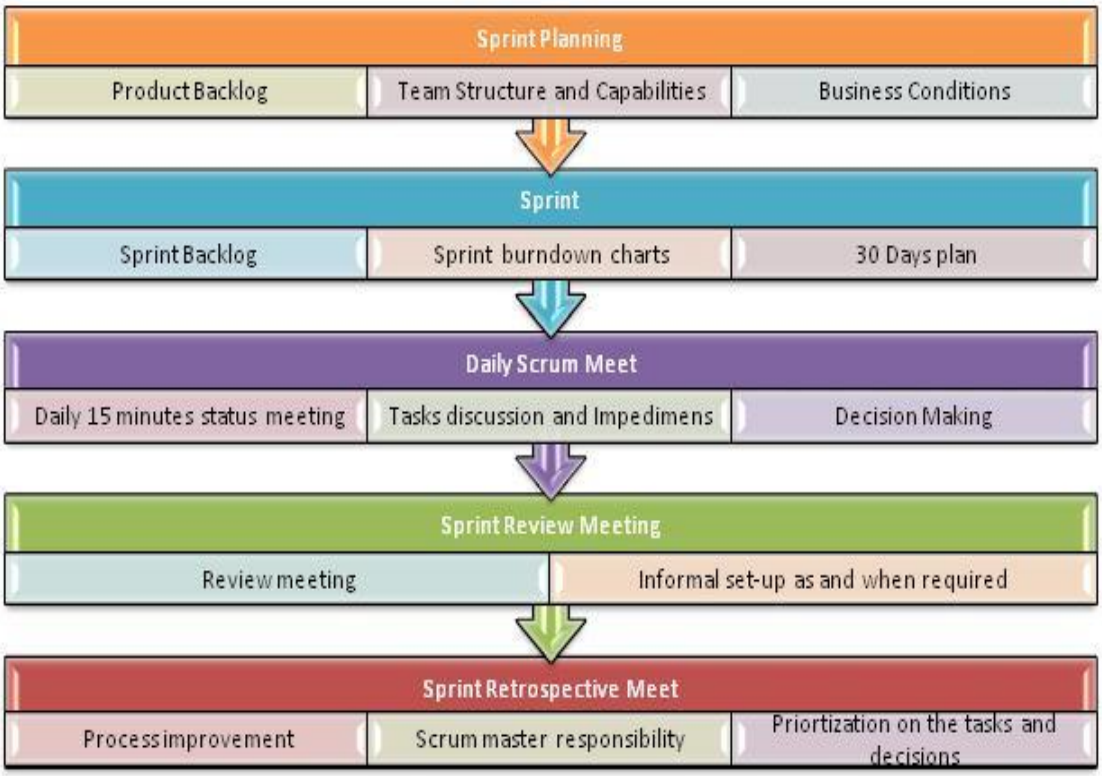
This is a repository where requirements are tracked with details on the no of requirements to be completed for each release. It should be maintained and prioritized by product owner, and it should be distributed to the scrum team. Team can also request for a new requirement addition or modification or deletion

Process flow of Scrum:

Process flow of scrum testing is as follows:

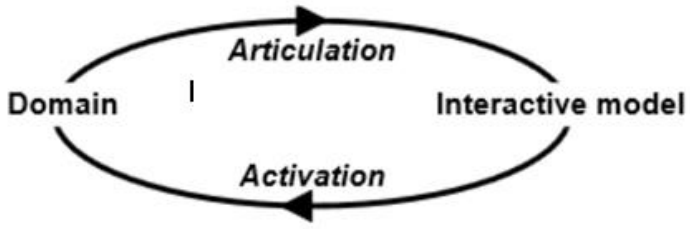
- Each iteration of a scrum is known as Sprint
- Product backlog is a list where all details are entered to get end product
- During each Sprint, top items of Product backlog are selected and turned into Sprint backlog

- Team works on the defined sprint backlog
- Team checks for the daily work
- At the end of the sprint, team delivers product functionality



Managing interactive processes

- Models are defined as explicit representations of some portions of reality as perceived by some actor. A model is active if it influences the reality it reflects; if changes to the representation also change the way some actors perceive reality.
- Model activation is the process by which a model affects reality. Activation involves actors interpreting the model and adjusting their behaviour to it.
- This process can be **Automated**, where a software component executes the model, **Manual**, where the model guides the actions of human actors, or **Interactive**, where prescribed aspects of the model are automatically interpreted and ambiguous parts are left to the users to resolve, with tool support.
- Fully automated activation implies that the model must be formal and complete, while manual and interactive activation also can handle informal and evolving models.
- The process of defining and updating an interactive model is called articulation.



The interplay of articulation and activation.

The Potential of Interactive Process Models

- The constantly changing nature of the competitive environment in the global network economy creates emergent organisations, where" every feature of social organisation culture, meaning, social relationships, decision processes and so on are continually emergent, following no predefined pattern" .
- This environment requires evolving information systems, adapting their behaviour to updated models of the usage environment.

Articulation: Simple and User-Oriented Process Modelling

- Our approach relies on the assumption that end users must be actively involved in creating, updating and interpreting models of their own work, as part of the work.
- Local participants are the only ones with sufficient knowledge of the process.
- Modelling by end users has met skepticism from the workflow research community.
- On the other hand, studies of user participation in IS development, tailoring, knowledge management and process improvement indicate that our approach is viable.
- In workflow management, users also deal creatively with change and exceptions, often by taking the process out of the system and handling it manually.
- Systems not designed for user involvement thus present a barrier to local innovation, and are unable to capture these contributions for further assessment and knowledge management.

- End user participation remains primarily an organisational problem, involving trust, power and community building, but simple, user-oriented, and adaptable modelling languages will remove many barriers.

Activation: Customised and Integrated Software Support

- Simple and useful tools motivate use. Information systems that offer a wide range of functionality often become overwhelmingly complex and incomprehensible. Consequently, only a small portion of the available functionality is utilized.
- This condition is known as featuritis. We need role and task specific user interfaces, containing just what is needed in the current context. Interfaces and semantics should also adapt to the local needs of each project. Process models, articulating who performs which tasks when and why, is a powerful resource for such customisation. Systems and processes should also adapt to the skills and preferences of each individual. Personalisation fosters a sense of ownership, further motivating active participation.
- In virtual enterprises, the unique nature of each project, and the changing set of partners, seldom makes it economically viable to integrate information systems through conventional development methods. Standardisation requires that the domain is static and well understood, and is thus seldom appropriate for knowledge work. Consequently, we need a flexible infrastructure that allows shared understanding and semantic interoperability to emerge from the project, rather than being a prerequisite for cooperation. Interactive models provide a simple, visual approach to capture shared understanding as it unfolds.

Reuse: Process Knowledge Management

- The gap between what people say and what they do, makes it difficult to use plans and other official descriptions of work as input to KM . Local articulation of process models must thus be straightforward, but still some knowledge cannot be modelled and will remain tacit. Process models will thus be incomplete while they are used, subject to an ongoing elaboration and interpretation. Models are completed only when they are no longer in use. Interactive modelling allows the system to handle incomplete, evolving descriptions of work, by involving users in resolving incompleteness and inconsistencies during activation. The openness of the approach allows local process innovation to be captured, assessed and packaged for reuse in similar future projects.

Basics of software estimation:

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)

Estimating size

An accurate estimate of the size of the software to be built is the first step to an effective estimate. Your source(s) of information regarding the scope of the project should, wherever possible, start with formal descriptions of the requirements

Two main ways you can estimate product size are:

- By analogy. Having done a similar project in the past and knowing its size, you estimate each major piece of the new project as a percentage of the size of a similar piece of the previous project. Estimate the total size of the new project by adding up the estimated sizes of each of the pieces. An experienced estimator can produce reasonably good size estimates by analogy if accurate size values are available for the previous project and if the new project is sufficiently similar to the previous one.
- By counting product features and using an algorithmic approach such as Function Points to convert the count into an estimate of size. Macro-level "product features" may include the number of subsystems, classes/modules, methods/functions.
- detailed "product features" may include the number of screens, dialogs, files, database
- tables, reports, messages, and so on.

Estimating effort

Once you have an estimate of the size of your product, you can derive the effort estimate. This conversion from software size to total project effort can only be done if you have a defined software development lifecycle and development process that you follow to specify, design, develop, and test the software. A software development project involves far more than simply coding the software – in fact, coding is often the smallest part of the overall effort. Writing and reviewing documentation, implementing prototypes, designing the deliverables, and reviewing and testing the code take up the larger portion of overall project effort. The project effort estimate requires you to identify and estimate, and then sum up all the activities you must perform to build a product of the estimated size.

Ways to derive effort size:

1) The best way is to use your organization's own historical data to determine how much effort previous projects of the estimated size have taken.

2) If you don't have historical data from your own organization because you haven't started collecting it yet or because your new project is very different in one or more key aspects, you can use a mature and generally accepted algorithmic approach such as

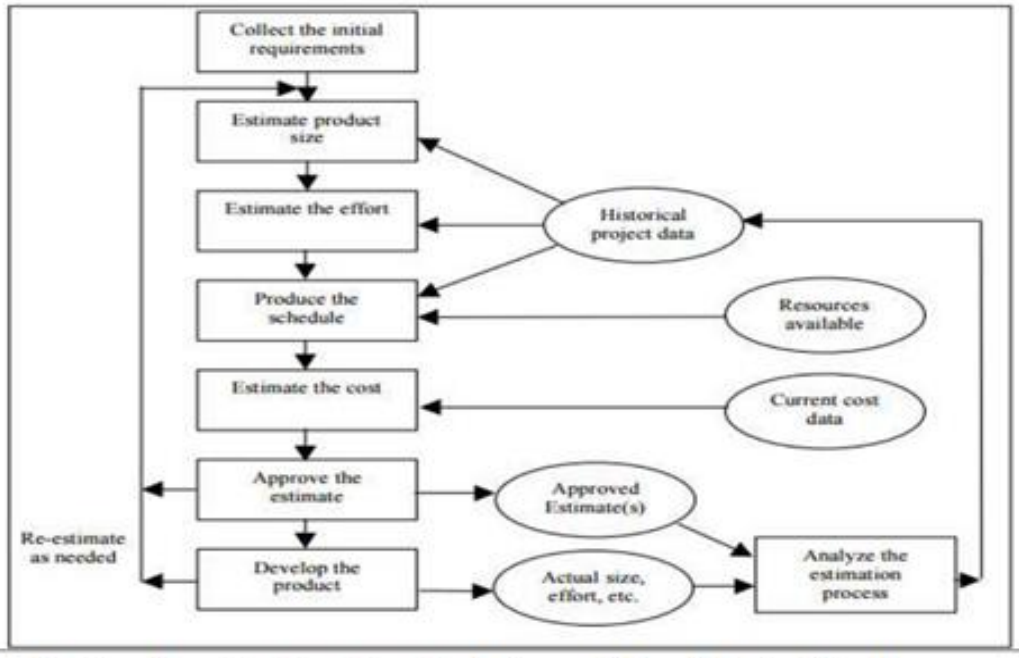
Barry Boehm's COCOMO model or the Putnam Methodology to convert a size estimate into an effort estimate.

Estimating schedule

- The third step in estimating a software development project is to determine the project schedule from the effort estimate.
- This generally involves estimating the number of people who will work on the project, what they will work on (the Work Breakdown Structure), when they will start working on the project and when they will finish (this is the "staffing profile").
- Again, historical data from your organization's past projects or industry data models can be used to predict the number of people you will need for a project of a given size and how work can be broken down into a schedule.

Estimating Cost

- There are many factors to consider when estimating the total cost of a project. These include labor, hardware and software purchases or rentals, travel for meeting or testing purposes, telecommunications (e.g., longdistance phone calls, video-conferences, dedicated lines for testing, etc.), training courses, office space, and so on.
- Exactly how you estimate total project cost will depend on how your organization allocates costs. Some costs may not be allocated to individual projects and may be taken care of by adding an overhead value to labor rates ($ per hour). Often, a software development project manager will only estimate the labor cost and identify any additional project costs not considered "overhead" by the organization.
- The simplest labor cost can be obtained by multiplying the project's effort estimate (in hours) by a general labor rate ($ per hour). A more accurate labor cost would result from using a specific labor rate for each staff position (e.g., Technical, QA, Project)



Effort and cost estimation techniques:

Size Oriented Metrics

i) Source Lines of Code (SLOC):

- is software metric used to measure the size of software program by counting the number of lines in the text of the program's source code.
- This metric does not count blank lines, comment lines, and library. SLOC measures are programming language dependent.
- They cannot easily accommodate nonprocedural languages. SLOC also can be used to measure others, such as errors/KLOC, defects/KLOC, pages of documentation/KLOC, cost/KLOC.

ii) Deliverable Source Instruction (DSI):

- is similar to SLOC.
- The difference between DSI and SLOC is that s "if-then-else" statement, it would be counted as one SLOC but might be counted as several DSI

Function Oriented Metrics:

- Function Point (FP): FP defined by Allan Albrecht at IBM in 1979, is a unit of measurement to express the amount software functionality .
- Function point analysis (FPA) is the method of measuring the size of software.
- The advantage is that it can avoid source code error when selecting different programming languages.
- FP is programming language independent, making ideal for applications using conventional and nonprocedural languages.
- It is base on data that are more likely to be known early in the evolution of project.
- Function types are as:
- External Inputs (EI): it originates from user or transmit ted from another application.
- External Outputs (EO) : it is derived data within application that provides information to the user.
- **External Enquiries (EQ)** : it is online i/p that results in the generation of some immediate s/w response in the form of an online output.
- **Internal Logical Files (ILF)** : is logical grouping of data that resides within the applications boundary and maintained.
  External Interface Files (EIF) : is logical grouping of data that resides external to application but provides information that may be of use to the application.
- **Test Point (TP)**
- Test point used for test point analysis (TPA), to estimate test effort for system and acceptance tests.
- However, it is important to note that TPA itself only covers functional testing.
- FPA and TPA merged together provide means for estimating both, white and black box testing efforts.
- Test effort estimation using UCP, is based upon use cases (UC).
- UC is system behaviour under various conditions, based on requests from a

stakeholder.

- The primary task of UCP is to map use cases (UC) to test cases (TC).

COCOMO (Constructive Cost Models):

- This family of models proposed by Barry Boehm , is the most popular method which is

categorized in algorithmic methods. This method uses some equations and parameters, which have been derived from previous experiences about software projects for estimation. The models have been widely accepted in practice.

- Code-size S is given in thousand LOC (KLOC) and Effort is in person-month. Three

models of COCOMO given by Barry Boehm Simple COCOMO: It was the first model suggested by Barry Boehm, which follows following formula: Effort = a*(KLOC)b where S is the code-size, and a, b are complexity factors. This model uses three sets of a, b depending on the complexity of the software only as given in table IX.

- COCOMO model is simple and easy to use. As many cost factors are not considered, it can only be used as a rough estimate.
- Intermediate COCOMO: In the intermediate COCOMO, a nominal effort estimation is obtained using the power function with three sets of a, b, with coefficient a being slightly different from that of the basic COCOMO

Model	a	b
Organic (Simple in terms of size and complexity)	3.2	1.05
Semi ditched (Average in terms of size and complexity)	3.0	1.15
Embedded (Complex)	2.8	1.20

Table 1: Depicting model and values for a and b

Expertise Based Estimation

It is the most frequently applied estimation strategy for software projects. There is no substantial evidence for use of estimation based models however there are situations where one can expect expert based estimation to be more precise than formal methods. This method is usually used when there is limitation in finding data and gathering requirements. Consultation is the basic issue in this method.

- Avoid conflicting estimation goals.
- Ask the estimators to justify and criticize their estimates.
- Avoid irrelevant and unreliable estimation information.
- Use documented data from previous development tasks.
- Find estimation experts with relevant domain background and good estimation records.
- Estimate top-down and bottom-up, independently of each other.
- Use estimation checklists.
- Combine estimates from different experts and estimation strategies.

COSMIC full function points:

- Function Point Analysis (FPA) is one of the most widely used methods to determine the size of software projects.
- FPA originated at a time when only a mainframe environment was available.
- Sizing of specifications was typically based on functional decomposition and modeled data. Nowadays, development methods like Object Oriented, Component Based and RAD are applied more often.
- There is also more attention on architecture and the use of client server and multitier environments.
- Another development is the growth in complexity caused by more integrated applications, real- time applications and embedded systems and combinations.
- FPA was not designed to cope with these various newer development approaches.
- The Common Software Measurement International Consortium (COSMIC), aimed to develop, test, bring to market and to seek acceptance of a new software sizing method to support estimating and performance measurement (productivity, time to market and quality).
- The measurement method must be applicable for estimating the effort for developing and maintaining software in various software domains. Not only business software (MIS) but also real time software (avionics, telecom, process control) and embedded software (mobile phones, consumer electronics) can be measured.
- The basis for measurement must be found, just as in FPA, in the user requirements the software must fulfil. The result of the measurement must be independent of the development environment and the method used to specify these requirements.
- Sizes depend only on the user require.

COSMIC Concepts

- The Functional User Requirements (FUR) are, according to the definition of a functional size measurement method, the basis for measurement. They specify user's needs and procedures that the software should fulfil.
- The FUR are analysed to identify the functional processes. A Functional Process is an elementary component of a set of FUR. It is triggered by one or more events in the world of the user of the software being measured. The process is complete when it has executed all that is required to be done in response to the triggering event.
- Each functional process consists of a set of sub processes that are either movements or manipulations of data. Since no one knows how to measure data manipulation, and since the aim is to measure 'data movement rich' software, the simplifying assumption is made that each functional process consists of a set of data movements.
- 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 recognises 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.

COCOMO II a parametric productivity model:

COCOMO a parametric model

- COCOMO (Constructive Cost Estimation Model) was proposed by Boehm.
- According to him, any software development project can be classified into one of the following three categories based on the development complexity: organic, semidetached, and embedded. The classification is done considering the characteristics of the product as well as those of the development team and development environment.
- Usually these three product classes correspond to application, utility and system programs, respectively.
- Data processing programs are normally considered to be application programs. Compilers, linkers, etc., are utility programs. Operating systems and real-time system programs, etc. are system programs.
- The definition of organic, semidetached, and embedded systems are elaborated below.
    - **Organic:** A development project can be considered of organic type, if the project deals with developing a well understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.
    - **Semidetached:** A development project can be considered of semidetached type, if the development consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.
    - **Embedded:** A development project is considered to be of embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational procedures exist.
- According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Effort = a * (KLOC)b PM Tdev = 2.5 * (Effort)c Months where

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code a, b, c are constants for each category of software products
- Tdev is the estimated time to develop the software, expressed in months
- Effort is the total effort required to develop the software product, expressed in person months (PMs)

The effort estimation is expressed in units of person-months (PM). The value of the constants a, b, c are given below:

Software project	a	b	c
Organic	2.4	1.05	0.38
Semi-detached	3.0	1.12	0.35
Embedded	3.6	1.20	0.32

Intermediate COCOMO model:

- The basic COCOMO model assumes that effort and development time are functions of the product size alone. However, many other project parameters apart from the product size affect the development effort and time required for the product.
- Therefore, in order to obtain an accurate estimation of the effort and project duration, the effect of all relevant parameters must be taken into account.
- The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development.
- For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1.
- 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) as shown below. An effort multiplier from the table below [i] applies to the rating. The product of all effort multipliers results in an Effort Adjustment Factor (EAF).

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 met hods	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	

EAF is used to refine the estimates obtained by basic COCOMO as follows:

Effort | corrected = Effort * EAF

Tdev | corrected = 2.5 * (Effort | corrected) c

Complete COCOMO model:

- Both the basic and intermediate COCOMO models consider a software product as a single homogeneous entity.
- However, most large systems are made up several smaller sub-systems, each of them in turn could be of organic type, some semidetached, or embedded.
- The complete COCOMO model takes into account these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual subsystems.
- This approach reduces the percentage of error in the final estimate.
- The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:
    - Database part
    - Graphical User Interface (GUI) part
    - Communication part
- Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

Staffing Pattern:

- Resource allocation in software development is important and many methods have been proposed.
- This introduces the staffing pattern as a metric of resource distribution among project phases, and verifies its effect on software quality and productivity using real project data. The main findings are:
    - there exist different staffing patterns in reality;
    - the staffing pattern has significant effect on software quality (post-release defect density);
    - the staffing pattern has no significant effect on productivity;
    - the effort invested on test, document or code inspection possibly explains the effect of staffing pattern on software quality;
    - the effort consumed by rework perhaps counteracts the effect of other potential factors on productivity.
- Preliminary heuristics are suggested to resource allocation practices.
- staffing patterns as follows:
    - **Rapid-team-buildup pattern** (abbreviated Rapid for later reference).
        - The staffing levels peak in requirement phase, and decrease in later phases.
        - This might mean the culture of excessive documentation or design, leading to low ability to respond rapidly for requirement change.
        - Another possible reason would be to outsource part of the system to other organization for design and development. In both situations, we suppose the software quality and productivity are low.
    - **Fix-staff pattern (abbreviated Fix)**.
        - The team size is fixed or stable across project lifecycle.
        - It is likely that the same team has done all work.
        - Due to sufficient learning time and communication within the team, we assumed that high software quality will be yielded as a result.
        - It is hard to assess its productivity: perhaps peoples work efficiently due to effective communication, perhaps this effect cannot counteract against the excessive human resources investment in a prolonged duration.