UNIT –I

Q1.List and explain phases of software

There are following six phases in every Software development life cycle model:

1. Requirement gathering and analysis
2. Design
3. Implementation or coding
4. Testing
5. Deployment
6. Maintenance

1) **Requirement gathering and analysis:** Business requirements are gathered in this phase. This phase is the main focus of the project managers and stake holders.

Meetings with managers, stake holders and users are held in order to determine the requirements like; Who is going to use the system? How will they use the system? What data should be input into the system? What data should be output by the system? These are general questions that get answered during a requirements gathering phase. After requirement gathering these requirements are analyzed for their validity and the possibility of incorporating the requirements in the system to be development is also studied.

Finally, a Requirement Specification document is created which serves the purpose of guideline for the next phase of the model. The testing team follows the Software Testing Life Cycle and starts the **Test Planning** phase after the requirements analysis is completed.

2) **Design:** In this phase the system and software design is prepared from the requirement specifications which were studied in the first phase. System Design helps in specifying hardware and system requirements and also helps in defining overall system architecture. The system design specifications serve as input for the next phase of the model.

In this phase the testers comes up with the **Test strategy**, where they mention what to test, how to test.

3) **Implementation / Coding:** On receiving system design documents, the work is divided in modules/units and actual coding is started. Since, in this phase the code is produced so it is the main focus for the developer. This is the longest phase of the software development life cycle.

4) **Testing:** After the code is developed it is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements phase. During this phase all types of **functional testing**like **unit**

testing, **integration testing**, **system testing**, **acceptance testing** are done as well as **non-functional testing** are also done.

5) Deployment: After successful testing the product is delivered / deployed to the customer for their use.

As soon as the product is given to the customers they will first do the **beta testing**. If any changes are required or if any bugs are caught, then they will report it to the engineering team. Once those changes are made or the **bugs** are fixed then the final deployment will happen.

6) Maintenance: Once when the customers starts using the developed system then the actual problems comes up and needs to be solved from time to time. This process where the care is taken for the developed product is known as maintenance.

Q 2.Explain Capability maturity model

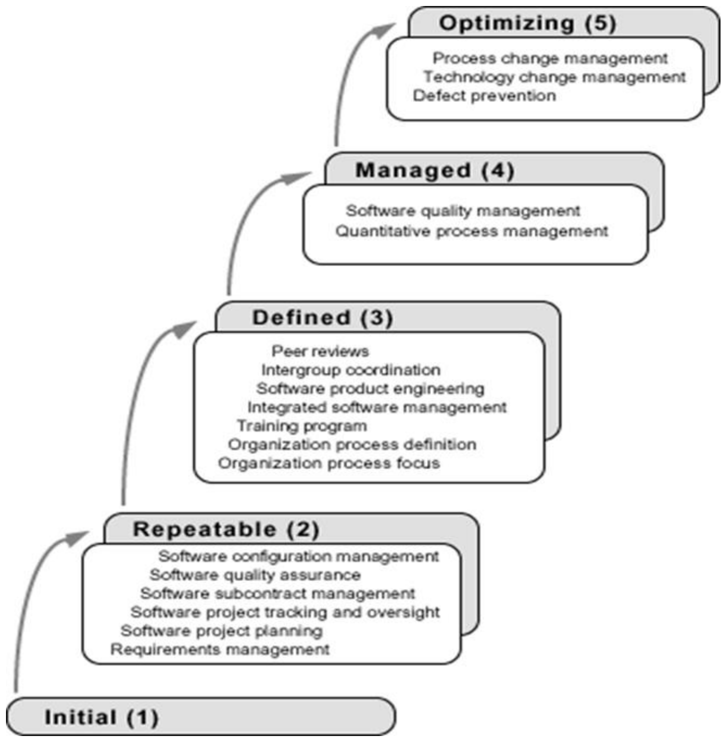
CMM(Capability Maturity Model) increasing series of levels of a software development organization.The higher the level, the better the software development process, hence reaching each level is an expensive and time-consuming process.

- Initial - The software process is characterized as inconsistent, and occasionally even chaotic. Defined processes and standard practices that exist are abandoned during a crisis. Success of the organization majorly depends on an individual effort, talent, and heroics. The heroes eventually move on to other organizations taking their wealth of knowledge or lessons learnt with them.
- Level Two: Repeatable - This level of Software Development Organization has a basic and consistent project management processes to track cost, schedule, and functionality. The process is in place to repeat the earlier successes on projects with similar applications. Program management is a key characteristic of a level two organization.
- Level Three: Defined - The software process for both management and engineering activities are documented, standardized, and integrated into a standard software process for the entire organization and all projects across the organization use an approved, tailored version of the organization's standard software process for developing,testing and maintaining the application.
- Level Four: Managed - Management can effectively control the software development effort using precise measurements. At this level, organization set a quantitative quality goal for both software process and software maintenance. At this maturity level, the

performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable.

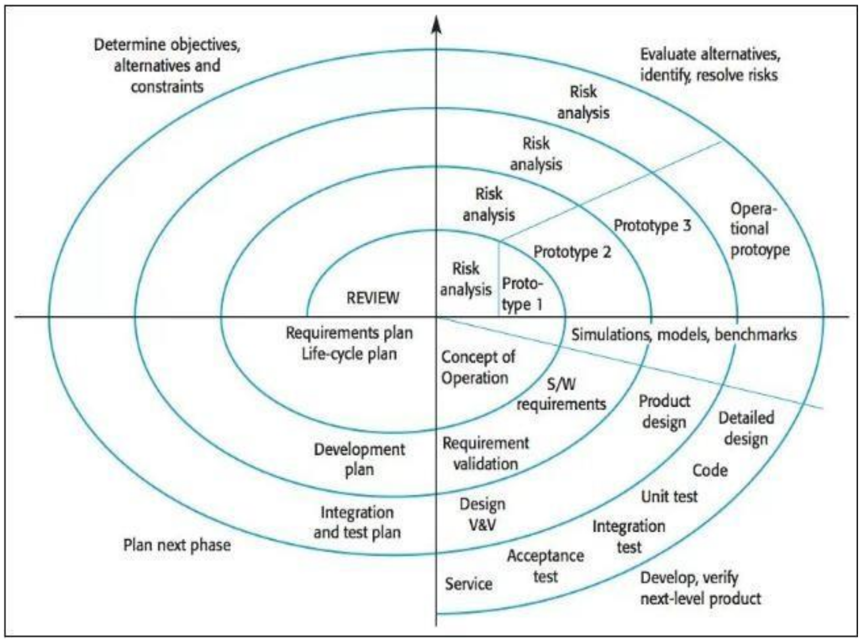
- Level Five: Optimizing - The Key characteristic of this level is focusing on continually improving process performance through both incremental and innovative technological improvements. At this level, changes to the process are to improve the process performance and at the same time maintaining statistical probability to achieve the established quantitative process-improvement objectives.

KPA’s


Optimizing (5)
Process change management
Technology change management
Defect prevention
Managed (4)
Software quality management
Quantitative process management
Defined (3)
Peer reviews
Intergroup coordination
Software product engineering
Integrated software management
Training program
Organization process definition
Organization process focus
Repeatable (2)
Software configuration management
Software quality assurance
Software subcontract management
Software project tracking and oversight
Software project planning
Requirements management
Initial (1)

Q 3.Explain Boehms Full Spiral Model.

Barry Boehm (Boehm, 1988) proposed a risk-driven software process framework (the spiral model) that integrates risk management and incremental development. The software process is represented as a spiral rather than a sequence of activities with some backtracking from one activity to another. Each loop in the spiral represents a phase of the software process.



Each loop in the spiral is split into four sectors:

1) Objective setting

Specific objectives for that phase of the project are defined. Constraints on the process and the product are identified and a detailed management plan is drawn up. Project risks are identified. Alternative strategies, depending on these risks, may be planned.

2) Risk assessment and reduction

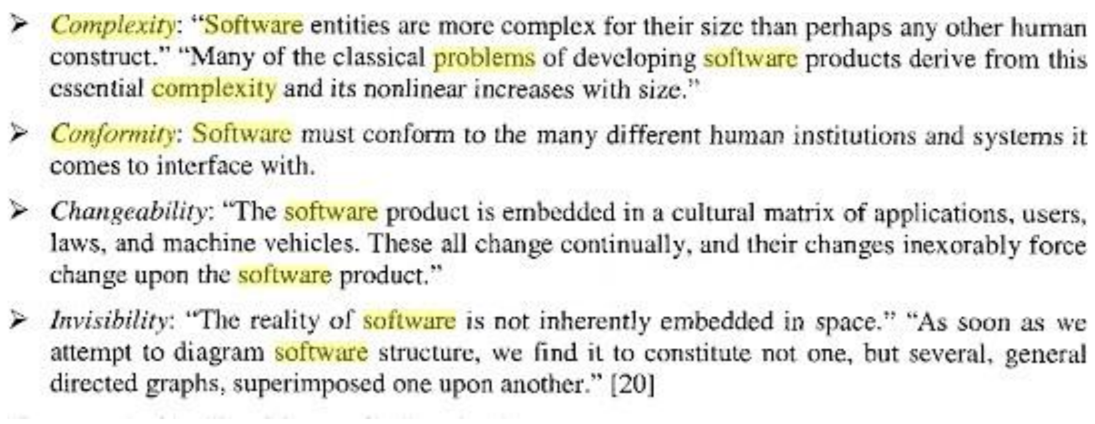
For each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3) Development and validation

After risk evaluation, a development model for the system is chosen. For example, throw-away prototyping may be the best development approach if user interface risks are dominant. If safety risks are the main consideration, development based on formal transformations may be the most appropriate process, and so on. If the main identified risk is sub-system integration, the waterfall model may be the best development model to use. Planning The project is reviewed and a decision made whether to continue with a further loop of the spiral. If it is decided to continue, plans are drawn up for the next phase of the project.

The main difference between the spiral model and other software process models is its explicit recognition of risk.
Once risks have been assessed, some development is carried out, followed by a planning activity for the next phase of the process. Informally, risk simply means something that can go wrong. For example, if the intention is to use a new programming language, a risk is that the available compilers are unreliable or do not produce sufficiently efficient object code.

Q 4.Discuss the problems associated with software production



- *Complexity*: "Software entities are more complex for their size than perhaps any other human construct." "Many of the classical problems of developing software products derive from this essential complexity and its nonlinear increases with size."
- *Conformity*: Software must conform to the many different human institutions and systems it comes to interface with.
- *Changeability*: "The software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product."
- *Invisibility*: "The reality of software is not inherently embedded in space." "As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs, superimposed one upon another." [20]

Q 5.Explain Classical chief programmer Approach

FIGURE 4.2 Communication paths between six software professionals.

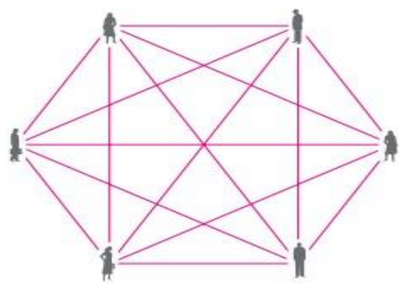
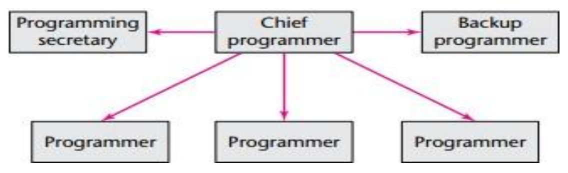


FIGURE 4.3 The structure of a classical chief programmer team.



The chief programmer team concept was formalized by Mills [Baker, 1972]. A classical chief programmer team, as described by Baker . It consisted of the **chief programmer**, who was assisted by the backup programmer, the **programming secretary**, and from one to three

programmers. When necessary, the team was assisted by specialists in other areas, such as legal or financial matters, or the job control language (JCL) statements used to give operating system commands to the mainframe computers of that era. The chief programmer was both a successful manager and a highly skilled programmer who did the architectural design and any critical or complex sections of the code. The other team members worked on the detailed design and the coding, under the direction of the chief programmer.

The position of backup programmer was necessary only because the chief programmer was human and could therefore become ill or change jobs.

The programming secretary was not a part-time clerical assistant but a highly skilled, well-paid, central member of a chief programmer team.

The programming secretary was responsible for maintaining the project production library, the documentation of the project. This included source code listings, JCL, and test data.

The programmers handed their source code to the secretary, who was responsible for its conversion to machine-readable form, compilation, linking, loading, execution, and running test cases. Programmers therefore did nothing but program. All other aspects of their work were handled by the programming secretary

Q .Explain Incremental model in detail

The incremental model combines elements of the linear sequential model (applied repetitively) with the iterative philosophy of prototyping.

When an incremental model is used, the first increment is often a core product. That is, basic requirements are addressed, but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed review)

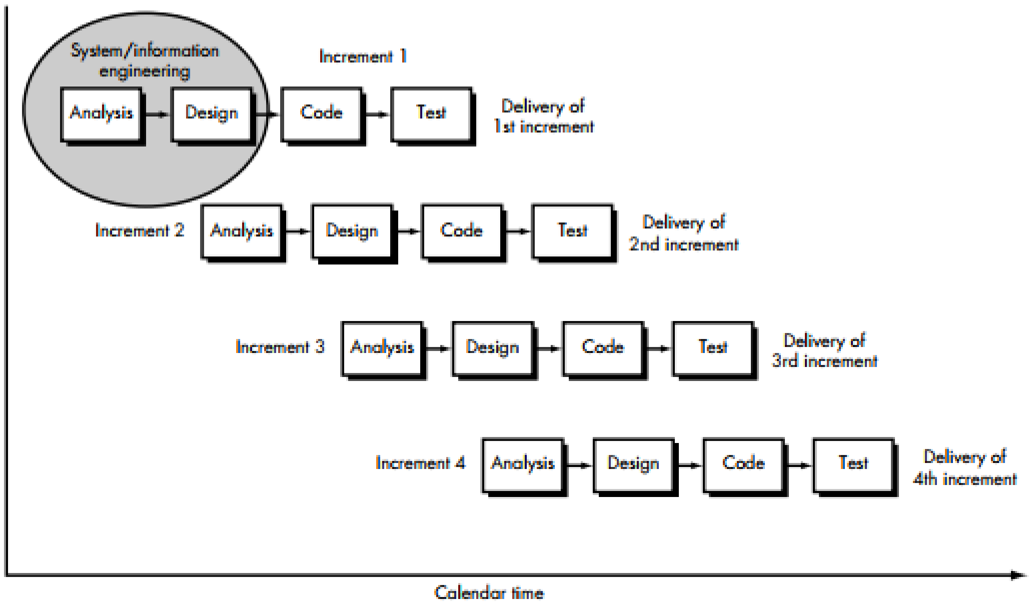
As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.

This process is repeated following the delivery of each increment, until the complete product is produced.

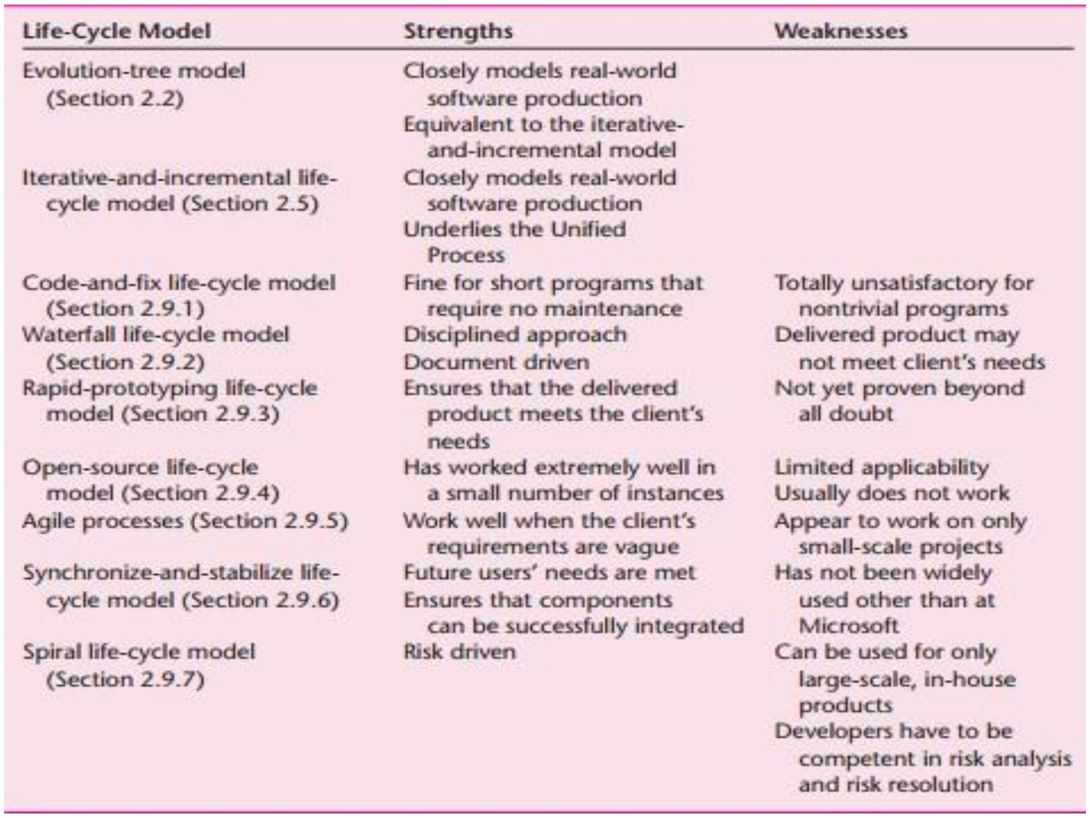
The incremental process model, like prototyping and other evolutionary approaches, is iterative in nature. But unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.


System/information engineering
Increment 1
Analysis
Design
Code
Test
Delivery of 1st increment
Increment 2
Delivery of 2nd increment
Increment 3
Delivery of 3rd increment
Increment 4
Delivery of 4th increment
Calendar time

Q.Comparison of different life cycle models



Life-Cycle Model	Strengths	Weaknesses
Evolution-tree model (Section 2.2)	Closely models real-world software production Equivalent to the iterative-and-incremental model	
Iterative-and-incremental life-cycle model (Section 2.5)	Closely models real-world software production Underlies the Unified Process	
Code-and-fix life-cycle model (Section 2.9.1)	Fine for short programs that require no maintenance	Totally unsatisfactory for nontrivial programs
Waterfall life-cycle model (Section 2.9.2)	Disciplined approach Document driven	Delivered product may not meet client's needs
Rapid-prototyping life-cycle model (Section 2.9.3)	Ensures that the delivered product meets the client's needs	Not yet proven beyond all doubt
Open-source life-cycle model (Section 2.9.4)	Has worked extremely well in a small number of instances	Limited applicability Usually does not work
Agile processes (Section 2.9.5)	Work well when the client's requirements are vague	Appear to work on only small-scale projects
Synchronize-and-stabilize life-cycle model (Section 2.9.6)	Future users' needs are met Ensures that components can be successfully integrated	Has not been widely used other than at Microsoft
Spiral life-cycle model (Section 2.9.7)	Risk driven	Can be used for only large-scale, in-house products Developers have to be competent in risk analysis and risk resolution

Q.List 5 metrics of software

There are five essential, fundamental metrics:

1. Size (in lines of code or, better, in a more meaningful metric.

2. Cost (in dollars).

3. Duration (in months).

4. Effort (in person-months).

5. Quality (number of faults detected).

Each of these metrics must be measured by workflow (metrics for the specification, analysis, design, and implementation workflows.

On the basis of the data from these fundamental metrics, management can identify problems within the software organization, such as high fault rates during the design workflow or code output that is well below the industry average.

Once problem areas have been highlighted, a strategy to correct these problems can be considered. To monitor the success of this strategy, more-detailed metrics can be introduced. For example, it may be deemed appropriate to collect data on the fault rates of each programmer or to conduct a survey of user satisfaction.

Q. Explain Democratic team approach

The democratic team organization was first described by Weinberg in 1971 [Weinberg, 1971]. The basic concept underlying the democratic team is egoless programming.

Weinberg points out that programmers can be highly attached to their code. Sometimes, they even name their modules after themselves:

They therefore see their modules as an extension of themselves. The difficulty with this is that a programmer who sees a module as an extension of his or her ego is certainly not going to try to find all the faults in “his” code or “her” code.

A group of up to 10 egoless programmers constitutes a democratic team. Weinberg warns that management may have difficulty working with such a team.

consider the managerial career path. When a programmer is promoted to a management position, his or her fellow programmers are not promoted and must strive to attain the higher level at the next round of promotions.

Q.Write a short note on build and fix model.

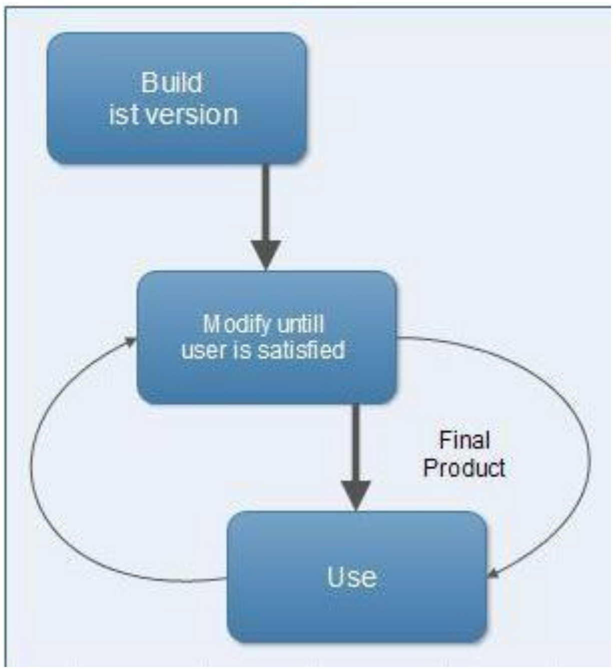
In the **build and fix model** (also referred to as an **ad hoc model**), the software is developed without any specification or design.

An initial product is built, which is then repeatedly modified until it (software) satisfies the user. That is, the software is developed and delivered to the user. The user checks whether the desired functions 'are present.

If not, then the software is changed according to the needs by adding, modifying or deleting functions. This process goes on until the user feels that the software can be used productively. However, the lack of design requirements and repeated modifications result in loss of acceptability of software.

This model includes the following two phases.

- **Build:** In this phase, the software code is developed and passed on to the next phase.
- **Fix:** In this phase, the code developed in the build phase is made error free. Also, in addition to the corrections to the code, the code is modified according to the user's requirements.



Q. Explain Intermediate COCOMO and COCOMO II

COCOMO actually is a series of three models, ranging from a macroestimation model that treats the product as a whole to a microestimation model that treats the product in detail. Computing development time using intermediate COCOMO is done in two stages.

First, a rough estimate of the development effort is provided. Two parameters have to be estimated: the le ngth of the product in KDSI and the product's development mode, a measure of the intrinsic level of difficulty of developing that product.

There are three modes: organic (small and straightforward), semidetached (medium sized), and embedded (complex).

From these two parameters, the nominal effort can be computed.

For example, if the project is judged to be essentially straightforward (organic), then the nominal effort (in person-months) is given by the equation Nominal effort 3.2 (KDSI) 1.05 person-months (9.6) The constants 3.2 and 1.05 are the values that best fitted the data on the organic mode products used by Boehm to develop intermediate COCOMO. For example, if the product to be built is organic and estimated to be 12,000 delivered source statements (12 KDSI), then the nominal effort is 3.2

---

(12) 1.05 43 person-months.

COCOMO II

COCOMO II [Boehm et al., 2000] is a major revision of the 1981 COCOMO. COCOMO II can handle a wide variety of modern software engineering techniques, including object-orientation, the various life-cycle models.

First, intermediate COCOMO consists of one overall model based on lines of code (KDSI). On the other hand, COCOMO II consists of three different models. The application composition model ,

A second difference lies in the effort model underlying COCOMO:

Effort = a *$(\text{size})^{b}$

where a and b are constants. In intermediate COCOMO, the exponent b takes on three different values, depending on whether the mode of the product to be built is organic (b = 1.05)

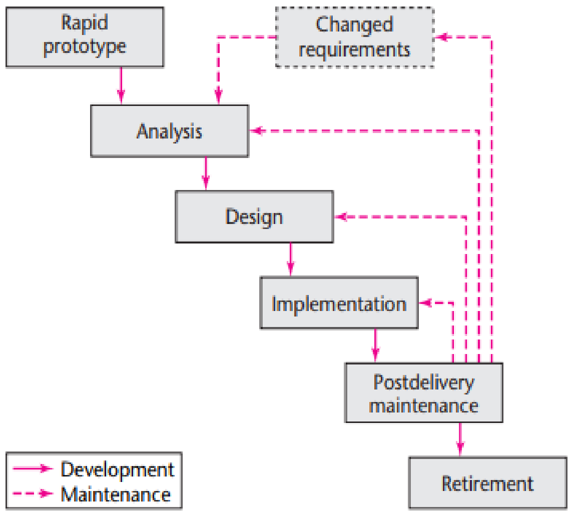
A third difference is the assumption regarding reuse. Intermediate COCOMO assumes that the savings due to reuse are directly proportional to the amount of reuse. COCOMO II takes into account that small changes to reused software incur disproportionately large costs.

Q. Explain Rapid Prototype model.

A rapid prototype is a working model that is functionally equivalent to a subset of the product. For example, if the target product is to handle accounts payable, accounts receivable, and warehousing, then the rapid prototype might consist of a product that performs the screen handling for data capture and prints the reports, but does no file updating or error handling.

The first step in the rapid-prototyping life-cycle model depicted in Figure 2.10 is to build a rapid prototype and let the client and future users interact and experiment with the rapid prototype. Once the client is satisfied that the rapid prototype indeed does most of what is required, the developers can draw up the specification document with some assurance that the product meets the client's real needs.

A major strength of the rapid-prototyping model is that the development of the product is essentially linear, proceeding from the rapid prototype to the delivered product; the feedback loops of the waterfall model.



Q. A target product has 8 simple inputs, 1 average input, and 9 complex inputs. There are 41 average outputs, 7 simple inquiries, 16 average master files, and 14 complex interfaces. Determine the unadjusted function points (UFP).

	Level of Complexity		
Component	**Simple**	**Average**	**Complex**
Input item	3	4	6
Output item	4	5	7
Inquiry	3	4	6
Master file	7	10	15
Interface	5	7	10

Answer:-

	Simple	Average	Complex
Input	8	1	9
Output		41	
Inquiry	7		
Master Files		16	
Interface			14

Therefore, Unadjusted function points = 8*3+1*4+9*6+41*5+7*3+16*10+14*10

= 24+4+54+205+21+160+140 = 608