	b)	Explain the architecture of a CASE environment.	L2	CO4	5 M

(25)

PVP 23

Code: 23CS3403, 23IT3403

II B.Tech - II Semester – Regular Examinations - MAY 2025

SOFTWARE ENGINEERING

(Common for CSE, IT)

Duration: 3 hours Max. Marks: 70

Note: 1. This question paper contains two Parts A and B.
2. Part-A contains 10 short answer questions. Each Question carries 2 Marks.
3. Part-B contains 5 essay questions with an internal choice from each unit. Each Question carries 10 marks.
4. All parts of Question paper must be answered in one place.

BL – Blooms Level CO – Course Outcome

PART – A

		BL	CO
1.a)	How has software development evolved?	L1	CO1
1.b)	What are the core values and principles of Agile development model?	L1	CO1
1.c)	What are the critical elements of software project planning?	L2	CO2
1.d)	What are the key advantages of empirical estimation techniques?	L1	CO2
1.e)	Define design process.	L2	CO3
1.f)	What is a Data Flow Diagram (DFD)?	L1	CO3
1.g)	Define system testing.	L1	CO3
1.h)	What is the importance of software documentation?	L1	CO4
1.i)	What is software reverse engineering?	L1	CO4
1.j)	What are the major factors affecting software maintenance costs?	L1	CO4

PART – B

			BL	CO	Max. Marks
		UNIT-I			
2	a)	Describe Waterfall model advantages and disadvantages.	L2	CO1	5 M
	b)	Write a short note on Software development projects.	L1	CO1	5 M
		OR			
3	a)	Discuss rapid application development model.	L2	CO1	5 M
	b)	Explain Exploratory style of software development.	L2	CO1	5 M
		UNIT-II			
4	a)	Summarize the Responsibilities of a Software Project Manager.	L2	CO1	5 M
	b)	Explain about COCOMO a Heuristic Estimation Technique.	L2	CO1	5 M
		OR			
5	a)	Discuss about Metrics for Project Size Estimation.	L2	CO2	5 M
	b)	What are the key challenges in gathering accurate and complete software requirements from stakeholders?	L2	CO2	5 M
		UNIT-III			
6	a)	What are the key characteristics of a good software design? Explain	L2	CO3	5 M

	b)	Discuss Shneiderman's Golden Rules of UI Design.	L2	CO3	5 M
		OR			
7	a)	Compare and contrast the different approaches to software design.	L3	CO3	5 M
	b)	Describe the essential characteristics of a good user interface.	L2	CO3	5 M
		UNIT-IV			
8	a)	Explain the difference between black-box testing and white-box testing.	L3	CO3	5 M
	b)	Demonstrate the ISO 9000 standard for software quality management.	L2	CO3	5 M
		OR			
9	a)	How does testing object-oriented programs, differ from testing procedural programs?	L3	CO3	5 M
	b)	What is debugging? Discuss the common debugging techniques used in software development.	L2	CO3	5 M
		UNIT-V			
10	a)	Explain the importance of Computer-Aided Software Engineering in modern software development.	L2	CO4	5 M
	b)	Describe the key characteristics of CASE tools.	L2	CO4	5 M
		OR			
11	a)	Explain the characteristics of Software maintenance.	L2	CO4	5 M

Subject Code: 23CS3403, 23IT3403 | PVP 23

PRASAD V POTLURI SIDDHARTHA INSTITUTE OF TECHNOLOGY

(AUTONOMOUS)

II B.Tech- II Semester- Regular Examinations-May 2025

SOFTWARE ENGINEERING

(Common for CSE,IT)

Duration: 3 Hours | **Scheme of Evaluation** | **Max.Marks : 70**

PART-A

Answer *all* the questions. All questions carry equal marks 10 x 2 = 20M

1.a) How has software development evolved? 2M
b) What are the core values and principles of Agile development Model? 2M
c) What are the critical elements of software project planning?(Any four) 2M
d) What are the key advantages of empirical estimation techniques? 2M

e) Define design process. 2M

f) What is a Data Flow Diagram (DFD)?2M

g) Define system testing.2M
h) What is the importance of software documentation? 2M
i) What is software reverse engineering?2M
J) What are the major factors affecting software maintenance costs?2M

PART – B

UNIT-1

2 a) Describe Waterfall model advantages and disadvantages. 5M

Advantages------2.5M

Disadvantages-------2.5M

2 b) Write a short notes on software development projects. 5M

(OR)

3 a) Discuss Rapid application development model. 5M
3 b) Explain Exploratory style of software development.
Diagram-1M, Explanation—4M 5M

UNIT-II

4 a) Summarize the Responsibilities of a Software Project Manager. 5M
4 b) Explain about COCOMO Heuristic Estimation Technique. 5M

(OR)

5 a) Discuss about Metrics for Project Size Estimation. 5M

LOC metrics(Any five)—2.5,Function point Metric(Any Five)-2.5M

5 b) What are the key challenges in gathering accurate and complete software requirements from stakeholders? 5M

UNIT-III

6 a) What are the key characteristics of good software design? –2M

Explanation-------3M. 5M

6 b) Discuss Shneiderman's Golden Rules of UI Design. 5M

(OR)

7 a) Compare and contrast the different approaches to Software design. 5M

(Any Five)

7 b) Describe the essential Characteristics of good user interface. 5M

(Any Five)

UNIT-IV

8 a) Explain the difference between black box testing and white box testing? 5M

(Any Five)

8 b) Demonstrate the ISO 9000 standard for software quality management. 5M

(OR)

9 a) How does testing object oriented programs, differ from testing procedural programs?5M

9 b) What is debugging?1M

Discuss the common debugging techniques used in software development.4M

UNIT-V

10 a) Explain the importance of Computer aided software Engineering in Modern Software Development.? 5M

10 b) Describe the key characteristics of CASE Tools. 5M

11 a) Explain the characteristics of Software maintenance? 5M

11 b) Explain the architecture of a CASE environment? 5M

Diagram-2M, Explanation—3M

Subject Code: 23CS3403, 23IT3403

PVP 23

PRASAD V POTLURI SIDDHARTHA INSTITUTE OF TECHNOLOGY

(AUTONOMOUS)

II B.Tech- II Semester- Regular Examinations-May 2025

SOFTWARE ENGINEERING

(Common for CSE,IT)

Duration: 3 Hours

Max. Marks : 70

PART-A

Answer *all* the questions. All questions carry equal marks **10 x 2 = 20M**

1.a) How has software development evolved?

It has evolved from an esoteric art form to a craft form, and then has slowly emerged as an engineering discipline. The good programmers knew certain principles (or tricks) that helped them write good programs, which they seldom shared with the bad programmers. Program writing in later years was akin to a craft. Over the next several years, all good principles (or tricks) that were discovered by programmers along with research innovations have systematically been organised into a body of knowledge that forms the discipline of software engineering.

b) What are the core values and principles of Agile development Model?

The agile model could help a project to adapt to change requests quickly. Thus a major aim of the agile models is to facilitate quick project completion. It gives the required flexibility so that the activities that may not be necessary for a specific project could be easily removed. Anything that wastes time and effort is avoided.

- A central principle of the agile model is the delivery of an increment to the customer after each time box.
- Working software over comprehensive documentation.
- Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.
- Requirement change requests from the customer are encouraged and are to be efficiently incorporated.

c) What are the critical elements of software project planning?

Project managers undertake project planning.

Estimation: The following project attributes are estimated.

Cost estimtion: How much is it going to cost to develop the software product?

Duration: How long is it going to take to develop the product?

Effort: How much effort would be necessary to develop the product?

Scheduling: The schedules for manpower and other resources are developed.

Staffing: Staff organization and staffing plans are made.

Risk management: This includes risk identification, analysis, and abatement planning.

Miscellaneous plans: This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

d) What are the key advantages of empirical estimation techniques?

- Empirical estimation techniques help in quoting an appropriate project cost to the customer.
- Form the basis for resource planning and scheduling.
- using this technique, prior experience with development of similar products is helpful

e) Define design process.

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document.

The design process starts using the SRS document and completes with the production of the design document.

The design process essentially transforms the SRS document into a design document.

f) What is a Data Flow Diagram (DFD)?

The DFD (also known as the *bubble chart*) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system.

It is simple to understand and use. A DFD model uses a very limited number of primitive symbols to represent the functions performed by a system and the data flow among these functions.

g) Define system testing.

System Testing

- After all the units of a program have been integrated together and tested, system testing is taken up.
- System tests are designed to validate a fully developed system to assure that it meets its requirements.
- The test cases are therefore designed solely based on the SRS document.

h) What is the importance of software documentation?

When a software is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc., are developed as part of the software engineering process. All these documents are considered a vital part of any good software development practice. Good documents are helpful in the following ways: Good documents help enhance understandability of code.

- Documents help the users to understand and effectively use the system.
- Good documents help to effectively tackle the manpower turnover problem
- Production of good documents helps the manager to effectively track the progress of the project.

i) What is software reverse engineering?

Software reverse engineering is the process of recovering the design and the requirements specification of a product from an analysis of its code. The purpose of reverse engineering is to facilitate maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system. Reverse engineering is becoming important, since legacy software products lack proper documentation, and are highly unstructured. Even well-designed products become legacy software as their structure degrades through a series of maintenance efforts.

J) What are the major factors affecting software maintenance costs?

The following factors effects the software maintenance costs.

When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface.Every software product continues to evolve after its development through maintenance efforts.

Software maintenance has a very poor image in industry. Therefore, an organisation often cannot employ bright engineers to carry out maintenance work. During maintenance it is necessary to thoroughly understand someone else's work, and then carry out the required modifications and extensions.

Poor documentation, unstructured and lack of personnel knowledgeable in the product. effects the software maintenance costs.

PART – B

UNIT-1

2 a) Describe Waterfall model advantages and disadvantages. 5M

The waterfall model offers several advantages, including its simple, easy-to-understand structure, well-defined phases, and clear milestones. It also promotes a disciplined, organized approach, facilitates documentation, and reinforces good coding habits.

This model is particularly well-suited for small projects with stable, well-defined requirements.

Clear Structure and Documentation: The waterfall model's sequential, well-defined phases provide a structured framework for project management, making it easy to understand and follow. Each phase has specific deliverables and a review process, facilitating management and documentation.

Easy to Manage and Understand: The rigid structure of the waterfall model makes it easy to manage, as each phase is completed one at a time, with no overlapping. This simplicity also makes it easy for developers and stakeholders to understand the project's progress.

Well-Defined Milestones and Deadlines: The model's clear phases and stages enable the definition of distinct milestones and deadlines, which are essential for project planning and progress tracking.

The waterfall model is relatively straightforward to learn and implement, requiring minimal training or specialized skills.

Disadvantages of the waterfall model

No feedback paths: In waterfall model, the evolution of software from one phase to the next is analogous to a waterfall. Just as water in a waterfall after having flowed down cannot flow back,

once a phase is complete, the activities carried out in it and any artifacts produced in this phase are considered to be final and are closed for any rework.
For example, a design defect might go unnoticed till the coding or testing phase. Once a defect is detected at a later time, the developers need to redo some of the work done during that phase and also redo the work of later phases that are affected by the rework. Therefore, in any non-trivial software development project, it becomes nearly impossible to strictly follow the classical waterfall model of software development.

Difficult to accommodate change requests: This model assumes that all customer requirements can be completely and correctly defined at the beginning of the project. There is much emphasis on creating an unambiguous and complete set of requirements. But, it is hard to achieve this even in ideal project scenarios. The customers' requirements usually keep on changing with time. But, in this model it becomes difficult to accommodate any requirement change requests made by the customer after the requirements specification phase is complete, and this often becomes a source of customer discontent.

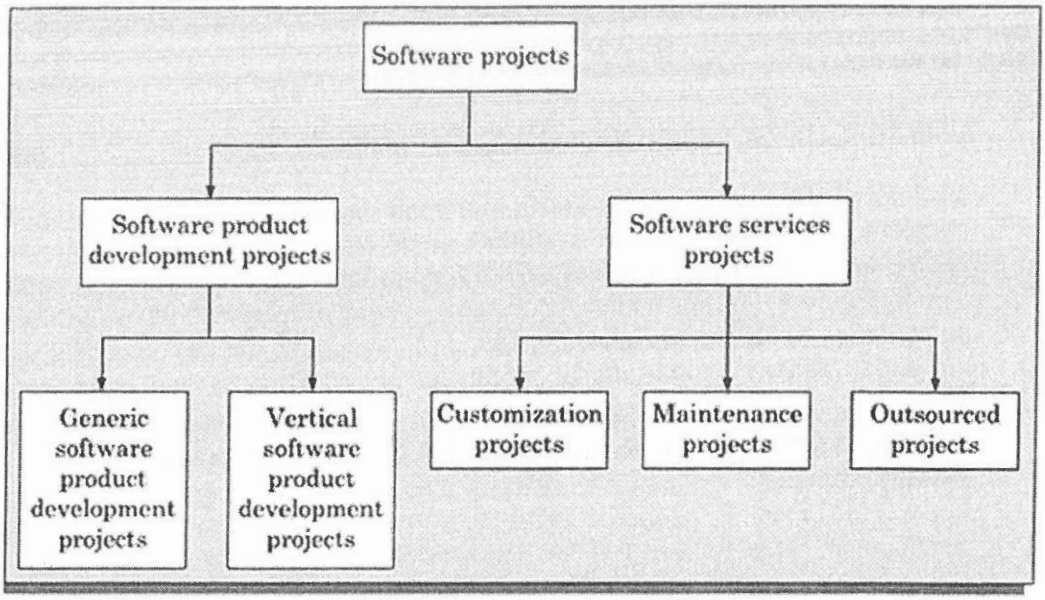
Inefficient error corrections: This model defers integration of code and testing tasks until it is very late when the problems are harder to resolve.

No overlapping of phases: This model recommends that the phases be carried out sequentially—new phase can start only after the previous one completes. However, it is rarely possible to adhere to this recommendation and it leads to a large number of team members to idle for extended periods. For example, for efficient utilisation of manpower, the testing team might need to design the system test cases immediately after requirements specification is complete.

2 b) Write a short notes on software development projects. 5M

The various types of development projects that are being undertaken by software development companies.
Software Development Projects: A software development company typically has a large number of on-going projects. Each of these projects may be classified into software product development projects or services type of projects. These two broad classes of software projects can be further classified into subclasses as shown in Figure.

A software product development project may be either to develop a generic product or a domain specific product. A generic software product development project concerns about developing software that would be sold to a large number of customers. Since a generic software product is sold to a broad spectrum of customers, it is said to have a horizontal market. On the other hand, the services projects may either involve customizing some existing software, maintaining or developing some outsourced software. Since a specific segment of customers are targeted, these software products are said to have a vertical market. In the following, we distinguish between these two major types of software projects



Types of software projects

Software products. A variety of software such as Microsoft's Windows operating system and Office suite, and Oracle Corporation's Oracle 8i database management software. These software are available off-the-shelf for purchase and are used by a diverse range of customers. These are called generic software products since many users essentially use the same software. These can be purchased off-the-shelf by the customers. When a software development company wishes to develop a generic product, it first determines the features or functionalities that would be useful to a large cross section of users. Based on these, the development team draws up the product specification on its own.

In contrast to the generic products, domain specific software products are sold to specific categories of customers and are said to have a vertical market. Domain specific software products target specific segments of customers (called *verticals*) such as banking, telecommunication, finance and accounts, and medical. Examples of domain specific software products are BANCS from TCS and FINACLE from Infosys in the banking domain and Aspen Plus from Aspen Corporation in the chemical process simulation.

Software services

Software services cover a large gamut of software projects such as customization, outsourcing, maintenance, testing, and consultancy. At present, there is a rapid growth in the number of software services projects that are being undertaken world-wide and software services are poised to become the dominant type of software projects. One of the reasons behind this situation is the steep growth in the available code base. Over the past few decades, a large number of programs have already been developed. Available programs can therefore be modified to quickly fulfil the specific requirements of any customer. At present, there is hardly any software project in which the program code is written from scratch, and software is being mostly developed by customizing some existing software. For example, to develop software to automate the payroll generation activities of an educational institute, the vendor may customize existing software that might has been developed earlier for a different client or educational institute.

The types of development projects that are being undertaken by a company can have an impact on its profitability.

(OR)

3 a) Discuss Rapid application development model. **5M**

The rapid application development (RAD) model was proposed in the early nineties in an attempt to overcome the rigidity of the waterfall model (and its derivatives) that makes it difficult to accommodate any change requests from the customer. It proposed a few radical extensions to the waterfall model. This model has the features of both prototyping and evolutionary models. It deploys an evolutionary delivery model to obtain and incorporate the customer feedbacks on incrementally delivered versions. In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer. But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction.

Working of RAD

In the RAD model, development takes place in a series of short cycles or iterations. At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time. The time planned for each iteration is called a time box.

Each iteration is planned to enhance the implemented functionality of the application by only a small amount. During each time box, quick-and-dirty prototype-style software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary. The prototype is refined based on the customer feedback.

The prototype is not meant to be released to the customer for regular use though the development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team. The development team usually consists of about five to six members, including a customer representative.

The customers usually suggest changes to a specific feature only after they have used it. Since the features are delivered in small increments, the customers are able to give their change requests pertaining to a feature already delivered. Incorporation of such change requests just after the delivery of an incremental feature saves cost as this is carried out before large investments have been made in development and testing of a large number of features.

The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in two main ways—minimal use of planning and heavy reuse of any existing code through rapid prototyping. The lack of long-term and detailed planning gives the flexibility to accommodate later requirements changes. Reuse of existing code has been adopted as an important mechanism of reducing the development cost.

RAD model emphasises code reuse as an important means for completing a project faster. In fact, the adopters of the RAD model were the earliest to embrace object-oriented languages and practices. Further, RAD advocates use of specialised tools to facilitate fast creation of working prototypes.

3 b) Explain Exploratory style of software development. **5M**

EXPLORATORY STYLE OF SOFTWARE DEVELOPMENT

The exploratory program development style refers to an informal development style where the programmer makes use of his own intuition to develop a program rather than making use of the systematic body of knowledge categorized under the software engineering discipline. The exploratory development style gives complete freedom to the programmer to choose the activities using which to develop software.

Though the exploratory style imposes no rules a typical development starts after an initial briefing from the customer. Based on this briefing, the developers start coding to develop a working program. The software is tested and the bugs found are fixed. This cycle of testing and bug fixing continues till the software works satisfactorily for the customer. A schematic of this work sequence in a build and fix style has been shown graphically in Figure. Observe that coding starts after an

initial customer briefing about what is required. After the program development is complete, a test and fix cycle continues till the program becomes acceptable to the customer. An exploratory development style can be successful when used for developing very small programs, and not for professional software.

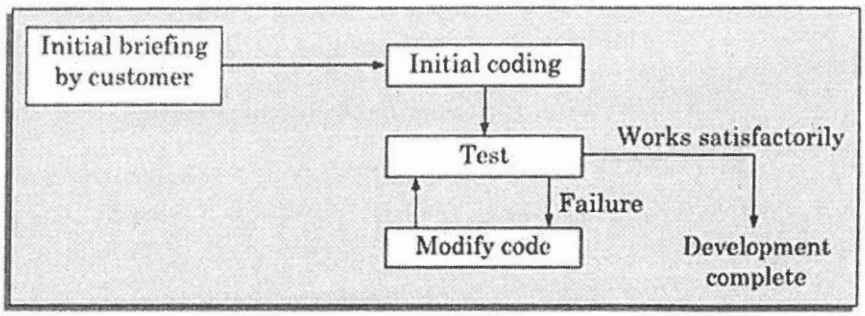


Figure: Exploratory program development

Short comings of the exploratory style of software development:

- The foremost difficulty is the exponential growth of development time and effort with problem size and large-sized software becomes almost impossible using this style of development.
- The exploratory style usually results in unmaintainable code. The reason for this is that any code developed without proper design would result in highly unstructured and poor quality code.
- It becomes very difficult to use the exploratory style in a team development environment. In the exploratory style, the development work is undertaken without any proper design and documentation. Thereafter it becomes very difficult to meaningfully partition the work among a set of developers who can work concurrently.

UNIT-2

4 a) Summarize the Responsibilities of a Software Project Manager. 5M

RESPONSIBILITIES OF A SOFTWARE PROJECT MANAGER

Job Responsibilities for Managing Software Projects

A software project manager takes the overall responsibility of steering a project to success. This surely is a very hazy job description. It is very difficult to objectively describe the precise job responsibilities of a project manager. The job responsibilities of a project manager range from invisible activities like building up of team morale to highly visible customer presentations.

The following are the two major categories of project manager responsibilities.

- Project planning, and
- Project monitoring and control.

Project planning: Project planning is undertaken immediately after the feasibility study phase and before the starting of the requirements analysis and specification phase. The initial project plans are revised from time to time as the project progresses and more project data become available.

Project monitoring and control: Project monitoring and control activities are undertaken once the development activities start. As the project gets underway, the details of the project

that were unclear earlier at the start of the project emerge and situations that were not visualized earlier arise. While carrying out project monitoring and control activities, a project manager usually needs to change the plan to cope up with specific situations at hand.

Skills Necessary for Managing Software Projects

A theoretical knowledge of various project management techniques is certainly important to become a successful project manager. However, a purely theoretical knowledge of various project management techniques would hardly make one a successful project manager. Effective software project management calls for good qualitative judgment and decision taking capabilities.

In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, and configuration management, etc., project managers need good communication skills and the ability to get work done. Some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. Never the less, the importance of a sound knowledge of the prevalent project management techniques cannot be overemphasized.

Three skills that are most critical to successful project management are the following:

- Knowledge of project management techniques.
- Decision taking capabilities
- Previous experience in managing similar projects.

4 b) Explain about COCOMO Heuristic Estimation Technique. 5M

Constructive Cost estimation Model (COCOMO) was proposed by Boehm [1981].

Basic COCOMO Model: Boehm postulated that any software development project can be classified into one of the following three categories based on the development complexity—*organic, semidetached, and embedded.*Three basic classes of software development projects

Organic: We can classify a development project to be 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 classify to be of semidetached type, if the development team 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 hardware, or if stringent regulations on the operational procedures exist. Development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

- Observe that in deciding the category of the development project, in addition to considering the characteristics of the product being developed, we need to consider the characteristics of

the team members. Thus, a simple data processing program may be classified as semidetached, if the team members are inexperienced in the development of similar products.

- For the three product categories, Boehm provided different sets of constant values for the coefficients for the two basic expressions to predict the effort (in units of person-months) and development time from the size estimation given in kilo lines of source code (KLSC).
- Person-month (PM) is a popular unit for effort measurement. It should be carefully noted that an effort estimation of 100 PM does not imply that 100 persons should work for 1 month. Neither does it imply that 1 person should be employed for 100 months to complete the project.
- The effort estimation simply denotes the area under the person-month curve for the project. plot Figure 1 shows that different number of personnel may work at different points in the project development. The number of personnel working on the project usually increases or decreases by an integral number resulting in the sharp edges in the plot.

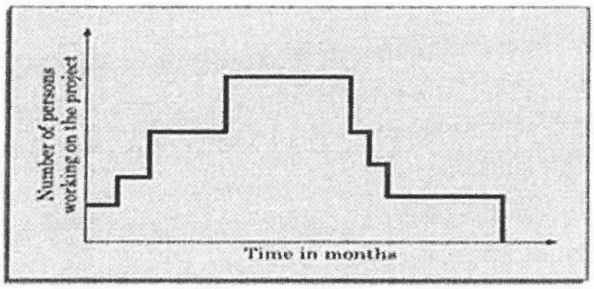


Figure 1: Person-month curve

General form of the COCOMO expressions

- The **basic COCOMO model** is a single variable heuristic model that gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by expressions of the following forms:
- Effort = $a1 \times (\text{KLOC})a2$ PM
- Tdev = $b1 \times (\text{Effort})b2$ months
- where, „ KLOC is the estimated size of the software product expressed in Kilo Lines Of Code.
- $a1$, $a2$, $b1$, $b2$ are constants for each category of software product.
- 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).
- According to Boehm, every line of source text should be calculated as one LOC n irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be n LOC. The values of $a1$, $a2$, $b1$, $b2$ for different categories of products .He derived these values by examining historical data collected from a large number of n actual projects.

Estimation of development effort: For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

- Organic: Effort = 2.4(KLOC)1.05 PM
- Semi-detached: Effort = 3.0(KLOC)1.12 PM
- Embedded: Effort = 3.6(KLOC)1.20 PM
- **Estimation of development time:** For the three classes of software products, the formulas for estimating the development time based on the effort are given below:
- Organic: Tdev = 2.5(Effort)0.38 Months
- Semi-detached: Tdev = 2.5(Effort)0.35 Months
- Embedded: Tdev = 2.5(Effort)0.32 Months We can gain some insight into the basic COCOMO model, if we plot the estimated effort and duration values for different software sizes. Figure2 shows the plots of estimated effort versus product size for different categories of software products.

Observations from the effort-size plot: From Figure 2, we can observe that the effort is somewhat super linear in the size of the software product. This is because the exponent in the effort expression is more than 1. Thus, the effort required to develop a product increases rapidly with project size. The reason for this is that COCOMO assumes that projects are carefully designed and developed by using software engineering principles

Observations from the development time—size plot

- The development time versus the product size in KLOC is plotted in Figure 3. The development time is a sublinear function of the size of the product. That is, when the size of the product increases by two times, the time to develop the product does not double but rises moderately.

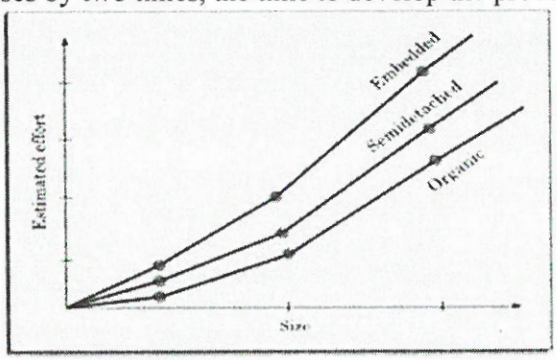


Figure 2: Effort vs Product size

- From Figure 3 we can observe that for a project of any given size, the development time is roughly the same for all the three categories of products. According to the COCOMO formulas,

embedded programs require much higher effort than either application or utility programs.

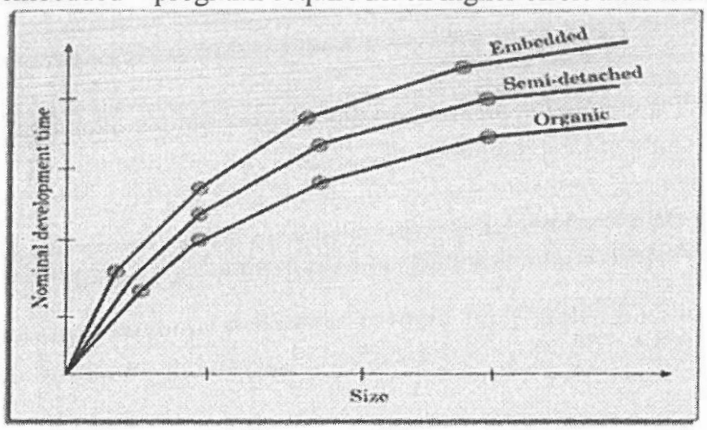


Figure 3. development time vs size

- **Cost estimation**

  From the effort estimation, project cost can be obtained by multiplying the estimated effort (in man-month) by the manpower cost per month. Implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. However, in addition to manpower cost, a project would incur several other types of costs which we shall refer to as the overhead costs. The overhead costs would include the costs due to hardware and software required for the project and the company overheads for administration, office space, electricity, etc. Depending on the expected values of the overhead costs, the project manager has to suitably scale up the cost arrived by using the COCOMO formula.

- **Implications of effort and duration estimate**

  The effort and duration values computed by COCOMO are the values for completing the work in the shortest time without unduly increasing manpower cost.

(OR)

5 a) Discuss about Metrics for Project Size Estimation. 5M

METRICS FOR PROJECT SIZE ESTIMATION

- The project size is a measure of the problem complexity in terms of the effort and time required to develop the product.
- Currently, two metrics are popularly being used to measure size—lines of code (LOC) and function point (FP). Each of these metrics has its own advantages and disadvantages.
- Based on their relative advantages, one metric may be more appropriate than the other in a particular situation.

Lines of Code (LOC)

- LOC is possibly the simplest among all metrics available to measure project size.
- Consequently, this metric is extremely popular. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are ignored.

- Determining the LOC count at the end of a project is very simple. However, accurate estimation of LOC count at the beginning of a project is a very difficult task. One can possibly estimate the LOC count at the starting of a project, only by using some form of systematic guess work. Systematic guessing typically involves the following. The project manager divides the problem into modules, and each module into sub-modules and
- **LOC is a measure of coding activity alone.** The implicit assumption made by the LOC metric that the overall product development effort is solely determined from the coding effort alone is flawed.
- **LOC count depends on the choice of specific instructions:** LOC gives a numerical value of problem size that can vary widely with coding styles of individual programmers.
- Even for the same programming problem, different programmers might come up with programs having very different LOC counts. This situation does not improve, even if language tokens are counted instead of lines of cod
- **LOC measure correlates poorly with the quality and efficiency of the code:** Larger code size does not necessarily imply better quality of code or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set or use improper algorithms.
- **LOC metric penalizes use of higher-level programming languages and code reuse:** A paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower. This would show up as smaller program size, and in turn, would indicate lower effort!
- **LOC metric measures the lexical complexity of a program and does not address the more important issues of logical and structural complexities:** Between two programs with equal LOC counts, a program incorporating complex logic would require much more effort to develop than a program with very simple logic.
- **It is very difficult to accurately estimate LOC of the final program from problem specification:** As already discussed, at the project initiation time, it is a very difficult task to accurately estimate the number of lines of code (LOC) that the program would have after development. The LOC count can accurately be computed only after the code has fully been developed. Since project planning is carried out even before any development activity starts, the LOC metric is of little use to the project managers during project planning.

Function Point (FP) Metric

- Function point metric was proposed by Albrecht and Gaffney in 1983. This metric overcomes many of the shortcomings of the LOC metric. Since its inception, function point metric has steadily gained popularity. Function point metric has several advantages over LOC metric.
- One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself.
- Using the LOC metric, on the other hand, the size can accurately be determined only after the code has been fully written.
- The conceptual idea behind the function point metric is the following. The size of a software product is directly dependent on the number of different high-level functions or features it

supports. This assumption is reasonable, since each feature would take additional effort to implement.

- Though each feature takes some effort to develop, different features may take very different amounts of efforts to develop. For example, in a banking software, a function to display a help message may be much easier to develop compared to say the function that carries out the actual banking transactions. Therefore, just determining the number of functions to be supported (with adjustments for number of fi les and interfaces) may not yield very accurate results.
- This has been considered by the function point metric by counting the number of input and output data items and the number of files accessed by the function. The implicit assumption made is that the more the number of data items that a function reads from the user and outputs and the more the number of files accessed, the higher is the complexity of the function.
- Now let us analyze why this assumption must be intuitively correct. Each feature when invoked typically reads some input data and then transforms those to the required output data. For example, the query book feature (see Figure 3.2) of a Library Automation Software takes the name of the book as input and displays its location in the library and the total number of copies available. Similarly, the issue book and the return book features produce their output based on the corresponding
- input data. It can therefore be argued that the computation of the number of input and
- output data items would give a more accurate indication of the code size compared to simply counting the number of high-level functions supported by the system.

5 b) What are the key challenges in gathering accurate and complete software requirements from stakeholders? 5M

The primary objective of the requirement's gathering task is to collect the requirements from the *stakeholders*.

Requirements gathering may sound like a simple task. However, in practice it is very difficult to gather all the necessary information from a large number of stakeholders and from information scattered across several pieces of documents. Gathering requirements turns out to be especially challenging if there is no working model of the software being developed.

Suppose a customer wants to automate some activity in his organisation that is currently being carried out manually. In this case, a working model of the system (that is, a manual system) exists. Availability of a working model is usually of great help in requirements gathering.

For example, if the project involves automating the existing accounting activities of an organisation, then the task of the system analyst becomes a lot easier as he can immediately obtain the input and output forms and the details of the operational procedures. In this context, consider that it is required to develop a software to automate the book-keeping activities involved in the operation of a certain office. In this case, the analyst would have to study the input and output forms and then understand how the outputs are produced from the input data. However, if a project involves developing something new for which no working model exists, then the requirements gathering activity becomes all the more difficult. In the absence of a working system, much more imagination and

creativity is required on the part of the system analyst. Typically, even before visiting the customer site, requirements gathering activity is started by studying the existing documents to collect all possible information about the system to be developed. During visit to the customer site, the analysts normally interview the end-users and customer representatives, carry out requirements gathering activities such as questionnaire surveys, task analysis, scenario analysis, and form analysis.

1. Studying the existing documentation: The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide statement of purpose (SoP) document to the analyst. Typically, these documents might discuss issues such as the context in which the software is required, the basic purpose, the stakeholders, and the broad category of features required.

2. Interview: Typically, there are many different categories of users of a software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each. For example, the different categories of users of a library automation software could be the library members, the librarians, and the accountants.

3. Task analysis: The users usually have a black-box view of a software and consider the software as something that provides a set of services (functionalities). A service supported by a software is also called a *task*. We can therefore say that the software performs various tasks of the users. In this context, the analyst tries to identify and understand the different tasks to be performed by the software. For each identified task, the analyst tries to formulate the different steps necessary to realize the required functionality in consultation with the users.

4. Scenario analysis: A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations. For different scenarios of a task, the behavior of the software can be different.

5. Form analysis: Form analysis is an important and effective requirement gathering activity that is undertaken by the analyst, when the project involves automating an existing manual system. During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn they receive several notifications (usually manually filled forms). In form analysis, the exiting forms and the formats of the notifications produced are analyzed to determine the data input to the system and the data that are output from the system. For the different sets of data input to the system, how the input data would be used by the system to produce the corresponding output data is determined from the users.

UNIT-III

6 a) What are the key characteristics of good software design? Explain. 5M

The following are the key characteristics of good software design.

Correctness: A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

Efficiency: A good design solution should adequately address resource, time, and cost optimisation issues.

Maintainability: A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

Understandability of a design: A Major concern

While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously, all incorrect designs have to be discarded first. Out of the correct design solutions, we can identify the best one.

A good design should help overcome the human cognitive limitations that arise due to limited short-term memory. A large problem overwhelms design would make the matter worse. Unless a design solution is easily understandable, it could lead to an implementation having a large number of defects and at the same time tremendously pushing up the development costs. Therefore, a good design solution should be simple and easily understandable. A design that is easy to understand is also easy to develop and maintain.

A complex design would lead to severely increased life cycle costs. Unless a design is easily understandable, it would require tremendous effort to implement, test, debug, and maintain it. that about 60 per cent of the total effort in the life cycle of a typical product is spent on maintenance.

If the software is not easy to understand, not only would it lead to increased development costs, the effort required to maintain the product would also increase manifold. Besides, a design solution that is difficult to understand would lead to a program that is full of bugs and is unreliable that understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition.

An understandable design is modular and layered

A design solution should have the following characteristics to be easily understandable:

It should assign consistent and meaningful names to various design components.

It should make use of the principles of decomposition and abstraction in good measures to simplify the design.

Modularity

A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution. A *modular design* implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.

Decomposition of a problem into modules facilitates taking advantage of the *divide and conquer* principle. If different modules have either no interactions or little interactions with each other, then each module can be understood separately. This reduces the perceived complexity of the design solution greatly.

Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterize the modularity of a design solution based on the cohesion and coupling existing in the design.

A software design with high cohesion and low coupling among modules is the effective problem decomposition .Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

Layered design

A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it. The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done.

A layered design can be considered to be implementing *control abstraction*, since a module at a lower layer is unaware of (about how to call) the higher layer modules. When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error.

This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error.

6 b) Discuss Shneiderman's Golden Rules of UI Design. 5M

The Golden rules actually form the basis for a set of user interface design principles that guides the important aspect of software design

- **1. Place the user in control.**
- **2. Reduce the user's memory load.**
- **3. Make the interface consistent.**

Place the user in control.

- During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface
- Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided.

- For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.
- But every action is not amenable to every interaction mechanism

Allow user interaction to be interruptible and undoable

- Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

- **Streamline interaction as skill levels advance and allow the interaction to be customized.** Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

- **Hide technical internals from the casual user.** The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

- **Design for direct interaction with objects that appear on the screen.** The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

. Reduce the user's memory load.

- The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory.
- It defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory.

When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results

Establish meaningful defaults.

The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.

Define shortcuts that are intuitive.

When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor.

For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion.

The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick

Make the user interface consistent

The interface should present and acquire information in a consistent fashion. This implies that

(1) all visual information is organized according to design rules that are maintained throughout all screen displays,

(2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and

(3) mechanisms for navigating from task to task are consistently defined and implemented.

Allow the user to put the current task into a meaningful context.

Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand.

Maintain consistency across a family of applications.

A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

(OR)

7 a) Compare and contrast the different approaches to Software design. 5M

Software design approaches can be categorized as function-oriented, object-oriented approaches. Function-oriented design focuses on the flow of data and how functions process it.

Object-oriented design focuses on the structure and interaction of objects, which encapsulate data and behavior. The choice of approach depends on factors like project complexity, team expertise, and the need for reusability and maintainability.

Function-Oriented Design:

- **Focus:** Decomposition of the system into functions or modules that perform specific operations on data.
- **Data Flow:** Emphasizes how data is passed between different functions.
- **Modularity:** Functions are designed to be independent and reusable.
- **Example:** Procedural programming where code is structured around functions.
- **Pros:** Simple to understand and implement for small projects, good for tasks where the data flow is clearly defined.
- **Cons:** Can become complex for large projects, may lead to code duplication, and can be less reusable than object-oriented design.

Object-Oriented Design:

- **Focus:**

  Identifying objects (entities with data and behavior) and their relationships.

- **Data Encapsulation:**

  Objects contain their own data (attributes) and methods (functions) to manipulate that data, promoting data integrity.

- **Modularity:**

  Supports Objects are designed to be self-contained and interact through well-defined interfaces.

- **Example:**

  Programming with classes and objects in languages like Java or C++.

- **Pros:**

  reusability, maintainability, and scalability, well-suited for complex projects.

- **Cons:**

  Can be more complex to learn and implement than function-oriented design, may require more upfront design effort.

Comparison:

Feature	Function-Oriented	Object-Oriented
Focus	Functions and data flow	Objects, their attributes, and methods
Modularity	Independent functions	Self-contained objects with well-defined interfaces
Reusability	Can be reusable, but less than object-oriented	Highly reusable
Complexity	Simpler to learn and implement for small projects	More complex to learn and implement, but scalable
Suitability	Suitable for smaller, simpler projects	Suitable for larger, more complex projects

7 b) Describe the essential Characteristics of good user interface. 5M

Speed of learning: A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memories commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning:

Use of metaphors and intuitive command names: Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar with. The abstractions of real-life objects or concepts used in user interface design are called *metaphors*. If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it.

Consistency: Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. Thus, the different commands supported by an interface should be consistent.

Component-based interface: Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with. This can be achieved if the interfaces of different applications are developed using some standard user interface components. This, in fact, is the theme of the component-based user interface

Speed of use: Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is sometimes referred to as *productivity support* of the interface. It indicates how fast the users can perform their intended

tasks. The time and user effort necessary to initiate and execute different commands should be minimal.

Speed of recall: Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.

Error prevention: A good user interface should minimize the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by an average users while using the interface. This monitoring can be automated by instrumenting the user interface code with monitoring code which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users.

Aesthetic and attractive: A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.

Consistency: The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate.

Feedback: A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic. If required, the user should be periodically informed about the progress made in processing his command.

Support for multiple skill levels: A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects. Very cryptic and complex commands discourage a novice, whereas elaborate command sequences make the command issue procedure very slow and therefore put off experienced users

Error recovery (undo facility): While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if they cannot recover from the errors they commit while using a software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

User guidance and on-line help: Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

UNIT-IV

8 a) Explain the difference between black box testing and white box testing? 5M

Black Box Testing Vs White Box Testing

Parameters	Black Box Testing	White Box Testing
Definition	Black Box Testing is a way of software testing in which the internal structure or the program or the code is	White Box Testing is a way of testing the software in which the tester has knowledge about the

Parameters	Black Box Testing	White Box Testing
	hidden and nothing is known about it.	internal structure or the code or the program of the software.
Testing objectives	Black box testing is mainly focused on testing the functionality of the software, ensuring that it meets the requirements and specifications.	White box testing is mainly focused on ensuring that the internal code of the software is correct and efficient.
Testing methods	Black box testing uses methods like **Equivalence partitioning**, **Boundary Value Analysis**, and **Error Guessing** to create test cases.	White box testing uses methods like **Control Flow Testing**, **Data Flow Testing** and **Statement Coverage Testing**.
Knowledge level	Black box testing does not require any knowledge of the internal workings of the software, and can be performed by testers who are not familiar with programming languages.	White box testing requires knowledge of programming languages, software architecture and designpatterns.
Scope	Black box testing is generally used for testing the software at the functional level.	White box testing is used for testing the software at the unit level, integration level and system level.
Implementation	Implementation of code is not needed for black box testing.	Code implementation is necessary for white box testing.
Done By	Black Box Testing is mostly done by software testers.	White Box Testing is mostly done by software developers.
Terminology	Black Box Testing can be referred to as outer or external software testing.	White Box Testing is the inner or the internal software testing.
Testing Level	Black Box Testing is a functional test of the software.	White Box Testing is a structural test of the software.

Parameters	Black Box Testing	White Box Testing
Testing Initiation	Black Box testing can be initiated based on the requirement specifications document.	White Box testing of software is started after a detail design document.
Programming	No knowledge of programming is required.	It is mandatory to have knowledge of programming.
Testing Focus	Black Box Testing is the behavior testing of the software.	White Box Testing is the logic testing of the software.
Applicability	Black Box Testing is applicable to the higher levels of testing of software.	White Box Testing is generally applicable to the lower levels of software testing.
Alternative Names	Black Box Testing is also called closed testing.	White Box Testing is also called as clear box testing.
Time Consumption	Black Box Testing is least time consuming.	White Box Testing is most time consuming.
Suitable for Algorithm Testing	Black Box Testing is not suitable or preferred for algorithm testing.	White Box Testing is suitable for algorithm testing.
Approach	Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.

8 b) Demonstrate the ISO 9000 standard for software quality management. 5M

ISO 9000 specifies:

- Guidelines for repeatable and high quality product development.
- Also addresses organizational aspects Responsibilities, reporting, procedures, processes, and resources for implementing quality management.

Several benefits:

- Confidence of customers in an organization increases.

ISO 9000 for Software Industry,

- A set of guidelines for the production process.
- Not directly concerned about the product it self.
- A series of three standards:

ISO 9001:

Applies to:

- Organizations engaged in design, development, production, and servicing of goods.
- Applicable to most software development organizations.

ISO 9002:

- This standard applies to those organizations which do not design products but are only involved in production.
- Examples of this category of industries include steel and car manufacturing industries who buy the product and plant designs from external sources and are involved in only manufacturing those products.
- Therefore, ISO 9002 is not applicable to software development organizations

ISO 9003:

- This standard applies to organizations involved only in installation and testing of products.

ISO 9000 is a generic standard that is applicable to a large gamut of industries, starting from a steel manufacturing industry to a service rendering company.

- Therefore, many of the clauses of the ISO 9000 documents are written using generic terminologies and it is very difficult to interpret them in the context of software development organisations.
- An important reason behind such a situation is the fact that software development is in many respects radically different from the development of other types of product manufacturing
- Software is intangible, making it difficult to control and manage since it remains invisible until fully developed and running. In contrast, physical product manufacturing, like car production, allows for visible progress tracking, making it easier to estimate work completed and remaining time.
- Software development primarily consumes data as its raw material, unlike other industries that require large quantities of physical materials. Consequently, many ISO 9000 clauses on raw material control are irrelevant to software development organizations.
  - Due to such radical differences between software and other types of product development, it was difficult to interpret various clauses of the original ISO standard in the context of software industry. Therefore, ISO released a separate document called ISO 9000 part-3 in 1991 to help interpret the ISO standard for software industry. At present, official guidance is inadequate regarding the interpretation of various clauses of ISO 9000 standard in the context of software industry and one has to keep on cross referencing the ISO 9000-3 document

(OR)

9 a) How does testing object oriented programs, differ from testing procedural programs? 5M

- During the initial years of object-oriented programming, it was believed that object orientation would, to a great extent, reduce the cost and effort incurred on testing. This thinking was based on the observation that object-orientation incorporates several good programming features such as encapsulation, abstraction, reuse through inheritance, polymorphism, etc., thereby chances of errors in the code is minimized.
- However, it was soon realized that satisfactory testing object-oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs.
- The main reason behind this situation is that various object-oriented features introduce additional complications and scope of new types of bugs that are present in procedural programs.
- For procedural programs, we had seen that procedures are the *basic units of testing*. That is, first all the procedures are unit tested. Then various tested procedures are integrated together

and tested. Thus, as far as procedural programs are concerned, procedures are the basic units of testing. Since methods in an object-oriented program are analogous to procedures in a procedural program, can we then consider the methods of object-oriented programs as the basic unit of testing?

- The main intuitive justification for the anticomposition axiom is the following. A method operates in the scope of the data and other methods of its object. That is, all the methods share the data of the class. Therefore, it is necessary to test a method in the context of these. Moreover, objects can have significant number of states. The behavior of a method can be different based on the state of the corresponding object. Therefore, it is not enough to test all the methods and check whether they can be integrated satisfactorily.
- A method has to be tested with all the other methods and data of the corresponding object. Moreover, a method needs to be tested at all the states that the object can assume. As a result, it is improper to consider a method as the basic unit of testing an object-oriented program.
- Thus, in an object-oriented program, unit testing would mean testing each object in isolation. During integration testing (called *cluster testing* in the object-oriented testing literature) various unit tested objects are integrated and tested. Finally, system-level testing is carried out.

9 b) What is debugging? Discuss the common debugging techniques used in software development. 5M

Debugging:

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed.

Debugging Approaches:

The following are some of the approaches that are popularly adopted by the programmers for debugging:

1. ***Brute force method:***
   - This is the most common method of debugging but is the least efficient method.
   - In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
   - This approach becomes more systematic with the use of a symbolic debugger, because values of different variables can be easily checked and breakpoints and watch points can be easily set to test the values of variables effortlessly.
2. **Backtracking:**
   - This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.
3. **Cause elimination method:**
   - In this approach, once a failure is observed, the symptoms of the failure are noted.
   - Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each.
4. **Program slicing:**
   - This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements.
   - However, the search space is reduced by defining slices.
   - A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

10 a) Explain the importance of Computer aided software Engineering in Modern Software Development.? 5M

Computer aided software engineering (CASE) and use of CASE tools help to improve software development effort and maintenance effort.

A CASE tool is a generic term used to denote any form of automated support for software engineering.

CASE tool can mean any tool used to automate some activity associated with software development.

Many CASE tools are now available. Some of these tools assist in phase-related tasks such as specification, structured analysis, design, coding, testing, etc. and others to non-phase activities such as project management and configuration management.

The primary objectives in using any CASE tool are:

- To increase productivity.
- To help produce better quality software at lower cost.
- Although individual CASE tools are useful, the true power of a tool set can be realised only when these set of tools are integrated into a common framework or environment.
- If the different CASE tools are not integrated, then the data generated by one tool would have to input to the other tools.
- This may also involve format conversions as the tools developed by different vendors are likely to use different formats.
- This results in additional effort of exporting data from one tool and importing to another. Also, many tools do not allow exporting data and maintain the data in proprietary formats.
- CASE tools are characterised by the stage or stages of software development life cycle on which they focus. Since different tools covering different stages share common information, it is required that they integrate through some central repository to have a consistent view of information associated with the software.
- This central repository is usually a data dictionary containing the definition of all composite and elementary data items. Through the central repository all the CASE tools in a CASE environment share common information among themselves. Thus, a CASE environment facilitates the automation of the step-by-step methodologies for software development. In contrast a CASE environment, a *programming environment* is an integrated collection of tools to support only the coding phase of software development.
- The tools commonly integrated in a programming environment are a text editor, a compiler, and a debugger. The different tools are integrated to the extent that once the compiler detects an error, the editor takes automatically goes to the statements in error and the error statements are highlighted.
- A key benefit arising out of the use of a CASE environment is cost saving through all developmental phases. Different studies carry out to measure the impact of CASE, put the effort reduction between 30 per cent and 40 per cent.
- Use of CASE tools leads to considerable improvements in quality. and the chances of human error is considerably reduced.

- CASE tools help produce high quality and consistent documents. Since the important data relating to a software product are maintained in a central repository, redundance in the stored data is reduced, and therefore, chances of inconsistent documentation are reduced to a great extent.
- Introduction of a CASE environment has an impact on the style of working of a company, and makes it oriented towards the structured and orderly approach.
-

10 b) Describe the key characteristics of CASE Tools. 5M

Hardware and Environmental Requirements

In most cases, it is the existing hardware that would place constraints upon the CASE tool selection. Thus, instead of defining hardware requirements for a CASE tool, the task at hand becomes to fit in an optimal configuration of CASE tool in the existing hardware capabilities. Therefore, we have to emphasise on selecting the most optimal CASE tool configuration for a given hardware configuration.

The heterogeneous network is one instance of distributed environment and we choose this for illustration as it is more popular due to its machine independent features. The CASE tool implementation in heterogeneous network makes use of client-server paradigm. The multiple clients which run different modules access data dictionary through this server.

The data dictionary server may support one or more projects. Though it is possible to run many servers for different projects but distributed implementation of data dictionary is not common.

The tool set is integrated through the data dictionary which supports multiple projects, multiple users working simultaneously and allows to share information between users and projects. The data dictionary provides consistent view of all project entities, e.g., a data record definition and its entity-relationship diagram be consistent. The server should depict the per-project logical view of the data dictionary. This means that it should allow backup/restore, copy, cleaning part of the data dictionary, etc.

The tool should work satisfactorily for maximum possible number of users working simultaneously. The tool should support multi-windowing environment for the users. This is important to enable the users to see more than one diagram at a time. It also facilitates navigation and switching from one part to the other.

Documentation Support

The deliverable documents should be organized graphically and should be able tom incorporate text and diagrams from the central repository. This helps in producing up-to date documentation. The CASE tool should integrate with one or more of the commercially available desk-top publishing packages. It should be possible to export text, graphics, tables, data dictionary reports to the DTP package in standard forms such as PostScript.

Project Management

It should support collecting, storing, and analysing information on the software project's progress such as the estimated task duration, scheduled and actual task start, completion date, dates and results of the reviews, etc.

External Interface

The tool should allow exchange of information for reusability of design.

The information which is to be exported by the tool should be preferably in ASCII format and support open architecture. Similarly, the data dictionary should provide a programming interface to access information.It is required for integration of custom utilities, building new techniques, or populating the data dictionary.

Reverse Engineering Support

The tool should support generation of structure charts and data dictionaries from the existing source codes. It should populate the data dictionary from the source code.

If the tool is used for re-engineering information systems, it should contain conversion tool from indexed sequential file structure, hierarchical and network database to relational database systems.

Data Dictionary Interface

The data dictionary interface should provide view and update access to the entities and relations stored in it. It should have print facility to obtain hard copy of the viewed screens. It should provide analysis reports like cross-referencing, impact analysis, etc. Ideally, it should support a query language to view its contents.

(OR)

11 a) Explain the characteristics of Software maintenance? 5M

CHARACTERISTICS OF SOFTWARE MAINTENANCE

Software maintenance is becoming an important activity of a large number of organisations. This is no surprise, given the rate of hardware obsolescence, the immortality of a software product *per se*, and the demand of the user community to see the existing software products run on newer platforms, run in newer environments, and/or with enhanced features.

When the hardware platform changes, and a software product performs some low-level functions, maintenance is necessary. Also, whenever the support environment of a software product changes, the software product requires rework to cope up with the newer interface.

For instance, a software product may need to be maintained when the operating system changes or the software needs to run over hand held devices. Thus, every software product continues to evolve after its development through maintenance efforts.

Characteristics of Software Evolution

Lehman and Belady studied the characteristics of evolution of several software products [1980]. They expressed their observations in the form of laws.

Their important laws are presented in the following subsection. Most of their observations concern large software projects and may not be appropriate for the maintenance and evolution of very small products.

Lehman's first law:

A software product must change continually or become progressively less useful. Every software product continues to evolve after its development through maintenance efforts. Larger products stay in operation for longer times because of higher replacement costs and therefore tend to incur higher maintenance efforts. This law clearly shows that every product irrespective of how well designed must undergo maintenance.

In fact, when a product does not need any more maintenance, it is a sign that the product is about to be retired/discarded. This is in contrast to the common intuition that only badly designed products need maintenance. In fact, good products are maintained and bad products are thrown away.

Lehman's second law:

The structure of a program tends to degrade as more and more maintenance is carried out on it. The reason for the degraded structure is that usually maintenance activities result in patch work. It is rarely the case that members of the original development team are part of the maintenance team.

The maintenance team, therefore, often has a partial and inadequate understanding of the architecture, design, and code of the software. Therefore, any modifications tend to be ugly and more complex than they should be. Due to quick-fix solutions, in addition to degradation of structure, the documentations become inconsistent and become less helpful as more and more maintenance is carried out.

Lehman's third law:

Over a program' s lifetime, its rate of development is approximately constant. The rate of development can be quantified in terms of the lines of code written or modified. Therefore, this law states that the rate at which code is written or modified is approximately the same during development and maintenance.

11 b) Explain the architecture of a CASE environment? 5M

ARCHITECTURE OF A CASE ENVIRONMENT

The architecture of a typical modern CASE environment is shown diagrammatically in Figure . The important components of a modern CASE environment are user interface, tool set, *object management system* (OMS), and a repository.

User interface: The user interface provides a consistent framework for accessing the different tools thus making it easier for the users to interact with the different tools and reducing the overhead of learning how the different tools are used.

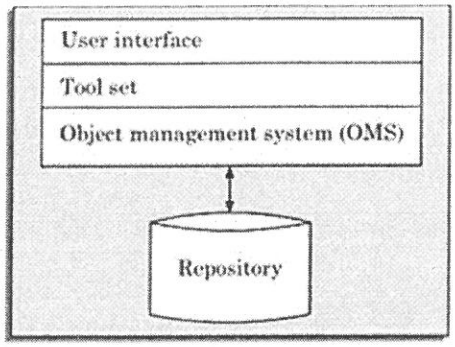
Object management system and repository

Different case tools represent the software product as a set of entities such as specification, design, text data, project plan, etc. The object management system maps these logical entities into the underlying storage management system (repository).

The commercial relational database management systems are geared towards supporting large volumes of information structured as simple relatively short records. There are a few types of

entities but large number of instances. By contrast, CASE tools create a large number of entities and relation types with perhaps a few instances of each.

Thus, the object management system takes care of appropriately mapping these entities into the underlying storage management system.



Architecture of a modern CASE environment

********** **THE END** **********