PVP 23

Code: 23CS3403, 23IT3403

II B.Tech - II Semester – Regular / Supplementary Examinations
APRIL 2026

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)	List the characteristics of good software.	L1	CO1
1.b)	Define Incremental Model with neat diagram.	L1	CO1
1.c)	How COCOMO model will be used for estimation.	L1	CO1
1.d)	Define Risk Identification.	L1	CO2
1.e)	What is Modularity in software design?	L1	CO3
1.f)	Tell about Object-Oriented Design.	L1	CO3
1.g)	What is Integration Testing?	L1	CO3
1.h)	Define Software Quality.	L1	C03
1.i)	What is Software Maintenance?	L1	CO4
1.j)	Sketch architecture of CASE environment.	L3	CO4

PART – B

			BL	CO	Max. Marks
UNIT-I					
2	a)	Explain Agile development models in detail.	L2	CO1	5 M
	b)	Describe the Waterfall model in detail with advantages and limitations.	L2	CO1	5 M
OR					
3	a)	Explain RAD models and its applicability with neat diagram.	L2	CO1	5 M
	b)	Describe exploratory style of software development with example.	L2	CO1	5 M
UNIT-II					
4	a)	Explain Software Project manager responsibilities and complexities.	L2	CO2	5 M
	b)	Demonstrate project planning activities in Software Project Management.	L3	CO2	5 M
OR					
5	a)	Explain Risk Analysis and Risk Mitigation strategies.	L2	CO2	5 M
	b)	Illustrate Empirical and Heuristic estimation techniques with examples.	L3	CO2	5 M
UNIT-III					
6	a)	Interpret about Function Oriented Design.	L3	CO3	5 M

	b)	Discuss Cohesion and Coupling with suitable examples.	L2	CO3	5 M
OR					
7	a)	Explain Structured Analysis methodology and DFD with example.	L2	CO3	5 M
	b)	Describe principles and golden rules of good User Interface Design.	L2	CO3	5 M
UNIT-IV					
8	a)	Demonstrate Black-box testing and White-box testing.	L3	CO3	5 M
	b)	Explain smoke Testing with suitable example.	L2	CO3	5 M
OR					
9	a)	Explain Software Quality and reliability.	L2	CO3	5 M
	b)	Discuss ISO 9000 certification process and quality standards.	L2	CO3	5 M
UNIT-V					
10	a)	Illustrate about CASE tools and their role in software development.	L3	CO4	5 M
	b)	Discuss reverse engineering and its uses.	L2	CO4	5 M
OR					
11	a)	Illustrate about Test case generator and its importance.	L3	CO4	5 M
	b)	Describe estimation of maintenance cost with suitable methods.	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-APRIL 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) List the characteristics of Good Software. 2M

- Any four or five characteristics

b) Define Incremental Model with neat diagram. 2M

- Definition-1M
- Diagram-1M

c) How COCOMO Model will be used for Estimation.2M

d)Define Risk Identification. 2M

- Definition-2M

e) What is Modularity in Software Design. 2M

f) Tell about Object Oriented Design. 2M

g) What is Integration Testing?2M

h) Define Software Quality. 2M

- Definition-2M

i)What is software Maintenance?2M

J) Sketch the Architecture of CASE environment. 2M

PART – B

UNIT-I

2a) Explain Agile Development Models in detail. 5M (CO1-L2)

- Iterative and Incremental model explanation-5M

2b) Describe the Waterfall Model in detail with advantages and Disadvantages.5M (CO1-L2)

- Waterfall model Importance-3M
- Advantages-1M
- Disadvantages-1M

(OR)

3a) Explain RAD Models and its applicability with neat diagram.5M(CO1-L2)

- RAD Model-3M
- Applications-2M

3 b) Describe Exploratory style of software development with example. 5M(CO1-L2)

- Exploratory style Importance-4M
- Example -1M

UNIT-II

4a) Explain Software project Manager Responsibilities and Complexities. 5M(CO2-L2)

- Responsibilities -2.5M
- Complexities-2.5M

4b) Demonstrate Project planning activities in Software project Management.5M(CO2-L3)

- Project planning activities -5M

(OR)

5a) Explain Risk Analysis and Risk Mitigation Strategies. 5M(CO2-L2)

- Risk Analysis and Mitigation-5M

5b) Illustrate Empirical and Heuristic Estimation Techniques with Examples. 5M(CO2-L3)

- Empirical -2.5M
- Heuristic -2.5M

UNIT-III

6a) Interpret about Function Oriented design.5M(CO3-L3)

- Function Oriented design -5M

6 b) Discuss Cohesion and coupling with suitable examples. 5M(CO3-L2)

- Cohesion with Examples-2.5M
- Coupling with Examples-2.5M

(OR)

7 a) Explain Structured Analysis Methodology and DFD with example. 5M(CO3-L2)

- Structured Analysis Methodology and DFD-4M
- Example-1M

7b) Describe Principles and golden rules of good user interface design. 5M(CO3-L2)

- Principles and golden rules of good user interface design-5M

UNIT-IV

8 a) Demonstrate black box testing and white box testing. 5M(CO3-L3)

- Black box testing-2.5M
- White box testing -2.5M

8 b) Explain Smoke Testing with suitable Example. 5M(CO3-L2)

- Smoke Testing-4M
- Example-1M

(OR)

9 a) Explain Software Quality and Reliability. 5M(CO3-L2)

- Software Quality -2.5M
- Software Reliability-2.5M

9 b) Discuss ISO 9000 Certification process and Quality Standards. 5M(CO3-L2)

- ISO 9000 Certification process and Quality Standards -5M

UNIT-V

10 a) Illustrate about CASE Tools and their role in software development. 5M(CO4-L3)

- CASE Tools and their role in software development-4M
- Diagram-1M

10 b) Discuss Reverse Engineering and its uses. 5M(CO4-L2)

- Reverse Engineering Importance-4M
- Uses-1M

(OR)

11 a) Illustrate about Test case Generator and its importance. 5M(CO4-L3)

- Test case Generator and its importance-5M

11 b) Describe Estimation of Maintenance cost with suitable methods. 5M(CO4-L2)

- Estimation of Maintenance cost with suitable methods-5M

Subject Code: 23CS3403, 23IT3403

PVP 23

PRASAD V POTLURI SIDDHARTHA INSTITUTE OF TECHNOLOGY

(AUTONOMOUS)

II B. Tech- II Semester- Regular Examinations-APRIL 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) List the characteristics of Good Software. **2M**

- Functionality
- Usability
- Reliability
- Performance
- Efficiency
- Maintainability
- Scalability
- Portability
- Security
- Flexibility

b) Define Incremental Model with neat diagram. **2M**

In the incremental life cycle model, the software is developed in increment. In this life cycle model, first the requirements are split into a set of increments. The first increment is a simple working system implementing only a few basic features. Over successive iterations, successive increments are implemented and delivered to the customer until the desired system is realized.

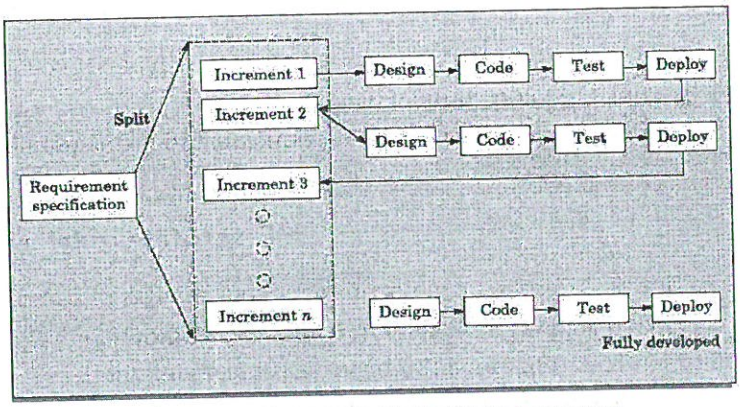

FIGURE 2.8 Incremental model of software development.

c) How COCOMO Model will be used for Estimation.2M

The COCOMO (Constructive Cost Model) model estimates software project efforts, costs, and schedules by analyzing project size (usually in Thousands of Delivered Source Instructions - KLOC) and using regression formulas based on historical data. It primarily calculates Person-Months.

d) Define Risk Identification. 2M

Risk identification is the foundational, often iterative, first step in risk management that involves systematically searching for, recognizing, and documenting potential internal or external threats and opportunities that could impact an organization's objectives or project success.

e) What is Modularity in Software Design. 2M

Modularity in software design is the technique of breaking a complex software system into smaller, self-contained, and manageable components called modules. It enhances maintainability and reduces complexity by ensuring each module handles a specific functionality and is loosely coupled with others. This approach allows for easier testing, debugging, and reusability of code.

f) Tell about Object Oriented Design. 2M

Object-Oriented Design (OOD) is a software engineering approach that models' systems as interacting objects, combining data (attributes) and behavior (methods) to improve modularity, maintainability, and reusability. It transforms analysis models into technical specifications using core principles like encapsulation, inheritance, and polymorphism.

g) What is Integration Testing?2M

Integration testing is a level of software testing where individual units or components of an application are combined and tested as a group. Its primary purpose is to verify that these modules interact and exchange data correctly, identifying bugs that occur at the interfaces between them.

h) Define Software Quality. 2M

Software quality is the degree to which a software product conforms to its specified requirements, technical standards, and user expectations, ensuring it is reliable, maintainable, and efficient.

i)What is software Maintenance?2M

Software maintenance is the process of modifying, updating, and optimizing a software application after its initial delivery to correct faults, improve performance, or adapt it to a changing environment.

J) Sketch the Architecture of CASE environment. 2M

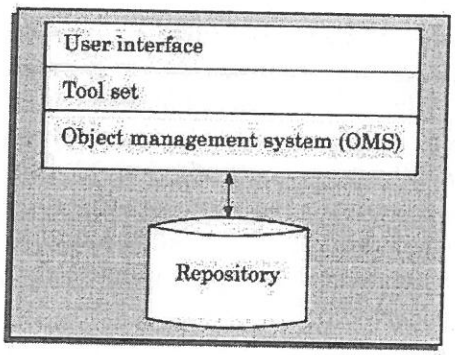

FIGURE 12.2 Architecture of a modern CASE environment.

PART – B

UNIT-I

2a) Explain Agile Development Models in detail. 5M (CO1-L2)

As already pointed out, though the iterative waterfall model has been very popular during the 1970s and 1980s, developers face several problems while using it on present-day software projects. The main difficulties include handling change requests from customers during product development, and the unreasonably high cost and time that is incurred while developing customized applications.

In the traditional iterative waterfall-based software development models, the requirements for the system are determined at the start of a development project and are assumed to be fixed from that point on. Later changes to the requirements after the SRS document has been completed are discouraged. If requirement changes become unavoidable later during the development life cycle, then the cost of accommodating these changes becomes prohibitively high. On the other hand, accumulated experience indicates that customers frequently change their requirements during the development period due to a variety of reasons. This is a major source of cost escalation in waterfall-based development projects.

over the last two decades or so, customized applications (services) have become common place. The prevalence of customized application development can be gauged from the fact that the sales revenue generated worldwide from services already exceeds that of the software products. Iterative waterfall model is not suitable for development of such software.

The main reason for this is that customization essentially involves reusing most of the parts of an existing application and carrying out only minor modifications to the other parts. For such development projects, the need for appropriate development models was acutely felt, and many researchers started to investigate this issue.

Waterfall model is called a heavy weight model, since there is too much emphasis on producing documentation and usage of tools. This is often a source of inefficiency and causes the project completion time to be much longer in comparison to the customer expectations.

Waterfall model prescribes almost no customer interactions after the requirements have been specified. In fact, in the waterfall model of software development, customer interactions are largely confined to the project initiation and project completion stages.

The agile software development model was proposed in the mid-1990s to overcome the shortcomings of the waterfall model of development identified above. The agile model could help a project to adapt to change requests quickly.1 Thus; a major aim of the agile models is to facilitate quick project completion. But, how is agility achieved in these models? Agility is achieved by fitting the process to the project. That is, it gave the required flexibility so that the activities that may not be necessary for a specific project could be easily removed. Also, anything that wastes time and effort is avoided.
Please note that agile model is being used as an umbrella term to refer to a group of development processes. While these processes share certain common characteristics, yet they do have certain subtle differences among themselves.

In an agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile models adopt an incremental and iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and lasts for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site. No long-term plans are made. The time to complete iteration is called a *time box*. The implication of the term *time box* is that the end date for iteration does not change. That is, the delivery date is considered sacrosanct.

2b) Describe the Waterfall Model in detail with advantages and Disadvantages.5M (CO1-L2).

The waterfall model and its derivatives were extremely popular in the 1970s and still are heavily being used across many development projects. The waterfall model is possibly the most obvious and intuitive way in which software can be developed through team effort. We can think of the waterfall model as a generic model that has been extended in many ways for catering to specific software development situations.

Waterfall Model

Classical waterfall model is intuitively the most obvious way to develop software. It is simple but idealistic. In fact, it is hard to put this model into use in any non-trivial software development project, since developers do commit many mistakes during various development activities many of which are noticed only during a later phase. This requires revisiting the work of a previous phase to correct the mistakes, but the classical waterfall model has no provision to go back to modify the artifacts produced during an earlier phase.

It can be easily observed from this figure that the diagrammatic representation of the classical waterfall model resembles a multi-level waterfall. This resemblance justifies the name of the Model

Phases of the waterfall model

The different phases of the classical waterfall model have been shown in Figure the different phases are—feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance. The phases starting from the feasibility study to the integration and system testing phase are known as the development phases. Software is developed during the development phases, and at the completion of the development phases, the software is delivered to the customer. After the delivery of software, customers start to use the software signaling the commencement of the operation phase. As the customers start to use the software, changes to it become necessary on account of bug fixes and feature extensions, causing maintenance works to be undertaken.

Therefore, the last phase is also known as the maintenance phase of the life cycle. It needs to be kept in mind that some of the text books have different number and names of the phases. An activity that spans all phases of software development is project management. Since it spans the entire project duration, no specific phase is named after it.

Project management, nevertheless, is an important activity in the life cycle and deals with managing all the activities that take place during software development and maintenance. In the waterfall model, different life cycle phases typically require relatively different amounts of efforts to be put in by the development team.

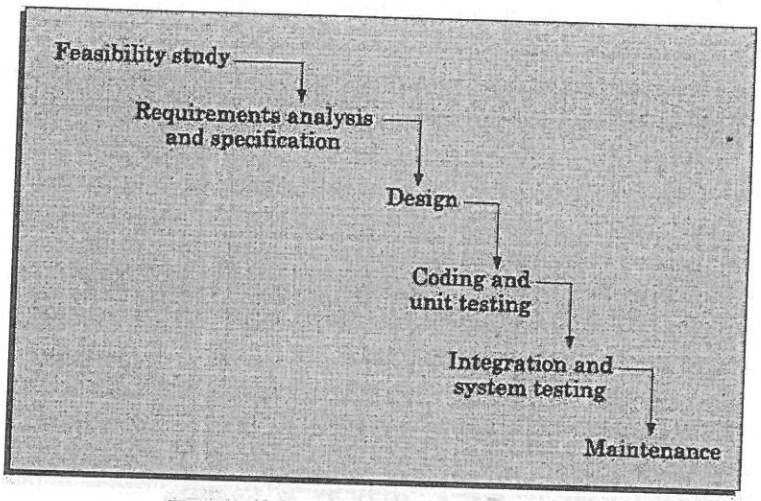

FIGURE 2.1 Classical waterfall model.

Advantages

- **Simple & Easy to Manage:** Due to its rigid structure, each phase has specific deliverables and a review process.
- **Clear Documentation:** Extensive documentation at each stage ensures that knowledge is not lost if team members leave.
- **Predictability:** Works well for projects where requirements are clearly defined and fixed.

Disadvantages

- **Inflexibility to Changes:** High difficulty in changing requirements once the development phase has started.
- **Late Testing & Integration:** Testing is done at the end, meaning defects are found late, which can be costly.
- **No Working Software Until Late:** A fully functional product is not produced until late in the life cycle.

(OR)

3a) Explain RAD Models and its applicability with neat diagram.5M(CO1-L2)

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 emphasizes 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 specialized tools to facilitate fast creation of working prototypes.

Applicability of RAD Model

The following are some of the characteristics of an application that indicate its suitability to RAD style of development:

Customized software: As already pointed out a customized software is developed for one or two customers only by adapting an existing software. In customized software development projects, substantial reuse is usually made of code from pre-existing software

Non-critical software: The RAD model suggests that quick and dirty software should first be developed and later this should be refined into the final software for delivery. Therefore, the developed product is usually far from being optimal in performance and reliability. In this regard, for well understood development projects and where the scope of reuse is rather restricted, the iterative waterfall model may provide a better solution.

Highly constrained project schedule: RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.

Large software: Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

3b) Describe Exploratory style of software development with example. 5M(CO1-L2)

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.

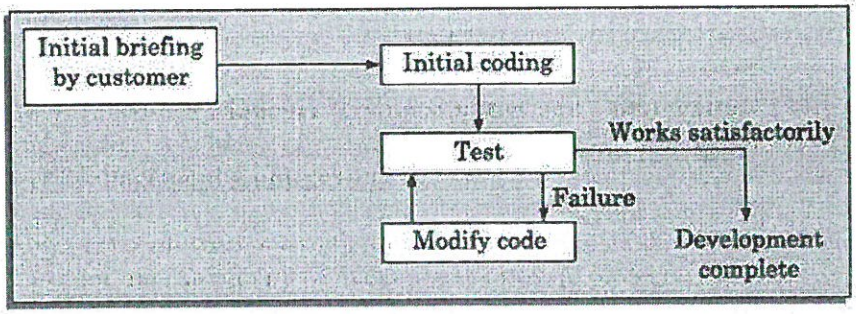

Figure 1.4 Exploratory program development

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-II

4a) Explain Software project Manager Responsibilities and Complexities. 5M(CO2-L2)

Software Project Management Complexities

- **Invisibility:** Invisibility of software makes it difficult to assess the progress of a project and is a major cause for the complexity of managing a software project.
- **Changeability:** Frequent changes to the requirements and the invisibility of software are possibly the two major factors making software project management a complex task.
- **Complexity:** Even a moderate sized software has millions of parts (functions) that interact with each other in many ways—data coupling, serial and concurrent runs, state transitions, control dependency, file sharing, etc.
- **Uniqueness:** Every software project is usually associated with many unique features or situations. This makes every project much different from the others. This is unlike projects in other domains, such as car manufacturing or steel manufacturing where the projects are more predictable. Due to the uniqueness of the software projects, a project manager in the course of a project faces many issues that are quite unlike the ones he/she might have encountered in the past.
- **Exactness of the solution:** Mechanical components such as nuts and bolts typically work satisfactorily as long as they are within a tolerance of 1 per cent or so of their specified sizes. However, the parameters of a function call in a program are required to be in complete conformity with the function definition.

- **Team-oriented and intellect-intensive work:** Software development projects are akin to research projects in the sense that they both involve team-oriented, intellect-intensive work. In contrast, projects in many domains are labor-intensive and each member works in a high degree of autonomy. Examples of such projects are planting rice, laying roads, assembly line manufacturing, constructing a multistoried building, etc.

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. In fact, 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.

We can broadly classify a project manager's varied responsibilities into the following two major categories:

- 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 andmore 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

4b) Demonstrate Project planning activities in Software project Management.5M(CO2-L3)

Project Planning

Project managers undertake project planning. Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale.

It can even cause project failure. For this reason, project planning is undertaken by the project managers with utmost care and attention. However, for effective project planning, in addition to a thorough knowledge of the various estimation techniques, past experience is crucial.

During project planning, the project manager performs the following activities. Note that we have given only a very brief description of the activities.

- **Estimation:** The following project attributes are estimated.

  Cost: 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?

The effectiveness of all later planning activities such as scheduling and staffing are dependent on the accuracy with which these three estimations have been made.

- **Scheduling:** After all the necessary project parameters have been estimated, 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.
- Figure 3.1 shows the order in which the planning activities are undertaken. Observe that size estimation is the first activity that a project manager undertakes during project planning.
- As can be seen from Figure 3.1, based on the size estimation, the effort required to complete a project and the duration over which the development is to be carried out are estimated. Based on

the effort estimation, the cost of the project is computed. The estimated cost forms the basis on which price negotiations with the customer is carried out. Other planning activities such as staffing, scheduling etc. are undertaken based on the effort and duration estimates made. In Section 3.7, we shall discuss a popular technique for estimating the project parameters. Subsequently, we shall discuss the staffing and scheduling issues.

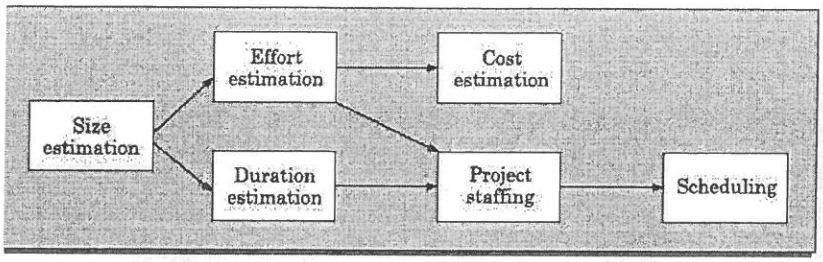

FIGURE 3.1 Precedence ordering among planning activities.

Sliding Window Planning

- It is usually very difficult to make accurate plans for large projects at project initiation. A part of the difficulty arises from the fact that large projects may take several years to complete.
- As a result, during the span of the project, the project parameters, scope of the project, project staff, etc., often change drastically resulting in the initial plans going haywire. In order to overcome this problem, sometimes project managers undertake project planning over several stages.
- That is, after the initial project plans have been made, these are revised at frequent intervals. Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning known as *sliding window planning*.
- At the start of a project, the project manager has incomplete knowledge about the nitty-gritty of the project. His information base gradually improves as the project progresses through different development phases. The complexities of different project activities become clear, some of the anticipated risks get resolved, and new risks appear. The project parameters are re-estimated periodically as understanding grows and also a periodically as project parameters change.

The SPMP Document of Project Planning

- Once project planning is complete, project managers document their plans in a software project management plan (SPMP) document. Listed below are the different items that the SPMP document should discuss. This list can be used as a possible organization of the SPMP document.

Organization of the software project management plan (SPMP) document

- **1. Introduction**

  (a) Objectives

  (b) Major Functions

(c) Performance Issues

(d) Management and Technical Constraints

- **2. Project estimates**

(a) Historical Data Used

(b) Estimation Techniques Used

(c) Effort, Resource, Cost, and Project Duration Estimates

- **3. Schedule**

(a) Work Breakdown Structure

(b) Task Network Representation

(c) Gantt Chart Representation

(d) PERT Chart Representation

- **4. Project resources**

(a) People

(b) Hardware and Software

(c) Special Resources

- **5. Staff organization**

(a) Team Structure

(b) Management Reporting

- **6. Risk management plan**

(a) Risk Analysis

(b) Risk Identification

(c) Risk Estimation

(d) Risk Abatement Procedures

- **7. Project tracking and control plan**

(a) Metrics to be tracked

(b) Tracking plan

(c) Control plan

- **8. Miscellaneous plans**

  (a) Process Tailoring

  (b) Quality Assurance Plan

  (c) Configuration Management Plan

(d) Validation and Verification

(e) System Testing Plan

(f) Delivery, Installation, and Maintenance Plan

(OR)

5 a) Explain Risk Analysis and Risk Mitigation Strategies. 5M(CO2-L2)

Risk Analysis and Risk Mitigation Strategies

After all the identified risks of a project have been assessed, plans are made to contain the most damaging and the most likely risks first. Different types of risks require different containment procedures. In fact, most risks require considerable ingenuity on the part of the project manager in tackling the risks.

There are three main strategies for risk containment:

- **Avoid the risk:** Risks can be avoided in several ways. Risks often arise due to project constraints and can be avoided by suitably modifying the constraints. The different categories of constraints that usually give rise to risks are:
- ***Process-related risk:*** These risks arise due to aggressive work schedule, budget, and resource utilization.
- ***Product-related risks:*** These risks arise due to commitment to challenging product features (e.g. response time of one second, etc.), quality, reliability, etc.
- ***Technology-related risks:*** These risks arise due to commitment to use certain technology (e.g., satellite communication).

  A few examples of risk avoidance can be the following: Discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover, etc.

- **Transfer the risk:** This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.
- **Risk reduction:** This involves planning ways to contain the damage due to a risk. For example, if there is risk that some key personnel might leave, new recruitment may be planned. The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use.

- For example, if you are using a compiler for recognizing user commands, you would have to construct a compiler for a small and very primitive command language first. There can be several strategies to cope up with a risk. To choose the most appropriate strategy for handling a risk, the project manager must consider the cost of handling the risk and the corresponding reduction of risk. risk leverage = cost of reduction
- Even though we identified three broad ways to handle any risk, effective risk handling cannot be achieved by mechanically following a set procedure, but requires a lot of ingenuity on the part of the project manager. As an example, let us consider the options available to contain an important type of risk that occurs in many software projects—that of schedule slippage.

5b) Illustrate Empirical and Heuristic Estimation Techniques with Examples. 5M(CO2-L3)5M

A large number of estimation techniques have been proposed by researchers. These can broadly be classified into three main categories:

- Empirical estimation techniques
- Heuristic techniques

Empirical Estimation Techniques

Empirical estimation techniques are essentially based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalized to a large extent. We shall discuss two such formalizations of the basic empirical estimation techniques known as expert judgement and the Delphi techniques in Sections

Heuristic Techniques

Heuristic techniques assume that the relationships that exist among the different project parameters can be satisfactorily modelled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the values of the independent parameters in the corresponding mathematical expression. Different heuristic estimation models can be divided into the following two broad categories—single variable and multivariable models.

Single variable estimation models assume that various project characteristic can be predicted based on a single previously estimated basic (independent) characteristic of the software such as its size. A single variable estimation model assumes that the relationship between a parameter to be estimated and the corresponding independent parameter can be characterized by an expression of the following form:

- Estimated Parameter = $c1 \times ed1$
- In the above expression, e represents a characteristic of the software that has already been estimated (independent variable).

Estimated Parameter is the dependent parameter (to be estimated). The dependent parameter to be estimated could be effort, project duration, staff size, etc., $c1$ and $d1$ are constants. The values of the constants $c1$ and $d1$ are usually determined using data collected from past projects (historical data). The COCOMO model discussed in Section 3.7.1, is an example of a single variable cost estimation model.

- A multivariable cost estimation model assumes that a parameter can be predicted based on the values of more than one independent parameter. It takes the following form:
- Estimated Resource = $c1 \times pd1 + c2 \times pd2 + L$
- where, $p1$, $p2$, ... are the basic (independent) characteristics of the software already estimated, and $c1$, $c2$, $d1$, $d2$, ... are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modelled by the different sets of constants $c1$, $d1$, $c2$, $d2$, ... Values of these constants are usually determined from an analysis of historical data. The intermediate COCOMO model discussed in Section 3.7.2 can be considered to be an example of a multivariable estimation model
- We have already pointed out that empirical estimation techniques have, over the years, been formalized to a certain extent. Yet, these are still essentially euphemisms for pure guess work. These techniques are easy to use and give reasonably accurate estimates. Two popular empirical estimation techniques are—Expert judgement and Delphi estimation techniques. We discuss these two techniques in the following subsection.
- **Expert Judgement**

  Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for then individual modules to arrive at the overall estimate. However, this technique suffers from several shortcomings. The outcome of the expert judgement technique is subject to human errors and individual bias. Also, it is possible that an expert may overlook some factors inadvertently. Further, an expert making an estimate may not have relevant experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts, but may not be very knowledgeable about the computer communication part. Due to these factors, the size estimation arrived at by the judgement of a single expert may be far from being accurate.

- A more refined form of expert judgement is the estimation made by a group of experts. Chances of errors arising out of issues such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates is minimized when the estimation is done by a group of experts.
- However, the estimate made by a group of experts may still exhibit bias. For example, on certain issues the entire group of experts may be biased due to reasons such as those arising out of

political or social considerations. Another important shortcoming of the expert judgement technique is that the decision made by a group may be dominated by overly assertive members.

Delphi Cost Estimation

- Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the *software requirements specification* (SRS) document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced their estimations. The coordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussions among the estimators is allowed during the entire estimation process.
- The purpose behind this restriction is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate. The Delphi estimation, though consumes more time and effort, overcomes an important shortcoming of the expert judgement technique in that the results cannot unjustly be influenced by overly assertive and senior members.

UNIT-III

6a) Interpret about Function Oriented design.5M(CO3-L3)

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.

6 b) Discuss Cohesion and coupling with suitable examples. 5M(CO3-L2)

Cohesion and Coupling

- We have so far discussed that effective problem decomposition is an important characteristic of a good design. Good module decomposition is indicated through *high cohesion* of the individual modules and *low coupling* of the modules with each other. Let us now define what is meant by cohesion and coupling.
- In this section, we first elaborate the concepts of cohesion and coupling. Subsequently, we discuss the classification of cohesion and coupling.
- **Coupling:** Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:
- If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
- If the interactions occur through some shared data, then also we say that they are highly coupled.
- If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.
- **Cohesion:** To understand cohesion, let us first understand an analogy. Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution.
- When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion. If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion
- **Functional independence**
- By the term *functional independence,* we mean that a module performs a single task and needs very little interaction with other modules. Functional independence is a key to any good design primarily due to the following
- **Error isolation:** Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules advantages it offers:
- Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.

- **Scope of reuse:** Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program. On the other hand, if a module interacts with several other modules or the functions of a module perform very different tasks, then it would be difficult to reuse it. This is especially so, if the module accesses the data (or code) internal to other modules.
- **Understandability:** When modules are functionally independent, complexity of the design is greatly reduced. This is because of the fact that different modules can be understood in isolation, since the modules are independent of each other that understandability is a major advantage of a modular design. Besides the three we have listed here, there are many other advantages of a modular design as well. We shall not list those here, and leave it as an assignment to the reader to identify them.

Classification of Cohesiveness

- Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective. The different modules of a design can possess different degrees of freedom. However, the different classes of cohesion that modules can possess
- The cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible. These different classes of cohesion are elaborated below.

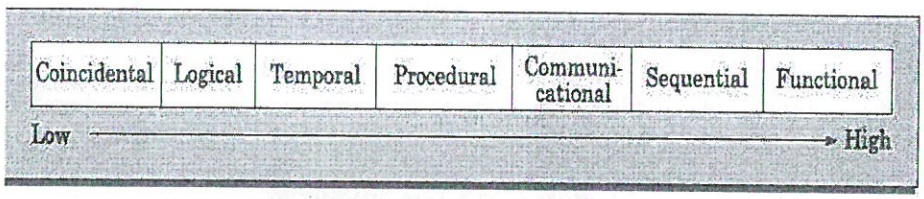

FIGURE 5.3 Classification of cohesion.

- **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions.
- It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design. The designs made by novice programmers often possess this category of cohesion, since they often bundle functions to modules rather arbitrarily.
- An example of a module with coincidental cohesion has been shown in Figure 5.4(a). Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

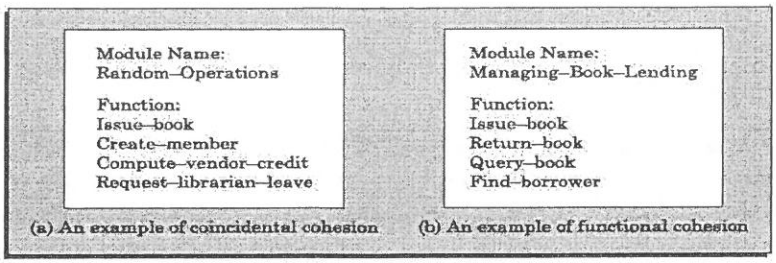

FIGURE 5.4 Examples of cohesion.

- **Logical cohesion:** A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc.
- As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.
- **Temporal cohesion:** When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.
- As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.
- Other examples of modules having temporal cohesion are the following. Similarly, a module would exhibit temporal cohesion, if it comprises functions for performing initialization, or start-up, or shut-down of some process.
- **Procedural cohesion:** A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.
- Consider the activities associated with order processing in a trading house. The functions login (), place-order (), check-order (), print-bill (), place-order-on-vendor (), update inventory(), and logout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.
- **Communicational cohesion:** A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admit Student, enter Marks, print Grade Sheet, etc. access and manipulate data stored in an array named student Records defined within the module.
- **Sequential cohesion:** A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence. As an example, consider the following situation. In an on-line store consider that after a customer request for some item, it is first determined if the item is in stock. In this case, if the functions create-order (), check-item-availability (), place-order-on-vendor () are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function

create-order () creates an order that is processed by the function check-item-availability () (whether the items are available in the required quantities in the inventory) is input to place-order-on-vendor ().

- **Functional cohesion:** A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., compute Overtime(), compute Work Hours(), compute Deductions(), etc.) work together to generate the pay slips of the employees.
- Another example of a module possessing functional cohesion. In this example, the functions issue-book(), return-book(), query-book(), and find-borrower(), together manage all activities concerned with book lending. When a module possesses functional cohesion, then we should be able to describe what the module does using only one simple sentence. For we can describe the overall responsibility of the module by saying "It manages the book V lendingprocedure of the library."
- A simple way to determine the cohesiveness of any given module is as follows. First examine what do the functions of the module perform. Then, try to write down a sentence to describe the overall work performed by the module. If you need a compound sentence to describe the functionality of the module, then it has sequential or communicational cohesion. If you need words such as "first", "next", "after", "then", etc., then it possesses sequential or temporal cohesion. If it needs words such as "initialize", "setup", "shut down", etc., to define its functionality, then it has temporal cohesion. We can now make the following observation. A cohesive module is one in which the functions interact among themselves heavily to achieve a single goal. As a result, if any of these functions is removed to a different module, the coupling would increase as the functions would now interact across two different modules.

Classification of Coupling

- The coupling between two modules indicates the degree interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. We can alternately state this concept as follows.
- The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.
- Let us now classify the different types of coupling that can exist between two modules. Between any two interacting modules, any of the following five different types of coupling can exist. These different types of coupling, in increasing order of their severities

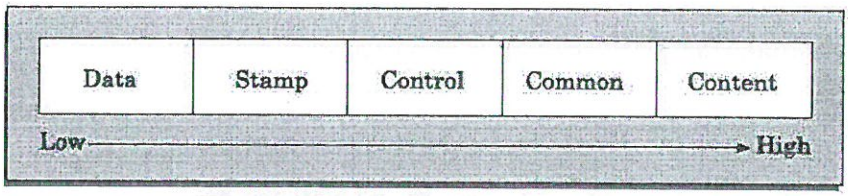

FIGURE 5.5 Classification of coupling.

- **Data coupling:** Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.
- **Stamp coupling:** Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.
- **Control coupling:** Control coupling exists between two modules, if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module and tested in another module.
- **Common coupling:** Two modules are common coupled, if they share some global data items.
- **Content coupling:** Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.
- The degree of coupling increases from data coupling to content coupling. High coupling among modules not only makes a design solution difficult to understand and maintain, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

(OR)

7 a) Explain Structured Analysis Methodology and DFD with example. 5M(CO3-L2)

Structured Analysis

We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analyzed, and the data flow among these processing tasks are represented graphically. Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978]. The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results using *data flow diagrams* (DFDs).
- DFD representation of a problem, as we shall see shortly, is very easy to construct. Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.
- Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed. In fact, it completely ignores aspects such as control flow, the specific algorithms used by the functions, etc. In the DFD terminology, each function is called a

process or a *bubble*. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.

- DFD is an elegant modelling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organization. how a DFD can be used to represent the processing activities and flow of material in an automated car assembling plant. We now elaborate how a DFD model can be constructed.

Data Flow Diagrams (DFDs)

- 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.
- The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism—it is simple to understand and use. A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the data flow among these functions.

Developing the DFD Model of a System

- A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
- The DFD model of a system is constructed by using a hierarchy of DFDs (see Figure 6.4). The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand.
- At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.
- To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model.
- All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.
- To develop the context diagram of the system, we have to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also include any external systems which supply data to or receive data from the system.

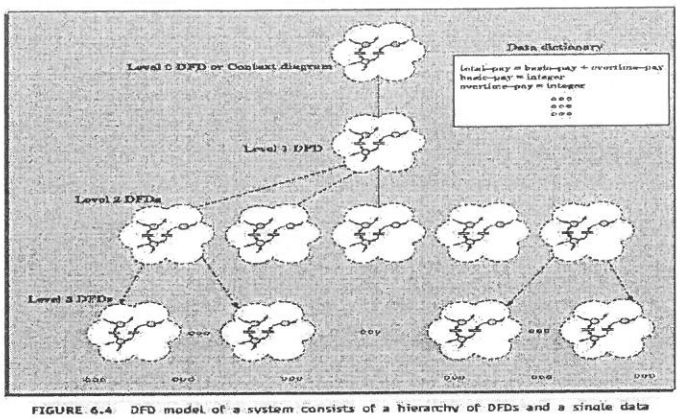

FIGURE 6.4 DFD model of a system consists of a hierarchy of DFDs and a single data

Level 1 DFD

- The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions. To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD. Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

Decomposition

- Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as *factoring* or *exploding* a bubble. Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles. A few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes trivial and redundant. On the other hand, too many bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm. We can now describe how to go about developing the DFD model of a system more systematically.

1. **Construction of context diagram:** Examine the SRS document to determine:

- Different high-level functions that the system needs to perform.
- Data input to every high-level function.
- Data output from every high-level function.
- Interactions (data flow) among the identified high-level functions.
- Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level *data flow diagram* (DFD), usually called the DFD 0.

2. **Construction of level 1 diagram:** Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-

level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

3. **Construction of lower-level diagrams:** Decompose each high-level function into its constituent subfunctions through the following set of activities:

- Identify the different subfunctions of the high-level function.
- Identify the data input to each of these subfunctions.
- identify the data output from each of these subfunctions.
- Identify the interactions (data flow) among these subfunctions.
- Represent these aspects in a diagrammatic form using a DFD.
- Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers helping uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubble sat level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is decomposed, its children bubble is numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and its successors.

Balancing DFDs

- The DFD model of a system usually consists of many DFDs that are organized in a hierarchy. In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD. We illustrate the concept of balancing a DFD in Figure 6.5. In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1, 0.1.2, 0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in. Please note that dangling arrows (d1, d2, d3) represent the data flows into or out of a diagram.

7b) Describe Principles and golden rules of good user interface design. 5M(CO3-L2)

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.

UNIT-IV

8 a) Demonstrate black box testing and white box testing. 5M(CO3-L3)

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 hidden and nothing is known about it.	White Box Testing is a way of testing the software in which the tester has knowledge about the 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 design patterns.
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.

Parameters	Black Box Testing	White Box Testing
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.
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) Explain Smoke Testing with suitable Example. 5M(CO3-L2)

Smoke Testing

Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail. The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing. For smoke testing, a few test cases are designed to check whether the basic functionalities are working

Smoke testing, also known as "Build Verification Testing" or "Build Acceptance Testing," is a type of software testing that is typically performed at the beginning of the development process to ensure that the most critical functions of a software application are working correctly. It is used to quickly identify and fix any major issues with the software before more detailed testing is performed.

For example, for a library automation system, the smoke tests may check whether books can be created and deleted, whether member records can be created and deleted, and whether books can be loaned and returned.

(OR)

9 a) Explain Software Quality and Reliability. 5M(CO3-L2)

Reliability also depends upon how the product is used, or on its *execution profile*. If the users execute only those features of a program that are "correctly" implemented, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if only those functions of the software which contain errors are invoked, then a large number of failures will be observed and the perceived reliability of the system will be very low. Different categories of users of a software product typically execute different functions of a software product.

For example, for a Library Automation Software the library members would use functionalities such as issue book, search book, etc., on the other hand the librarian would normally execute features such as create member, create book record, delete member record, etc. So, defects which show up for the librarian, may not show up for the members.

Software Quality shows how good and reliable a product is. To convey an associate degree example, think about functionally correct software. It performs all functions as laid out in the SRS document. But it has an associate degree virtually unusable program. even though it should be functionally correct, we tend not to think about it to be a high-quality product.

Factors of Software Quality

The modern read of high-quality associates with software many quality factors like the following:

1. Portability: A software is claimed to be transportable, if it may be simply created to figure in several package environments, in several machines, with alternative code products, etc.
2. Usability: A software has smart usability if completely different classes of users (i.e. knowledgeable and novice users) will simply invoke the functions of the products.

3. Reusability: A software has smart reusability if completely different modules of the products will simply be reused to develop new products.
4. Correctness: Software is correct if completely different needs as laid out in the SRS document are properly enforced.
5. Maintainability: A software is reparable, if errors may be simply corrected as and once they show up, new functions may be simply added to the products, and therefore the functionalities of the products may be simply changed, etc
6. Reliability: Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make.

9 b) Discuss ISO 9000 Certification process and Quality Standards. 5M(CO3-L2)

- ISO 9000 certification serves as a reference for contract between independent parties. In particular, a company awarding a development contract can form his opinion about the possible vendor performance based on whether the vendor has obtained ISO 9000 certification or not. In this context, the ISO 9000 standard specifies the guidelines for maintaining a quality system. We have already seen that the quality system of an organization applies to all its activities related to its products or services.
- The ISO standard addresses both operational aspects (that is, the process) and organizational aspects such as responsibilities, reporting, etc. In a nutshell, ISO 9000 makes a set of recommendations for repeatable and high-quality product development. It is important to realize that ISO 9000 standard is a set of guidelines for the production process and is not directly concerned about the product itself.
- ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO 9003.
- The types of software companies to which the different ISO standards apply are as follows:
- **ISO 9001:** This standard applies to the organizations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organisations.
- **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 for Software Industry

- 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 organizations.

- 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 activities. Two major differences between software development and development of other kinds of products are as follows:
- Software is intangible and therefore difficult to control. It means that software would not be visible to the user until the development is complete and the software is up and running. It is difficult to control and manage anything that you cannot see and feel. In contrast, in any other type of product manufacturing such as car manufacturing, you can see a product being developed through various stages such as fitting engine, fitting doors, etc. Therefore, it becomes easy to accurately determine how much work has been completed and to estimate how much more time will it take.
- During software development, the only raw material consumed is data. In contrast, large quantities of raw materials are consumed during the development of any other product. As an example, consider a steel making company. The company would consume large amounts of raw material such as iron-ore, coal, lime, manganese, etc. Not surprisingly then, many clauses of ISO 9000 standards are concerned with raw material control. These clauses are obviously not relevant for software development organizations.

UNIT-V

10 a) Illustrate about CASE Tools and their role in software development. 5M(CO4-L3)

CASE tools are characterized 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.

Examples of popular programming environments are Turbo C environment, Visual Basic, Visual C++, etc. A schematic representation of a CASE environment

The standard programming environments such as Turbo C, Visual C++, etc. come equipped with a program editor, compiler, debugger, linker, etc. All these tools are integrated. If you click on an error reported by the

compiler, not only does it take you into the editor, but also takes the cursor to the specific line or statement causing the error.

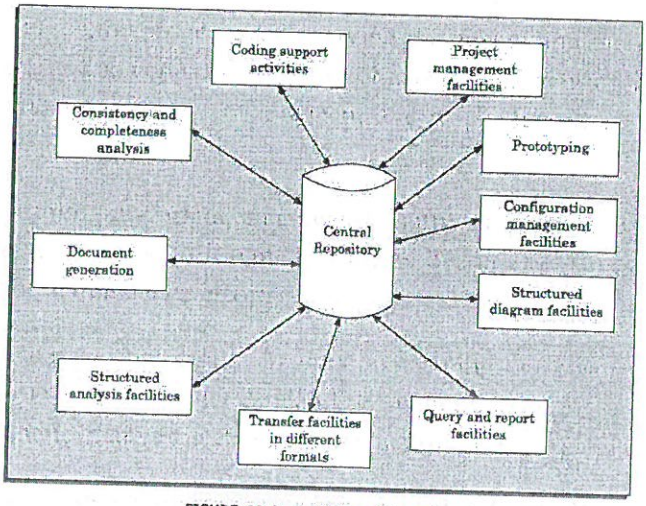

FIGURE 12.1 A CASE environment.

10b) Discuss Reverse Engineering and its uses. 5M(CO4-L2)

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.

The first stage of reverse engineering usually focuses on carrying out cosmetic changes to the code to improve its readability, structure, and understandability, without changing any of its functionalities. A way to carry out these cosmetic changes is shown schematically in Figure 13.1.

A program can be reformatted using any of the several available Pretty Printer programs which layout the program neatly. Many legacy software products are difficult to comprehend with complex control structure and unthoughtful variable names. Assigning meaningful variable names is important that meaningful variable names is the most helpful code documentation. All variables, data structures, and functions should be assigned meaningful names wherever possible.

Complex nested conditionals in the program can be replaced by simpler conditional statements or whenever appropriate by case statements.

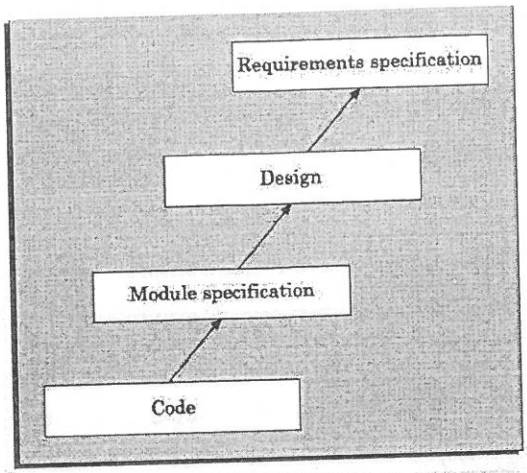

FIGURE 13.1 A process model for reverse engineering.

After the cosmetic changes have been carried out on a legacy software, the process of extracting the code, design, and the requirements specification can begin. These activities are schematically shown in Figure 13.2. In order to extract the design, a full understanding of the code is needed. Some automatic tools can be used to derive the data flow and control flow diagram from the code. The structure chart (module invocation sequence and data interchange among modules) should also be extracted. The SRS document can be written once the full code has been thoroughly understood and the design extracted.

(OR)

11 a) Illustrate about Test case Generator and its importance. 5M(CO4-L3)

Test CASE Generator

The CASE tool for test case generation should have the following features:

▶It should support both design and requirement testing.

▶It should generate test set reports in ASCII format which can be directly imported into the test plan document.

A **Test Case Generator** is a software tool, often leveraging artificial intelligence (AI) and machine learning (ML), that automatically creates test scenarios, steps, and expected outcomes from input sources such as requirements documents, user stories, wireframes, or application URLs

These tools are designed to move software testing from a manual, time-intensive process to an automated, intelligent, and scalable workflow, overcoming the common bottleneck where test creation cannot keep pace with development velocity.

Test case generators are essential for modern Agile and DevOps teams due to their ability to provide high ROI and efficiency.

- **Drastically Reduces Manual Effort:** By automating the creation of test steps and documentation, testers save significant time, shifting their focus from tedious documentation to strategic testing activities.
- **Increases Test Coverage:** AI can identify and generate test cases for edge cases, complex logic, and boundary conditions that human testers might overlook.
- **Accelerates Testing Cycles (Faster Feedback):** Instead of spending weeks on test design, generators create comprehensive test suites in hours, enabling faster feedback for developers and quicker software releases.

11 b) Describe Estimation of Maintenance cost with suitable methods. 5M(CO4-L2)

Estimation of Maintenance Cost

We had earlier pointed out that maintenance efforts require about 60 per cent of the total life cycle cost for a typical software product. However, maintenance costs vary widely from one application domain to another. For embedded systems, the maintenance cost can be as much as 2 to 4 times the development cost.

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model. Boehm's maintenance cost estimation is made in terms of a quantity called the *annual change traffic* (ACT). Boehm defined ACT as the fraction of a software product's source instructions which undergo change during a typical year either through addition or deletion.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

where, KLOC added is the total kilo lines of source code added during maintenance. KLOC deleted is the total KLOC deleted during maintenance. Thus, the code that is changed, should be counted in both the code added and code deleted.

The annual change tra ffi c (ACT) is multiplied with the total development cost to arrive at the maintenance cost:

Maintenance cost = ACT × Development cost

Most maintenance cost estimation models, however, give only approximate results because they do not take into account several factors such as experience level of the engineers, and familiarity of the engineers with the product, hardware requirements, software complexity, etc.