Study Guide

For

Software Engineering

Department of Computer Science

Faculty of Informatics

ቅድስት ማርያም ዩኒቨርስቲ ኮሌጅ

St. Mary's University College

Compiled By:

Mohammed_hussen Abubeker

January 2012

Summary of the Study Guide

The aim this course is to provide a general introduction to software Engineering and identify the important phases of any software project developments. Chapter 1 provides a general introduction to the field in order to give some sense of the magnitude and importance of software in today's world, the kinds of problems that make software development difficult, and an outline of how software development is undertaken. Chapter 2 provides more detail on the idea of a "software process", that is, on the various stages software goes through, from the planning stages to its delivery to the customer and beyond. Different models of the process are introduced, and the types of project features for which each is most appropriate are discussed. Chapters 3 through 12 follow, in order, the major phases in the life of a software system. Chapter 3 deals with requirement elicitation and analysis principles. This chapter concentrates on the methods that are necessary to fully capture the customer's requirements for the system, and how to specify them in a way that will be useful for future needs. Chapter 4 discusses how this requirement could be modeled through analysis model. Chapter 5 discusses the design of the software using structured, introducing broad architectural styles that may be useful for different types of systems as well as more specific design characteristics. This chapter sketches the roles of the people involved in producing the design, as well as measures that can be used to assess a design's quality. Chapter 7 explores structured approach software testing.

Chapter 8 and 9 explores an important concepts, analysis and design paradigm of Object Orientation, in more detail and shows how the design notation captures useful information about several aspects of the problem and the resulting system. An overview of different types of testing, as well as testing tools and methods, are presented in chapter 10. Chapter 11 discusses system maintenance, that is, the part of the life-cycle that comes after delivery. The nature of the problems that may arise with the system in this phase, as well as techniques and tools for performing maintenance, are presented. Finally chapter 12 deals with software project management: how resources and cost are estimated, how risks are identified and planned, and how schedules are created.

CHAPTER ONE
INTRODUCTION

General Objectives:

Students are expected to understand what software engineering in general and the system approach to software development.

Specific objectives:

After studying this chapter, you should be able to:

- Describe what is meant by software engineering
- Identify the characteristics of "good software".
- Define what is meant by a systems approach to building software and
- Describe why a systems approach is important.
- Describe how software engineering has changed since the 1970s.

Summary:

The main emphasis of this chapter is to lay the groundwork for the remaining topics or chapters in the guide. After brief history of software engineering is discussed to motivate the students, major points are discussed. The chapter explores the need to take a systems approach to building software. Software engineers use their knowledge of computers and computing to help solve problems. For problem-solving, software engineering makes use of analysis and synthesis. Software engineers begin investigating a problem by analyzing it, breaking it into pieces that are easier to deal with and understand. Once a problem is analyzed, a solution is synthesized based on the analysis of the pieces. To help solve problems, software engineers employ a variety of methods, tools, procedures and paradigms.

The development of software involves requirements analysis, design, implementation, testing, configuration management, quality assurance and more. Software engineers must select a development process that is appropriate for the team size, risk level and application domain. Tools that are well –integrated and support the type of communication the project demands must be selected. Measurements and supporting tools should be used to supply as much visibility and understanding as possible.

Software engineering has had both positive and negative results in the past. Existing software have enabled us to perform tasks more quickly and effectively than ever before. In addition, software has enabled us to do things never done before. However, software is not without its problems. Often software systems function, but not exactly as expected. In some cases, when a system fails, it is a minor annoyance. In other cases, system failures can be life-threatening.

This has led software engineers to find methods to assure that their products are of acceptable quality and utility. Quality must be viewed from several different perspectives. Software engineers must understand that technical quality and business quality may be very different.

Exercises with answer key:

1. What is software?
2. What is software engineering?
3. What is the difference between technical and business quality? Explain why each is important.
4. Look through several issues of software magazines (IEEE Computer and IEEE Software are good choices) from the 1970's, 1980's and recent issues. Compare the types of problems and solutions described in the older issues with those described in the more recent issues.

Answer keys:

1. The answer to this question could be:

a. Computer programs and associated documentation

Software products may be developed for a particular customer or may be developed for a general market**.** Software products may be

- Generic - developed to be sold to a range of different customers
- Custom - developed for a single customer according to their specification

2. Software engineering is the study or practice of using computers and computing technology to solve real-world problems. Or Software engineering is an engineering discipline which is concerned with all aspects of software production. Software engineers should adopt a systematic and organised approach to their work and use appropriate tools and techniques depending on the problem to be solved, the development constraints and the resources available.
3. Answers will be specific to the types of failures that you identify. The purpose of this exercise is to make the distinction between errors, faults and failures clear. Review the definitions for errors, faults and failures.
4. Answers to this question will vary depending upon which articles are involved. To answer this question, you may want to use the seven key factors that have altered software engineering (from Wasserman (1996) and presented in Section 1.8 of the textbook) to make your comparison among articles from the past and recent articles. In your comparison, cite specific examples of how the problems and solutions have changed.

Question without answer key:

1. Give two or three examples of failures you have encountered while using software. Describe how these failures affected the quality of the software product.
2. Examine failures that have occurred in software that you have written. Identify and list the faults and errors that caused each failure.
3. What are the essential characteristics of software engineering?
4. Discuss the major differences between software engineering and some other engineering discipline, such as bridge design or house building. Would you consider state-of-the-art software engineering as a true engineering discipline?

CHAPTER TWO

SOFTWARE PROCESS MODELS AND LIFE CYCLES

General Objectives:

Students are expected to understand various types of process modes and software development life cycles.

Specific Objectives:

After studying this chapter, you should be able to:

- Define what is meant by the term "process" and how it applies to software development.
- Describe the activities, resources and products involved in the software development process.
- Describe several different models of the software development process and understand their drawbacks and when they are applicable.
- Describe the characteristics of several different tools and techniques for process modeling.

Summary:

This chapter presents an overview of different types of process and life- cycle models. It also describes several modeling techniques and tools. The chapter examines a variety of software development process models to demonstrate how organizing process activities can make development more effective. A process is a series of steps involving activities, constraints and resources that produce an intended output of some kind. A process usually involves a set of tools and techniques. Processes are important because they impose consistency and structure on a set of activities. The process structure guides actions by allowing software engineers to examine, understand, control, and improve the activities that comprise the software process. In software development, it is important to follow a software development process in order to understand, control and improve what happens as software products are built for customers.

Each stage of software development is itself a process (or a collection of processes) that can be described by a set of activities. A process can be described in a variety of ways, using text, pictures or a combination. In the software engineering literature, descriptions of process models are prescriptions (or the way software development should progress) or descriptions (the way software development is done in actuality). In theory, the two should be the same, but in practice, they are not. Building a process model and discussing its sub processes helps the team to understand the gap between the two. Every software development process model includes system requirements as input and a delivered product as output. Some of the more common models include the waterfall model, the V model, the spiral model and various prototyping models.

The waterfall model was one of the first models to be proposed. The waterfall model presents a very high-level view of what goes on during development and suggests the sequence of events a developer should expect to encounter. The V model is a variation of the waterfall model that demonstrates how testing activities are related to analysis and design. The spiral model combines development activities with risk management. No matter what process model is used, many activities are common to all.

There are many choices for modeling tools and techniques. There are two major categories of model types: static and dynamic. A static model depicts the process, showing that the inputs are transformed to outputs. A dynamic model can enact the process, so that the user can see how intermediate and final products are transformed over time. The Lai notation is an example of a static modeling notation. The systems dynamics approach has also been applied to dynamically model software development processes.

Exercises with answer key:

1. Describe the process you use to get to ready for class or work in the morning. Draw a diagram to capture the process.
2. Describe three software development life-cycle models. For each, name the main activities performed, and the inputs and outputs of each activity.

3. For each give an example of the kind of software development project where the life-cycle model would be well-suited, and an example of where the life-cycle model would be inappropriate; explain why.
4. What is the difference between static and dynamic modeling? Explain how each type of modeling is useful.
5. Use the five desirable properties of process modeling tools and techniques identified by Curtis, Kellner and Over (1992) and presented in Section 2.4 of the textbook to evaluate one process modeling tool or technique. You may use an example from the book and/or consult outside sources.
6. Explain the difference between prescriptive and descriptive process models. What is the purpose for each? When is it appropriate to use each?

Answer keys:

1. When answering this question, consider the definition of a process. Your answer should include the following:
   a. the activities involved
   b. the steps required to complete the tasks
   c. the inputs and outputs to each activity
   d. the constraints involved
2. Answers to this question will vary depending upon the life-cycle models chosen. Your answer should include the activities, the inputs and the outputs involved with each process model. In addition, you should provide examples and reasons why a particular process model would be appropriate as well as situations where a process model would be inappropriate. For example, if a development project is highly risky (development team is inexperienced with the domain, time pressures exist) a spiral life-cycle model would be appropriate because development activities are combined with risk management to minimize and control risk. However, if the development project is low risk, a spiral model may not be the best choice.

3. A static process model describes the elements of a process. It depicts where the inputs are transformed to outputs. A dynamic process model enacts the process and allows the user to view how the products are transformed over time. A static model is useful to identify the elements of the process. A dynamic model may be useful to simulate how changes to the process affect the outputs of the process over time. For more details on static and dynamic process models, re-read Section.
4. Your answer to this question will depend upon which process modeling technique or tool is chosen. Your answer should address the five desirable properties of process modeling tools and techniques outlined in .Section 2.4. Does the tool or technique you are evaluating possess the desirable characteristic? Which features of the tool or technique satisfy? The desirable property? Are there areas where the tool or technique lacks support for a desirable property?
5. Descriptive models attempt to describe what is actually happening in the process. Prescriptive process models attempt to describe what should be happening with the process. For more details on prescriptive and descriptive process models, you may find it helpful to re-read Section 2.2. In your answer to this question, use the reasons for modeling a process (in Section 2.2) to describe how and when prescriptive and descriptive models are useful. Can you think of cases where a prescriptive process model may be inappropriate? How does a descriptive model help in building a prescriptive model?
6. What are the major phases in a software development project?

Exercises without answer key:

1. Describe the following process models
   a. Waterfall model of software development.
   b. Rapid Application Development (RAD) approach and
   c. incremental development
2. Discuss the main differences between prototyping and incremental development.
3. How does the spiral model subsume prototyping, incremental development, and the waterfall model?

4. Discuss the key values of the agile movement.

CHAPTER THREE

REQUIRMENT ELICITATION AND ANALYSIS PRINCIPLES

General Objectives:

Students are required to understand how requirements can be gathered and basic principles used for requirement analysis.

Specific Objectives:

After studying this chapter, you should be able to:

- Explain why it is necessary to elicit requirements from software customers, and the role of requirements in the software life-cycle;
- Identify requirement Initiating the Process
- Identify the characteristics that make individual requirements good or bad;
- Describe the types of requirements that should be included in a requirements document;
- Describe the notations and methods that can be used for capturing requirements, and the types of situations in which each may be appropriate;
- Discuss the requirement Analysis Principles
- Define Software Prototyping its methods and approach
- Discuss requirement, representation and principles
- Explain how and why a requirement reviews should be done to ensure quality;
- Describe how to document requirements for use by the design and test teams.

Summary:

This chapter focuses on Requirements analysis, an important component of any model of the software development process. It is important to remember that the purpose of requirements is to specify the problem that the system is intended to solve, leaving the details of the solution to the system designers. Requirements analysis is the first technical step in the software process. It is at this point that a general statement of software scope is refined into a concrete specification that becomes the foundation for all software engineering activities that follow. Analysis must focus on the information, functional, and behavioral (non functional) domains of a problem. To better understand what is required, models are created, the problem is Software Requirements Specification Review partitioned, and representations that depict the essence of requirements and, later, implementation detail, are developed.

In many cases, it is not possible to completely specify a problem at an early stage. Prototyping offers an alternative approach that results in an executable model of the software from which requirements can be refined. To properly conduct prototyping special tools and techniques are required. The Software Requirements Specification is developed as a consequence of analysis.

Review is essential to ensure that the developer and the customer have the same perception of the system. Unfortunately, even with the best of methods, the problem is that the problem keeps changing.

Any requirements document should include both functional and non-functional requirements. The functional requirements explain what the system will do, and the non-functional ones constrain the behavior in terms of safety, reliability, budget, schedule and other issues. Since mistakes made during the requirements process can cause additional problems later in the software life-cycle, the complete set of requirements should be validated by checking for completeness, correctness, consistency, realism, and other attributes. Measures reflecting requirements quality are especially important since they may indicate useful activities; e.g. when indicators show that the requirements are not well - understood, prototyping of some requirements may be appropriate.

There are many different types of definition and specification techniques that can be used for capturing requirements. Some are static (e.g. data flow diagrams), while others are dynamic (i.e. they include information about timing and time-related dependencies).

Exercises with answer keys:

1. Most of a system's requirements specify that the system should do what it is intended to do. Is it also appropriate to specify that the system should not do what it is not intended to do? If your answer is no, explain why; if your answer is yes, give an example.
2. Describe the different consumers of software requirements i.e. the different users, or types of users, of a software requirements document). For each consumer, explain how he or she would use the requirements, and how the requirements should be documented to make them useful for this consumer.
3. One source of problems in the requirements phase can be the relationship between system developers and their customers. What are some negative stereotypes customers may hold about developers? What could you, as a developer on a project, do to minimize the impact of those negative stereotypes?

Answer keys:

1. An example of the latter type of requirement is a security requirement. In this case, it is necessary to specify exactly what the system should NOT allow a user or other system to do.
2. Requirements need to be used by:

a. The customer, who should check that the system described actually matches his or her needs. For use by the customer, requirements should be easy to understand, with a minimum of jargon, to facilitate clear communication with the customer.

b. Designers, who need to construct a design of the system described in the requirements. The requirements will need to be as complete, clear, and correct as possible so that designs developed from it are correct. Also, they will need to identify all of the constraints on the system so that the design can correctly incorporate them.

c. Testers, who need to develop test scripts. To support testers the requirements should be as precise as possible, so that the values that need to be tested and the expected system behavior are well -specified.

d. Documentation writers, who will write the user manuals based on the requirements. As for the customer, the requirements should clearly communicate the features of the system.

Exercises without answer key:

1. What is requirements elicitation?
2. Why is requirements traceability important?
3. List and discuss the major quality requirements for a requirements document.
4. For an office information system, identify different types of stakeholders. Can you think of ways in which the requirements of these stakeholders might conflict?
5. What are the major uses of a requirements specification? In what ways do these different uses affect the style and contents of a requirements document?

Chapter 4: ANALYSIS MODELING

Learning Objectives:

After studying this chapter, you should be able to:

- Identify elements of the Analysis Model
- Discuss on Data Modeling
- Explain Data Objects, Attributes, Relationships ,Cardinality Modality
- Define Functional Modeling and Information Flow Diagrams(DFD),Behavioral Modeling and functional decompositions
- Define the Control Specification and model
- Create a Data Flow Model
- Creating a Control Flow Model
- Create process model
- Draw Entity/Relationship Diagrams or ER diagram
- Identify The Process Specification and pr
- Explain Data Dictionary

Summary:

Structured analysis, a widely used method of requirements modeling, relies on datamodeling and flow modeling to create the basis for a comprehensive analysis model.Using entity-relationship diagrams, the software engineer creates a representation of all data objects that are important for the system. Data and control flow diagrams are used as a basis for representing the transformation of data and control. At the same time, these models are used to create a functional model of the software and to provide a mechanism for partitioning function. A behavioral model is created using the state transition diagram, and data content is developed with a data dictionary. Process and control specifications provide additional elaboration of detail. The original notation for structured analysis was developed for conventional data processing applications, but extensions have made the method applicable to real-time systems. Structured analysis is supported by an array of CASE tools that assist in the creation of each element of the model and also help to ensure consistency and correctness.

Exercise with answer key:

1. What is a data flow diagram? Why do systems analysts use data flow diagrams?
2. What is decomposition? What is balancing? How can you determine if DFDs are not balanced?
3. How can data flow diagrams be used as analysis tools?
4. Explain what the term DFD completeness means and provide an example.
5. What is the purpose of logic modeling? What techniques are used to model decision logic and what techniques are used to model temporal logic?
6. What is Structured English? How can Structured English be used to represent sequence, conditional statements, and repetition in an information systems process?
7. What is the formula that is used to calculate the number of rules a decision table must cover?

Answer keys:

1. Data flow diagram is a picture of the movement of data between external entities and the processes and data stores within a system. Systems analysts use data flow diagrams to help them model the processes internal to an information system as well as how data from the system's environment enter the system, are used by the system, and are returned to the environment. DFDs help analysts understand how the organization handles information and what its information needs are or might be. Analysts also use DFDs to study alternative information handling procedures during the process of designing new information services.
2. Decomposition is the iterative process by which a system description is broken down into finer and finer detail, creating a set of diagrams in which one process on a given diagram is explained in greater detail on a lower–level diagram. Balancing is the conservation of inputs and outputs to a data flow diagram process when that process is decomposed to a lower level. You can determine if a set of DFDs are balanced or not by observing whether or not a process that appears in a level-n diagram has the same inputs and outputs when decomposed for a lower-level diagram.
3. DFDs can be used as analysis tools to help determine the completeness of a system model and a model's internal consistency, as a way to determine when system events occur through analyzing timeliness, and, through iterative use, to develop and check models. Analysts can study DFDs to find excessive information handling, thus identifying areas for possible efficiencies.
4. DFD completeness is the extent to which all necessary components of a data flow diagram have been included and fully described. A data store that does not have any data flows coming into or out of it is a completeness violation.
5. The purpose of logic modeling is to show the rules that govern the behavior of processes represented in data flow diagrams. Structured English and decision tables model decision logic. State diagrams model temporal logic.
6. Structured English is a modified form of the English language used to specify the logic of information system processes. Sequence is represented by listing statements at the same level of indentation. Conditional statements are represented by BEGIN IF/END IF

statements and by case statements. Repetition is represented by DO ... UNTIL and DO…While statements.

7. To determine the number of rules a decision table must cover, simply determine the number of values each condition may have and multiply the number of values for each condition by the number of values for every other condition.

Exercise without answer keys:

1. Draw DFDs for each of these scenarios:
   a. A customer goes into a bookshop and asks for this book. The member of staff looks for the book in the online stock catalogue and reports that the book is sold out.
   b. Every month, the Medical Centre receives a list of current drugs available from the drug companies. These lists are collated into a catalogue of drugs which is copied and given to each doctor.
2. Draw an entity model to model the following car rental business scenario:
   - Cars are always rented from one location and are brought back to the same location.
   - Customers may pay by cash or credit card
   - Customers who call the agency may request a particular car make, or model, etc if available
   - A bill is presented to the customers prior to releasing the rental car
   - A further bill may be presented to the customers once the rented car has been returned to cover any damage or excessive mileage.
3. Produce a decision table to model the logic in this scenario:
   - A postal delivery company delivers parcels by air or rail transport. The price of delivery by air depends upon the weight of the parcel. There is a basic charge of 5 Euros per kilo up to 50 kilos. Excess weight over 50 kilos is charged at 3 Euros per kilo. Delivery by rail is charged at 3 Euros per kilo up to 50 kilos and then 2 Euros per kilo. There is a special

service guaranteeing same day delivery which carries an additional flat rate charge of 20 Euros. Any deliveries overseas are charged at double the normal rate.

4. Produce a structured English specification for this scenario:
   - A travel agent has account customers and individual customers. Account customers who have spent over 25,000 Euros in the past year get a discount of 25%. Otherwise, they get 10% discount. Individual customers who have booked holidays previously get 5% discount. New customers get no discount. Account customers who have spent over 10,000 Euros in any previous year will receive offers of free tickets on selected routes.

CHAPTER 5 DESIGN CONCEPTS AND PRINCIPLES

General Objectives:

Students are expected to grasp basic software design concepts and priniciles

Learning Objectives:

After studying this chapter, you should be able to:

- Define Software Design and Design Process
- Discuss Design Principles and concepts like Abstraction ,Refinement Modularity, Software Architecture ,Control Hierarchy etc
- Explain how come up with Effective Modular Design through Functional Independence ,Cohesion and Coupling
- Indentify the Design Model
- Discuss Design Documentation
- Explain What Is Architecture? Why Is Architecture Important Software Architecture
- Discuss about Data Design
- Identify Alternative Architectural Designs
- Map Requirements into a Software Architecture
- Transform Mapping

- Identify user interface design and its activities
- Discuss component level design such as: Structured Programming, Graphical Design Notation, Tabular Design Notation, Program Design Language and Comparison of Design Notations.

Summary:

Design is the technical essential part of software engineering. During design, progressive refinements of data structure, architecture, interfaces, and procedural detail of software components are developed, reviewed, and documented. Design results in representations of software that can be assessed for quality. A number of fundamental software design principles and concepts have been proposed over the past four decades. Design principles guide the software engineer as the design process proceeds. Design concepts provide basic criteria for design quality. Modularity (in both program and data) and the concept of abstraction enable the designer to simplify and reuse software components. Refinement provides a mechanism for representing successive layers of functional detail. Program and data structure contribute to an overall view of software architecture, while procedure provides the detail necessary for algorithm implementation. Information hiding and functional independence provide heuristics for achieving effective modularity.

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them. Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture. Architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative system structures. Data design translates the data objects defined in the analysis model into data structures that reside within the software. The attributes that describe the object, the relationships between data objects and their use within the program all influence the choice of data structures. At a higher level of abstraction, data design may lead to the definition of architecture for a database or a data warehouse. A number of different architectural styles and patterns are available to the software engineer. Each style describes a system category that encompasses a set of components that perform a function required by a system, a set of connectors that enable communication, coordination and cooperation among components, constraints that define how components can be integrated to form the system and semantic models that enable a designer to understand the overall properties of a system. Once one or more architectural styles have been proposed for a system, an architecture trade-off analysis method may be used to assess the efficacy of each the proposed architecture. This is accomplished by determining the sensitivity of selected quality attributes (also called design dimensions) to various realization mechanisms that reflect properties of the architecture. The architectural design method presented in this chapter uses data flow characteristics described in the analysis model to derive a commonly used architectural style. A data flow diagram is mapped into program structure using one of two mapping approaches—transform mapping or transaction mapping. Transform mapping is applied to an information flow that exhibits distinct boundaries between incoming and outgoing data. The DFD is mapped into a structure that allocates control to input, processing, and output along three separately factored module hierarchies. Transaction mapping is applied when a single information item causes flow to branch along one of many paths. The DFD is mapped into a structure that allocates control to a substructure that acquires and evaluates a transaction. Another substructure controls all potential processing actions based on a transaction. Once architecture has been derived, it is elaborated and then analyzed against quality criteria. Architectural design encompasses the initial set of design activities that lead to a complete design model of the software. In the chapters that follow, the design focus shifts to interfaces and components.

The user interface is the window into the software. Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user's memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted. User interface design begins with the identification of user, task, and environmental requirements. Task analysis is a design activity that defines user tasks and actions using either an elaborative or object-oriented approach.

Component-level design depicts the software at a level of abstraction that is very close to code. At the component level, the software engineer must represent data structures, interfaces, and algorithms in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of a number of design notations that represent component-level detail in either graphical, tabular, or text-based formats. Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

Exercise with answer keys:

1. Suppose a team is developing a web-based ticket distribution system. Which of the following decisions was most likely made during system design?

   a. The ticket distributor will include a user interface subsystem.

   b. The ticket distributor will follow web-accessibility standards.

   c. The ticket distributor will provide the traveler with on-line help.

   d. The ticket distributor requirements have been met and satisfy customer needs.

2. In designing an object-oriented class hierarchy to represent binary expressions, which structural pattern and behavioral pattern could be used most effectively?

   a. Composite and Interpreter
   b. Decorator and Command
   c. Flyweight and Strategy
   d. Proxy and Visitor

3. To support the reuse of component systems, which of the following mechanisms is often used to insulate client code from the specific details of the components by exporting only the portion of a component's interfaces that are needed by the client system?

   a. Abstract Class
   b. Façade
   c. Parameterized Classes
   d. Publisher-Subscriber

4. One of the following is characteristics a structural model to represent an architectural design?

    a. A structural model allows identifying repeatable architectural design frameworks that are encountered in similar applications

    b. A structural model addresses behavioral aspects which indicate how the system changes as a function of external events

    c. A structural model represents the architecture as an organized collection of program modules and components

    d. A structural model focuses on the business or technical processes that a system must accommodate

Answer keys:

1.A 2.A 3.B 4.C

Exercise without answer keys:

1. Give a definition of the term 'software architecture'. Explain the different elements in this definition.
2. Define the following connector types: data flow, message passing, shared data.
3. What is the main purpose of software architecture?
4. In what sense does the abstract-data-type architectural style constrain the designer?
5. What is a software architecture assessment?
6. What is the difference between procedural abstraction and data abstraction?
7. Explain the notions cohesion and coupling.
8. In what sense are the various notions of coupling technology-dependent?
9. What is the essence of information hiding?
10. Discuss the relative merits and drawbacks of deep and narrow versus wide and shallow inheritance trees.
11. What is the main difference between problem-oriented and product-oriented design methods?

12. What are the differences between object-oriented design and the simple application of the information hiding principle?
13. What are the properties of a design pattern?
14. Discuss the pros and cons of:
    - functional decomposition,
    - data flow design,
    - design based on data structures, and
    - object-oriented design

For the design of each of:

- a compiler,
- a patient monitoring system, and
- a stock control system.

CHAPTER 6 SOFTWARE TESTING

General Objectives:

Students will comprehend the basic approach and strategies of structured approach software testing.

Learning Objectives:

After studying this chapter, you should be able to:

- Explain Software Testing Fundamentals like:
    - Testing Objectives
    - Testing Principles
    - Testability
- Define Test Case Designs as: White-Box Testing ,Basis Path Testing ,Flow Graph Notation ,Cyclomatic Complexity ,Deriving Test Cases and Graph Matrices
- Discuss Control Structure Testing like: Condition Testing, Data Flow Testing and Loop Testing
- Define Black-Box Testing

- Define a Strategic Approach to Software Testing :Verification and Validation
- Discuss type of testing: Unit Testing , Integration Testing, Validation Testing, and System Testing

Summary:

The primary objective for test case design is to derive a set of tests that have the highest likelihood for uncovering errors in the software. To accomplish this objective, two different categories of test case design techniques are used: white-box testing and black-box testing. White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity. Black-box testing, on the other hand, broadens our focus and might be called "testing in the large." Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough tests coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems will small numbers of input parameters.

Experienced software developers often say, "Testing never ends, it just gets transferred from you [the software engineer] to your customer. Every time your customer uses the program, a test is being conducted." By applying test case design, the software engineer can achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

Unit and integration tests concentrate on functional verification of a component and incorporation of components into a program structure. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system. Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened. Unlike testing (a systematic, planned activity), debugging must be viewed as an art. Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error.

Exercises without answer keys:

1. Describe the following categories of test technique: coverage-based testing, fault-based testing, and error-based testing.
2. What is the difference between black-box testing and white-box testing?
3. Define the following terms: error, fault, and failure.
4. What is test-driven development?
5. Define the following categories of control-flow coverage: All-Paths coverage, All-Edges coverage, All-Statements coverage.
6. What is the difference between a system test and an acceptance test?
7. Contrast top-down and bottom-up integration testing.

CHAPTER 7 OBJECT-ORIENTED CONCEPTS AND PRINCIPLES

General Objectives:

Students will understand concepts and principles used in Object Oriented software engineering approach.

Specific Objectives:

After studying this chapter, you should be able to:

- Explain the Object-Oriented Paradigm

- Identify Object-Oriented Concepts like: Attributes, Operations, Methods, Services Messages, Encapsulation, Inheritance, and Polymorphism
- Identify the Elements of an Object Model
- Identify Classes and Objects
- Specify Attributes
- Define Operations

Summary:

Object-oriented technologies reflect a natural view of the world. Objects are categorized into classes and class hierarchies. Each class contains a set of attributes that describe it and a set of operations that define its behavior. Objects model almost any identifiable aspect of the problem domain. External entities, things, occurrences, roles, organizational units, places, and structures can all be represented as objects. As important, objects (and the classes from which they are derived) encapsulate both data and process. Processing operations are part of the object and are initiated by passing the object a message. A class definition, once defined, forms the basis for reusability at the modeling, design, and implementation levels. New objects can be instantiated from a class.

Three important concepts differentiate the OO approach from conventional software engineering. Encapsulation packages data and the operations that manipulate the data into a single named object. Inheritance enables the attributes and operations of a class to be inherited by all subclasses and the objects that are instantiated from them. Polymorphism enables a number of different operations to have the same name, reducing the number of lines of code required to implement a system and facilitating changes when they are made.

Object-oriented products and systems are engineered using an evolutionary model, sometimes called a recursive/parallel model. OO software evolves iteratively and must be managed with the recognition that the final product will be developed over a series of increments.

Exercises with answer keys:

1. In what ways do the Object-Oriented characteristics of encapsulation and information hiding support reuse? What kind of criteria would you use in deciding whether to reuse a class in a new system?
2. Why is it useful to have separate phases for system and program design?
3. You are designing a system to help run a bookstore. Revenue for the store comes from two distinct services: customers can purchase books, or bring their books in for rebinding. You are considering making a separate class for each service, both of which would be subclasses of a general "sale item" class. What are the likely benefits of such an approach? Are there any possible arguments against using inheritance in this case? Be sure to specify what factors could influence your decision.
4. "Report" class inherits from "Document" class. Document class defines a print() method. Assume polymorphic behavior. Which of the following scenarios would not result in the client calling Document.print() on a Report object called TermReport?
   a. Report.print() does not refer to Document.print()
   b. Report class does not define print() method.
   c. Report.print() calls Document.print() and does nothing else.
   d. Client casts the TermReport object to Document class before calling print.

Answer keys:

1. Some goals could be: it should be easy for the developer to understand what functionality is available to be reused, related functionalities should be somehow reusable together, it should be easy to understand how to reuse the functionality, and the reusable components should be of high quality. Then, address whether it would be harder or easier to achieve these characteristics in an OO environment, and why.
2. System design gives developers a chance to solidify the broad outlines of the proposed system before having to decide on more specific details of the implementation

3. To answer this question, think about how someone reading an OO program knows where the specific definition of a method is located. If there is no inheritance involved, a method is defined within the class to which it belongs. But if that class is part of an inheritance hierarchy, the method need not be defined in the class. Is there any indication of which parent class contains the definition of a method? Can a method be redefined multiple times within a particular hierarchy? How do those factors contribute to the ease and accuracy with which a method definition can be found?
4. A

Exercises without answer keys:

1. Using your own words and a few examples, define the terms class, encapsulation, inheritance, and polymorphism.
2. You have been assigned the job of engineering new word-processing software. A class named document is identified. Define the attributes and operations that are relevant for document.

CHAPTER 8 OBJECT-ORIENTED ANALYSIS

General Objectives:

Students will use object oriented analysis methods to solve real life problems

Specific Objectives:

After studying this chapter, you should be able to:

- Explain Object-Oriented Analysis
- Discuss Conventional vs object oriented Approaches
- Identify the OOA Landscape
- A Unified Approach to OOA
- Discuss Reuse and Domain Analysis

- Discuss on Generic Components of the OO Analysis Model: The OOA Process, Use-Cases ,Class-Responsibility-Collaborator Modeling, Defining Structures and Hierarchies, Defining Subsystems, the Object-Relationship Model, the Object-Behavior Model etc

Summary:

Object-oriented analysis methods enable a software engineer to model a problem by representing both static and dynamic characteristics of classes and their relationships as the primary modeling components. Like earlier OO analysis methods, the Unified Modeling Language (UML) builds an analysis model that has the following characteristics :(1) representation of classes and class hierarchies, (2) creation of object relationship models, and (3) derivation of object-behavior models. Analysis for object-oriented systems occurs at many different levels of abstraction. At the business or enterprise level, the techniques associated with OOA can be coupled with a business process engineering approach. This technique is often called domain analysis. At an application level, the object model focuses on specific customer requirements as those requirements affect the application to be built. The OOA process begins with the definition of use-cases—scenarios that describe how the OO system is to be used. The class-responsibility-collaborator modeling technique is then applied to document classes and their attributes and operations. It also provides an initial view of the collaborations that occur among objects. The next step in the OOA process is classification of objects and the creation of a class hierarchy. Subsystems (packages) can be used to encapsulate related objects. The object relationship model provides an indication of how classes are connected to one another, and the object-behavior model indicates the behavior of individual objects and the overall behavior of the OO system.

Exercises with Answer key:

1. One of the following statements describes use cases.
    a. Use case diagrams are the primary tool to document requirements
    b. Use cases provide the basis of communication between sponsors and developers in planning phase
    c. Use cases description provides a good source to identify domain concepts
    d. A fully-dressed use case should include both "whats" and "hows" so that they are ready for "realization"

e. A use case is an interaction between a user and a system.

B, C &E

2. One of the following is true about UML stereotypes?
    a. A stereotype is used for extending the UML language.
    b. A stereotyped class must be abstract.
    c. The stereotype {frozen} indicates that the UML element cannot be changed.
    d. UML Profiles can be stereotyped for backward compatibility.
3. One of the following is usage UML interfaces?
    a. to provide concrete classes with the stereotype <<interface>>
    b. to program in Java and C++, but not in C#
    c. to define executable logic that can be reused in several classes
    d. to specify required services for types of objects
4. One of the following statements is true about use-case driven development.
    a. Requirements are primarily captured in use cases.
    b. Use cases are the essential part of iterative planning by choosing some use case scenarios.
    c. Use cases are key input to project sizing
    d. User manuals are normally organized based on use cases
    e. Business use cases should not be developed within the project scope
    f. All of the above

Answer: 1. B, C &E 2.A 3. D 4. F

Exercises without Answer key:

1. Conduct an abbreviated domain analysis for one of the following areas:

    a. A university student record-keeping system.

    b. An e-commerce application (e.g., clothes, books, electronic gear).

    c. Customer service for a bank.

    d. A video game developer.

    e. An application area suggested by your instructor.

2. In your own words describe the difference between static and dynamic views of an OO system.

3. Develop a set of use-cases for any one of the following applications:
   a. Software for a general-purpose personal digital assistant.
   b. Software for a video game of your choosing.
   c. Software that sits inside a climate control system for a car.
   d. Software for a navigation system for a car.
   e. A system (product) suggested by your instructor.
4. In your own words, describe how collaborators for a class are determined.
5. What strategy would you propose for defining subsystems for a collection of classes?
6. What role does cardinality play in the development of an object-relationship model?
7. What is the difference between an active and a passive state for an object?

CHAPTER 9 OBJECT-ORIENTED DESIGN

General Objectives

Students will grasp basic concepts of design in Object oriented approach

Learning Objectives:

After studying this chapter, you should be able to:

- Discuss Design for Object-Oriented Systems
  - Conventional vs. OO Approaches
  - Design Issues
  - The OOD Landscape
  - A Unified Approach to OOD
- Identify System Design Process
- Recognize Object Design Process
- Explain Design Patterns
- Be acquainted with Object-Oriented Programming

Summary

Object-oriented design translates the OOA model of the real world into an implementation-specific model that can be realized in software. The OOD process can be described as a pyramid composed of four layers. The foundation layer focuses on the design of subsystems that implement major system functions. The class layer specifies the overall object architecture and the hierarchy of classes required to implement a system. The message layer indicates how collaboration between objects will be realized, and the responsibilities layer identifies the attributes and operations that characterize each class. Like OOA, there are many different OOD methods. UML is an attempt to provide a single approach to OOD that is applicable in all application domains. UML and other methods approach the design process through two levels of abstraction, design of subsystems (architecture) and design of individual objects.

During system design, the architecture of the object-oriented system is developed. In addition to developing subsystems, their interactions, and their placement in architectural layers, system design considers the user interaction component, a task management component, and a data management component. These subsystem components provide a design infrastructure that enables the application to operate effectively. The object design process focuses on the description of data structures that implement class attributes, algorithms that implement operations and messages that enable collaborations and object relationships.

Design patterns allow the designer to create the system architecture by integrating reusable components. Object-oriented programming extends the design model into the executable domain. An OO programming language is used to translate the classes, attributes, operations, and messages into a form that can be executed by a machine.

Exercise with answer key:

1. One of the following is true about a Sequence Diagram? [2 answers]
   a. It describes the behavior in many Use Cases.
   b. It describes the behavior in a single Use Case.
   c. It describes the behavior of a single object.

   d. It describes the behavior of several objects.

2. One of the following belong to UML behavioral diagrams:
   a. Cass and object diagram
   b. Use case diagram and sequence diagram
   c. Collaboration diagram and state-char diagram
   d. A and BB and C
3. If you need to show the physical relationship between software components and the hardware in the delivered system, which diagram can you use?
   a. Component diagram
   b. Deployment diagram
   c. Class diagram
   d. Network diagram
4. One of the following statements is true about Responsibilities?
   a. Responsibilities are related to the obligations of an object in terms of its behavior. Basically, responsibilities are of two types: Doing and Knowing
   b. Responsibilities are assigned to objects during object design while creating interaction diagrams.
   c. A responsibility is the same thing as a method that is designed to fulfill responsibility

Answer: 1. B & D 2. B 3. C 4.A & B

Exercises without answer key:

1. How do OOD and structured design differ? What aspects of these two design methods are the same?
2. You are responsible for the development of an electronic mail (e-mail) system to be implemented on a PC network. The e-mail system will enable users to create letters to be mailed to another user, general distribution, or a specific address list. Letters can be read,

copied, stored, and the like. The e-mail system will use existing word-processing capability to create letters. Using this description as a starting point, derive a set of requirements and apply OOD techniques to create a top-level design for the e-mail system.

3. How do design patterns impact the quality of a design?

CHAPTER 10 OBJECT-ORIENTED TESTING

General Objective:

Students will understand the basic idea behind software testing

Specific Objectives:

- Recognize Testing OOA and OOD Models: Correctness of OOA and OOD Models ,Consistency of OOA and OOD Models
- Distinguish Object-Oriented Testing Strategies: Unit Testing in the OO Context, Integration Testing in the OO Context, Validation Testing in an OO Context
- Identify Testing Methods Applicable at the Class Level such as :Random Testing for OO Classes , Partition Testing at the Class Level
- Use Interclass Test Case Design and Multiple Class Testing for OO software

Summary:

The overall objective of object-oriented testing—to find the maximum number of errors with a minimum amount of effort—is identical to the objective of conventional software testing. But the strategy and tactics for OO testing differ significantly. The view of testing broadens to include the review of both the analysis and design model. In addition, the focus of testing moves away from the procedural component (the module) and toward the class. Because the OO analysis and design models and the resulting source code are semantically coupled, testing (in the form of formal technical reviews) begins during these engineering activities. For this reason, the review of CRC, object-relationship, and object-behavior models can be viewed as first stage testing.

Once OOP has been accomplished, unit testing is applied for each class. The design of tests for a class uses a variety of methods: fault-based testing, random testing, and partition testing. Each of these methods exercises the operations encapsulated by the class. Test sequences are designed to ensure that relevant operations are exercised. The state of the class, represented by the values of its attributes, is examined to determine if errors exist. Integration testing can be accomplished using a thread-based or use-based strategy. Thread-based testing integrates the set of classes that collaborate to respond to one input or event. Use-based testing constructs the system in layers, beginning with those classes that do not use server classes. Integration test case design methods can also use random and partition tests. In addition, scenario-based testing and tests derived from behavioral models can be used to test a class and its collaborators. A test sequence tracks the flow of operations across class collaborations. OO system validation testing is black-box oriented and can be accomplished by applying the same black-box methods discussed for conventional software. However, scenario-based testing dominates the validation of OO systems, making the use-case a primary driver for validation testing.

Exercises with answer key:

1. Equivalence Partitioning is a testing technique used in the following
   a. White box testing
   b. Black box testing
   c. Stress testing
   d. Usability testing
2. An incorrect step, process, or data definition in a computer program is a
   a. Failure
   b. Mistake
   c. Fault
   d. Consequence
3. The Dynamic process used to check whether we have developed the product according to the customer requirements is
4. The Dynamic process used to check whether we have developed the product according to the customer requirements is

a. Validation

b. Verification

c. Quality Assurance

d. Quality Control

5. A Non-Functional Software testing done to check if the user interface is easy to use and understand

a. Usability Testing

b. Security Testing

c. Unit testing

d. Block Box Testing

Answer: 1. B 2. C 3. A 4. A

Exercises without answer key:

1. In your own words, describe why the class is the smallest reasonable unit for testing within an OO system.
2. Why should "testing" begin with the OOA and OOD activities?
3. What is the difference between thread-based and use-based strategies for integration testing? How does cluster testing fit in?
4. Why do we have to retest subclasses that are instantiated from an existing class, if the existing class has already been thoroughly tested? Can we use the test cases designed for the existing class?

CHAPTER 11 SOFTWARE MAINTENANCE

Genera Objectives:

Students will gain basic notion of software maintenance and apply of real life software problems**.**

Learning Objectives:

- Identify categories of maintenance tasks and data on their distribution
- Distinguish major causes of maintenance problems
- Identify different ways in which maintenance activities can be organized
- Differentiate between development and maintenance and the consequences thereof.

Summary:

Software maintenance encompasses all modifications to a software product after delivery. The following breakdown of maintenance activities is usually made: **Corrective maintenance** concerns the correction of faults. **Adaptive maintenance** deals with adapting software to changes in the environment. **Perfective maintenance** mainly deals with accommodating new or changed user requirements. **Preventive maintenance** concerns activities aimed at increasing a system's maintainability. **'Real' maintenance**, the correction of faults, consumes approximately 25% of maintenance effort. By far the larger part of software maintenance concerns the evolution of software. This evolution is inescapable. Software models part of reality. Reality changes, and so does the software that models it. Major causes of maintenance problems are the existence of a vast amount of unstructured code, insufficient knowledge about the system or application domain on the part of maintenance programmers, insufficient documentation, and the bad image of the software maintenance department. Some of these problems are accidental and can be remedied by proper actions. Through a better organization and management of software maintenance, substantial quality and productivity improvements can be realized. Improved maintenance should start with improved development. A particularly relevant issue for software maintenance is that of reverse engineering, the process of reconstructing a lost blueprint. Before changes can be realized, the maintainer has to gain an understanding of the system. Since the majority of operational code is unstructured and undocumented, this is a major problem. The fundamental problem is that maintenance will remain a big issue. Because of the changes made to software, its structure degrades. Specific attention to preventive maintenance activities aimed at improving system structure are needed from time to time to fight system entropy. Software maintenance used to be a rather neglected topic in the software engineering literature.

Exercises with answer key:

1. The failure rate of a software product decreases first and then increases with time approximating a bath tub like curve as for other tangible products. One of the following may be the reason(s) behind this phenomenon?
    a. Software does not wear and tear.
    b. Software products change with time as customer requirements change.
    c. Maintenance is the costliest phase of the software development life cycle.
    d. A change to a piece of code may implicitly affect the functions of the rest of the code.
    e. Software maintenance is given little consideration by companies developing software.

    Answer: B

Exercises without answer key:

1. Why does corrective maintenance have more service-like aspects than product-like aspects?
2. Discuss the iterative-enhancement and quick-fix models of software maintenance.
3. Discuss advantages of software configuration control support during software maintenance.
4. Discuss the possible structure and role of an acceptance test by the maintenance organization prior to the release of a system.
5. Discuss the impact of component reuse on maintainability.
6. Discuss the possible contribution of object-oriented software development to software maintenance.

CHAPTER 12 SOFTWARE PRJECT MANAGEMENT

General Objectives:

Students will grasp basic issues on how to manage software projects.

Specific Objectives:

- Identify Project Planning Objectives
- Identify the Relationship Between People and Effort
- Use Scheduling methods: Timeline Charts, Tracking the Schedule
- Explain Software Project Estimation
- Distinguish Reactive versus Proactive Risk Strategies : Software Risks and Risk
- Recognize software Quality Concepts: Quality, Quality Control, Quality Assurance and Cost of Quality.
- Distinguish Software Configuration Management :Baselines, Software Configuration Items ,The SCM Process
- Identification of Objects in the Software Configuration: Version Control, Change Control, Configuration Audit, SCM Standards.

Summary:

The software project planner must estimate three things before a project begins: how long it will take, how much effort will be required, and how many people will be involved. In addition, the planner must predict the resources (hardware and software) that will be required and the risk involved. The statement of scope helps the planner to develop estimates using one or more techniques that fall into two broad categories: decomposition and empirical modeling. Decomposition techniques require a delineation of major software functions, followed by estimates of either (1) the number of LOC, (2) selected values within the information domain, (3) the number of person-months required to implement each function, or (4) the number of person-months required for each software engineering activity. Empirical techniques use empirically derived expressions for effort and time to predict these project quantities. Automated tools can be used to implement a specific empirical model. Accurate project estimates generally use at least two of the three techniques just noted. Software project estimation can never be an exact science, but a combination of good historical data and systematic techniques can improve estimation accuracy.

Risk analysis can absorb a significant amount of project planning effort. Identification, projection, assessment, management, and monitoring all take time. But the effort is worth it.

Scheduling is the culmination of a planning activity that is a primary component of software project management. When combined with estimation methods and risk analysis, scheduling establishes a road map for the project manager.

Scheduling begins with process decomposition. The characteristics of the project are used to adapt an appropriate task set for the work to be done. A task network depicts each engineering task, its dependency on other tasks, and its projected duration.

The task network is used to compute the critical path, a timeline chart and a variety of project information. Using the schedule as a guide, the project manager can track and control each step in the software process.

SQA encompasses procedures for the effective application of methods and tools, formal technical reviews, testing strategies and techniques, poka-yoke devices, procedures for change control, procedures for assuring compliance to standards, and measurement and reporting mechanisms. SQA is complicated by the complex nature of software quality—an attribute of computer programs that is defined as "conformance to explicitly and implicitly specified requirements." But when considered more generally, software quality encompasses many different product and process factors and related metrics. Software reviews are one of the most important SQA activities. Statistical SQA helps to improve the quality of the product and the software process itself. Software reliability models extend measurements, enabling collected defect data to be extrapolated into projected failure rates and reliability predictions.

SCM identifies controls, audits, and reports modifications that invariably occur while software is being developed and after it has been released to a customer. All information produced as part of software engineering becomes part of a software configuration. The configuration is organized in a manner that enables orderly control of change. The software configuration is composed of a set of interrelated objects, also called software configuration items that are produced as a result of some software engineering activity. In addition to documents, programs, and data, the development environment that is used to create software can also be placed under configuration control. Once a configuration object has been developed and reviewed, it becomes a baseline. Changes to a baseline object result in the creation of a new version of that object. The evolution of a program can be tracked by examining the revision history of all configuration objects. Basic and composite objects form an object pool from which variants and versions are created. Version control is the set of procedures and tools for managing the use of these objects. Change control is a procedural activity that ensures quality and consistency as changes are made to a configuration object. The change control process begins with a change request, leads to a decision to make or reject the request for change, and culminates with a controlled update of the SCI that is to be changed. The configuration audit is an SQA activity that helps to ensure that quality is maintained as changes are made. Status reporting provides information about each change to those with a need to know.

Exercises with key:

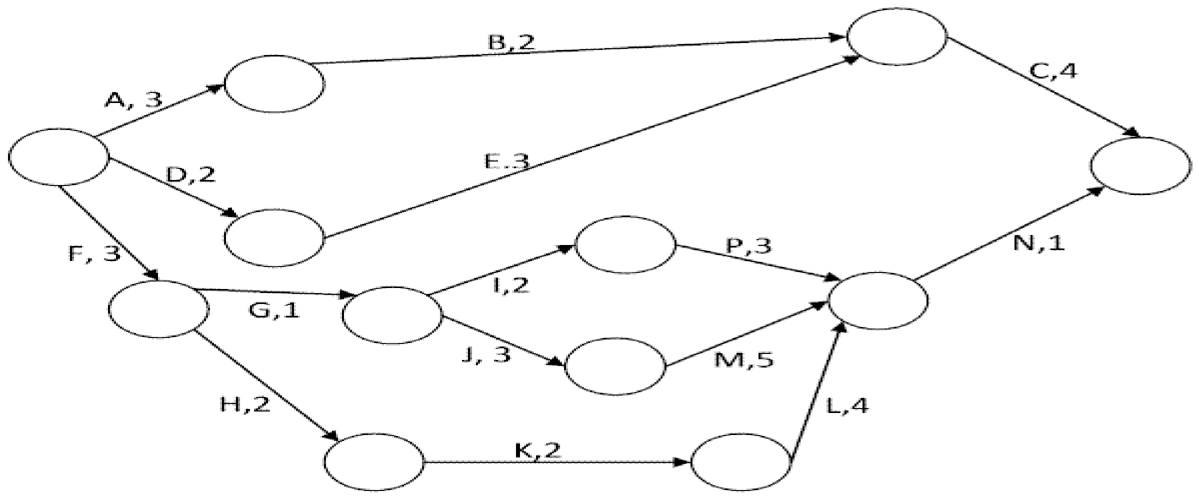
1. As a project manager, a project is considered a success if all of the below are true EXCEPT?
   a. system is delivered on time
   b. resulting software system is acceptable to the customer
   c. system is delivered by as few stakeholders as possible
   d. system development process had minimal impact on ongoing business operations
2. project management consists of all of the following EXCEPT
   a. scoping
   b. planning
   c. staffing
   d. controlling
   e. none
3. Effective software project management focuses on four P's which are
   a. people, performance, payoff, product
   b. people, product, performance, process

c. people, product, process, project
d. people, process, payoff, product

4. The first step in project planning is to:
   a. Determine the budget.
   b. select a team organizational model
   c. determine the project constraints
   d. Establish the objectives and scope.
5. Which of the following is not generally considered a player in the software process?
   a. Customers
   b. end-users
   c. project managers
   d. sales people
6. How does a software project manager need to act to minimize the risk of software failure?
   a. double the project team size
   b. request a large budget
   c. start on the right foot
   d. track progress
   e. both c and d

Exercises without key:

1. Discuss the reasons for baselines in your own words.
2. What is the difference between an SCM audit and a formal technical review?
3. Can a program be correct and still not be reliable? Explain.
4. What are the major constituents of a project plan?



For question number 5,6,7,8 and 9 refer to the above figure.

5. What possible paths from the start of the project until the end are there in this PERT chart?

- *A,B,C*
- *D,E,C*
- *F,G,I,P,N*
- *F,G,J,M,N*
- *F,H,K,L,N*

6. Calculate the lengths of the paths.

- *A,B,C = 9*
- *D,E,C = 9*
- *F,G,I,P,N = 10*
- *F,G,J,M,N = 13*
- *F,H,K,L,N = 12*

7. Which path is longest? *F, G,J, M, N.* What is this path called? *The critical path.*
8. How much slack does task J have? *None. (No task on the critical path can have any slack.)*

9. If task P ran 2 days over time, what effect would this have on the finishing date of the project? *None. It would use 2 of its 3 slack days but still would not exceed the length of the critical path.*
10. How many predecessors does task N have? *Three (P, M,L)*
11. Is task E dependent on any tasks? *Yes, D. So task E can't start until D has finished.*