Unit V : Architectural Design and Documentation **07 Hours**

Architecture in the Life Cycle: Architecture in Agile Projects, Architecture and Requirements, Designing an Architecture. Documenting Software Architecture: Notations, Choosing and Combining views, Building the documentation Package, Documenting Behavior, Documenting Architecture in an Agile Development Project.

#Exemplar/Case Studies : Air Traffic Control.

Unit VI : Design Patterns **07 Hours**

Design Patterns : Introduction, Different approaches to select Design Patterns. Creational patterns: Singleton, Factory, Structural pattern: Adapter, Proxy. Behavioral Patterns: Iterator, Observer Pattern with applications.

#Exemplar/Case Studies : Flight Simulation

❑❑❑

UNIT I

Chapter 1 : Concepts of Software Modeling 1-1 to 1-44

Syllabus : Software Modeling : Introduction to Software Modeling, Advantages of modeling, Principles of modeling. **Evolution of Software Modeling and Design Methods :** Object oriented analysis and design methods, Concurrent, Distributed Design Methods and Real-Time Design Methods, Model Driven Architecture (MDA), 4+1 Architecture, Introduction to UML, UML building Blocks, COMET Use Case–Based Software Life Cycle. **Requirement Study :** Requirement Analysis, SRS design, Requirements Modeling. **Use Case :** Actor and Use case identification, Use case relationship (Include, Extend, Use case Generalization, Actor Generalization), Use case template.

1.1 Introduction to Software Modeling 1-1

1.1.1 Advantages of Modeling 1-2

1.1.2 Why do We Model ? 1-2

1.1.3 Benefits of Visual Modeling 1-3

1.1.4 Principles of Modeling 1-3

1.2 Evolution of Software Modeling and Design Methods 1-3

1.2.1 Object Oriented Analysis and Design Methods 1-4

1.2.2 Concurrent, Distributed Design Methods and Real-Time Design Methods 1-5

1.2.3 Model Driven Architecture (MDA) 1-6

1.2.4 4+1 Architecture 1-7

1.2.4(A) Why is it Called the 4 + 1 Instead of just 5 ? 1-8

1.2.4(B) 4+1 View of UML Diagrams 1-9

1.2.4(C) Why 4+1 Architecture 1-9

1.2.5 Introduction to UML 1-9

1.2.5(A) Introduction to Unified Modeling Language 1-10

1.2.5(B) UML Supports Requirements Modeling 1-11

1.2.6 UML Building Blocks 1-11

1.2.6(A) Things in UML 1-11

1.2.6(B) Relationship in UML 1-14

1.2.6(C) Diagrams in UML 1-15

1.2.7 COMET Use Case - Based Software Life Cycle 1-15

1.3 Requirement Study 1-17

1.3.1 Types of Requirements 1-17

1.3.1(A) Functional Requirements 1-18

1.3.1(B) Non-Functional Requirements 1-18

1.3.2 Requirement Analysis 1-19

1.3.3 SRS Design 1-20

1.3.4 Organization of SRS Document 1-21

1.3.5 SRS for UMS (University Management System) 1-25

1.3.5(A) General Description 1-25

1.3.5(B) Overall Description 1-25

1.3.6 Specific Requirements 1-26

1.3.6(A) External Interface Required 1-26

1.3.6(B) Performance Requirements 1-26

1.3.6(C) Design Constraints 1-26

1.3.6(D) Attributes 1-26

1.3.6(E) Other Requirements 1-26

1.3.7 Requirements Modeling 1-27

1.3.8 Managing Changing Requirement 1-29

1.4 Use Case Modeling 1-30

1.4.1 Use Case Diagram 1-30

1.4.2 Actor and Use Case Identification 1-31

1.4.3 Use Case Relationship 1-33

1.4.3(A) Include 1-33

1.4.3(B) Extend 1-34

1.4.3(C) Use Case Generalization 1-35

1.4.3(D) Actor Generalization 1-36

1.4.4 Use Case Specification 1-37

1.4.5 Use Case Template 1-39

1.4.6 Solved Examples 1-41

UNIT II

Chapter 2 : Static Modeling 2-1 to 2-48

Syllabus : Study of classes (analysis level and design level classes). **Methods for identification of classes** : RUP (Rational Unified Process), CRC (Class, Responsibilities and Collaboration), Use of Noun Verb analysis (for identifying entity classes, controller classes and boundary classes).

Class Diagram : Relationship between classes, Generalization/Specialization Hierarchy, Composition and Aggregation Hierarchies, Associations Classes, Constraints.

Object diagram, Package diagram, Component diagram, Composite Structure diagram, Deployment Diagram.

2.1 Introduction to Static Modeling 2-1

2.2 Study of Classes 2-1

2.3 Method of Identifying Classes 2-2

2.3.1 Rational Unified Process Model 2-2

2.3.2 CRC (Class Responsibilities Collaboration) 2-4

2.3.3 Use of Noun Verb Analysis 2-4

2.3.4 Common Class Pattern 2-6

2.3.5 Identifying Boundary, Entity and Control Classes 2-6

2.3.5(A) Boundary Class 2-6

2.3.5 (B) Control Classes 2-8

2.3.5 (C) Entity Classes 2-8

2.4 Class Diagram 2-9

2.4.1 Class Structure 2-9

2.4.2 Identifying Attributes 2-11

2.4.3 Identifying Operation 2-11

2.4.4 Relationship between Classes 2-13

2.4.4(A) Association 2-13

2.4.4(B) Generalization/Specialization Hierarchy 2-15

2.4.4(C) Aggregate /Composite Hierarchy 2-16

2.4.4(D) Composite Relationships 2-17

2.4.5 Association Classes 2-18

2.4.6 Constraints 2-18

2.4.7 Solved Examples 2-20

2.5 Object Diagram 2-29

2.6 Package Diagram 2-31

2.7 Component Diagram 2-32

2.7.1 Types of Components / Elements of Component Diagram 2-33

2.7.2 Interface 2-34

2.8 Composite Structure Diagrams 2-35

2.8.1 Part 2-35

2.8.2 Port 2-35

2.8.3 Interfaces 2-36

2.8.4 Delegate 2-36

2.9 Deployment Diagram 2-37

2.9.1 Elements of a Deployment Diagram 2-37

2.9.2 When to use Deployment Diagram 2-37

UNIT III

Chapter 3 : Dynamic Modeling 3-1 to 3-39

Syllabus : **Activity diagram** : Different Types of nodes, Control flow, Activity Partition, Exception handler, Interruptible activity region, Input and output parameters, Pins.

Interaction diagram : Sequence diagram, Interaction Overview diagram, State machine diagram, Advanced State Machine diagram, Communication diagram, Timing diagram.

3.1 Activity Diagram 3-1

3.1.1 Different Types of Nodes 3-2

3.1.1(A) Initial and Final Activity 3-2

3.1.1(B) Decision and Merge Points 3-3

3.1.1(C) Forking and Joining 3-3

3.1.2 Activity Edge / Control Flow 3-4

3.1.3 Activity Partitions 3-4

3.1.3(A) Swimlanes 3-4

3.1.4 Exception Handler 3-5
3.1.5 Interruptible Activity Region 3-6
3.1.6 Input and Output Parameters And Pin 3-6
3.1.6(A) Input and Output Pins 3-6
3.1.7 Creating Activity Diagram 3-7
3.1.8 Solved Examples 3-8
3.2 Interaction Diagram 3-14
3.3 Sequence Diagram 3-15
3.3.1 Elements of Sequence Diagram 3-16
3.3.1(A) Objects and Roles 3-16
3.3.1(B) Link 3-16
3.3.1(C) Object Life Line 3-17
3.3.1(D) Message or Stimulus 3-17
3.3.1(E) Focus of control 3-18
3.3.1(F) End of a Lifeline 3-18
3.3.1(G) Conditional Message 3-18
3.3.2 Creating Sequence Diagram 3-18
3.3.3 Modeling Interactions 3-19
3.4 Interaction Overview Diagram 3-23
3.5 State Diagrams 3-23
3.5.1 Element of State Diagram 3-24
3.5.1(A) State Machine 3-24
3.5.1(B) Initial and Final State 3-26
3.5.1(C) Triggers and Ports 3-26
3.5.1(D) Transitions 3-26
3.6 Advanced State Machine Diagram 3-27
3.6.1(A) Composite States 3-27
3.6.1(B) Orthogonal State 3-28
3.6.1(C) Submachine States 3-29
3.7 Communication Diagram 3-35
3.8 Timing Diagram 3-36

UNIT IV

Chapter 4 : Software Architecture and Quality Attributes 4-1 to 4-13

Syllabus : Introduction to Software Architecture, Importance of Software Architecture, Architectural Structure and Views. **Architectural Pattern** : common module, Common component-and-connector, Common allocation.
Quality Attributes : Architecture and Requirements, Quality Attributes and Considerations

4.1 Introduction 4-1
4.2 Overview of Software Architecture 4-2
4.3 Architectural Structure and Views 4-3
4.3.1 Architectural Structure 4-3
4.3.2 Architectural View 4-3
4.4 Architectural Pattern 4-4
4.4.1 Common Module Patterns 4-4
4.4.1(A) Layered Pattern 4-4
4.4.2 Component and Connector Pattern 4-5
4.4.2(A) The Broker Architectural Pattern 4-5
4.4.2(B) Model-View-Controller Architectural Pattern 4-5
4.4.2(C) Client Server Pattern 4-6
4.4.2(D) Publish Subscribe Pattern 4-7
4.4.2(E) Service Oriented Architecture Pattern 4-8
4.4.3 Allocation Pattern 4-9
4.4.3(A) Map Reduce Pattern 4-9
4.4.3(B) Multi-tier Pattern 4-9
4.5 Quality Attributes 4-10
4.6 Architecture and Requirements 4-11
4.7 Quality Attributes and Considerations 4-12

UNIT V

Chapter 5 : Architectural Design and Documentation 5-1 to 5-8

Syllabus : Architecture in the Life Cycle : Architecture in Agile Projects, Architecture and Requirements, Designing an Architecture.

Documenting Software Architecture : Notations, Choosing and Combining views, Building the documentation Package, Documenting Behavior, Documenting Architecture in an Agile Development Project.

5.1 Architecture in the Life Cycle **5-1**

5.1.1 Architecture in Agile Projects 5-2

5.1.2 Architecture and Requirements 5-2

5.1.3 Designing an Architecture 5-3

5.1.3(A) Attribute Driven Design (ADD) 5-3

5.2 Documenting Software Architecture **5-5**

5.2.1 Notations 5-5

5.2.2 Choosing and Combining View 5-5

5.2.3 Building the Documentation Package 5-6

5.2.4 Documenting Behavior 5-6

5.2.5 Documenting Architecture in an Agile Development Project 5-7

5.2.5(A) Architecture Decision Records 5-7

5.2.5(B) The C4 Model 5-7

UNIT VI

Chapter 6 : Design Patterns 6-1 to 6-22

Syllabus : Design Patterns : Introduction, Different approaches to select Design Patterns.

Creational patterns : Singleton, Factory,

Structural pattern : Adapter, Proxy.

Behavioral Patterns : Iterator, Observer Pattern with applications.

6.1 Design Patterns: Introduction **6-1**

6.1.1 What is Design Pattern ? 6-1

6.1.2 Why to Use Design Pattern ? 6-1

6.1.3 Classification of Design Patterns 6-2

6.1.3(A) Creational Design Patterns 6-2

6.1.3(B) Structural Design Patterns 6-3

6.1.3(B) Behavioral Design Patterns 6-4

6.1.4 Design Pattern Template/ Documenting Design Pattern 6-5

6.2 Different Approaches to Select Design Patterns **6-6**

6.3 Creational Patterns **6-7**

6.3.1 Singleton Pattern 6-7

6.3.2 Factory Pattern 6-8

6.4 Structural Pattern **6-11**

6.4.1 Adapter Pattern 6-11

6.4.2 Proxy Pattern 6-15

6.5 Behavioral Patterns **6-17**

6.5.1 Iterator Pattern 6-18

6.5.2 Observer Pattern with Applications 6-19

➢ **Case Studies** ..... **C-1 to C-65**

1 Concepts of Software Modeling

Syllabus

Software Modeling : Introduction to Software Modeling, Advantages of modeling, Principles of modeling. **Evolution of Software Modeling and Design Methods :** Object oriented analysis and design methods, Concurrent, Distributed Design Methods and Real-Time Design Methods, Model Driven Architecture (MDA), 4+1 Architecture, Introduction to UML, UML building Blocks, COMET Use Case–Based Software Life Cycle. **Requirement Study :** Requirement Analysis, SRS design, Requirements Modeling. **Use Case :** Actor and Use case identification, Use case relationship (Include, Extend, Use case Generalization, Actor Generalization), Use case template.

1.1 Introduction to Software Modeling

- Modeling is proven and well accepted engineering technique.
- Modeling enables people to understand what has gone on before their project, and also allows them to get a better understanding of what needs to happen next.
- As an example of the "gone on before", take the all to typical situation where a developer team inherits requirements; sometimes of good quality, sometimes of not so good quality.
- Developers are asked to design or develop based on these requirements, and frequently don't even know where to start.
- Modeling and diagramming is an excellent way to comb through these artifacts and apply some critical thinking around them so as to make some initial directional setting decisions.
- As an example of the "what needs to happen next", the same example also applies.
- The modeling done by developer type resources can also be used to apply critical thinking around what the data structures will look like, what the interface screens will look like, what services will be required, ceteria.
- This modeling exercise is also the best way for these developer resources to actually understand any legacy documentation.
- Simply asking someone to read a pound of documentation will rarely result in comprehension of that documentation.
- Giving someone some documentation and asking them to perform a discrete activity will result in exponentially more understanding.
- Examples include reading requirements to do initial analysis/domain modeling, initial vendor analysis, usability/interface modeling, actual development, et cetera.
- Simple models that abstract and structure a large amount of requirements (or any other system information for that matter) are worth their weight in gold.
- The maxim "a picture is worth a thousand words" is no truer than in the system development space.

- Being able to illustrate a vast and incredibly complex system with 10 to 20 pictures (or 1 or 2 depending on the level of abstraction) goes a long way toward ensuring that a lot of distributed teams have the same understanding of what the system needs to do.

1.1.1 Advantages of Modeling

- A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it.
- You can take a piece of paper and a paper clip, and, in a few minutes, have a paper airplane that entertains your kids. If it isn't built just right, you can always start over and build another airplane.
- Would it be smart for you to build a fighter jet in the same way? That is, start with some steel, nuts, bolts, and wiring and go right to work. Of course not. You're building an airplane that costs millions of dollars, and the cost of failure is high. You're also being part of a much larger team, needing blueprints and models to effectively communicate with one another.

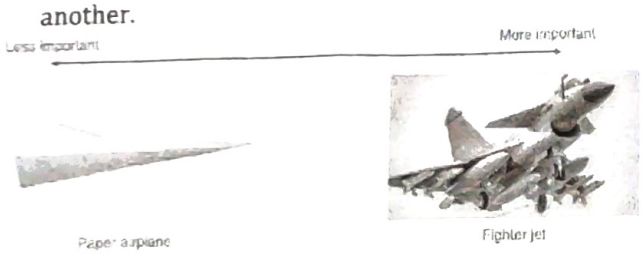

Fig. 1.1.1

- Many software development organizations begin wanting to build complex software systems, but approach the problem as though they were building a paper airplane.
- With the increasing demand to build more complex software in shorter time, development teams often retreat to the only thing they know how to do well - pound out lines of code.
- Developers start working longer hours and frequently produce code with a requirements document as their only source of input.
- However, there eventually comes a time when the application collapses due to the lack of a well thought-out architecture.
- Consequently, many of these software projects result in failure.
- Therefore we need simpler representations for complex systems and modeling is a mean for dealing with complexity.
- Thus, Modeling is a common thread to successful projects.

1.1.2 Why do We Model ?

- We build models so that we can better understand the system we are developing.
- The model provides an understanding of the business, of the project and of the enterprise perspective. This understanding improves communications among the data modelers, other IT representatives and the business stakeholders, and it provides a strong foundation for the physical structures that must be built
- The model would provide a way to understand the business, a basis for the physical structure needed to support the business. The model also helps us understand the project.
- As we create the model, we speak with the business representatives to comprehend the business in general, and also to understand what needs to be provided by the project under way.
- This ensures that we will include the required entities and attributes in our model.
- The model is useful for verifying that business needs can be met and that the desired navigation capabilities are being provided.
- This understanding goes two ways - the model not only reflects the modeler's interpretation of the project scope and business needs, but it also provides a means to communicate that understanding so that its accuracy can be confirmed by the business community.

- I have found that business users can comprehend a logical dimensional model and hence are better positioned to confirm its accuracy before the physical structure is designed and built.

1.1.3 Benefits of Visual Modeling

- Other benefits of using visual modeling include accelerated time to launch, reduced development cost, and increased user adoption of applications.
- Most of these benefits come from the development of a clear understanding of the true business needs and a common understanding of how the applications will operate. These results in accelerated development, fewer missed requirements, reduced rework, and quicker deployment.
- Additionally, visual modeling emphasizes collaboration among multiple communities, helping to catalyse innovation in the user-functionality of applications and the rethinking of business functions.

1.1.4 Principles of Modeling

There are four main principles of modeling :

1. *Models help us to visualize a system as it is or as we want it to be.*
2. *Models permit us to specify the structure or behavior of a system.*
3. *Models give us a template that guides us in constructing a system.*
4. *Models document the decisions we have made.*

1. Models help you to **visualize** a system, as you want it to be. A model helps the software team communicate the vision for the system being developed. It is difficult for a software team to have a unified vision of a system that is described only in specification and requirement documents. Models bring about understanding of the system.
2. Models permit you to **specify** the structure of behavior of a system. A model allows how to document system behavior and structure before coding the system.
3. Models give a template that guides you in **constructing** a system. A model is an invaluable tool during construction. It serves as a road map for a developer. Have you experienced a situation where a developer coded incorrect behavior because he or she was confused over the wording in a requirements document? Modeling helps alleviate that situation.
4. Models **document** the decisions you've made. Models are valuable tools in the long term because they give "hard" information on design decisions. You don't need to rely on someone's memory.

1.2 Evolution of Software Modeling and Design Methods

- The Structured methodologies is based on the concept of functional decomposition where the analyst breaks down the system into the basic processes that make it up and then breaks these down into smaller ones and so on until the analyst understands all the essential components of the system being investigated.
- The OOAD methodology uses an object-oriented perspective. OOAD focuses on objects. OOAD treat system is a collection of interacting objects and Objects interact with each other. Objects send and respond to messages. Object-oriented analysis is the process of developing an object-oriented model of the problem domain where the initial objects represent the entities and methods related to the problem that needs to be resolved. Object-oriented design is the process of developing an object-oriented model of the system necessary to meet the specified requirements. So in this methodology we think in terms of things (objects) rather than functions.
- Jackson System Development (JSD) is a linear software development methodology developed by Michael A. Jackson and John Cameron in the 1980s. It comprises the whole software life cycle directly or providing a framework for more specialized techniques.

1.2.1 Object Oriented Analysis and Design Methods

- Object Oriented Methodology is a new system development approach, encouraging and facilitating re-use of software components. Numbers of methodology are based on modeling the business problem and implementing the application in an object oriented fashion.
- The major differences between different methodologies lie primarily in the documentation of information, and modeling notations and language. The three major methodologies and their modeling notations developed by Rumbaugh et al., Booch and Jacobson which are the origins of the Unified Modeling Language.
- Each method has its strengths. Rambaugh method is well-suited for describing the object model or the static structure of the system. The Jacobson et al. method is good for producing user – driven analysis models. The Booch method produces detailed object–oriented design models.

A. Object Oriented Design –Booch

- The Booch method helps to design systems using the object paradigm. It covers the analysis- and design phases of an object-oriented system. The method defines different models to describe a system and it supports the iterative and incremental development of systems.
- The Booch software engineering methodology provides an object-oriented development in the analysis and design phases. The analysis phase is split into steps. The first step is to establish the requirements from the customer perspective. This analysis step generates a high-level description of the system's function and structure. The second step is a domain analysis. The domain analysis is accomplished by defining object classes; their attributes, inheritance, and methods. State diagrams for the objects are then established. The analysis phase is completed with a validation step. The analysis phase iterates between the customer's requirements step, the domain analysis step, and the validation step until consistency is reached.
- Once the analysis phase is completed, the Booch software engineering methodology develops the architecture in the design phase. The design phase is iterative. A logic design is mapped to a physical design where details of execution threads, processes, performance, location, data types, data structures, visibility, and distribution are established. A prototype is created and tested. The process iterates between the logical design, physical design, prototypes, and testing.

B. Object Modeling Techniques – Rumbaugh

- The object-modeling technique (OMT) is an object modeling language for software modeling and designing. It is developed in 1991 by James Rumbaugh as a method to develop object-oriented systems and to support object-oriented programming. A fundamental assumption of OMT is that object-oriented thinking represents a more natural and intuitive way for people to reason about reality.
- The purposes of modeling according to Rumbaugh are:
  - testing physical entities before building them (simulation),
  - communication with customers,
  - visualization (alternative presentation of information), and
  - reduction of complexity.

C. Object – Oriented Analysis - Cood Yourdon

- Coad-Yourdon methodology has its primary strength in system analysis. Coad and Yourdon propose a model, or architecture, for an object-oriented system that is partitioned ``horizontally'' into layers and ``vertically'' into components, as shown in the following figure.
- Coad-Yourdon methodology is based on a technique called "SOSAS", which stands for the five steps that help make up the analysis part of their methodology.

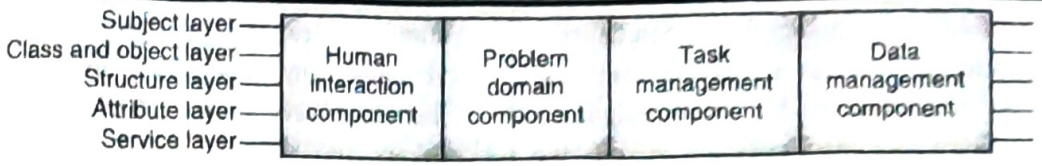

Fig. 1.2.1: Coad-Yourdon methodology

- The first step in system analysis is called "Subjects", which are basically data flow diagrams for objects. The second step is called "Objects", where they identify the object classes and the class hierarchies. The third step is called "Structures", where they decompose structures into two types, classification structures and composition structures. the structure layer would display the system's generalization-specialization and whole-part structure. The next step in analysis is called "Attributes" which show attributes and instance connections,, and the final step is called "Services", where all of the behaviors or methods for each class are identified. Coad and Yourdon recommend the use of a CASE tool that would allow you to select which layers you wish to see, when viewing the model of the system that's being developed.

D. Object – Oriented Software Engineering – Ivar Jacobson

- This methodologies covers the entire life cycle and stress traceability between the different phases both forward & backward. It consists of :
  - OOBE (Object-oriented Business Engineering): Object oriented business engineering (OOBE) is object modeling at enterprise level. Use cases are the important tool for modeling. OOBE is a framework for architecture, business engineering, business process management and object-oriented development.
  - OOSE (Object-oriented Software Engineering) : OOSE is developed by Ivar Jacobson in 1992. OOSE is the first object-oriented design methodology that employs use cases in software design. OOSE is also called Objectory. The system development method based on OOSE is a process for the industrialized development of the s/w. The development process called use–case driven development stresses that use cases are involved in several phases of the development including analysis, design, validation and testing. The use - case scenario begins with a user of the system initiating a sequence of interrelated events. Objectory (Object Factory for Software Development) has been developed and applied to numerous application areas and embodied in the CASE tool systems.

1.2.2 Concurrent, Distributed Design Methods and Real-Time Design Methods

- When we design real-time software target system is analyzed and then we structure sequential executive modules into concurrent tasks. As a result of the analysis, control flow and dataflow diagram based on the RTSA notation is produced. The criteria structuring concurrent tasks are introduced as Concurrent Design Approach for Real-Time System(CODARTS) by Gomaa. CODARTS emphases both information-hiding module structuring and concurrent task structuring. CODARTS method is developed around 1993 by Gomaa
- Real-Time Object-Oriented Modeling (ROOM) is a domain specific language. ROOM describes a software system along three dimensions: structure, behavior and inheritance. ROOM a development method specifically designed for dealing with large real-time systems.
- It supports automatic code generation to ensure reliability and increase productivity. ROOM introduces the notions of service access point (SAP) for the client side and service provision point (SPP) for the server side. From the point of view of an actor implementation the SAPs and SPPs work like ports.

- Octopus is a Design Methodology for Motion Capture Wearables. Octopus provides a tool that can help define design requirements for multidisciplinary teams, generating a common framework and offering a new method of communication between them . For UML-based real-time software development, Douglass has provided a comprehensive description of how UML can be applied to real-time systems
- The Collaborative Object Modeling and Design Method, or COMET, uses the UML notation to describe the design. COMET is based on the design concepts of information hiding, classes, inheritance, and concurrent tasks. It uses a design strategy of concurrent object design, which addresses the structuring of a software system into active and passive objects and defines the interfaces between them. It provides structuring criteria to help structure the system into objects during analysis, and additional criteria to determine the subsystems and concurrent tasks during design. COMET an iterative use case driven and object-oriented software development method that addresses the requirements, analysis, and design modeling phases of the software development life cycle.

1.2.3 Model Driven Architecture (MDA)

- Model Driven Architecture (MDA) is an approach to software design, development and implementation spearheaded by the OMG. MDA supports model-driven engineering of software systems. MDA provides a set of guidelines for structuring specifications expressed as models.
- The reason why models were selected as the central object in this design principle is because it helps designers to reason out the system design by allowing them to ignore extra details and giving them more focus on the relevant issues. Models are also used all throughout the engineering and design field to understand complex and real-world systems.
- MDA separates business and application logic from underlying platform technology. Platform-independent models of an application or integrated system's business functionality and behavior, built using UML and the other associated OMG modeling standards, can be realized through the MDA on virtually any platform, open or proprietary, including Web Services, .NET, CORBA R, J2EE, and others.

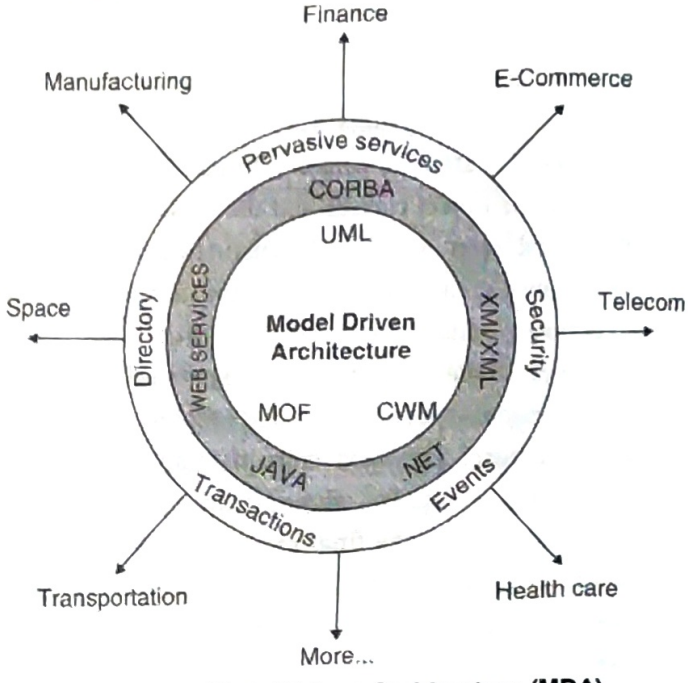

Fig. 1.2.2 : Model Driven Architecture (MDA)

- These platform-independent models document the business functionality and behavior of an application separate from the technology-specific code that implements it, insulating the core of the application from technology and its relentless churn cycle while enabling interoperability both within and across platform boundaries.
- **Model driven architecture** is a model-driven engineering (MDE) approach that uses three types of models:
  - **CIM** - computation independent model that describes how a system should be behave in terms of a language that is appropriate for a user (e.g. an educational designer, a business analyst, etc.)
  - **PIM** - Platform Independent Model that describes a CIM in computational terms, i.e. a language that makes sense to a given class of computer persons but is independent from specific computer technology
  - **PSM** - platform specific model that links PIM specification with specific information about a specific platform and it should be executable.

Benefits from using MDA are

- **Portability**, increasing application re-use and reducing the cost and complexity of application development and management, now and into the future.
- **Cross-platform Interoperability**, using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions.
- **Platform Independence**, greatly reducing the time, cost and complexity associated with re-targeting applications for different platforms-including those yet to be introduced.
- **Productivity**, by allowing developers, designers and system administrators to use languages and concepts they are comfortable with, while allowing seamless communication and integration across the teams.

1.2.4 4+1 Architecture

University Questions

Q. What is 4+1 architecture view model? Explain with suitable diagram. **SPPU : Dec. 16, Dec. 19, 5 Marks**

Q. What is 4+1 view architecture? Why it is called as 4+1 view architecture. **SPPU : April 19, 5 Marks**

- The "4 + 1 Architectural Views" were proposed in 1995 to solve increasing challenges communicating about software architectures. The purpose of separating the architecture into multiple concurrent views was to isolate and illustrate different aspects of the design with information specifically oriented towards different stakeholders. There are many different stakeholders related to a software system, which all have different perspectives.
- Project/Project Managers, Developers, System Engineers, End Users all view a system in completely different ways. They view the system based on their own context. In order to describe a system, it would be useful to define all the different viewpoints and how the overall use cases of the system.
- Each view should :
  - Define the system in terms of Components, Connectors and Containers.
  - Use appropriate styles, forms and patterns for each view.
  - Consider applicable constraints in each view.
  - Consider the relationship between the views.

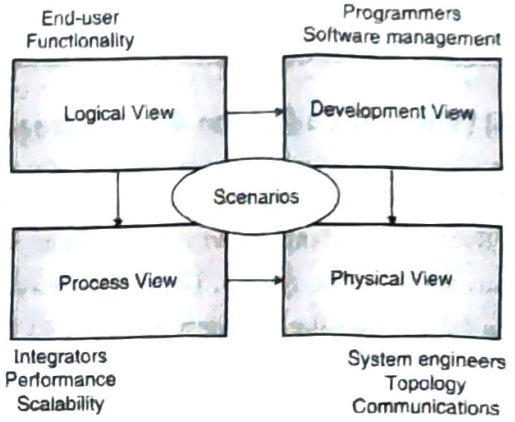

Fig. 1.2.3 : 4 + 1 Architectural Views

Let's have a look at each of the different views.

1. **The logical view :** The logical view is concerned with the functionality that the system provides to end-users. This contains information about the various parts of the system. In UML the logical view is modelled using Class, Object, State machine and Interaction diagrams (e.g Sequence diagrams). Its relevance is really to developers.
2. **The process view :** This describes the concurrent processes within the system. It encompasses some non-functional requirements such as performance and availability. The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the run time behavior of the system. The process view addresses concurrency, distribution, integrator, performance, and scalability, etc. In UML, Activity diagrams - which can be used to model concurrent behaviour - are used to model the process view.
3. **The development view :** The development view focuses on software modules and subsystems. In UML, Package and Component diagrams are used to model the development view.

4. **The physical view :** The physical view describes the physical deployment of the system .The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer as well as the physical connections between these components. The physical view concerns some non-functional requirements such as scalability and availability. In UML, Deployment diagrams are used to model the physical view.

5. **The use case view :** This view describes the functionality of the system from the perspective from outside world. It contains diagrams describing what the system is supposed to do from a black box perspective. This view typically contains Use Case diagrams. All other views use this view to guide them.

1.2.4(A) Why is it Called the 4 + 1 Instead of just 5 ?

- Well this is because of the special significance the use case view has. When all other views are finished, it's effectively redundant. However, all other views would not be possible without it. It details the high levels requirements of the system. The other views detail how those requirements are realised.
- Table 1.2.1 shows the 4+1 view in detail.

Table 1.2.1 : 4+1 view

	Logical	Process	Development	Physical	Scenario
Description	Shows the component (Object) of system as well as their interaction.	Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system.	Gives building block views of system and describe static organization of the system modules.	Shows the installation, configuration and deployment of software application.	Shows the design is complete by performing validation and illustration.
Viewer / Stake holder	End-User, Analysts and Designer.	Integrators and developers.	Programmer and software project managers.	System engineer, operators, system administrators and system installers.	All the views of their views and evaluators.
Consider	Functional requirements	Non-Functional Requirements	Software Module organization (Software management reuse, constraint of tools)	Non-functional requirement regarding to underlying hardware	System Consistency and validity
UML – Diagram	Class, State, Object, sequence, Communication Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	

1.2.4(B) 4+1 View of UML Diagrams

- Considering that the UML diagrams can be used in different stages in the life cycle of a system, let us take a look at the "4+1 view" of UML diagrams. The 4+1 view offers a different perspective to classify and apply UML diagrams. The 4+1 view is essentially how a system can be viewed from a software life cycle perspective. Each of these views represents how a system can be modelled. This will enable us to understand where exactly the UML diagrams fit in and their applicability.
- These different views are :

1. **Structural View/ Design View :** The *design view* of a system is the structural view of the system. This gives an idea of what a given system is made up of. Class diagrams and object diagrams form the design view of the system.
2. **Behavioural View/ Process View :** The dynamic behaviour of a system can be seen using the process view. The different diagrams such as the state diagram, activity diagram, sequence diagram, and collaboration diagram are used in this view.
3. **Implementation View/Component View :** Next, you have the *component view* that shows the grouped modules of a given system modelled using the component diagram.
4. **Environmental View/Deployment View :** The deployment diagram of UML is used to identify the deployment modules for a given system. This is the *deployment view* of the system.
5. **User's View/Use case View :** Finally, we have the *use case view*. Use case diagrams of UML are used to view a system from this perspective as a set of discrete activities or transactions.

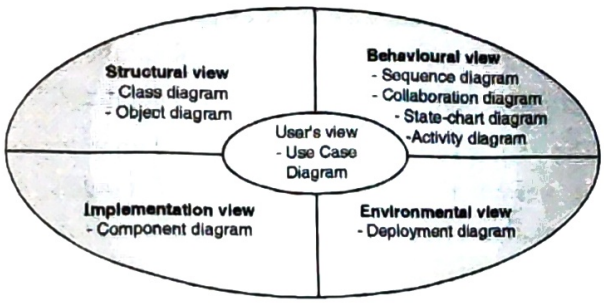

Fig. 1.2.4: Diagrams and views in UML

1.2.4(C) Why 4+1 Architecture

- The 4 + 1 approach isn't just about satisfying different stakeholders. It makes modelling easier to do because it makes it easier to organise. A typical project will contain numerous diagrams of the various types. For example, a project may contain a few hundred sequence diagrams and several class diagrams. Grouping diagrams of similar types and purpose means there is an emphasis in separating concerns. Sure isn't it just the same with Java? Grouping Java classes of similar purpose and related responsibilities into packages means organisation is better.
- Similarly, grouping different components into different jar files means organisation is better. Modelling tools will usually support the 4 + 1 approach and this means projects will have templates for how to split the various types of diagrams. In a company when projects follow industry standard templates again it means things are better organised.
- The 4 + 1 approach also provides a way for architects to be able to prioritise modelling concerns. It is rare that a project will have enough time to model every single diagram possible for architecture. Architects can prioritise different views. For example, for a business domain intensive project it would make sense to prioritise the logical view. In a project with high concurrency and complex timing it would make sense to ensure the process view gets ample time. Similarly, the 4 + 1 approach makes it possible for stakeholders to get the parts of the model that are relevant to them.

1.2.5 Introduction to UML

- During the 1990s many different methodologies, along with their own set of notations, were introduced to the market. Three of the most popular methods were OMT (Rumbaugh), Booch, and OOSE (Jacobson). Each method had its own value and emphasis. OMT was strong in analysis and weaker in the design area.

- Booch 1991 was strong in design and weaker in analysis. Jacobson was strong in behaviour analysis and weaker in the other areas.
- The methods were beginning to converge but they still had their own unique notations. The use of different notations brought confusion to the market since one symbol meant different things to different people. For example, a filled circle was a multiplicity indicator in OMT and an aggregation symbol in Booch. You will hear the term "method wars" being used to describe this period of time is a class, a cloud or a rectangle? Which one is better?
- The end of the method wars as far as notation is concerned comes with the adoption of the Unified Modelling Language (UML). "UML is a language used to specify, visualize and document the artifacts of an object-oriented system under development.
- It represents the unification of the Booch, OMT and Objectory notations, as well as the best ideas from a number of other methodologists. By unifying the notations used by these object-oriented methods, the Unified Modelling Language provides the basis for a de-facto standard in the domain of object-oriented analysis and design founded on a wide base of user experience."

1.2.5(A) Introduction to Unified Modeling Language

- As its name implies, UML is language for creating models. UML has its syntax and semantics. It provides a set of notations to create models of system that would be useful in documenting the design and analysis result.
- The UML is a language for **visualizing, specifying, constructing and documenting** the artifacts of a software-intensive system.

The UML is a Language

- A language provides a vocabulary and the rules for combining words in that vocabulary for the purpose of communication.
- A modelling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modelling language such as the UML is thus a standard language for software blueprints. The vocabulary and rules of a language such as the UML tell you how to create and read well formed models.

A. The UML is a Language for Visualizing

- Something's are best modelled textually; others are best modelled graphically. Indeed, in all interesting systems, there are structures that transcend what can be represented in a programming language. The UML is such a graphical language.
- The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously.

B. The UML is a Language for Specifying

- Specifying means building models that are precise, unambiguous and complete.
- In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

C. The UML is a Language for Constructing

- The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database or the persistent store of an object-oriented database.
- The UML is sufficiently expressive and unambiguous to permit the direct execution of models, the simulation of systems, and the instrumentation of running systems.

D. The UML is a Language for Documenting

- A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include requirements, architecture, design, source code, project plans, tests, prototypes and releases.
- The UML addresses the documentation of a system's architecture and all of its details. The UML also provides a language for expressing requirements and for tests. Finally, the UML provides a language for modelling the activities of project planning and release management.

1.2.5(B) UML Supports Requirements Modeling

University Question

Q. State and explain how UML supports requirements modeling ? **SPPU : May 18, 5 Marks**

- UML supports requirements modelling by means of use cases. Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used. Use case modelling is a useful tool for requirements elicitation. It provides a graphical representation of the software system's requirements.
- Use case diagram allows for the specification of high level user goals that the system must carry out. These goals are not necessarily tasks or actions, but can be more general required functionality of the system. You apply use cases to capture the intended behaviour of the system you are developing, without having to specify how that behaviour is implemented. For example, you can specify how an ATM system should behave by stating in use cases how users interact with the system; you don't need to know anything about the inside of the ATM at all.

  Use case diagram shows the overall functionality of the system. Users, project managers, analysts, developers, quality assurance engineers, and anyone else interested in the system as a whole can view these diagrams and understand what the system is supposed to accomplish.
- We use stereotypes to enhance UML diagram clarity, to store additional element properties, and to develop automated RE process support. Stereotypes are one of the core extension mechanisms of the Unified Modelling Language (UML). The benefits founds in their use in a software development organization support the claims that stereotypes play a significant role in model comprehension, reduce errors and increase productivity during the software development cycle.

1.2.6 UML Building Blocks

There are three kinds of building blocks :

1. Things
2. Relationships
3. Diagrams

1.2.6(A) Things in UML

- Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.
- These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.
- There are four kinds of things in the UML :

A. Structural Things
B. Behavioural Things
C. Grouping Things
D. Annotational Things

A. Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

These seven elements are the basic structural things that you may include in a UML model and these are :

1. Classes	2. Interfaces
3. Collaborations	4. Use cases
5. Active classes	6. Components
7. Nodes	

1. Class

- A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. A class also has properties that reflect unique features of a class. The properties of a class are called *attributes*.
- The UML representation of a class is a rectangle containing three compartments stacked vertically. The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations.

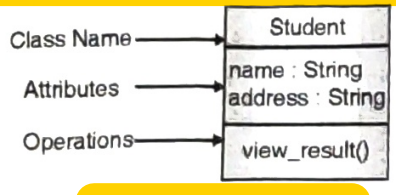

Fig. 1.2.5 : Class structure

2. Interface

- An interface is a collection of operations that specify a service of a class or component.
- An interface therefore describes the externally visible behaviour of that element. An interface might represent the complete behaviour of a class or component or only a part of that behaviour. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface.

3. Use case

- A use case in a use case diagram is a visual representation of distinct business functionality in a system. Use cases include anything that is within the system.
- A Use-Case can be defined as : A sequence of actions a system performs that yields a measurable (observable) result of value to a particular actor. A use case is shown as an ellipse in a use case diagram.

Fig. 1.2.6 : Use cases

- Fig. 1.2.6 shows two uses cases : "issue Book" and "Renew Book".

4. Collaborations

- Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behaviour that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioural, dimensions.
- A given class might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, collaboration is rendered as an ellipse with dashed lines, usually including only its name.

Fig. 1.2.7 : Collaboration

5. Active class

- An active class is a class whose objects own one or more processes or threads.
- Therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behaviour is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations, as in Fig. 1.2.8.

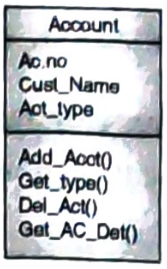

Fig. 1.2.8 : Active class

6. Component

- The Component icon represents a software module with a well-defined interface. In the component specification, you specify the type of component in the Stereotype field (e.g., ActiveX, Applet, Application, DLL, and Executables).
- A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name.

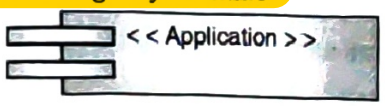

Fig. 1.2.9 : Component

7. Node

- A node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability
- Use nodes to model the topology of the hardware on which your system executes.
- A node typically represents a processor or a device on which artifacts may be deployed

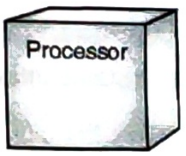

Fig. 1.2.10 : Node

B. Behavioural Things

Behavioural things are the dynamic parts of UML models. These are the verbs of a model, representing behaviour over time and space. There are two primary kinds of behavioural things.

i. An interaction
ii. A state machine

i. Interaction

- An interaction is a behaviour that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.
- The behaviour of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of other elements, including messages, action sequences and links. Graphically, a message is rendered as a directed line, almost always including the name of its operation.

Fig. 1.2.11

ii. State machine

- A state machine is a behaviour that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behaviour of an individual class or a collaboration of classes may be specified with a state machine.
- A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name.

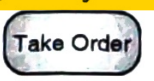

Fig. 1.2.12

C. Grouping Things

- A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioural things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time).

- Graphically, a package is rendered as a tabbed folder, usually including only its name. Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages)

Fig. 1.2.13 : Package

D. Annotational Things

- Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note.
- A note is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

Fig. 1.2.14 : Note

1.2.6(B) Relationship in UML

There are four types of relationships are in UML.

1. Association	2. Realization
3. Generalization	4. Dependency

1. Association

- An **Association** can be defined as : The semantic relationship between two or more classifiers that specifies connections among their instances. In other words, an association is a structural relationship that specifies that objects (instances of classes) of one thing are connected to objects of another thing.
- Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names.

Fig. 1.2.15 : Association

2. Realization

- A **realization** is a semantic relationship between classifiers, wherein one classifier
- Specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship.

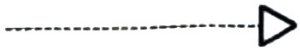

Fig. 1.2.16 : Realization

3. Generalization

- It is a relationship among classes where one class shares the structure and/or behaviour of one or more classes. Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. But the child has attributes and operations in addition to those found in its parents. Graphically, a
- Generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent.

Fig. 1.2.17 : Generalization

4. Dependency

- Dependency is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing).

- Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label

Fig. 1.2.18 : Dependency

1.2.6(C) Diagrams in UML

UML is made up of nine diagrams that can be used to model a system at different points of time in the software life cycle of a system. The nine UML diagrams are: Use case diagram, Class diagram, Object diagram, State diagram, Activity diagram, Sequence diagram, Collaboration diagram, Component diagram, Deployment diagram.

UML Diagram Classification

- A software system can be said to have two distinct characteristics: a structural, "static" part and a behavioural, "dynamic" part. In addition to these two characteristics, an additional characteristic that a software system possesses is related to implementation. Before we categorize UML diagrams into each of these three characteristics, let us take a quick look at exactly what these characteristics are :

  Static (static model) : The static characteristic of a system is essentially the structural aspect of the system. The static characteristics define what parts the system is made up of. A static model can be viewed as "snapshot" of a system's parameters at rest or at a specific point in time. The goal of static model is to capture those concepts from the real world that are important to an application. Static model assume stability and absence of change in data over time. The classes' structure and their relationships to each other frozen in time are examples of static models i.e. UML class diagram is an example of class diagram

- **Dynamic (dynamic model) :** The behavioural features of a system; for example, the ways a system behaves in response to certain events or actions are the dynamic characteristics of a system. The dynamic model describe those aspect of the system concerned with time and the sequencing of operations-events that mark changes, sequences of events, states that define context for events and organization of events and states. The dynamic model captures control , that aspect of system which describes the sequences of operation that occur, without regard for what operation do, what they operate on, or how they are implemented.

- **Implementation :** The implementation characteristic of a system is an entirely new feature that describes the different elements required for deploying a system.

- The UML diagrams that fall under each of these categories are :

1. **Static**
   - Use case diagram
   - Class diagram
2. **Dynamic**
   - Object diagram
   - State diagram
   - Activity diagram
   - Sequence diagram
   - Collaboration diagram
3. **Implementation**
   - Component diagram
   - Deployment diagram

1.2.7 COMET Use Case–Based Software Life Cycle

University Question

Q. Explain COMET and phases of COMET.

SPPU : Feb. 16, 5 Marks

Q. Comment on how COMET is useful in software design and analysis. **SPPU : April 18, 5 Marks**

Q. What is COMET ? Explain phases of COMET.

SPPU : April 19, 5 Marks

- The **Collaborative Object Modelling and Design Method**, or **COMET**, uses the **UML** notation to describe the design. COMET is based on the design concepts of information hiding, classes, inheritance, and concurrent tasks.
- It uses a design strategy of concurrent object design, which addresses the structuring of a software system into active and passive objects and defines the interfaces between them.
- It provides structuring criteria to help structure the system into objects during analysis, and additional criteria to determine the subsystems and concurrent tasks during design.
- COMET an iterative use case driven and object-oriented software development method that addresses the requirements, analysis, and design modelling phases of the software development life cycle.
- The functional requirements of the system are defined in terms of actors and use cases.
- Each use case defines a sequence of interactions between one or more actors and the system.
- A use case can be viewed at various levels of detail. In a requirements model, the functional requirements of the system are defined in terms of actors and use cases.

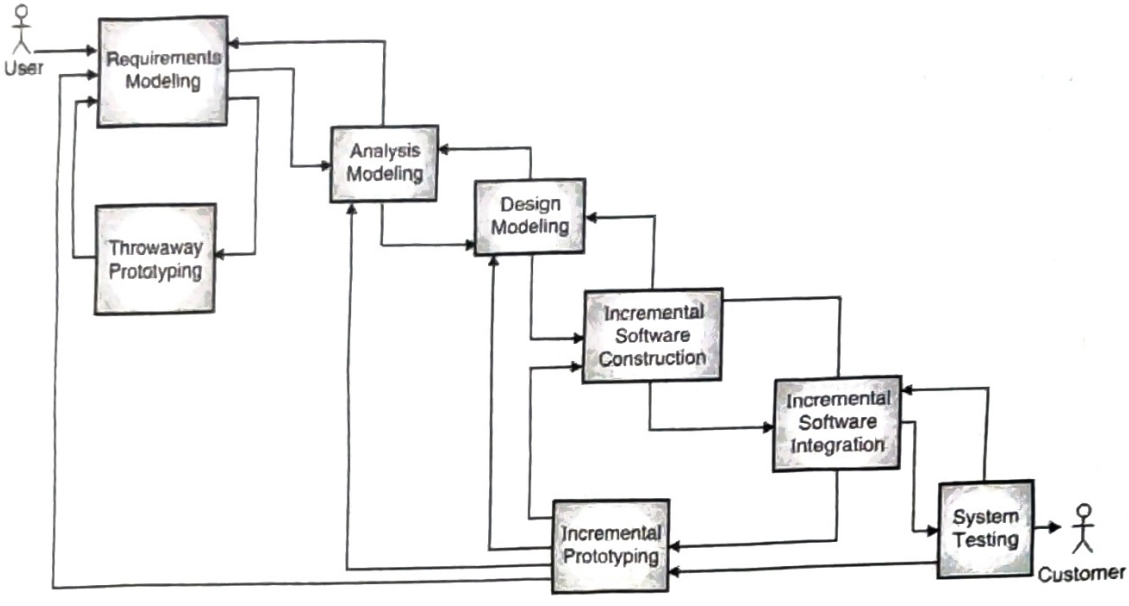

Fig. 1.2.19: COMET

- In an analysis model, the use case is realized to describe the objects that participate in the use case and their interactions.
- In the design model, the software architecture is developed, addressing issues of distribution, concurrency and information hiding.

The basic phases of COMET are:

1. Requirements Modeling
2. Analysis Modeling
3. Design Modeling
4. Incremental Software Construction
5. Incremental Software Integration
6. Software Testing

1. **Requirements modeling** : In the Requirements Modeling phase, a use case model is developed in which the functional requirements of the system are defined in terms of actors and use cases. each use case is documented with a flow of events. The flow of events for a use case is a description of the events needed to accomplish the required behavior of the use case. Throwaway prototypes can help to clarify the requirements.

2. **Analysis modeling :** In the Analysis modeling phase, static and dynamic models of the system are developed. The static model defines the structural relationships among problem domain classes. The dynamic model represents the time-dependent aspects of a system. It is concerned with the temporal changes in the states of the objects in a system. In the dynamic model, state dependent objects are defined using state charts.

3. **Design modeling** : In design modeling software architecture is designed. Problem Domain (Analysis Mode) is mapped to Solution Domain (Design Model). Subsystems are identified and structured. Emphasis is on designing distributed subsystems as configurable components that communicate with each other via messaging.

4. **Incremental software construction** : It involves selecting subset of system based on use cases. Incremental Software Construction consists of, detailed design, coding and unit testing. For each class in subset detailed design in Pseudo code or Structured English is written. Coding is done as per specification using programming language such as Java. Each subset is unit tested using testing framework e.g. Junit/Mokito.

5. **Incremental software integration** : It involve Integration testing of each system increment is performed. Integration tests are based on use cases.

6. **System testing** : It involves testing of software functional requirements based on use cases. The use cases of the requirements model are tested black box, ensuring the software meets specifications.

1.3 Requirement Study

A requirement is information/statement about what the projected system will do and how the system will be developed and all the stakeholders have agreed on it.

1.3.1 Types of Requirements

– Requirements can be divided into two major types :

1. **Functional requirement:** It describes the interaction between system and its environment independent of its implementation. Functional requirements are associated with specific functions, tasks or behaviours the system must support, while non-functional requirements are constraints on various attributes of these functions or tasks. Normally, a Software Architect makes use of the Use Cases and Scenarios (instances of Use Cases) generally following a standard UML notations, in order to capture the functional requirements through a system analysis process.

2. **Non-functional requirement:** Non-Functional Requirements which are primarily based on parameters of system performance, software quality attributes, reliability and security, cost, constraints in design/implementation etc.

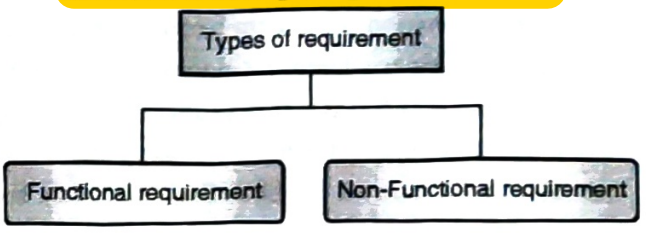

Fig. 1.3.1 : Types of requirements

– Non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. This should be contrasted with functional requirements that define specific behavior or functions. The plan for implementing functional requirements is detailed in the system design. The plan for implementing non-functional requirements is detailed in the system architecture.

- In general, functional requirements define what a system is supposed to do whereas non-functional requirements define how a system is supposed to be. Functional requirements are usually in the form of "system shall do requirement", while non-functional requirements are "system shall be requirement".

1.3.1(A) Functional Requirements

- The functional requirements discuss the functionality required by the users from the system.
- The functional requirements should include everything user does with system and everything that system does for other system .
- Functional requirement are mainly depend upon type of system where software to be used.
- The functional requirements can specifies what are inputs to system(data and commands from the users and other system) , what are output to system(screen or printed), what data system store, and what computations system should perform on data.
- Consider system as performing a set of functions {fi}. These functions can be considered similar to mathematical function. $F : I \rightarrow O$ meaning that a function transforms an element (ii) in input domain to value (oi)in output domain.
- Fig. 1.3.2 shows each function takes input data and then transformation input data to output data. For example consider Library automation system, searching-book is functional requirement. Searching book function takes book name as input , run matching algorithm on the book list, and return the matched books names.

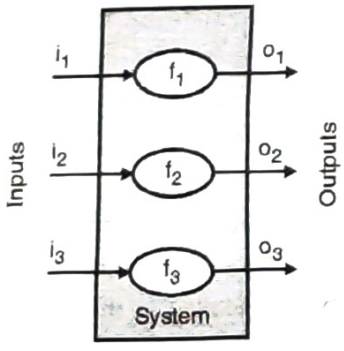

Fig. 1.3.2 : Black box view of function of system

1.3.1(B) Non-Functional Requirements

- A non-functional requirement is a statement of how a system must behave; it is a constraint upon the systems behaviour.
- Non-functional requirements specify all the remaining requirements not covered by the functional requirements. They specify criteria that judge the operation of a system, rather than specific behaviours, for example:"Display of the patient's vital signs must respond to a change in the patient's status within 2 seconds."
- Non-Functional Requirements can be categorized as
  - Quality Constraints
  - Performance Constraints
  - Design Constraints
  - Environmental constraints

Quality Constraints

Quality Constraints reflects the three qualities attributes reliability, maintainability and reusability. These requirements constrain the design to meet specified levels of quality.

- **Maintainability and enhancement:** A software product is said maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product and functionalities of the product can be easily modified. Thus the software product that is maintainable should be well-documented, should not be complex, and should have spare capacity for memory, storage and processor utilization and other resources.
- **Reusability:** Reusability is the likelihood a segment of source code that can be used again to add new functionalities with slight or no modification. Reusability reduces implementation time as reusable module is already tested.
- **Reliability:** Reliability is how much time system is available between failures. It is specified by using mean time to failure. Reliability includes robustness, safety and security. It is the ability of a system or component to perform its required functions under stated conditions for a specified period of time.

Performance Constraints

- **Response time** : Response time is time system take to compute the result. Low response time is desirable.
- **Throughput** : It is number of computation or transactions per unit time. Some systems takes long time for processing or some system continually responding to client it is necessary to consider throughput.
- **Utilization** : utilization is resource consumed by the system. Utilization of resources such memory or network bandwidth are specified. E.g. system should not use more than 20MBPS of bandwidth.

Design Constraints

- **Availability** : Availability is period of time system is available to user to use(responding to user. E.g. system should not down for more than 5 minutes
- **Recovery from failure** : If system is down because of some reason such as software crash, recovery from failure specifies the amount of time system will take to restore to previous state.(recover).
- **Development process to be used** : In an organization for quality control some processes are followed. In requirement document reference to these processes can be placed.

Environmental Constraints

- **Platform** : platform specifies the hardware and operating system used for developing system. Linux operating systems with 2GB RAM.
- **Technology to be used** : It is signifies the programming language (framework) used to develop a system. Sometimes client places constraint on programming language to be used for developing system. E.g. system should be developed by using Java language.

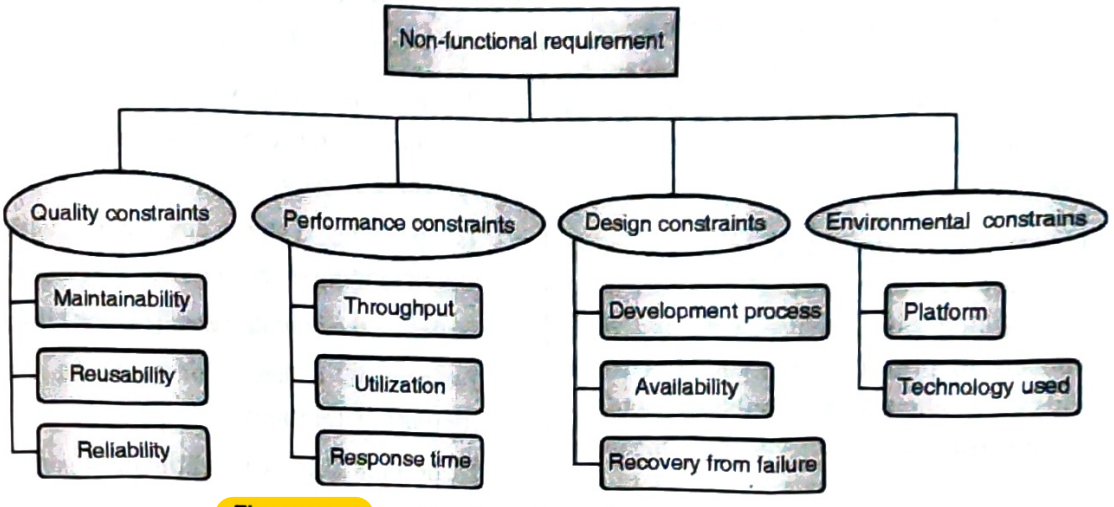

Fig. 1.3.3 : Classification of non-functional requirement

1.3.2 Requirement Analysis

- Requirement engineering gives the appropriate mechanism for understanding what customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, validating specification and managing the requirements as they are transformed into operational system.
- The requirement engineering process is accomplished through the execution of seven distinct functions:

1. Inception	2. Elicitation
3. Elaboration	4. Negotiation
5. Specification	6. Validation
7. Management	

1. **Inception**

In this phase you should understand the feasibility, scope of problem, stakeholders that need solution and how different stakeholder communicate with each other.

2. **Elicitation**

Requirement elicitation is the action of discovering system requirements through consultation with stakeholders. Some of the sources of this discovery could be system documents, domain knowledge, and market studies. From the customer, user find-out the objectives of system, how to accomplish these objectives, and how system is used for day to day operation. Number of problem comes in requirement elicitation such as scope of system is not defined properly, requirements are conflicting and ambiguous. Also over the time requirement changes.

3. **Elaboration**

In elaboration requirement obtain from above steps are refined. Elaborations finds out user scenarios that describe how end user going to use the system. Then analysis of each user scenario is done by drawing various UML diagrams. And finally requirements are clarified.

4. **Negotiation**

Some requirements are conflicting, such conflicting requirements are reconcile through a process of negotiation. We first rank requirements and then discuss conflicts in priority. By negotiation some requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

5. **Specification**

Requirement specification or documentation is the action of documenting agreed requirements at a certain level of detail. Requirements Specification is a complete description of the behaviour of the system to be developed. It includes a set of use cases that describe all the interactions the users will have with the software. Use cases are functional requirements. In addition to use cases it contains non-functional requirement. Non-functional requirements are requirements which impose constrains on the design or implementation.

6. **Validation**

Requirement validation is the process of reviewing requirements for clarity, consistency and completeness. Requirements validation examines that requirement specified in SRS are the stated unambiguously; and that the work products conform to the standards established for the process, the project, and the product. Requirement is validated by using technical review. The review team consists of software engineers, customers, users, and other stakeholders.

7. **Requirements management**

When a change is needed in a requirement, there might be many artifacts affected by this change such as design component, test cases and source code. For these reasons it is important that changes to requirements are carefully traced, analysed. Requirement management also keep tacks of different versions of the requirement documents. Every time you change requirement , you should give the document a new version number.

Thus, the purpose of the Requirements workflow is :

- To establish and maintain agreement with the customers and other stakeholders on what the system should do.
- To provide system developers with a better understanding of the system requirements.
- To define the boundaries of (delimit) the system.
- To provide a basis for planning the technical contents of iterations.
- To provide a basis for estimating cost and time to develop the system.
- To define a user-interface for the system, focusing on the needs and goals of the users.

1.3.3 SRS Design

- SRSs are typically developed during the first stages of "Requirements Development," which is the initial product development phase in which information is gathered about what requirements are needed--and not. The actual specification, then, is written after the requirements have been gathered and analyzed.

- It's important to note that an SRS contains functional and non-functional requirements only; it doesn't offer design suggestions, possible solutions to technology or business issues, or any other information other than what the development team understands the customer's system requirements to be.
- The SRS is often referred to as the "parent" document because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans, and documentation plans, are related to it.
- A well-designed, well-written SRS accomplishes four major goals:

1. It provides feedback to the customer. An SRS is the customer's assurance that the development organization understands the issues or problems to be solved and the software behaviour necessary to address those problems. Therefore, the SRS should be written in natural language, in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables etc.
2. It decomposes the problem into component parts. The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas, and helps break down the problem into its component parts in an orderly fashion.
3. It serves as an input to the design specification. The SRS serves as the parent document to subsequent documents, such as the software design specification and statement of work. Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised.
4. It serves as a product validation check. The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

1.3.4 Organization of SRS Document

Organization of SRS document is given below . Depending on type of project, some sections can be omitted or added by system analyst.

General Outline of a SRS

1. Introduction :

- 1.1 Product Overview
- 1.2 Purpose
- 1.3 Scope
- 1.4 Reference
- 1.5 Definition and Abbreviation

2 Overall descriptions :

- 2.1 Product Perspective
- 2.2 Product Functions
- 2.3 User Characteristics
- 2.4 General Constraints
- 2.5 Assumptions and Dependencies

3 Specific requirements

- 3.1 External Interface Requirements
  - 3.1.1 User Interfaces
  - 3.1.2 Hardware Interfaces
  - 3.1.3 Software Interfaces
  - 3.1.4 Communications interfaces
  - 3.1.5 Memory Constraints
  - 3.1.6 Operation
- 3.2 Software Product Features
- 3.3 Software System Attributes
  - 3.3.1 Reliability
  - 3.3.2 Availability
  - 3.3.3 Security
  - 3.3.4 Maintainability
  - 3.3.5 Portability
- 3.4 Database Requirements
- 3.5 Other Requirements

4 Additional material(Optional)

1. Introduction

The following subsections of the Software Requirements Specifications (SRS) document should provide an overview of the entire SRS. The thing to keep in mind as you write this document is that you are telling what the system must do – so that designers can ultimately build it.

1.1 Overview

In this subsection :

1. Describe what the SRS contains
2. Explain how the SRS is organized

1.2 Purpose

Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.

1.3 Scope

Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies

1.4 References

List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.

1.5 Definitions, Acronyms, and Abbreviations

Provide the definitions of all terms, acronyms, and abbreviations required to properly interpret the SRS. This information may be provided by reference to one or more appendices in the SRS or by reference to documents. This information may be provided by reference to an Appendix.

2. The Overall Description

This section gives information about products ,its function, constrains which will help to understand requirement

2.1 Product Perspective

Describe the context and origin of the product being specified in this SRS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SRS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.

2.2 Product Functions

Summarize the major functions the product must perform or must let the user perform. Details will be provided in Section 3, so only a high level summary (such as a bullet list) is needed here. Organize the functions to make them understandable to any reader of the SRS. A picture of the major groups of related requirements and how they relate, such as a top level data flow diagram or object class diagram, is often effective.

2.3 User Characteristics

Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the most important user classes for this product from those who are less important to satisfy.

2.4 General Constraints

Describe any items or issues that will limit the options available to the developers. These might include : corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer's organization will be responsible for maintaining the delivered software).

2.5 Assumptions and Dependencies

List each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption might be that a specific operating system would be available on the hardware designated for the software product. If, in fact, the operating system were not available, the SRS would then have to change accordingly. This section is catch-all for everything else that might influence the design of the system and that did not fit in any of the categories above.

3. Specific Requirements

This section contains all the software requirements at a level of detail sufficient to enable designers to design a system to satisfy those requirements, and testers to test that the system satisfies those requirement

3.1 External Interface Requirements

This contains a detailed description of all inputs into and outputs from the software system.

3.1.1 User Interfaces

Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.

3.1.2 Hardware Interfaces

Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.

3.1.3 Software Interfaces

Describe the connections between this product and other specific software components (name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.

3.1.4 Communications Interfaces (Protocols)

Describe the requirements associated with any communications functions required by this product, including e-mail, web browser, network server communications protocols, electronic forms, and so on. Define any pertinent message formatting. Identify any communication standards that will be used, such as FTP or HTTP. Specify any communication security or encryption issues, data transfer rates, and synchronization mechanisms.

3.1.5 Memory Constraints

Specify any applicable characteristics and limits on primary and secondary memory. If all the customer's machines have only 128K of RAM, then your target design has got to come in under 128K so there is an actual requirement.

3.1.6 Operations

Specify the normal and special operations required by the user such as:

1. The various modes of operations in the user organization
2. Periods of interactive operations and periods of unattended operations
3. Data processing support functions
4. Backup and recovery operations

3.2 Software Product Features

Software product feature is nothing but functional requirements. Functional requirements define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs.

These include :

- Validity checks on the inputs
- Exact sequence of operations
- Responses to abnormal situation, including overflow, communication facilities, error handling and recovery.
- Relationship of outputs to inputs, including input/output sequences, formulas for input to output conversion

3.3 Software System Attributes

- There are a number of attributes of software that can serve as requirements. It is important that required attributes by specified so that their achievement can be objectively verified. The following items provide a partial list of examples. These are also known as non-functional requirements or quality attributes.
- These are characteristics the system must possess, but that pervade (or cross-cut) the design. These requirements have to be testable just like the functional requirements. Its easy to start philosophizing here, but keep it specific.

3.3.1 Reliability

Specify the factors required to establish the required reliability of the software system at time of delivery. If you have MTBF requirements, express them here. This doesn't refer to just having a program that does not crash. This has a specific engineering meaning.

3.3.2 Availability

Specify the factors required to guarantee a defined availability level for the entire system such as checkpoint, recovery, and restart. This is somewhat related to reliability. Some systems run only infrequently on-demand (like MS Word). Some systems have to run 24/7 (like an e-commerce web site). The required availability will greatly impact the design. What are the requirements for system recovery from a failure ? "The system shall allow users to restart the application after failure with the loss of at most 12 characters of input".

3.3.3 Security

Specify the factors that would protect the software from accidental or malicious access, use, modification, destruction, or disclosure. Specific requirements in this area could include the need to :

- Utilize certain cryptographic techniques
- Keep specific log or history data sets
- Assign certain functions to different modules
- Restrict communications between some areas of the program
- Check data integrity for critical variables

3.3.4 Maintainability

Specify attributes of software that relate to the ease of maintenance of the software itself. There may be some requirement for certain modularity, interfaces, complexity, etc. Requirements should not be placed here just because they are thought to be good design practices if someone else will maintain the system.

3.3.5 Portability

Specify attributes of software that relate to the ease of porting the software to other host machines and/or operating systems. This may include :

- Percentage of components with host-dependent code.
- Percentage of code that is host dependent.
- Use of a proven portable language.
- Use of a particular compiler or language subset.
- Use of a particular operating system.

3.4 Logical Database Requirements

- This section specifies the logical requirements for any information that is to be placed into a database. This may include :
  - Types of information used by various functions
  - Frequency of use
  - Accessing capabilities
  - Data entities and their relationships
  - Integrity constraints
  - Data retention requirements
- If the customer provided you with data models, those can be presented here. ER diagrams (or static class diagrams) can be useful here to show complex data relationships. Remember a diagram is worth a thousand words of confusing text.

1.3.5 SRS for UMS (University Management System)

1.3.5(A) General Description

UMS is University Management System for managing the records of the alumni's of the university as well as staff, faculty and higher authorities.

Purpose

The purpose for developing this type of software or introducing this UMS is to facilitate everyone who is concerned with the university.

Scope

The scope of UMS is global i.e. it should be able to be accessed from anywhere through internet i.e. registered users must be able to login to their accounts by directly accessing the university's website and then signing in with their username and password anytime and anywhere.

Abbreviation

UMS University Management System

Overview

The UMS is able to have a user interface. It should have drop down boxes and if we drag mouse on any control at our welcome screen information regarding that the control should be displayed. Help menu should be there. As a teacher it should provide them to upload the various assignments and the attendance of the students. As a developer it should make a user interface which is user friendly. He should make the UMS as simple as he can. Backup at the main server should be made.

1.3.5(B) Overall Description

Product perspective

Product i.e. UMS should be able to provide a basic and easy interchange of information i.e. it should be able to remove the communication gaps between a teacher and the student. It should have chat facilities for all the users that are online. It should be compatible with all the operating systems.

Product functions

The following are the product functions of the UMS:

- The UMS login box should on the official website of the university.
- The password field should be secured.
- After signing in all updates and new announcements for users should be displayed.
- By clicking on the dropdown box of the options the user should be able to view progress reports, assignments, notes, attendance, placement services and results.

- User should be able to change the passwords.
- Web pages should supportpdf, ppt, doc and similar supported formats so that they can be easily downloaded and unloaded.

User Characteristics

A user can only have his/her registration number as username so if he joins the university then only he can then only he can login. This prevents misuse, unauthorized access and hacking if the product.

General Constraints

Server capacity is how many users can access or can be online at once. More is the number of users more will be the network traffic and hence the server comes in a down state. Personal firewall and updating is a tough task, it should be such that it should not block the network traffic, making the system slower. Firewall of the UMS should not collide with the firewall of the user system.

Assumptions and Dependencies

UMS should work even at when the network traffic is high. Server should have a power backup as well as a database backup. The UMS should be compatible with most of the operating systems i.e. previous and latest ones.

1.3.6 Specific Requirements

1.3.6(A) External Interface Required

User Interfaces

The external users are the students and the teachers of the university. The students can have an access to their accounts for their attendance, assignments etc. The teachers have also an account to access their account for uploading of the students' attendance and the assignments to be submitted by them.

Hardware Interfaces

The external hardware interface used for accessing the UMS is the personal computers of the teachers and the students. The PCs may be laptops with wireless LAN as the internet connections provided will be wireless.

Software Interfaces

The Operating Systems can be any version of Windows, Linux, UNIX or Mac which supports TCP/IP protocols.

Communication Interfaces

The communication interface is a local area network through wireless network routers.

1.3.6(B) Performance Requirements

The PCs used must be at least Pentium 4 machines so that they can give optimum performance of the product.

1.3.6(C) Design Constraints

The constraints at the designing time are that the needs of the university students and the teachers may keep on changing so the designers must keep this in view and design the product in this way that it is easily updatable.

1.3.6(D) Attributes

The following are the attributes of the product UMS:

- It should be equipped with current and archive database.
- All records can easily be updated.
- It should have its personal firewall.
- It should facilitate student with updating his/her account, downloading or uploading of assignments from anywhere.
- It should also do the same for teachers they can also have their pay checks online i.e. UMS should be capable of online transaction.

1.3.6(E) Other Requirements

The software is such that as the time goes by the need of the university management, students and teachers may keep on changing thus it is made to change from time to time.

1.3.7 Requirements Modeling

- Requirements modeling is the process used in software development projects where requirements and solutions constantly evolve through collaborative efforts and teamwork. By using this method of cross-functional and self-organizing teams, you can ensure that your team meets the exact needs of the stakeholders.
- A requirement needs to meet several criteria to be considered a "good requirement" Good requirements should have the following characteristics:

1. Unambiguous	2. Testable (verifiable)
3. Clear (concise, terse, simple, precise)	
4. Correct	5. Understandable
6. Feasible (realistic, possible)	
7. Independent	8. Atomic
9. Necessary	10. Implementation-free (abstract)
11. Consistent	12. Non-redundant
13. Complete	

1. Unambiguous

There should be only one way to interpret the requirement .Language used in the statement must not leave a doubt in the reader's mind as to the intended descriptive or numeric value. Sometimes ambiguity is introduced by undefined acronyms. The requirement is concisely stated without recourse to technical jargon, acronyms other esoteric verbiage. Negative statements and compound statements are prohibited.

For example :

- **Bad requirement** : The system shall be implemented using ASP. Here ASP means Active Server Pages or Application Service Provider? To solve this, we can mention a full name and provide an acronym in parentheses
- **Good requirement** : The system shall be implemented using Active Server Pages (ASP).
- Here's another example :
- **Bad requirement** : The system shall not accept passwords longer than 10 characters. It is not clear what the system is supposed to do. The requirement is corrected to give the clarification.
- **Good requirement** : The system shall not accept passwords longer than 10 characters. If the user enters more than 10 characters while choosing the password, an error message shall ask the user to correct it.

2. Testable (Verifiable)

Requirement is stated in such a way that it can be tested by inspection, analysis or demonstration. Testers should be able to verify whether the requirement is implemented correctly. The test should either pass or fail. Testable requirement makes possible to evaluate whether system meet requirement. The expected methods are usually test or demonstration. The inspection or analysis are done just in special cases.

For example :

- **Bad requirement** : System must be user friendly. Here problem is that how to measure friendliness.
- **Good requirement** : User interface should be menu driven. It shall provide textbox, dialog box, help screen, radio button to give input.

3. Complete

A set of requirement is complete, if and only if t describes all significant requirements of concern to the user, including the requirement associated with functionality, performance, design constrains, attributes or external interfaces. The requirement is fully stated in one place with no missing information. It should not give any space for guessing. A requirement should be specified for all conditions that can occur. High degree of completeness is achieved through reviews, prototyping and cross checking with other similar system and application.

4. Clear, Concise, Simple, Precise

Each requirement should be expressed using language that customer can understand. Requirements should not contain unnecessary verbiage or information. They should be stated clearly and simply. The requirement statement includes only one requirement stating what must be done and stated simply and clearly. It is easy to read and understand.

For example :

- **Bad requirement :** Sometimes the user will enter Airport Code, which the system will understand, but sometimes the closest city may replace it, so the user does not need to know what the airport code is, and it will still be understood by the system. This sentence may be replaced by a simpler one
- **Good requirement :** The system shall identify the airport based on either an Airport Code or a City Name.

5. Feasible (Realistic, Possible)

The requirement can be implemented within the constraints (such as time, money, and available resources) of the project.

For Example :

- **Bad requirement :** There placement control system shall be installed with no disruption to production. This is an unrealistic expectation.
- **Good requirement :** The replacement control system shall be installed causing no more than 1 days of production disruption.

6. Independent

To understand the requirement, there should not be a need to know any other requirement.

For example :

Req1 : The list of available flights shall include flight numbers, departure time, and arrival time for every leg of a flight.

Req2 : It should be sorted by price.

The word "It" in the second sentence refers to the previous requirement. However, if the order of the requirements changes, this requirement will not be understandable

7. Atomic

The requirement should contain a single traceable element.

For Example :

- The system shall provide the opportunity to book the flight, purchase a ticket, reserve a hotel room, reserve a car, and provide information about attractions.
- This requirement combines five atomic requirements, which makes traceability very difficult. Sentences including the words "and" or "but" should be reviewed to see if they can be broken into atomic requirements.

8. Traceable

- Requirement is traceable if and only if the origin of each of its component requirement is clearly linked to the need and if there is a mechanism that makes it feasible to trace the requirement to the need.
- In practice, this is made possible through structuring the requirement in parts and the components with proper code, so that one can navigate backward or forward.

Theoretical model of requirements could be as under :

Needs :	A,B, C,D	
Requirements :	A1, A2, A3	An.
Functions :	A1.1, A1.2, A1.3, A1.4	A1.n.
Features :	A1.11, A1.12, A1.13, A1.14	A1.1n

9. Non-redundant

Each requirement should be expressed only once and should not overlap with another requirement.

For example

Req1 : Calendar shall be available to help with entering the flight date.

Req2 : The system shall display a pop-up calendar when entering any date. The first requirement (related to only the flight date) is a subset of the second one (related to any date entered by the user).

10. Necessary

- The stated requirement is an essential capability, physical characteristic, or quality factor of the product or process. If it is removed or deleted, a deficiency will exist, which cannot be fulfilled by other capabilities of the product or process.

11. Consistent

- The set of requirements does not have individual requirements which are contradictory. The same term is used for the same item in all requirements. Two requirements are said to be inconsistent, if one of the requirements contradicts the other, or two-end users of the systems give inconsistent description of the requirement.
- One of the most important ways to help ensure consistency is to avoid duplicating requirement. It is very common, for example, for a requirement to be stated in the in introduction to the document, and then repeated in more detail in the body of the document. This is dangerous because if a change is made in one place, there is a tendency to forget to make the change in the other place.

12. Implementation free

The requirement states what is required, not how the requirement should be met. A requirement statement should not reflect a design or implementation nor should it describe an operation. However, the treatment of interface requirements is generally an exception. Implementation free allows system developer to decide what technology is best suited to achieve the function.

For example :

- **Bad requirement :** After 3 unsuccessful attempts to log on, JavaScript routine must run and lock the user out of system. Here specifying JavaScript routine concerns how the requirement will be implemented.
- **Good requirement :** After 3 unsuccessful attempts to log on, user must be locked out of system.

13. Correct

The requirement meets all or part of a business need as authoritatively stated by stakeholders

14. Understandable

Requirements should be grammatically correct and written in a consistent style. Standard conventions should be used. The word "shall" should be used instead of "will," "must," or "may."

1.3.8 Managing Changing Requirement

- Change is inherent in the development of most software systems. Software requirements change and evolve from inception to deployment. Although the majority of the requirements efforts are performed at the beginning, they often continue just before code freeze.
- The changes in requirement can affect both system functionality and the wider business goals of the organization for which the system is developed and as well the project cost, resources, and schedule.
- When a change is needed in a requirement, there might be many artifacts affected by this change such as design components, test cases, and source code. For these reasons it is important that changes to requirements are carefully traced, analyzed and their effects on the system overall is properly assessed.
- The most common problem with software projects is that the requirements defined in the first phase change as the project progresses. This may occur because as development progresses and prototypes are developed, customers are able to more clearly see problems with the original plan and make necessary course corrections; it may also occur because changes in the external environment require reshaping of the original business problem and hence necessitates a different solution than the one originally proposed.
- Requirements analysis never stops. The development team continue to interact with the clients and users, asking about their problems and their ideas and showing them prototypes of the system. Changes to requirement are made when the benefits of changes must outweigh the costs.

- Certain small changes (e.g. look and feel of the UI) are usually quick and easy to make at relatively little cost. Larger-scale changes have to be carefully assessed; forcing unexpected changes into a partially built system will probably result in a poor design and late delivery. Some changes are enhancements in disguise. Avoid making the system bigger, only make it better.
- Requirement management also keeps track of different versions of the requirement documents. Every time you change requirements, you should give the document a new version number. You should use version control mechanism because sometime it is important to go back and look at decision that were previously made, but subsequently changed.

Why Requirements Change

People change their minds for many reasons, and do so on a regular basis. This happens because:

1. **They missed a requirement :** A stakeholder will be working with an existing system and realize that it's missing a feature.
2. **They identified a defect :** A bug, or more importantly the need to address the bug, should also be considered a requirement.
3. **They realize they didn't understand their actual need :** It's common to show a stakeholder your working system to date only to have them realize that what they asked for really isn't what they want after all. This is one reason why **active stakeholder participation** and short iterations are important to your success.
4. **Politics :** The political landscape within your organization is likely dynamic. When the balance of political power shifts amongst your stakeholders, and it always does, so do their priorities. These changing political priorities will often motivate changes to requirements.
5. **The marketplace changes :**Perhaps a competitor will release a new product which implements features that your product doesn't.
6. **Legislation changes :** Perhaps new legislation requires new features, or changes to existing features, in your software.
7. **Technology changes :** A new release of the operating system or some other system with which your system interacts may force you to reassess the requirement.

- Good project managers are aware of these possibilities and typically already have backup plans in place to deal with these changes.
- To solve this problem, you should:
  - Have a clearly defined process for receiving, analyzing and incorporating change requests, and make your customer aware of his/her entry point into this process.
  - Set milestones for each development phase beyond which certain changes are not permissible - for example, disallowing major changes once a module reaches 75 percent completion.
  - Ensure that change requests (and approvals) are clearly communicated to all stakeholders, together with their rationale, and that the master project plan is updated accordingly.

1.4 Use Case Modeling

Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how a system is used.

1.4.1 Use Case Diagram

- Shows what software is suppose to do from user point of view.
- Describes the behaviour of system from user stand point.
- Provide functional description of system and its major processes.
- Define the scope of the system you are building.

- Use case diagram allows for the specification of high level user goals that the system must carry out. These goals are not necessarily tasks or actions, but can be more general required functionality of the system.
- You apply use cases to capture the intended behaviour of the system you are developing, without having to specify how that behaviour is implemented. For example, you can specify how an ATM system should behave by stating in use cases how users interact with the system; you don't need to know anything about the inside of the ATM at all.
- Use case diagram shows the overall functionality of the system. Users, project managers, analysts, developers, quality assurance engineers, and anyone else interested in the system as a whole can view these diagrams and understand what the system is supposed to accomplish.

Definition : *A use-case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalization among the use cases.*

OR

Definition : *A Use case diagram is a graphical view of the actors, use cases, and their interactions identified for a system.*

1.4.2 Actor and Use Case Identification

1. Actor

- An actor is a role that a user plays in the system. It is important to distinguish between a user and an actor (better thought of as a role).
- A user of the system may play several different roles through the course of his, her or its job. Examples of actors are salesperson, manager, support person, and web store system. It is possible that the same person may be a sales person and also provide support. When creating a use case model, we are not concerned with the individuals, only the roles that they play.
- An actor in a use case diagram interacts with a use case. For example, for modelling a banking application, a customer entity represents an actor in the application. Similarly, the person who provides service at the counter is also an actor.
- It is up to you to consider what actors make an impact on the functionality that you want to model. If an entity does not affect a certain piece of functionality that you are modelling, it makes no sense to represent it as an actor.
- An actor is shown as a stick figure in a use case diagram depicted "outside" the system boundary as shown in Fig. 1.4.1.

Fig. 1.4.1 : An actor in a use case diagram

a. Actor identification

- To identify an actor, search in the problem statement for business terms that portray roles in the system. For example, in the statement "patients visit the doctor in the clinic for medical tests," "doctor" and "patients" are the business roles and can be easily identified as actors in the system.
- Actors include anything that is external to the system. Actor Provide input, Consumes output or Interact with system (software).
- There are two primary types of actors :
  1. Users of the system.
  2. Other systems that will interact with the system being built.
- The first type of actor is a physical person, or a user. These are the most common actors. For our flight reservation system, actors are the people who will be directly using the system.
- The second type of actor is another External system (Computer hardware and device) For example; the airline's reservation system may need to interface with an external application to validate credit cards for purchases. In this example, the external credit application is an actor.

- External system is outside the scope of the current project, but it does need to interface with our new system i.e. External system that needs some information from current system is also actor.
- Thus, Actor is anything that exchanges data with the system and is external to system. An actor may actively interchange information with the system or can be a passive recipient of information, or can be giver of giver of information.
- Actors are found in the problem statement and by conversations with customers and domain experts. The following questions may be used to help identify the actors for a system :
  - Who will use main functionality of the system?
  - Who is affected by the system?
  - Which external hardware or other systems (if any) use the system to perform tasks?
  - Who will need support from system to do their task?
  - Who need to maintain administrate and keep system working?
  - Which h/w device does the system need to handle?
  - What other system does system need to interact?
  - Who has interest in result produced?
  - Who will supply the system with this information, use this information, and remove this information?
  - Who will support and maintain the system?
  - Does the system interact with a legacy system?

b. Actor Classification

- Actors can be primary or secondary actors. Primary actors initiate a use case, while secondary actors support a use case or receive something of value from the use case.
  - i. **Primary Actors** : The Actor(s) using the system to achieve a goal.
  - ii. **Secondary Actors** : Actors that the system needs assistance from to achieve the primary actor's goal.
- **Example** : A bank loan officer wants to review a loan application from a customer, and part of the process involves a real-time credit rating check.
  - **Use Case Name** : Review Loan Application
  - **Primary Actor** : Loan Officer
  - **Secondary Actors** : Credit Rating System
- In above example we need to define the Secondary Actor because without the "Credit Rating System" we cannot successfully complete the Use Case. In other words, the goal of the Primary Actor is to successfully complete the Loan Application, but they need the explicit "help" of the Secondary Actor (Credit Rating System) to achieve this goal.

2. Use Case Identification

- A use case in a use case diagram is a visual representation of distinct business functionality in a system. Use cases include anything that is within the system.
- A use case is shown as an ellipse in a use case diagram.
- Fig. 1.4.2 shows two uses cases : "issue Book" and "Renew Book" in the use case diagram of a library management system.

Fig. 1.4.2 : Use cases in a use case diagram

Use-Case can be defined as

A sequence of actions a system performs that yields a measurable / (observable) result of value to a particular actor.

- **Actions** : An action is a computational or algorithmic procedure. It is invoked either when the actor provides a signal to the system or when the system receives a time event.
- **System performs** : The system provides the use case. An actor communicates with a use-case instance of the system.

- **An observable result of value :** A use case should ensure that an actor can perform a task that has an identifiable value.
- **A particular actor :** Each of these actors has its own demands on the system and requires its own set of use cases. The actor is key to finding the correct use case. Consider a visual modelling tool. The two actors to this application are developer and system administrator. Developer develops systems using modelling tool and a system administrator manages the tool. Each of them has its own demands on the system and requires its own set of use cases.
- The following questions may be used to help identify the use cases for a system :
  - Which function does actor require from system?
  - What does actor need to do?
  - Does actor need to read, create destroy, modify, or store some kind of information in the system
  - What do these events represent in terms of functionality?
  - Where does this input/output come from or go to?
  - What are the tasks of each actor?
  - Will any actor create, store, change, remove, or read information in the system?
  - Will any actor need to inform the system about sudden, external changes?
  - Does any actor need to be informed about certain occurrences in the system?
  - What use cases will support and maintain the system?
  - Can all functional requirements be performed by the use cases?
  - What use case will create, store, change, remove, or read this information?

3. Communication (Associations)

- Association represents communication between an actor and a use case i.e. association shows participation of actor in use case.
- A use case can be carried out by many actors and an actor may carry out many use cases. Actors may be connected to use cases only by an association.
- An association between an actor and a use case indicates that the actor and the use case instance of the system communicate with one another, each one able to send and receive messages. The arrow head is optional but it's commonly used to denote the initiator.

4. System Boundary

- A system boundary defines the scope of what a system will be. A system cannot have infinite functionality. So, it follows that use cases also need to have definitive limits defined.
- A system boundary of a use case diagram defines the limits of the system. The system boundary is shown as a rectangle spanning all the use cases in the system.
- Note that the actors in the system are outside the system boundary.

1.4.3 Use Case Relationship

Use cases share different kinds of relationships. A relationship between two use cases is basically a dependency between the two use cases.

1.4.3(A) Include

University Questions

Q. What is include stereotype in use case diagram ? Explain with an appropriate example.

SPPU : Feb. 16, April 17, May 17, 5 Marks

Q. Diagrammatically show include relationship on the context of use case diagram.

SPPU : May 16, 2 Marks

Q. Mention the significance of includes relation in the use case diagram with a suitable example.

SPPU : April 18, 5 Marks

- An includes relationship allows one use case to use the functionality provided by another use case. This relationship can be used in one of two cases.

- First, if two or more use cases have a large piece of functionality that is identical. This identical functionality is placed in a separate use case rather than documenting it in every use case that needs it. Each of the other use cases can then have include relationship with this new use case.
- The second case where an includes relationship is helpful is a situation in which a single use case has an unusually large amount of functionality. An include relationship can be used to model two smaller use cases instead. Include relationships are created between the new use case and use case that "uses" its functionality.

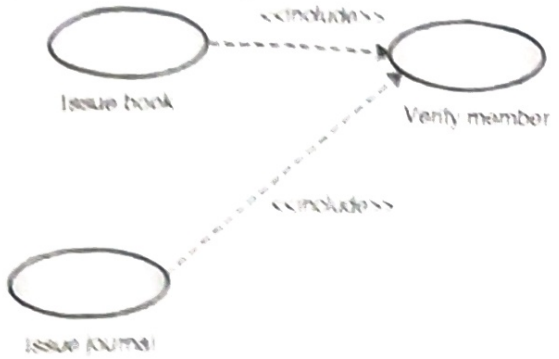

Fig. 1.4.3 : Example of include relationship

- An include relationship is depicted with a directed arrow having a dotted shaft. The tip of the arrowhead points to the parent use case and the child use case is connected at the base of the arrow. The stereotype "<<include>>" identifies the relationship as an include relationship.
- An include relationship between use cases means that the base use case explicitly incorporates the behaviour of another use case at a location specified in the base. The included use case never stands alone, but is only instantiated as part of some larger base that includes it. You can think of include as the base use case pulling behaviour from the supplier use case.
- The include arrow is drawn from a use case A to another use case B to indicate that the process of doing A always involves doing B at least once. A simple way to think about the include arrow is that it can be read A uses B means that "A has a B" as part of its behaviour.

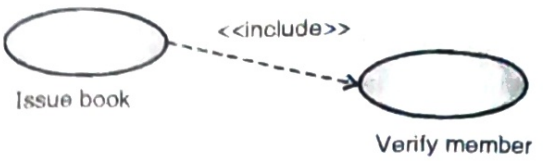

Fig. 1.4.4 : An example of an include relationship

- For example, in Fig. 1.4.4, you can see that the functionality defined by the "Verify member" use case is contained within the "issue books "use case. Hence, whenever the "Issue book" use case executes, the business steps defined in the "Verify member" use case are also executed.

1.4.3(B) Extend

University Questions

Q. What is extends stereotype in use case diagram ? Explain with an appropriate example.

SPPU : Feb. 16, April 17, May 17, 5 Marks

Q. Diagrammatically show extend relationship on the context of use case diagram.

SPPU : May 16, 2 Marks

Q. Mention the significance of Extends relation in the use case diagram with a suitable example.

SPPU : April 18, 5 Marks

- In an extend relationship between two use cases, the child use case adds to the existing functionality and characteristics of the parent use case.
- An extend relationship is depicted with a directed arrow having a dotted shaft, similar to the include relationship. The tip of the arrowhead points to the parent use case and the child use case is connected at the base of the arrow. The stereotype "<<extend>>" identifies the relationship as an extend relationship.

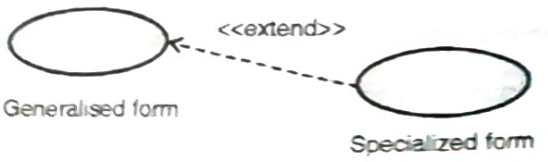

Fig. 1.4.5 : An example of an extend relationship

- An extend relationship between use cases means that the base use case implicitly incorporates the behaviour of another use case at a location specified indirectly by the extending use case.
- The base use case may stand alone, but under certain conditions, its behaviour may be extended by the behaviour of another use case. You can think of extend as the extension use case pushing behaviour to the base use case.
- An extend relationship is used to show
  - Optional behavior.
  - Behavior that is run only under certain conditions such as triggering an alarm.
  - Several different flows that may be run based on actor selection.

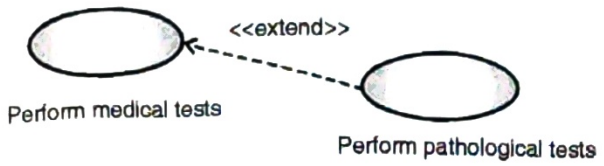

Fig. 1.4.6 : An example of an extend relationship

- Fig. 1.4.6 shows an example of an extend relationship between the "Perform medical tests" (parent) and "Perform Pathological Tests" (child) use cases. The "Perform Pathological Tests" use case enhances the functionality of the "Perform medical tests" use case. Essentially, the "Perform Pathological Tests" use case is a specialized version of the generic "Perform medical tests" use case.

Difference between uses and extends

- "X uses Y" indicates that the task "X" **has a** subtask "Y"; that is, in the process of completing task "X", task "Y" will be completed at least once.
- "X extends Y" indicates that "X" **is a** task of the same type as "Y", but "X" is a special, more specific case of doing "Y". That is, doing X is a lot like doing Y, but X has a few extra processes to it that go above and beyond the things that must be done in order to complete Y.

1.4.3(C) Use Case Generalization

- A generalization relationship is also a parent-child relationship between use cases. The child use case in the generalization relationship has the underlying business process meaning, but is an enhancement of the parent use case.
- In a use case diagram, generalization is shown as a directed arrow with a triangle arrowhead. The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.
- On the face of it, both generalizations and extends appear to be more or less similar. But there is a subtle difference between a generalization relationship and an extend relationship.

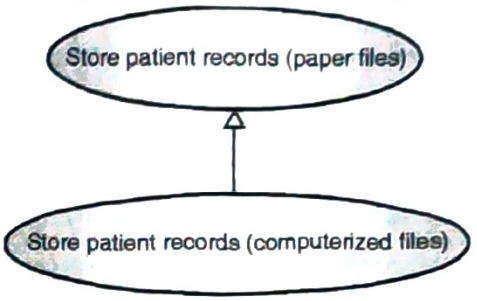

Fig. 1.4.7 : An example of a generalization relationship

- When you establish a generalization relationship between use cases, this implies that the parent use case can be replaced by the child use case without breaking the business flow. On the other hand, an extend relationship between use cases implies that the child use case enhances the functionality of the parent use case into a specialized functionality. The parent use case in an extend relationship cannot be replaced by the child use case.
- Let us see if we understand things better with an example. From Fig. 1.4.7, the diagram of a generalization relationship, you can see that "Store patient records (paper file)" (parent) use case is depicted as a generalized version of the "Store patient records (computerized file)" (child) use case. Defining a generalization relationship between the two implies that you can replace any occurrence of the "Store patient records (paper file)"
- Use case in the business flow of your system with the "Store patient records (computerized file)" use case without impacting any business flow.

- This would mean that in future you might choose to store patient records in a computerized file instead of as paper documents without impacting other business actions.
- Now, if we had defined this as an extend relationship between the two use cases, this would imply that the "Store patient records (computerized file)" use case is a specialized version of the "Store patient records (paper file)" use case. Hence, you would not be able to seamlessly replace the occurrence of the "Store patient records (paper file)" use case with the "Store patient records (computerized file)" use case.
- Generalization means that the child use case inherits the behaviour and meaning of the parent use case; the child may add to or override the behaviour of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances).
- For example, in a banking system, you might have the use case Validate User, which is responsible for verifying the identity of the user. You might then have two specialized children of this use case (Check password and Retinal scan), both of which behave just like Validate User and may be applied anywhere Validate User appears, yet both of which add their own behaviour (the former by checking a textual password, the latter by checking the unique retina patterns of the user.

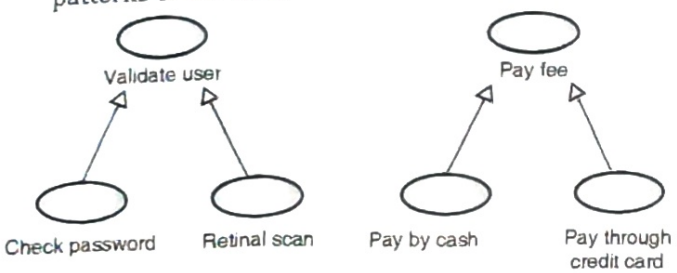

Fig. 1.4.8 : An example of a generalization relationship

1.4.3(D) Actor Generalization

- Actor generalization refers to the relationship, which can exist between two actors, and Generalization of an actor means that one actor (descendant) inherits the role and properties of another actor (ancestor).
- The generalization relationship also implies that the descendant actor can use all the use cases that have been defined for its ancestor.

Ex. 1.4.2 : When it comes to air travel, both a "Business Traveller" and a "Tourist" are "Passengers". The fact that they are passengers allow them to have common behaviour such as "Buy Ticket" but the fact that they are separate actors implies they can also have differences. The "Business Traveller" might be able to "Redeem Business Miles" while the "Tourist" cannot.

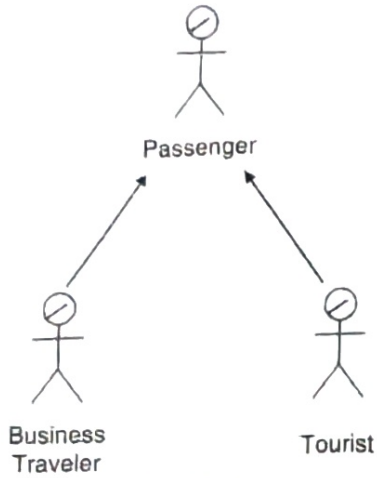

Fig. P.1.4.1

Soln. :

- A user can play several roles in relation to the system, which means that the user may, in fact, correspond to several actors. To make the model clearer, you can represent the user by one actor who inherits several actors. Each inherited actor represents one of the user's roles relative to the system.
- A Teller and an Accountant, both of whom check the balance of an account, are seen as the same external entity by the use case that does the checking. The shared role is modelled as an actor, Balance Supervisor, inherited by the two original actors. This relationship is shown with actor-generalizations.

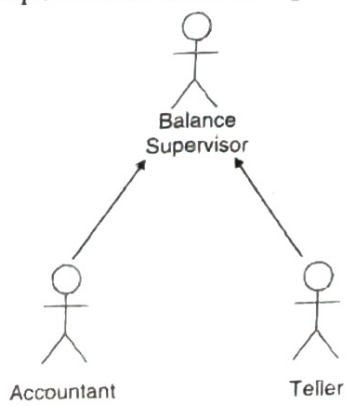

Fig. P.1.4.1(a)

- The actors Teller and Accountant inherit all the properties of a Balance Supervisor. Thus, both these actors can act as Balance Supervisors.

1.4.4 Use Case Specification

University Question

Q. Write a use case specification for the use case: Userauthentication. **SPPU : Dec. 18, May 19, 5 Marks**

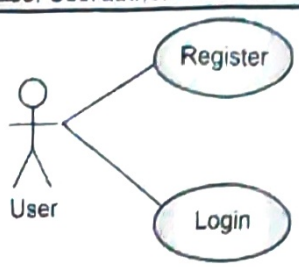

Fig. 1.4.9

Name	Login
Brief Description	A user of the System logs in to the System.
Actor(s)	Logged In User
Flow of Events	
Basic Flow	
This use case starts when a system user is not logged in to the system and goes to the login page. The System prompts the user for a username and password or register new account The user enters his/her username and password. The system validates the entered username and password, making sure that the entered username is a valid username in the System, and that the required password is entered for the entered username. The user is signed in and returned to the home page as a Logged In User. The use case ends.	
Alternate Flows	
Title	**Description**
User Fails Authentication	If the User entered an invalid username and/or password, the following occurs: The system describes the reasons why the User failed authentication. The system presents the User with suggestions for changes necessary to allow the User to pass authentication. The system prompts the User to re-enter the valid information. The Basic Flow continues where the User enters new information (see step 2 of the Basic Flow).

Pre-Conditions	
Title	**Description**
(none)	
Post-Conditions	
Title	**Description**
Success	The User is authenticated and the system displays a home page based on the user type.
Failure	User is unable to log in for one or more reasons.
Extension Points	
None	

Name	Register
Brief Description	A user of the System creates an account
Actor(s)	Guest
Flow of Events	
Basic Flow	
This use case starts when a system user is not logged in to the system and goes to the login page. The System prompts the user for a username and password or register new account. The user selects registration option. The System prompts user for registration information, Username, password, etc The user enters in their information. System verifies information and creates account. The use case ends.	
Alternate Flows	
Title	**Description**
Cancel Registration	The user selects the cancel option. The system returns the user to the home page without the user being logged in and any information entered has been erased.
Invalid Information Entered	User clicks submit after entering information system asked for. System displays information with appropriate message to correct invalid information. User re-enters information.

Pre-Conditions	
Title	**Description**
(none)	
Post-Conditions	
Title	Description
Success	The user entered successful information and is returned to the home page as a Logged In User
Failure	User is unable to log in for one or more reasons and is returned to the home page as a Guest.
Extension Points	
None	

Ex. 1.4.2 : Draw Use case diagrams for automated cold-drink machine with scenario ? **SPPU- April 19, 6 Marks**

Soln. :

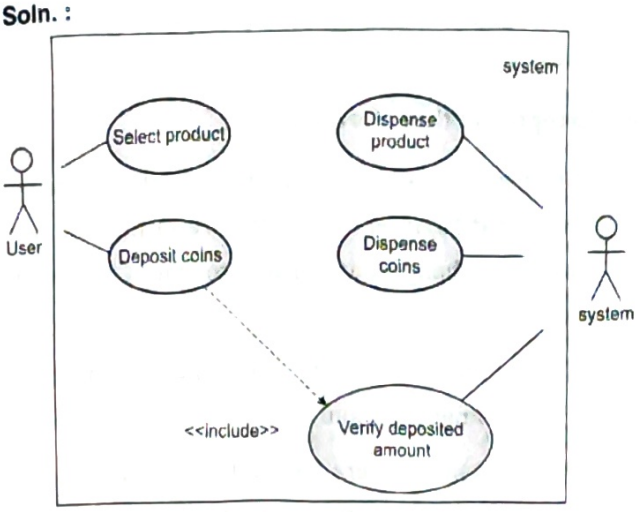

Fig. P.1.4.2

1.4.5 Use Case Template

A use case represents a class of functionality provided by the system as an event flow. Therefore each use case is documented with a flow of events. The flow of events for a use case is a description of the events needed to accomplish the required behavior of the use case. When you write this flow of events, you should include how and when the use case starts and ends, when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

1. **Actors :** A use case identifies all actors participating in it.
2. **Pre-conditions :** A use case may require specific conditions to hold for it to be successfully invoked. The preconditions for a use case list any conditions that have to be met before the use case can start at all. For example, a precondition might be that another use case has been executed or that the user has the necessary access rights to run the current use case. Not all use cases will have preconditions. A pre-condition identifies the mode required and any other conditions that must hold for the use case to be valid. This includes information such as the action requested, confirmation of user identity, values that must hold, and any other factors that affect successful conclusion of the use case. All such conditions must be identified.
3. **Main event flow :** A use case describes details about how a system provides some service. The details of a use case identify the details of the sequences of interactions. The details are captured as step-by-step interactions among domain objects. Each step provides sufficient detail to identify which entities are involved, what each entity does, and the result of the step. This can be accomplished using text or sequence diagrams.

4. **Post-conditions** : The execution of a use case is intended to bring about some desired computation or state. The post-conditions identify exactly what results are expected from execution of the use case. This includes any side effects produced such as any objects created and/or destroyed.

5. **Alternate flow (Exceptions)** : Every action performed in a use case is susceptible to error. Desired data may not be located, computations might be aborted, and connectivity lost. It is necessary to identify all possible erroneous situations that can occur in the use case. In addition, it is useful to identify the specific actions to be taken in order to recover from the erroneous state. Hence, for each exception we wish to know the circumstances in which it can occur and the actions that should be taken.

6. **Special condition (requirement)** : It is necessary to identify all constraints that might apply to the use case. Such constraints can be on the values being manipulated, resources allocated to it, and resource allocations to various steps. These are usually the invariant conditions (i.e., the conditions that must always be true). Violation of these constraints can also give rise to errors and these errors should be identified as an exception.

Example : Consider a use case purchasing ticket.

Name : Purchase ticket

1. **Participating actor** : Passenger

2. **Pre-conditions**

Passenger standing in front of ticket distributor. Passenger has sufficient money to purchase ticket.

3. **Event flow**

1. Passenger selects the number of zones to be traveled.
2. Distributor displays the amount due.
3. Passenger inserts money, of at least the amount due.
4. Distributor returns change.
5. Distributor issues ticket.

7. **Post-condition**

Passenger has ticket.

Now it should be noted that not every use case have alternate flow and some usecase have more than one alternate flow. To clarify this statement considers ATMs ValidateUser use case.

Example : ValidateUser use case in ATM.

5. **Normal flow**

- The use case starts when the system prompts the Customer for a PIN number. The Customer can now enter a PIN number via the keypad.
- The Customer commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.

6. **Exceptional flow of events**

- The Customer can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the Customer's account.
- The Customer can clear a PIN number anytime before committing it and re-enter a new PIN number.
- If the Customer enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the Customer from interacting with the ATM for 60 seconds.
- Now see that validating user use case have three exceptional flows.

1.4.6 Solved Examples

Ex. 1.4.3 : Construct the use-case diagram for trading system.

Soln. :

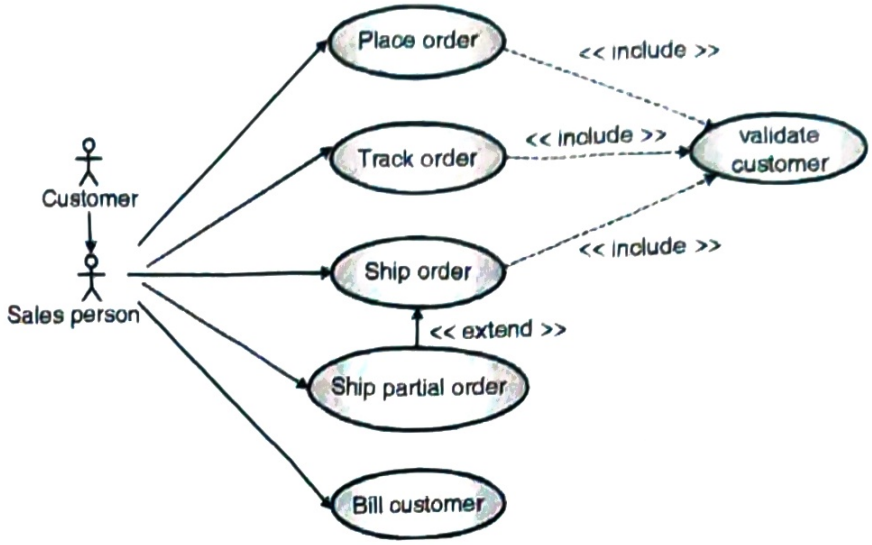

Fig. P. 1.4.3: Use case diagram for trading system

Ex. 1.4.4 : Construct the use-case diagram for following scenario; student information system. The University conducts' many courses and students can register for those courses. The registrar of the University will introduce any new course. Also the registrar maintains the curriculum and students details. The examination will be conducted for the students and the degree is issued for the students who have completed the course successfully.

Soln. :

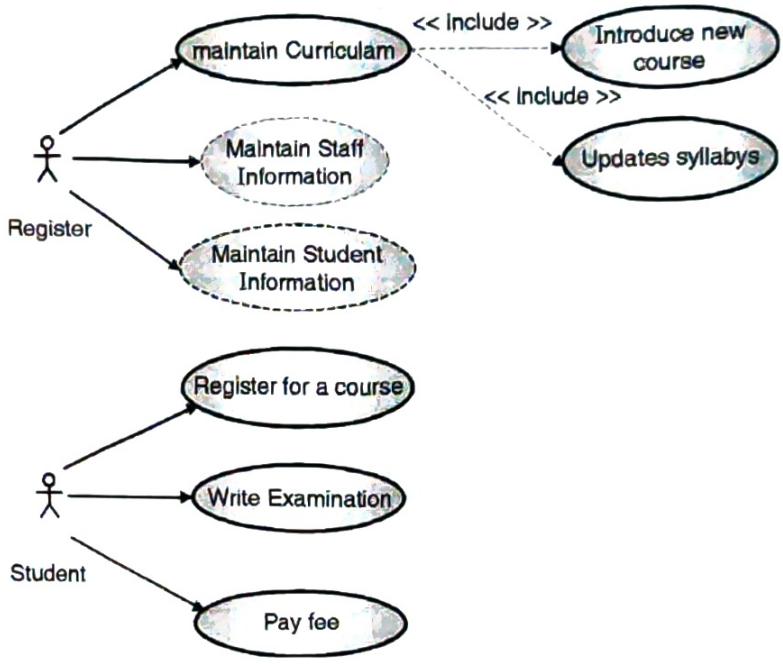

Fig. P.1.4.4 : Use case diagram

Ex. 1.4.5 : Draw a use case diagram for the system : Credit card authentication system. The scenario is : The merchant submits a credit card transaction request to the credit card payment gateway on behalf of a customer. Bank which issued customer's credit card is actor which could approve or reject the transition. If transaction is approved, funds will be transferred to merchant's bank account. **SPPU- April 18, 5 Marks**

Soln. :

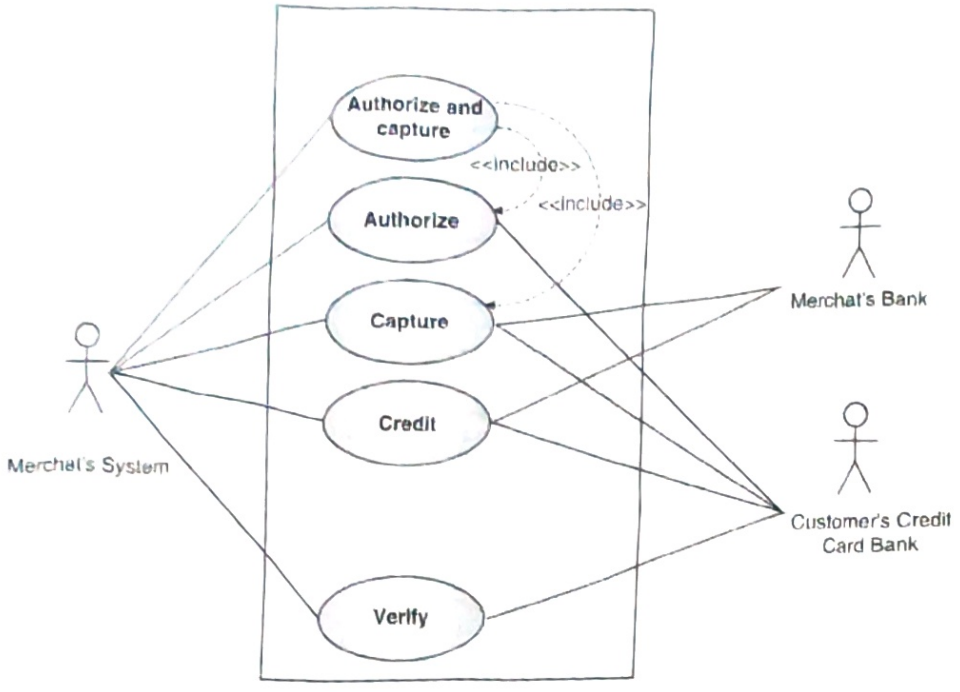

Fig. P.1.4.5 : Credit card authentication system

Ex. 1.4.6 : Draw Use Case diagram for Online Shoppe system. **SPPU – Dec. 19, 5 Marks**

Soln. :

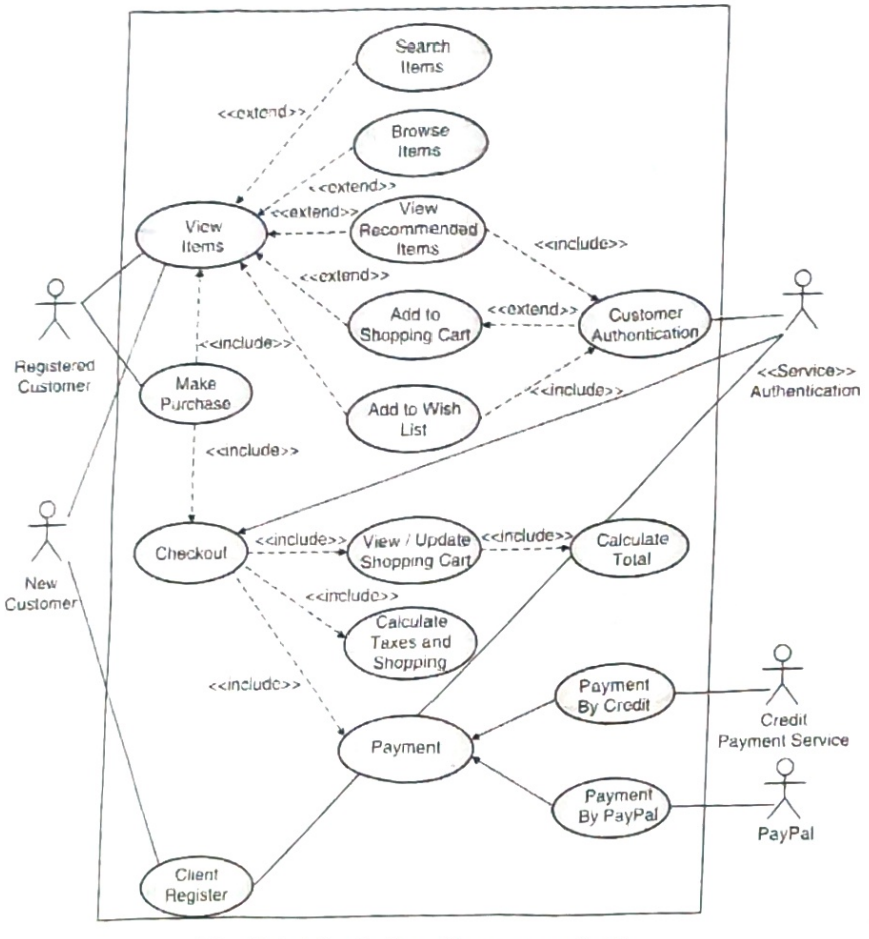

Fig. P.1.4.6 : Online Shoppe system

Ex 1.4.7 : Draw Use Case diagram for an Automated Teller Machine (ATM) system.

Soln. :

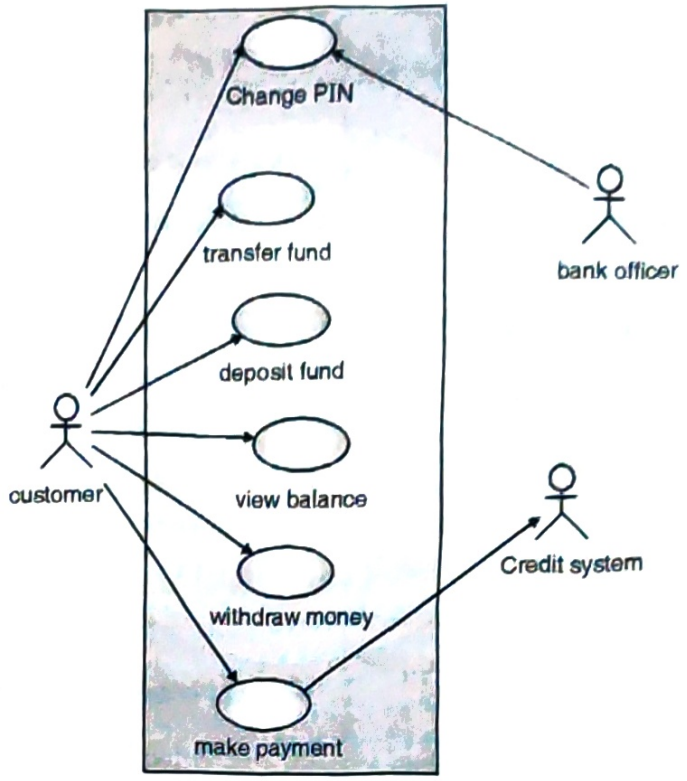

Fig. P.1.4.7 : Use Case diagram for an Automated Teller Machine (ATM)

Ex 1.4.8 : Draw Use case diagram for Airline reservation system

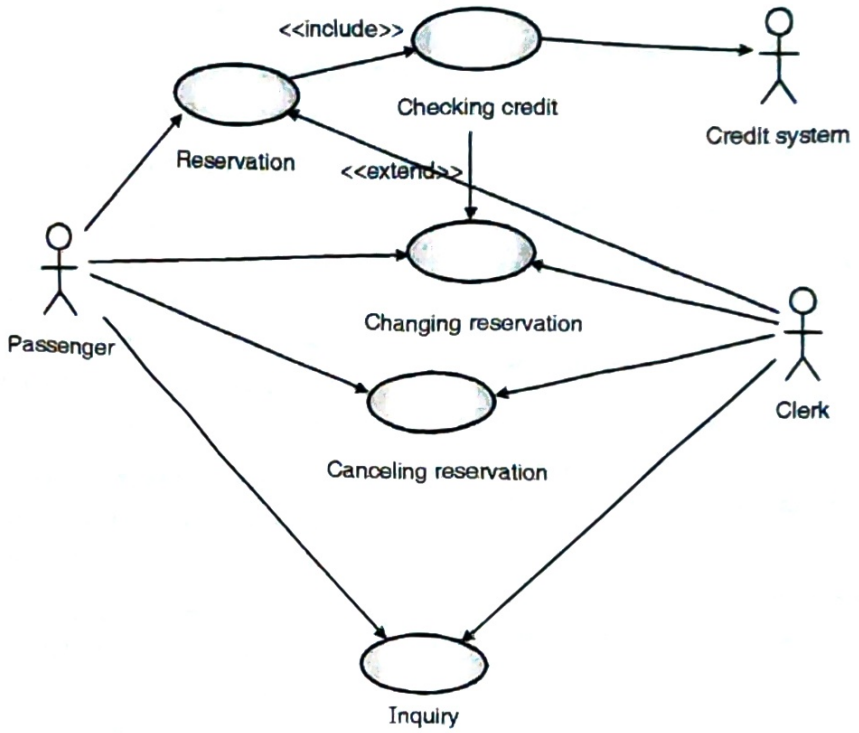

Fig. P.1.4.8: Use case diagram for Airline reservation system

Review Questions

Q.1 Explain Object oriented analysis and design methods with example

Q.2 Explain COMET and phases of COMET.

Q.3 Comment on how COMET is useful in software design and analysis.

Q.4 What is COMET ? Explain phases of COMET.

Q.5 Explain various types of non functional requirement.

Q.6 What are the characteristics of good requirement ? Explain with example.

Q.7 Explain various steps involved in SRS with suitable example.

Q.8 Give a SRS for developing software for Hospital Management system.

Q.9 Explain the organization of SRS document.

Q.10 Discuss the importance of standard template for SRS.

Q.11 How would you identify actor and use case ?

Q.12 What is a Use Case Diagram ? And when should we use it ?

Q.13 What are the different elements of use case ?

Q.14 Explain different types of relationship in use case.

Q.15 What the difference is between include and extends ? Explain with example

Q.16 What is the difference between uses and extends ?

Q.17 How can you use use-case analysis for object oriented analysis ? Constructs object model (class diagram) by using use case analysis for airline reservation system.

Q.18 Who are the actor in use case scenario ? How would you identify them? How is the actor different from the user

Q.19 What is use case model ? How is it useful in OOA ?

Q.20 What is a use case and actor ? How can you identify the actor and use case? Identify the actors and use case for library system and draw use case diagram

Q.21 What is purpose of producing use cases ?

Q.22 Draw use case diagram for Hospital management system.

Q.23 What is UML ? Explain need for UML

Q.24 What is model ? Why do we need to model' a problem ? Explain basic principles of modeling

Q.25 State UML diagrams. Explain any one in detail with example

Q.26 What is model ? Why do we need to model' a problem ?

Q.27 What are the building blocks of the UML ?

Q.28 Write a note on UML.

Q.29 Explain 4+1 View of UML.

Q.30 What is 4+1 architecture view model ? Explain with suitable diagram.

Q.31 What is 4+1 view architecture? Why it is called as 4+1 view architecture.

Q.32 What are the different relationships in UML

Q.33 What are the different system views that can be modeled using UML ? What are the different UML diagrams which can be used to capture each of the views ? Do you need to develop all the views of a system using all the modeling diagrams supported by UML? Justify your answer.

Q.34 What are the different models of a problem that can be constructed using UML? Why is it necessary to construct more than one type of model of problem ?

Q.35 What is difference between static and dynamic models in the context object oriented modeling of system ? Identify the UML diagrams which provide these two models respectively.

❑❑❑

2 Static Modeling

Syllabus

Study of classes (analysis level and design level classes). **Methods for identification of classes :** RUP (Rational Unified Process), CRC (Class, Responsibilities and Collaboration), Use of Noun Verb analysis (for identifying entity classes, controller classes and boundary classes).

Class Diagram : Relationship between classes, Generalization/Specialization Hierarchy, Composition and Aggregation Hierarchies, Associations Classes, Constraints.

Object diagram, Package diagram, Component diagram, Composite Structure diagram, Deployment Diagram.

2.1 Introduction to Static Modeling

- The static characteristic of a system is essentially the structural aspect of the system. The static characteristics define what parts the system is made up of. A static model can be viewed as "snapshot" of a system's parameters at rest or at a specific point in time. The goal of static model is to capture those concepts from the real world that are important to an application. Static model assume stability and absence of change in data over time. The classes' structure and their relationships to each other frozen in time are examples of static models.
- System analysis can be defined as a deep analysis of a part of a structure of a module which is been designed before. System design means to make any module or a part of the structure from scratch and build it completely without estimation.
- System analysis is the process which is done on the existing model or the newly created model. But in case of system design module is made from the raw data and has to build to a new structure using his own new ideas and concept
- System analysis deals with the client issues and modification that has to be done in the project whereas system design deals with the requirement of the client and need of the customer. System design incorporates collecting and reviewing problems, further solving it with the help of team or tool. System design is planning a new module by defining its components and satisfying its key requirements.

2.2 Study of Classes

- The analysis level a classes can help you to understand the requirements of your problem domain and to identify its components. In an object-oriented software project, the class diagrams that you create during the early stages of the project contain classes that often translate into actual software classes and objects when you write code. Later, you can refine your earlier analysis and conceptual models into class diagrams that show the specific parts of your system, user interfaces, logical implementations, and so on. Your class diagrams then become a snapshot that describes exactly how your system works, the relationships between system components at many levels, and how you plan to implement those components.

- The main difference between analysis level class and design level class is the analysis phase focus on designing a class diagram without concerning about the implementation detail. whereas, desgin modeling concern with the implementaion and architechture since, it the high-level static view of the system.
- Analysis classes represent an early conceptual model for 'things in the system which have responsibilities and behaviour'. Analysis classes are used to capture a 'first-draft', rough-cut of the object model of the system.
- Analysis classes handle primarily functional requirements. They model objects from the problem domain. There are three aspects of the system that are likely to change:
  - The boundary between the system and its actors,
  - The information the system uses, and
  - The control logic of the system.
- In an effort to isolate the parts of the system that will change, different types of analysis classes as boundary class, entity class and control class.

2.3 Method of Identifying Classes

- Identifying the classes is first step in creating class diagram. There are various methods (approach) to find classes. Each of them is explained in detail in following section.
- There are three approaches for identifying classes.
  - Rational Unified Process Model
  - The noun phase approach.
  - The common class patterns approach.
  - The Class Responsibilities Collaboration (CRC) approach.
- The first three approaches have been included to increase understanding of subject. CRC is used for identifying responsibilities and therefore their attributes and method.

2.3.1 Rational Unified Process Model

- Unified process model is proposed by I.Jacobson, G.Booch and J.Rumbaugh. Unified process is framework for object oriented models. In this model software is developed through a series of fixed length mini projects called iterations.
- This model is maintained and enhanced by Rational Software Corporation and thus referred to as Rational Unified Process (RUP).
- The **Rational Unified Process (RUP)** is an iterative software development process framework created by the Rational Software Corporation, a division of IBM since 2003. RUP is a specific implementation of the Unified Process.
- Rational Unified Process model contains four phases and those are :
  - Inception Phase
  - Elaboration Phase
  - Construction Phase
  - Transition Phase

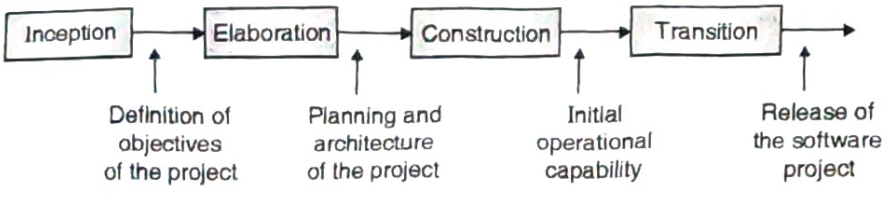

Fig. 2.3.1 : Unified Process

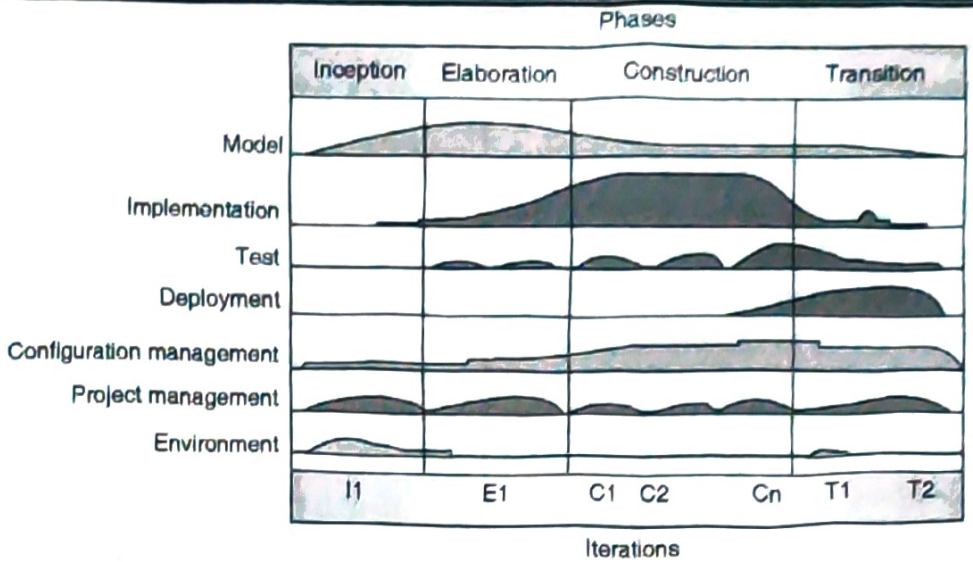

Fig. 2.3.2 : Iterations and Workflow of Unified Process

Inception Phase

- The inception phase has the following objectives:
- Gathering and analyzing the requirements.
- Planning and preparing a business case and evaluating alternatives for risk management, scheduling resources etc.
- Estimating the overall cost and schedule for the project.
- Studying the feasibility and calculating profitability of the project.
- Outcomes of Inception phase are Project plan, Initial risk assessment, Initial business case, Vision document, Initial usecase model, Initial project glossary, Business model and Prototypes.

Elaboration Phase

- The elaboration phase has the following objectives:
- Establishing architectural foundations.
- Design of use case model.
- Elaborating the process, infrastructure & development environment.
- Selecting component.
- Demonstrating that architecture supports the vision at reasonable cost & within specified time.
- Outcomes of Elaboration phase are Revised risk document, Executable architectural prototype, Usecase model, Supplementary requirements with non functional requirement, Architecture description document, Preliminary user manual and Development plan.

Construction Phase

- The construction phase has the following objectives :
- Implementing the project.
- Minimizing development cost.
- Management and optimizing resources.
- Testing the product.
- Assessing the product releases against acceptance criteria.
- Outcomes of Construction phase are Operational manuals, Test Suite, Description of the current release, Software product, User manuals, Documentation manuals and Test Outline.

Transition Phase

- The transition phase has the following objectives:
- Starting of beta testing
- Analysis of user's views.
- Training of users.
- Tuning activities including bug fixing and enhancements for performance & usability.
- Assessing the customer satisfaction.
- Outcomes of Transition Phase are Product release, Beta test reports and User feedback.

2.3.2 CRC (Class Responsibilities Collaboration)

CRC stands for Class, Responsibilities and Collaborators developed by Cunningham, Wilkerson and Beck. CRC can be used for identifying classes and their responsibilities. An object can accomplish either a certain responsibility itself, or it may require the assistance of other objects. If it requires an assistance of other objects, it must collaborate with those objects to fulfill its responsibility.

1. Class

A Class represents a collection of similar objects. Objects are things of interest in the system being modeled. They can be a person, place, thing, or any other concept important to the system at hand. The Class name appears across the top of the CRC card.

2. Responsibility

A Responsibility is anything that the class knows or does. These responsibilities are things that the class has knowledge about itself, or things the class can do with the knowledge it has. For example, a person class might have knowledge (and responsibility) for its name, address, and phone number. In another example an automobile class might have knowledge of its size, its number of doors, or it might be able to do things like stop and go. The Responsibilities of a class appear along the left side of the CRC card.

3. Collaborator

- A Collaborator is another class that is used to get information for, or perform actions for the class at hand. It often works with a particular class to complete a step (or steps) in a scenario. The Collaborators of a class appear along the right side of the CRC card.
- CRC cards are 4" x 6" index cards upon which the analyst write name of the class at top of card, its responsibility is written in one half of card and its collaborator are written in another half. One card is created for each class identified as relevant to the scenario. As team walks through the scenario, they may assign new responsibility to an existing class, group certain responsibility to form a new class, or divide responsibility of one class into more fine-grained ones, and perhaps distributes these responsibilities to different class.

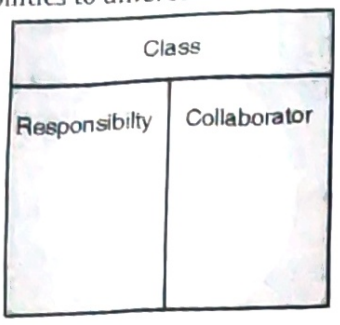

Fig. 2.3.3 : CRC card layout

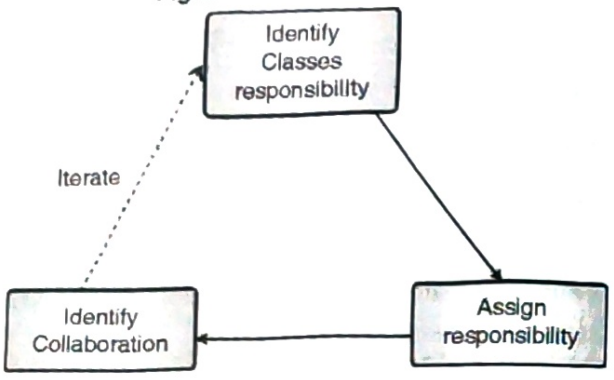

Fig. 2.3.4 : CRC card approach

- CRC cards can be spatially arranged to represent pattern of collaboration. As viewed from the dynamic semantics of the scenario, the card are arranged to show the flow of message among prototype instances of each class. As viewed from the static semantics of the scenario, the card are arranged to represent generalization/specialization hierarchies among the classes.

2.3.3 Use of Noun Verb Analysis

- Using this method, you have to read through the use cases, interviews, and requirements specification carefully, looking for noun phrases.
- Nouns in textual description are considered as classes and verbs are considered as methods of the classes.
- By changing all plurals to singular we make a list of nouns, which can then be divided into three categories i.e. Relevant Classes, fuzzy classes(not sure) and Irrelevant Classes.

- It is safe to scrap the Irrelevant Classes, which don't have any purpose. You must then select candidate classes from the other two categories. If you are unable to formulate a statement of purpose for each candidate class then simply eliminate it.

Fig. 2.3.5 : Different classes in problem domain

- Process of selecting classes from relevant classes and fuzzy
- The process of eliminating the redundant classes and refining the remaining classes is not sequential, You can move back and forth among these steps as often as you like.

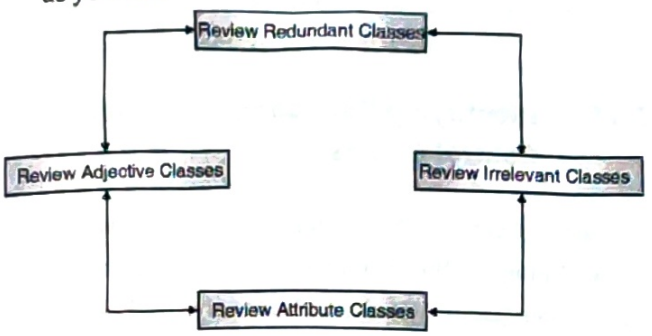

Fig. 2.3.6 : Noun phrase approach

1. Review redundant classes

Do not keep two classes that express same information. If more than one word is being used to describe same idea, select the one that is most meaningful in context of the system. For example consider a banking application in which Client, BankClient are two known then we will choose BankClient.

2. Review adjective classes

An adjective can suggest different kind of object, different use of same object. If the object represented by the noun behave differently when the adjective is applied to it then make new class. If not, simply ignore adjective i.e. If the use of the adjective signals that the behavior of the object is different, then make a new class.

Example

If Adult Membership and Youth Membership behave differently, than they should be classified as different classes.

3. Review Attribute Classes

"Tentative objects which are only used as values probably should be attributes, not classes".

Examples

- Age is value, not a class.
- Account balance is an attribute of Account class.
- The demographics of Membership are not classes but attributes of the Membership class.

4. Review irrelevant classes

- Each class must have a purpose and every class should be clearly defined and necessary. If you cannot come up with a statement of purpose, simply eliminate the candidate class.
- The followings are **guidelines** for selecting classes in your application :
  - Look for nouns and noun phrases in the problem statement.
  - Some classes are implicit or taken from general knowledge.
  - All classes must make sense in the application domain.
  - Avoid computer implementation classes; defer it to the design stage.
  - Carefully choose and define class names.

Advantages of noun phrase approach

This technique has many advantages.

- Narrative language is well understood by everyone on a project.
- An effective communication medium for both technical and non-technical project staff.
- Usually one-to-one mapping from nouns to objects or classes or interfaces.
- Using nouns requires no learning curve.

- The technique is straightforward and well defined and does not require a complete paradigm shift for the beginner.

Disadvantages of noun phrase approach

- A weakness of this approach is the imprecision of natural language. Different noun phrases may represent the same conceptual class or attribute, among other ambiguities.
- Nouns do not always result in classes, or interfaces, or objects in the problem domain.
- Many sentences in a functional specification are in the wrong form for easy identification of the objects, classes, and/or interfaces.

2.3.4 Common Class Pattern

This approach is based on the knowledge-base of the common classes that have been proposed by various researchers.

1. **Candidate Classes : Events**

- Each event classes are points in time that must be recorded and remembered. Things happen, usually to something else, at a given date and time, or as a step in an ordered sequence. For example order which is an event that must be remembered

  For examples, Landing, interrupt, request

2. **Candidate Classes : Organization**

- Organization class is a collection of people, resource; groups to which the user belongs, their capabilities have a defied mission, whose existence is largely independents of the individuals.
- For example, accounting department might be considered as a potential class.

3. **Candidate Classes :**

- People and Person (Roles and Roles Played)
- The different roles users play in interacting with the application. It can be divided into two type's .Those representing users of the system, such as an operator, or a clerk; those people who do not use the system but about whom information is kept.

  Some examples are Client, Employee, Teacher and Manager.

4. **Candidate Classes :** Places classes

- These are physical locations, such as buildings, stores, sites or offices that the system must keep information about.

5. **Candidate Classes :** Tangible Things and Devices

- Physical objects or group of objects, that are tangible, and devices with which the application interacts.
- For example, cars are example of tangible things and pressure sensors are example of device.

6. **Candidate Classes :** Concepts

- Concepts are principles or ideas not tangible but used to organize or keep track of business activities and/or communications. For example, Performance.

2.3.5 Identifying Boundary, Entity and Control Classes

- Analysis classes represent an early conceptual model for 'things in the system which have responsibilities and behaviour'. Analysis classes are used to capture a 'first-draft', rough-cut of the object model of the system.
- Analysis classes handle primarily functional requirements. They model objects from the problem domain. There are three aspects of the system that are likely to change :
  1. The boundary between the system and its actors.
  2. The information the system uses and
  3. The control logic of the system.
- In an effort to isolate the parts of the system that will change, different types of analysis classes as boundary class, entity class and control class.

2.3.5(A) Boundary Class

- A boundary class is used to model interaction between the system's surroundings and its inner workings. Such interaction involves transforming and translating events and noting changes in the system presentation.

- Boundary class insulate the system from changes in the surroundings (e.g. changes in interfaces to other systems, changes in user requirements, etc.), keeping these changes from affecting the rest of the system. Boundary classes model the parts of the system that depend on its surroundings. They make it easier to understand the system because they clarify the system's boundaries and aid design by providing a good point of departure for identifying related services
- A boundary class intermediates between the interface and something outside the system. A system may have several types of boundary classes :
  1. **User interface classes** : Classes which intermediate communication with human users of the system.
  2. **System interface classes** : Classes which intermediate communication with other systems. A boundary class which communicates with an external system is responsible for managing the dialogue with the external system; it provides the interface to that system for the system being built.
  3. **Device interface classes** : Classes which provide the interface to devices which detect external events. These boundary classes capture the responsibilities of the device or sensor.
- Each physical actor/scenario pair is examined to discover boundary classes. The boundary classes chosen in the Elaboration Phase of development are typically at a high level. For example, you may model a window but not model each of its dialogue boxes and buttons. Because boundary classes are used between Actors and the internal system's working, they insulate external forces from internal mechanisms and vice versa. Thus, changing the GUI or communication protocol should mean changing only the boundary classes, not the entity and control classes.
- One recommendation for the initial identification of boundary classes is one boundary class per actor/use-case pair. This class can be viewed as having responsibility for coordinating the interaction with the actor. This may be refined as a more detailed analysis is performed.
- When identifying user interface classes concentrate on what information is presented to the user and not on the UI details. When identifying system and device interface classes concentrate on what protocols must be defined not on how the protocols will be implemented.

UML notation for boundary class

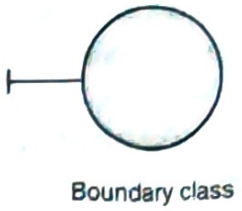

Fig. 2.3.7

Example

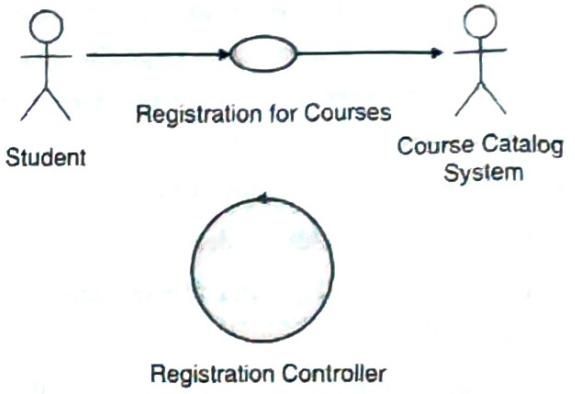

Fig. 2.3.8

In the above example :

- The Registration Form contains a Student's "schedule-in-progress". It displays a list of Course Offerings for the current semester from which the Student may select courses to be added to his/her Schedule.
- The Course Catalog System interfaces with the legacy system that provides the unabridged catalog of all courses offered by the university. This class replaces the Course Catalog abstraction originally identified in Architectural Analysis.

2.3.5 (B) Control Classes

- Control classes can contribute to understanding the system because they represent the dynamics of the system, handling the main tasks and control flows.
- When the system performs the use case, a control object is created. Control objects usually die when their corresponding use case has been performed.
- A control class is a class used to model control behaviour specific to one or more use cases. Control objects (instances of control classes) often control other objects, so their behaviour is of the coordinating type. Control classes encapsulate use-case-specific behaviour. The behaviour of a control object is closely related to the realization of a specific use case.
- Control classes effectively decouple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also decouple the use-case specific behaviour from the entity objects, making them more reusable across use cases and systems.
- Control classes provide behaviour that :
  - Is surroundings-independent.
  - Defines control logic and transactions within a use case.
  - Changes little if the internal structure or behaviour of the entity classes changes.
- One recommendation is to identify one control class per use case. Each control class is responsible for orchestrating/controlling the processing that implements the functionality described in the associated use case.

UML notation for control class

Control class

Fig. 2.3.9

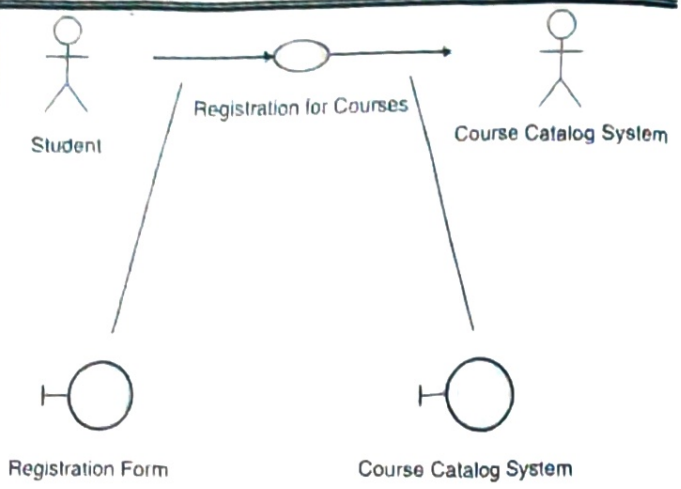

Fig. 2.3.10

In the above example, the Registration Controller class has been defined to control Registration within the system.

2.3.5 (C) Entity Classes

- Entity classes represent stores of information in the system. They are typically used to represent the key concepts that the system manages. Entity objects (instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or some real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the lifetime of the system.
- The main responsibilities of entity classes are to store and manage information in the system. Entity classes provide another point of view from which to understand the system because they show the logical data structure. Knowing the data structure can help you understand what the system is supposed to offer its users.
- Frequent sources of inspiration for entity classes are the :
  - Glossary (developed during requirements).
  - Business-domain model (developed during business modeling).
  - Use-case flow of events (developed during requirements).
  - Key abstractions (identified in Architectural Analysis).

- The first step is to examine the responsibilities documented in the flow of events for the identified use cases (i.e., what the system must do). Entity classes typically are classes that are needed by the system to accomplish some responsibility. The nouns and noun phrases used to describe the responsibility may be a good starting point.
- The initial list of nouns must be filtered because it could contain nouns that are outside the problem domain, nouns that are just language expressions, nouns that are redundant, and nouns that are descriptions of class structures. Next, go through a series of filtering steps where some candidate classes are eliminated. This is necessary due to the ambiguity of the English language. The result of the filtering exercise is a refined list of candidate entity classes.

UML notation for Entity class

For example, Register for Courses (Create Schedule) use case store information about course offered and their schedule of course hence we have two classes.

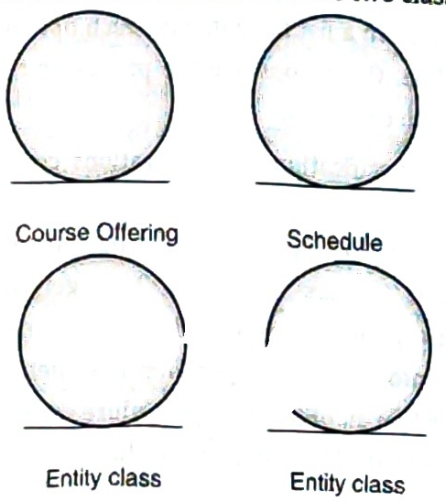

Fig. 2.3.11

2.4 Class Diagram

- A class is the descriptor for a set of objects with the same attributes and operations. It serves as a template for object creation. Its purpose is to declare a collection of attributes and operations that fully describe the structure and behaviour of objects.
- In other words, an object represents a particular instance of a class. It always originates from a class. It is structured and it behaves according to the description in a class.
- A class is an abstraction in object-oriented modeling and in object-oriented programming languages. A class encapsulates both structure and behaviour.
- We can define class from different perspectives :
  - **Conceptual perspective** : The diagrams are interpreted as describing things in the real world.
  - **Specification perspective** : The diagrams are interpreted as describing software abstractions or components with specifications and interfaces but with no commitment to a particular implementation.
  - **Implementation perspective** : The diagrams are interpreted as describing software implementations in a particular technology and language.

2.4.1 Class Structure

- A class represents an entity of a given system that provides an encapsulated implementation of certain functionality of a given entity. These are exposed by the class to other classes as methods. Apart from business functionality, a class also has properties that reflect unique features of a class. The properties of a class are called *attributes.*
- The UML representation of a class is a rectangle containing three compartments stacked vertically. The top compartment shows the class's name. The middle compartment lists the class's attributes. The bottom compartment lists the class's operations.

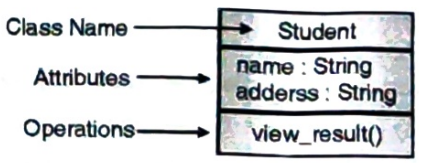

Fig. 2.4.1 : Class structure

1. Class Name

- A class name must be unique within its enclosing package. Every class must have a name that distinguishes it from other classes.
- A name is a textual string. That name alone is known as a simple name; a path name is the class name prefixed by the name of the package in which that class lives.

Fig. 2.4.2

2. Class Attributes

- An attribute is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all.
- An attribute represents some property of the thing you are modeling that is shared by all objects of that class. You might model your customers in such a way that each has a name, address, phone number, and date of birth.
- An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass.
- The attribute section of a class (the middle compartment) lists each of the class's attributes on a separate line. The attribute section is optional, but when used it contains each attribute of the class displayed in a list format.
- The line uses the following format :
- Attribute Name: attribute type
- In business class diagrams, the attribute types usually correspond to units that make sense to the likely readers of the diagram (i.e., minutes, dollars, etc.).
- However, a class diagram that will be used to generate code needs classes whose attribute types are limited to the types provided by the programming language, or types included in the model that will also be implemented in the system.
- Sometimes it is useful to show on a class diagram that a particular attribute has a default value (For example, in a banking account application a new bank account would start off with a zero balance).
- The UML specification allows for the identification of default values in the attribute list section by using the following notation:

  Name : attribute type = default value

  For e.g. Balance: Dollars = 0

3. Class Operations List

- An operation is the implementation of a service that can be requested from any object of the class to affect behaviour.
- In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class.
- The class's operations are documented in the third (lowest) compartment of the class diagram's rectangle, which again is optional.
- Like the attributes, the operations of a class are displayed in a list format, with each operation on its own line. Operations are the processes that a class knows to carry out.
- At the specification level operations correspond to public methods on a type. Operations that simply manipulate attributes must not be shown in a class diagram (getters and setters). A getting method returns a value from a field. A setting method puts value into a field. An operation is something that is invoked on an object (the procedure call), whereas a method is the body of the procedure.
- The syntax for operations is:
- Visibility name (parameter-list) : return-type-expression {property-string}

  Where,

  - Visibility is + (public), #(protected) or - (private).
  - Name is a string.

- Parameter-list contains a comma-separated parameters whose syntax is similar to that for attributes: direction name : type = default value.
- Return-type-expression is a comma-separated list of return types.
- Property-string indicates property values that apply to the given operation.

- The structure of a system is defined by a Class Diagram by showing its attributes, relationships among objects, and so on. It is the backbone of object-oriented modeling, and could also be used for Data modeling. Class Diagrams helps in making pre plans which ease the programming process.
- It is a design plan based on which a system is built. It is easy to understand without much technical knowledge required.
- Class Diagram provides a static view of the application, and its mapping ability with object-oriented language makes it ready to be used in construction.
- The foundation of software products are the class diagrams which are an essential part of any project.
- A sense of orientation is given by the class diagrams. The structure of the system is analyzed in detail by the class diagram, and also the synergy among different elements is overviewed by them along with their properties. It is fast, and easy to read, and could be created easily if the right software is in place. Any system that needs to be created, the class diagrams forms the foundation for that.
- Below is the purpose of a Class diagram.
  1. The static view of an application is designed and analyzed.
  2. A system's responsibilities are described by it.
  3. The components and the deployment diagram's base is the class diagram.
  4. The forward and reverse engineering is influenced by the Class Diagram.

2.4.2 Identifying Attributes

- Attributes are properties of individual objects, such as name, velocity or colour.
- Attributes usually correspond to nouns followed by possessive phrases such as" the colour of the car' or the position of cursor". Adjective often represent specific enumerated attribute values such Red, On, Expired.
- Attributes can be identified by looking at the description of the system and searching for information that must be maintained about instances of each class.
- Several of nouns you have originally identified, but rejected as classes, may now become attributes. For example, your flow of events may read "The user enters the employee's name, address, social security number, and phone number," letting you know that the Employee class has attributes called Name, Address, SSN, and Phone.
- In our airline example, the user is looking for a flight and enters a departure and destination city as well as a departure date. From this, we know that a Flight class would have attributes called Departure City, Destination City, and Departure Date.

Guideline for identifying attributes

- Examine possessive phrases
- Represent stored state as an attribute of the entity object
- Do not represent an attribute as an object, use an association instead
- Do not waste time describing fine details before the object structure is stable.

2.4.3 Identifying Operation

- An operation is a behaviour associated with a class. Finding operations is fairly straightforward. As you created your Sequence and Collaboration diagrams, you did most of the work necessary to find operations.
- There are four different types of operations to consider: implementer, manager, access, and helper.

1. Implementor operations

- Implementor operations implement some business functionality. They are found by examining Interaction diagrams. The Interaction diagrams focus on business functionality, and each message on the diagram will most likely be mapped to an implementor operation.
- Each implementor operation should be able to be traced back to a requirement. This is achieved through the various pieces of the model. Each operation comes from a message on an Interaction diagram, which comes from the details in the flow of events, which comes from the use case, which comes from the requirements. This ability to trace can help you ensure that each requirement is implemented in the code and that each piece of code can be traced back to a requirement.

2. Manager operations

Manager operations manage the creation and destruction of objects. For example, the constructor and destructor operations of a class fall into this category.

3. Access operations

- Attributes are typically private or protected. However, other classes may need to view or change the attributes of a particular class. This can be accomplished through access operations.
- For example, if we have an attribute called Salary in an Employee class, we wouldn't want all of the other classes to be able to go in and change the Salary. Instead, we add two access operations to the Employee class : GetSalary and SetSalary.
- This approach keeps the attributes safely encapsulated inside a class and protected from other classes, but still allows controlled access to the attributes. The industry standard has been to create a Get and Set operation for each attribute in a class.

4. Helper operations

- Helper operations are those operations that one class needs to carry out its responsibilities but that other classes don't need to know about. These are the private and protected operations of a class.
- Like implementor operations, helper operations are found by examining the Sequence and Collaboration diagrams. Frequently, helper operations appear as reflexive messages on a Sequence or Collaboration diagram.
- To identify operations, you can perform the following series of steps :
  - Examine Sequence and Collaboration diagrams. Most messages will become implementor operations. Reflexive messages may become helper operations.
  - Consider manager operations. You may want to add constructors and destructors.
  - Consider access operations. Create a Get and Set operation for any attribute that will need to be viewed or changed by another class.
  - As you identify operations and examine your classes, keep a few things in mind :
  - Be suspicious of any classes with only one or two operations. It may be perfectly legitimate, but it may also indicate that the class should be combined with another.
  - Be very suspicious of classes with no operations. A class typically encapsulates both behaviour and information. A class with no behaviour might be better modelled as an attribute or two.
- Be wary of classes with too many operations. A class should have a manageable set of responsibilities. If you have a class with too many operations, it may be difficult to maintain. Dividing it instead into two smaller classes may ease maintenance.

2.4.4 Relationship between Classes

- A class may be involved in one or more relationships with other classes. Classes are interrelated to each other in specific ways. In particular, relationships in class diagrams include different types of logical connections. The following are such types of logical connections that are possible in UML:
  - Association
  - Directed Association
  - Reflexive Association
  - Multiplicity
  - Aggregation
  - Composition
  - Inheritance/Generalization

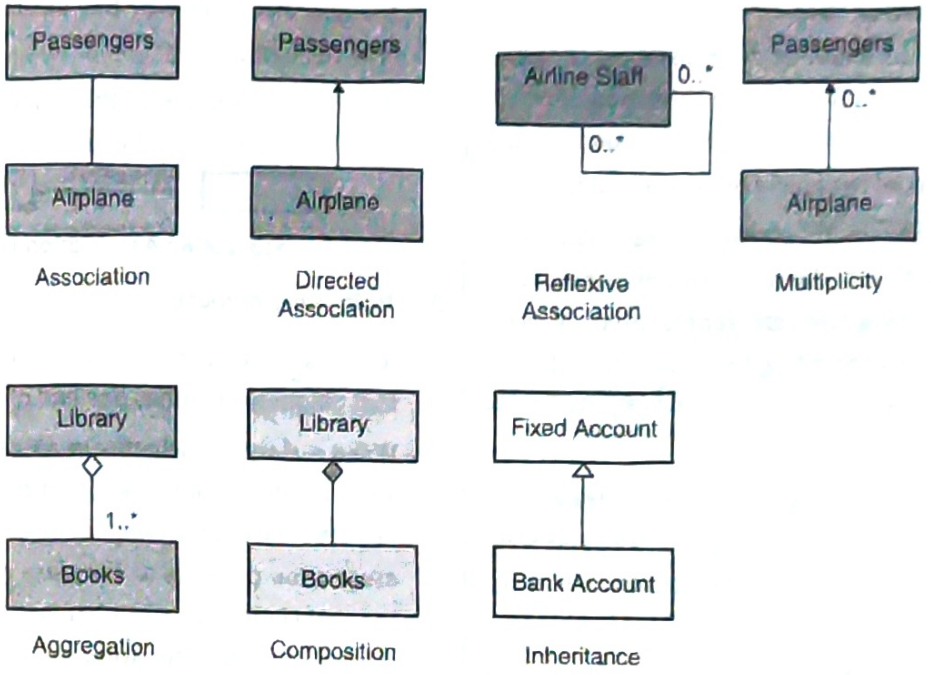

Fig. 2.4.3 : Relationships in UML class diagrams

2.4.4(A) Association

University Questions

Q. What are the different degrees of Multiplicity in association relationships ? **SPPU : April 19, 4 Marks**

Q. Write short note on : Association. **SPPU : April 19, 4 Marks**

- Association is reference from one class to another. Basically it is dependency between two or more classes. Some associations are implicit or taken from general knowledge.

- **For example** : A "student studies in a college" association can be shown as :

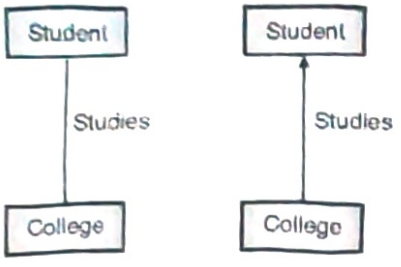

Fig. 2.4.4 : Association

- Association between classes is bi-directional by default. You can define the flow of the association by using a directed association. The arrowhead identifies the container-contained relationship.
- An **Association** can be defined as : The semantic relationship between two or more classifiers that specifies connections among their instances. In other words, an association is a structural relationship that specifies that objects (instances of classes) of one thing are connected to objects of another thing.
- The way that you show these structural relationships between classes is through the use of associations. Associations are represented on class diagrams by a line connecting the associating classes. Data may flow in either direction or both directions across a link.
- Most associations are simple. That is, they exist between exactly two classes. They are drawn as solid paths connecting pairs of class symbols. Ternary relationships are also possible.
- **Reflexive Associations :** An association can connect a class to itself. Sometimes, a class has an association to itself. This does not always mean that an instance of that class has an association to itself. More often, it means that one instance of the class has associations to other instances of the same class. For e.g. Person can have up to 2 parents recorded.

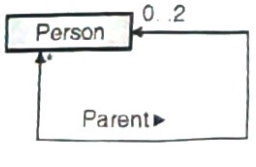

Fig. 2.4.5 : Reflexive Associations

- Associations in a class diagram can be further defined by

1. Association Names
2. Role Names
3. Multiplicity

1. Association Names

- An association name is a label that clarifies the meaning of the association.
- An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name.
- We can give a name to the association by putting a label in the middle of the line.

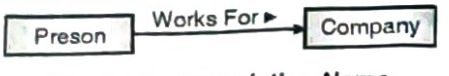

Fig. 2.4.6 : Association Name

2. Role (Role names)

- A role is one end of an association. A role is the name that uniquely identifies one end of association.
- When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the near end of the association presents to the class at the other end of the association. You can explicitly name the role a class plays in an association.
- In Fig. 2.4.7, a Person playing the role of employee is associated with a Company playing the role of employer.
- Thus, a role name is a noun that denotes the purpose or capacity wherein one class associates with another. Role names are written next to association line near the class that plays the role.

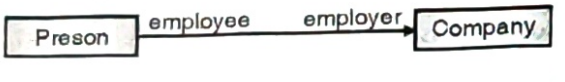

Fig. 2.4.7 : Role names

3. Multiplicity

- Multiplicity can be defined as the number of instances one class relates to one instance of another class.
- It is often important to define a limit on the number of participating objects in a particular association. For each role, you can specify the multiplicity of its class and how many objects of the class can be associated with one object of the other class.
- Multiplicity is indicated by a text expression on the role. The expression is a comma-separated list of integer ranges.
- Multiplicity must be defined on both ends of the association. For each association, there are two multiplicity decisions to make, one for each end of the association.

  For example as shown in Fig. 2.4.8, for each instance of Professor, many Course Offerings may be taught. For each instance of Course Offering, there may be either one or zero Professor as the instructor.

Fig. 2.4.8 : Multiplicity indicator

- The multiplicity (or cardinality) is shown as a range of integers *n1, n2*. The integer *n1* defines the minimum number of connected objects, and *n2* defines the maximum number. The maximum number can be an asterisk *, if we do not know the precise maximum integer value, or if it is indefinite.

Multiplicity indicators

Table 2.4.1: Multiplicity Indicators

Unspecified	
Exactly One	1
Zero or More	0..*
Zero or More	*
One or More	1..*
Zero or one	0 ..1
Specified range	3..8
Multiple ,Disjoint ranges	2,5..9

Navigation

- It's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional. However, there are some circumstances in which you'll want to limit navigation to just one direction.
- For example, as Fig. 2.4.9 shows, when modeling the services of an operating system, you'll find an association between User and Password objects. Given a User, you'll want to be able to find the corresponding Password objects; but given a Password, you don't want to be able to identify the corresponding User.
- You can explicitly represent the direction of navigation by adorning an association with an arrowhead pointing to the direction of traversal. If we put an arrow on an association, it means there is a way to navigate from one class to another.

Fig. 2.4.9 : Navigation

2.4.4(B) Generalization/Specialization Hierarchy

University Question

Q. Explain generalization with reference to class diagram. **SPPU : May 17, 2 Marks**

- There are two ways to identify generalization viz. bottom up and top down. The bottom up approach groups together similar classes creating a new superclass, whereas the top down approach divides up a complex class, creating new subclasses.
- To use the bottom up approach, you look for classes which have feature common in both. In general, if you find two or more classes that have similar attributes, associations or operations then you should consider creating a common super class. For. e.g. four legged animal and two legged animals are similar except their number of legs ,can be generalized by Animal.

- Top-down specialization is often apparent from the application domain. Look for noun phrases composed of various adjectives at the same class name. For example an account class refined into Checking account and saving account. When the same association name appears more than once with substantially the same meaning try to generalize the associated classes.

Generalization

- It is a relationship among classes where one class shares the structure and/or behaviour of one or more classes. Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse.
- In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. But the child has attributes and operations in addition to those found in its parents.
- Generalization is transitive. You can always test your generalization by applying the "is a kind of" rule. You should always be able to say that your specialized class "is a kind of" the parent class. It is called as "is a" relationship, because the child class is a type of the parent class.
- The notation for generalization is a triangle connecting a super class to its subclasses. The super class is connected by a line to the apex of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of triangle.

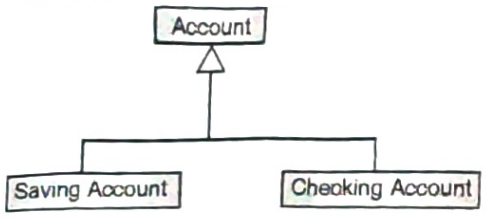

Fig. 2.4.10 : Generalization

- The terms "generalization" and "inheritance" are generally interchangeable, but if you need to distinguish, generalization is the name of the relationship. Inheritance is the mechanism that the generalization relationship represents/models. Inheritance is a mechanism by which more-specific elements incorporate the structure and behaviour of more-general elements.

2.4.4(C) Aggregate /Composite Hierarchy

University Question

Q. Explain aggregation with reference to class diagram.

SPPU : May 17, 2 Marks

- Aggregation is a stronger form of a relationship expressing ownership semantics. It is also referred to as "part of" relationship.
- An **Aggregation** can be defined as : A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
- Aggregation is used to model a whole-part relationship between model elements. There are many examples of whole-part relationships : a Library contains Books, Departments are made up of Employees, a Computer is composed of a number of Devices.
- A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.
- Destroying the aggregate does not necessarily destroy the parts. Aggregation represents the situation where a class consists of several component classes.
- Situations where aggregation may be appropriate include :
  - An object is physically composed of other objects (for example, a car being physically composed of an engine and four wheels).
  - An object is a logical collection of other objects (for example, a family is a collection of parents and children).
  - An object physically contains other objects (for example, an airplane physically contains a pilot).

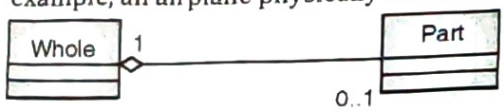

Fig. 2.4.11

- Fig. 2.4.12 shows two Examples Company has many departments and department is part of company. Similarly team consists of many players and player is part of team .

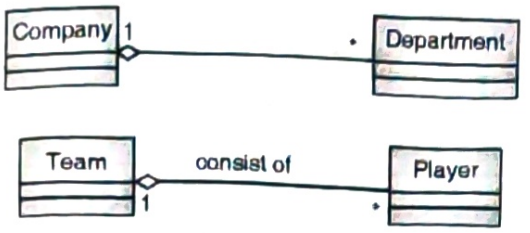

Fig. 2.4.12 : Aggregation

- Two major properties of a-part-of relationship are

1. Transitivity
2. Anti-symmetry

1. Transitivity

- If A is part of B and B is part of C, then A is part of C.
- In our example, this means that *Hand A* contains *Finger B*, and *Finger B* contains *Nail C*, then *Nail C* is part of *Hand A*.

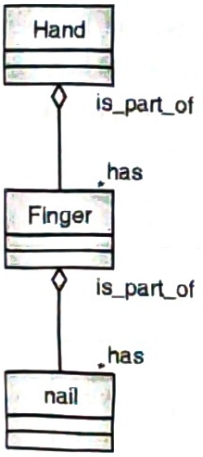

Fig. 2.4.13

2. Anti-symmetry

- If A is part of B, then B is not part of A.
- In our example, this means that Hand *A* contains Finger *B*, then *Finger B* cannot contain Hand *A*. Take another example, an engine is part of a car, but a car is not part of an engine.

2.4.4(D) Composite Relationships

University Question

Q. Explain composition with reference to class diagram.

SPPU : May 17, 2 Marks

- Composition is a stronger variety of aggregation where the part may belong only to one whole. Here the parts are usually expected to live and with the whole.
- Thus, composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole.
- The UML notation for composition is a solid diamond at the whole end. This implies that if a superset object is deleted (destroyed), then its subset objects must also be deleted. If you cut off a hand, the finger cannot longer exist and function as a finger.
- In a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts.
- For example, when you create a Frame in a windowing system, you must attach it to an enclosing Window. Similarly, when you destroy the Window, the Window object must in turn destroy its Frame parts.

Fig. 2.4.14 : Composition

- A CD contains one or more tracks. A Track is part of a CD – won't exist on its own. Tracks can belong to only one CD. If you delete the CD, the Tracks go away too.

Fig. 2.4.15: Composition

Difference between aggregation and composition

University Question

Q. Explain with an example the difference between aggregation and composition.

SPPU : May 18, Dec. 19, 5 Marks

- Top-down specialization is often apparent from the application domain. Look for noun phrases composed of various adjectives at the same class name. For example an account class refined into Checking account and saving account. When the same association name appears more than once with substantially the same meaning try to generalize the associated classes.

Generalization

- It is a relationship among classes where one class shares the structure and/or behaviour of one or more classes. Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse.
- In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. But the child has attributes and operations in addition to those found in its parents.
- Generalization is transitive. You can always test your generalization by applying the "is a kind of" rule. You should always be able to say that your specialized class "is a kind of" the parent class. It is called as "is a" relationship, because the child class is a type of the parent class.
- The notation for generalization is a triangle connecting a super class to its subclasses. The super class is connected by a line to the apex of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of triangle.

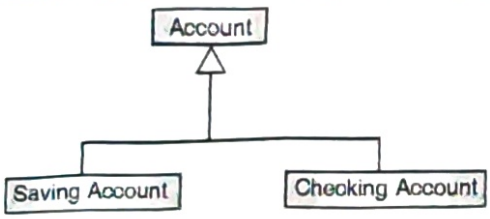

Fig. 2.4.10 : Generalization

- The terms "generalization" and "inheritance" are generally interchangeable, but if you need to distinguish, generalization is the name of the relationship. Inheritance is the mechanism that the generalization relationship represents/models. Inheritance is a mechanism by which more-specific elements incorporate the structure and behaviour of more-general elements.

2.4.4(C) Aggregate /Composite Hierarchy

University Question

Q. Explain aggregation with reference to class diagram.

SPPU : May 17, 2 Marks

- Aggregation is a stronger form of a relationship expressing ownership semantics. It is also referred to as "part of" relationship.
- An **Aggregation** can be defined as : A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.
- Aggregation is used to model a whole-part relationship between model elements. There are many examples of whole-part relationships : a Library contains Books, Departments are made up of Employees, a Computer is composed of a number of Devices.
- A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.
- Destroying the aggregate does not necessarily destroy the parts. Aggregation represents the situation where a class consists of several component classes.
- Situations where aggregation may be appropriate include :
  - An object is physically composed of other objects (for example, a car being physically composed of an engine and four wheels).
  - An object is a logical collection of other objects (for example, a family is a collection of parents and children).
  - An object physically contains other objects (for example, an airplane physically contains a pilot).

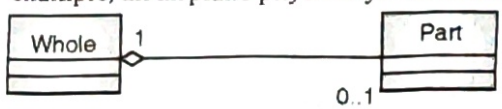

Fig. 2.4.11

- Fig. 2.4.12 shows two Examples Company has many departments and department is part of company. Similarly team consists of many players and player is part of team .

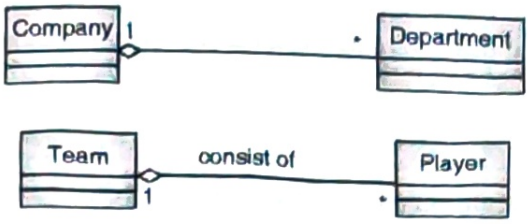

Fig. 2.4.12 : Aggregation

- Two major properties of a-part-of relationship are

1. Transitivity
2. Anti-symmetry

1. Transitivity

- If A is part of B and B is part of C, then A is part of C.
- In our example, this means that *Hand A* contains *Finger B*, and *Finger B* contains *Nail C*, then *Nail C* is part of *Hand A*.

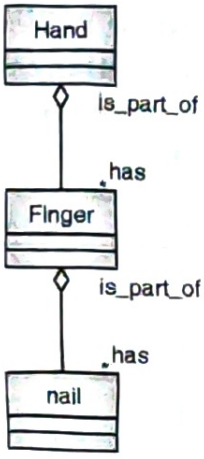

Fig. 2.4.13

2. Anti-symmetry

- If A is part of B, then B is not part of A.
- In our example, this means that Hand *A* contains Finger *B*, then *Finger B* cannot contain Hand *A*. Take another example, an engine is part of a car, but a car is not part of an engine.

2.4.4(D) Composite Relationships

University Question

Q. Explain composition with reference to class diagram.

SPPU : May 17, 2 Marks

- Composition is a stronger variety of aggregation where the part may belong only to one whole. Here the parts are usually expected to live and with the whole.
- Thus, composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole.
- The UML notation for composition is a solid diamond at the whole end. This implies that if a superset object is deleted (destroyed), then its subset objects must also be deleted. If you cut off a hand, the finger cannot longer exist and function as a finger.
- In a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts.
- For example, when you create a Frame in a windowing system, you must attach it to an enclosing Window. Similarly, when you destroy the Window, the Window object must in turn destroy its Frame parts.

Fig. 2.4.14 : Composition

- A CD contains one or more tracks. A Track is part of a CD – won't exist on its own. Tracks can belong to only one CD. If you delete the CD, the Tracks go away too.

Fig. 2.4.15: Composition

Difference between aggregation and composition

University Question

Q. Explain with an example the difference between aggregation and composition.

SPPU : May 18, Dec. 19, 5 Marks

Table 2.4.2

Aggregation	Composition
Aggregation relationship is denoted using a straight line with an empty arrowhead at one end.	Composition relationship is denoted using a straight line with a filled arrowhead at any one of the ends.
Aggregation is a part of an association relationship.	The composition is a part of an association relationship.
Aggregation is considered as a weak type of association.	The composition is considered as a strong type of association.
In an aggregation relationship, objects that are associated with each other can remain in the scope of a system without each other.	In a composition relationship, objects that are associated with each other cannot remain in the scope without each other.
Linked objects are not dependent upon the other object.	Objects are highly dependent upon each other.
In aggregation, deleting one element does not affect another associated element.	In composition, deleting one element affects another associated element.
Example : A car needs a wheel, but it doesn't always require the same wheel. A car can function adequately with another wheel as well.	Example : A file is placed inside the folder. If one deletes the folder, then the file associated with that given folder is also deleted.

2.4.5 Association Classes

A class diagram is rarely correct after a single pass. The entire software development process is one of continual iteration; different parts of model are often at different stages of completion. If a deficiency is found, go back to an earlier stage if necessary to correct it. Some refinements can only come after the dynamic and functional models are completed.

Association classes

- In an association between two classes, the association itself might have properties. For example, in an employer/employee relationship between a Company and a Person, there is a Job that represents the properties of that relationship that apply to exactly one pairing of the Person and Company.
- It wouldn't be appropriate to model this situation with a Company to Job association together with a Job to Person association. That wouldn't tie a specific instance of the Job to the specific pairing of Company and Person.
- In the UML, you'd model this as an association class, which is a modeling element that has both association and class properties.
- An association class can be seen as an association that also has class properties, or as a class that also has association properties. You render an association class as a class symbol attached by a dashed line to an association.

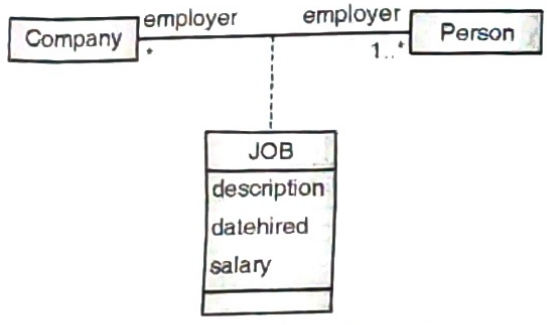

Fig. 2.4.16 : Association class

2.4.6 Constraints

- Constraints are functional relationship between entities of an object model. The term entity includes objects, classes, attributes, links, and associations. A constraint restricts the values that entities can assume.

- A constraint is packageable element used to express a condition, restriction or assertion related to some of the semantics of an element (that owns the constraint).
- Some kinds of constraints are predefined in UML, and constraint element allows to introduce user defined conditions and restrictions.
- Constraint is a condition (a Boolean expression) which must evaluate to a Boolean value - true or false. Constraint must be satisfied (i.e. evaluated to true) by a correct design of the system.
- In general there are many possible kinds of owners for a Constraint. Owning element must have access to the constrained elements to verify constraint. The owner of the constraint will determine when the constraint is evaluated. For example, Operation can have pre-condition and/or post-condition constraints.
- **OCL** is a constraint language predefined in UML but UML specification does not restrict languages which could be used to describe constraints. If some tool is used to draw UML diagrams, it could be any constraint language supported by that tool.

A Constraints on Objects

- Example of Constraints on Objects are :
- No employee's salary can exceed the salary of the employee boss.
- No window can have an aspect ratio (length/width) of less than 0.7 or greater than 1.4.

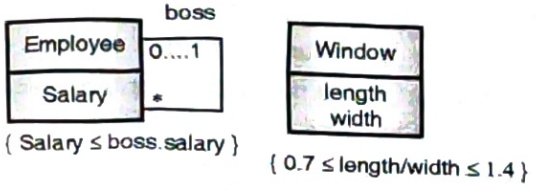

Fig. 2.4.17

B Constraints on Links

- Constraints between relationships are shown as dashed lines between pairs of associations and labeled by the constraint string in curly braces. Such a line is called a constraint.
- If it is annotated {xor} or simply {or}, it is an "exclusive or". (Each occurrence of the base entity must be (or may be) related to either an occurrence of one entity, or to an occurrence of the other, but not both.) . For example Account owner is either Person or Corporation; {xor} is predefined UML constraint.

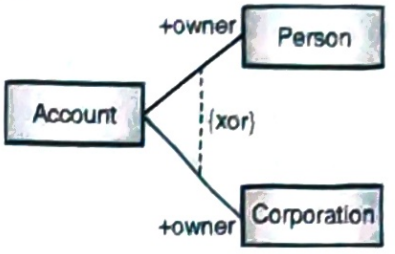

Fig. 2.4.18 : Constraints on association links

- If the constraint is shown as a dashed line between two elements, then an arrowhead may be placed on one end. The direction of the arrow is relevant information within the constraint.
- The element at the tail of the arrow is mapped to the first position and the element at the head of the arrow is mapped to the second position in the constrained Elements collection. For example table has many columns. The constraints, primary key columns are a subset of overall column is represented by a dotted arrow.

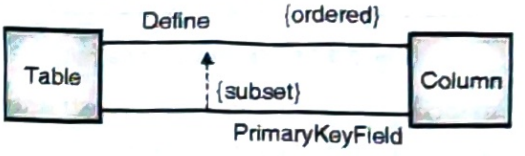

Fig. 2.4.19 :Subset constraint between associations

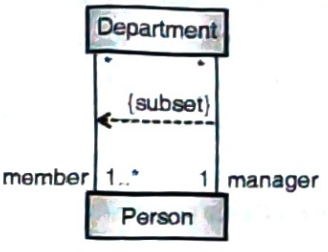

Fig. 2.4.20 :Subset constraint between associations

- This diagram shows that each Person may be a member of zero or more Departments and that each Department must have at least one Person as a member. This diagram goes on to indicate that each Department must have exactly one Person as a manager and every Person may be the manager of zero or more Departments.

- All of these semantics can be expressed using simple UML. However, to assert that a manager must also be a member of the department is something that cuts across multiple associations and cannot be expressed using simple UML.
- To state this invariant, you have to write a constraint that shows the manager as a subset of the members of the Department, connecting the two associations and the constraint by a dependency from the subset to the superset.
- Multiplicity constraints an association. It restricts the number of objects related to a given object. The notation "{ordered}" indicates that the elements of the "many" end of an association have an explicit order that must be preserved.
- For example, Customer has none to many purchases. Purchases are ordered and each one is unique (by default). If multiplicity element is multivalued and specified as **ordered**, then the collection of values in an instantiation of this element is sequentially ordered. By default, collections are not ordered.

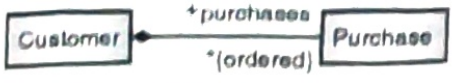

Fig. 2.4.21 : Ordered constraint

- An association class implies constraints. An association class is a class in every right; for example it can have attributes and operations, participate in associations, and participate in generalization. But an association class has a constraint that an ordinary class does not; it derives identity from instances of the related classes.

C Constraints on Generalization Sets

- Each instance of an abstract superclass corresponds to exactly one subclass instance. Each instance of a concrete superclass corresponds to at most one subclass instance.
- Class models capture many constraints through their very structure. For example, the semantics of generalization imply certain structural constraints with single inheritance the subclasses are mutually exclusive.
- The UML defines the following keywords for generalization sets.
  - Complete: specifies that all children in the generalization have been specified in the model and that no additional children are permitted.
  - Incomplete: specifies that not all children in the generalization have been specified in the model and that additional children are permitted.
  - Disjoint: specifies that objects of the parent may have no more than one of the children as a type.
  - Overlapping: specifies that objects of the parent may have more than one of the children as a type.

D Use of Constraints

- Constraints provide one criterion for measuring the quality of a class model a "good" class model captures many constraints through its structure. It often requires several iterations to get the structure of a model right from the perspective of constraints. Also, in practice, you cannot enforce every constraint with a model's structure, but you should try to enforce the important ones.

2.4.7 Solved Examples

Ex. 2.4.1 : Draw a class diagram for Hospital Management System.

Case study :

The hospital wishes to maintain a database to assist with the administration of its wards and operating theatres, and to maintain information relating to its patients, surgeons and nurse. Most patients are assigned to a ward on admittance and each ward may contain many patients. However senior surgeon at the hospital may have private patients who are assigned to private rooms a nurse may or may not be assigned to a ward and he/she cannot assign to more than one ward.

A ward may have many nurses assigned to it. Each ward is dedicated to a particular type of patient (e.g. generic, pediatric, maternity, etc).A nurse may or may not be assigned to a theatre and he/she cannot be assigned to more than one theatre.

A theatre may have many nurses assigned to it. A patient may have a number of operations. The information to be recorded about an operation includes the type of operations, the patients, the surgeons involves, date, time and location. Only one surgeon may perform an operation, any other surgeon present being considered as assisting at operation. Each consultant has a specialization.

Soln. :

Step 1 :For finding classes we use Noun Phrase Approach

Classes Identified Using Noun Phrase Approach

Patient, ward, nurse, theatre, operation, surgeon, private room, consultant, administration, hospital, private patient, general patient, information, specialization, database, operation theatre.

Redundant Classes

- **Hospital :** vague class
- **Database :** vague class,
- **Theatre :** redundant class
- **Information :** vague class
- **Administration :** vague class
- **Specialization :** attribute of surgeon

Final classes

Patient, ward, nurse, operation, surgeon, private room, consultant, private patient, general patient, operation theatre.

Step 2 : Classes with role

Patient, Ward, Nurse, Operation, Doctor, PvtRoom, Consultant, Pvt Patient, Gen Patient, Opr theatre and Hospital.

Hospital : Stores information such as no of staff, address etc.

Gen Patient : Patient stores information about patients admitted in the hospital.

Ward : Patients are admitted to the various rooms in the hospital called wards.

Nurse : Stores all personal information about the nurses in the hospital.

Opr Theatre : It stores the information related to operation such as operation theatre no.

Doctor : Stores information such as name, address and specialization.

Consultant : Stores information about consultants.

Pvt Patient : Consultants diagnose the illness of private patients. They are admitted to private rooms.

Operation : Operations are performed by surgeon in operation theatre. It stores information about operation type, patient name and surgeons in operation theatre. It stores information about operation type, patient name and surgeon.

Step 3 : Associations between Objects/classes

- A patient is assigned to a ward.
- A nurse is assigned to 0 or 1 ward.
- A nurse is assigned to 0 or 1 operation theatre.
- An operation is performed by a surgeon.
- A surgeon is assisted by one or more assistant surgeons.
- A private patient is assigned to a private room.
- A patient undergoes 0 or 1 operations.
- An operation is performed in an operation theatre.

Step 4 : Attributes and Operations of classes

Doctor : Attributes : Name, ID, Qualification and Specialist.

Nurse : Attributes : Name, Qualification and Experience.

Ward : Attributes : Ward no, Type and facilities.

Gen Patient : Attributes : Name, address, ward of patient and disease.

Operations : time()

Operation : Attributes : Type, patient and doctor.

Opr Theatre : Attributes : theatre no, facilities.

Pvt Patient : Attributes: Name, address, disease, ward of patient.

Pvt Room : Attributes: room no, facilities.

Hospital :

Attributes : address, no of staff.

Operations : update_info().

Step 5 : Simplify classes using Inheritance

Patients are of two types: general and private. Therefore we can create superclass patient.

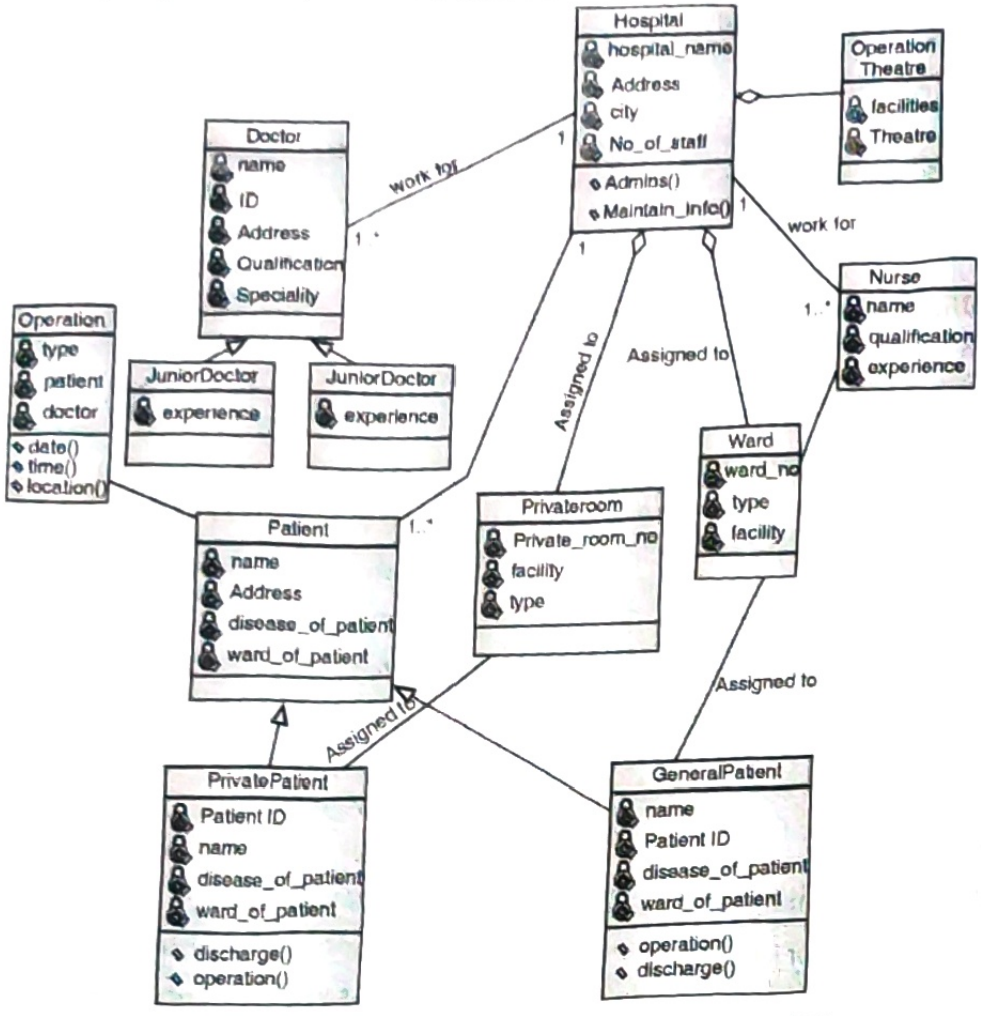

Fig. P. 2.4.1 : Class diagram for hospital management system

Attributes and methods

- **userId:** The user's userid to login.
- **password:** The user's password to login.
- **loginStatus:** The user's login status.
- **verifyLogin():** Responsible for authenticating and authorizing the user to user secure sites.

2. Keyword

This class represents a collection of all the keywords used for the searching.

Keyword
– keyword : string
+ find key word (in keyword : stating) : keyword

Attributes and methods

- **keyword :** The keyword used for searching.
- **findkeyword() :** Responsible for matching the keyword from the database.

3. Shipping info

This class represents the shipping information of each order.

Shipping info
– Shipping ID : int – order ID : int – shipping type : staring – shipping cost : int – Shipping region ID : int
+ update shipping info ()

Attributes and methods

- **shipping Id :** The shipping id of the order.
- **Order Id :** The order id.
- **Shipping Type :** The type of shipping for the order.
- **Shipping Cost :** The cost of shipping.
- **Shipping Region Id :** The id of the shipping region for the order.
- **Update Shipping Info() :** Responsible for updating the shipping information of the order.

4. Search Facade

This class has methods to get the search results according to the options chosen by the user.

Search Facade
+ get by artist (in artist : string) : Produce set + get by product name () : product set + get by key word (in keyword : string) : product set

Attributes and methods

- **Getby artist():** Responsible for getting the product information based on artist.
- **Getby product name():** Responsible for getting the product information based on product name.
- **Getby keyword():** Responsible for getting the product information based on keywords.

5. Recommendation set

This class represents a collection of products recommended for the product selected by the user.

Recommendation set
– product : int
+ get recs (in product : int) : product set

Attributes and methods

- **productId:** The id of the recommended product.
- **getrecs():** Responsible for getting product recommendations.

6. Product Set

This class represents a collection of all the products that were searched by the user.

Product set
– set ID : int – product : int
+ Match product (in keyword : staring) : product set

Attributes and methods

- **set Id :** The id of the set of products.
- **product Id :** The id of the product in the set.
- **Match to Product() :** Responsible for matching the user's text with the product information.

7. Order Detail

This class represents the detail of each order.

Order detail
– order ID : int – product ID : int – product name : staring – quantity : int – unit cost : float – sub total : float
+ cacl price ()

Attributes and methods

- **ordered :** The order id.
- **product Id :** The id of the product contained in the order.
- **productName :** The name of product contained in the order
- **quantity :** The quantity of products in the order
- **unitCost :** The cost of each product in the order
- **subtotal :** The total cost of the order

8. Customer

This class will handle the customer actions. It inherits all the User class responsibilities and its functions.

Customer
- customer name : staring - address : staring - email : staring - Phone no. : int - credit card info : string - shipping info : staring
+ **resister (in user ID :** staring in password : string in email ID : string) : void + **login (in user ID** : void , in password : void) : void + **update profile** (in customer name : string in emailid : string , in address : string): void + **search** (in keyword : string) : void

Attributes and methods

- **customerId :** The user's customerid
- **password :** The user's password
- **customerName :** The user's name
- **address :** The user's address
- **email :** The user's email id
- **phoneno :** The user's contact number
- **creditcardInfo** : The user's credit card information for payment
- **shippingInfo :** The user's shipping information
- **register() :** Responsible for user registration
- **login() :** Responsible for user login
- **updateProfile() :** Responsible for updating user's profile information.
- **search()** : Responsible for searching an available MusicCD.

9. Administrator

This class will handle the administrator actions. It inherits all the Userclass responsibilities and its functions.

Administrator
- admin name : string - email : string
+ create department (in department ID : int in name : string in description : string) : department + create category (in category : int, name : string, in description : strain, in price : int) : product + create product (in department ID : int) : string, in description : string, in price : int) : product + delete department (in department : int) : void + delete category : (in category : int) : void + delete product (in product : int) : void + edit catalog details ()

Attributes and methods

- **adminId :** The administrator's Id.
- **password :** The administrator's password.
- **adminName :** The administrator's name.
- **email :** The administrator's email id.
- **createGenre()** : Responsible for creating a new genre.
- **createProduct()** : Responsible for creating a new product.
- **deleteGenre() :** Responsible for deleting an existing genre.
- **deleteProduct()** : Responsible for deleting an existing product.

10. Session Manager

The Session Manager class supports the User's required operations.

Session manager
- user ID : string - department : string
+ get user (in user ID : string) : user + get department (in department name : string) : department

Attributes and methods

- **userId:** The user's Id
- **genreId:** The genre id
- **getuser():** Responsible for getting user information
- **getallGenres():** Responsible for getting information of all available genres

11. Genre

This class represents a Genre

Genre
- department id - name - description
+ get category department (in category : int) category

Attributes and methods

- **genreId :** The genre's id
- **genreName :** The genre's name
- **description() :** The genre's description
- **getallProducts() :** Responsible for getting information of all available products

12. Product

This class represents collection of products of a particular genre.

Product
- product : int - name string - description : string - price : int - image file name : string
+ display product (in product id : int) : void + get product details (in product id : int) : void

Attributes and methods

- **productId:** The product's id
- **productName:** The product's name
- **genreName():** The products' genre name
- **artist:** The artist of the product
- **description:** The description of product
- **price:** The price of product
- **imageFileName:** The file name of the image for the product
- **getProductDetails():** Responsible for getting details of all available products

13. Shopping cart

This class represents a shopping cart that has all the products that are added by a customer to buy.

Shopping cart
- cart id : int - product id : int - quantity : int Date added : int
+ add cart item (in cart id : int, in product id : int) : void + delete cart item (in cart id : int, in product id : int) : void + view cart details (in product id : int, in quantity : int) void + check out () : void + calc total price ()

Attributes and methods

- **cartId:** The shopping carts' id
- **productId:** The id of a product contained by the shopping cart
- **quantity:** The quantity of cart items in the shopping cart

- **dateAdded** : The date when a new item is added to the cart
- **item** : The shopping cart item
- **addCartItem()** : Responsible for adding an item to the shopping cart
- **deleteCartItem()** : Responsible for deleting an item from the shopping cart
- **updatequantity()** : Responsible for updating the quantity of items in the shopping cart
- **viewCartDetails()** : Responsible for viewing the shopping cart details
- **checkout()** : Responsible for checking out
- **calcUnitPrice()** : Responsible for calculating the price of each cart item based on its quantity
- **calcTotalPrice()** : Responsible for calculating the total price of the cart items

14. Cart Item

This class represents a cart item object.

Cart item
– cart id : int – customer id : int – product id : int – quantity : int – unit cost : float – sub total : float
+ new (in user id : string, in product id : int)

Attributes and methods

- **cartId** : The items' cart id
- **customerId** : The id of the user
- **name** : The name of the user
- **productId** : The id of the product in the shopping cart
- **quantity** : The quantity of cart items in the shopping cart
- **unitCost** : The cost of the cart item
- **subtotal** : The cost of the cart item based on its quantity

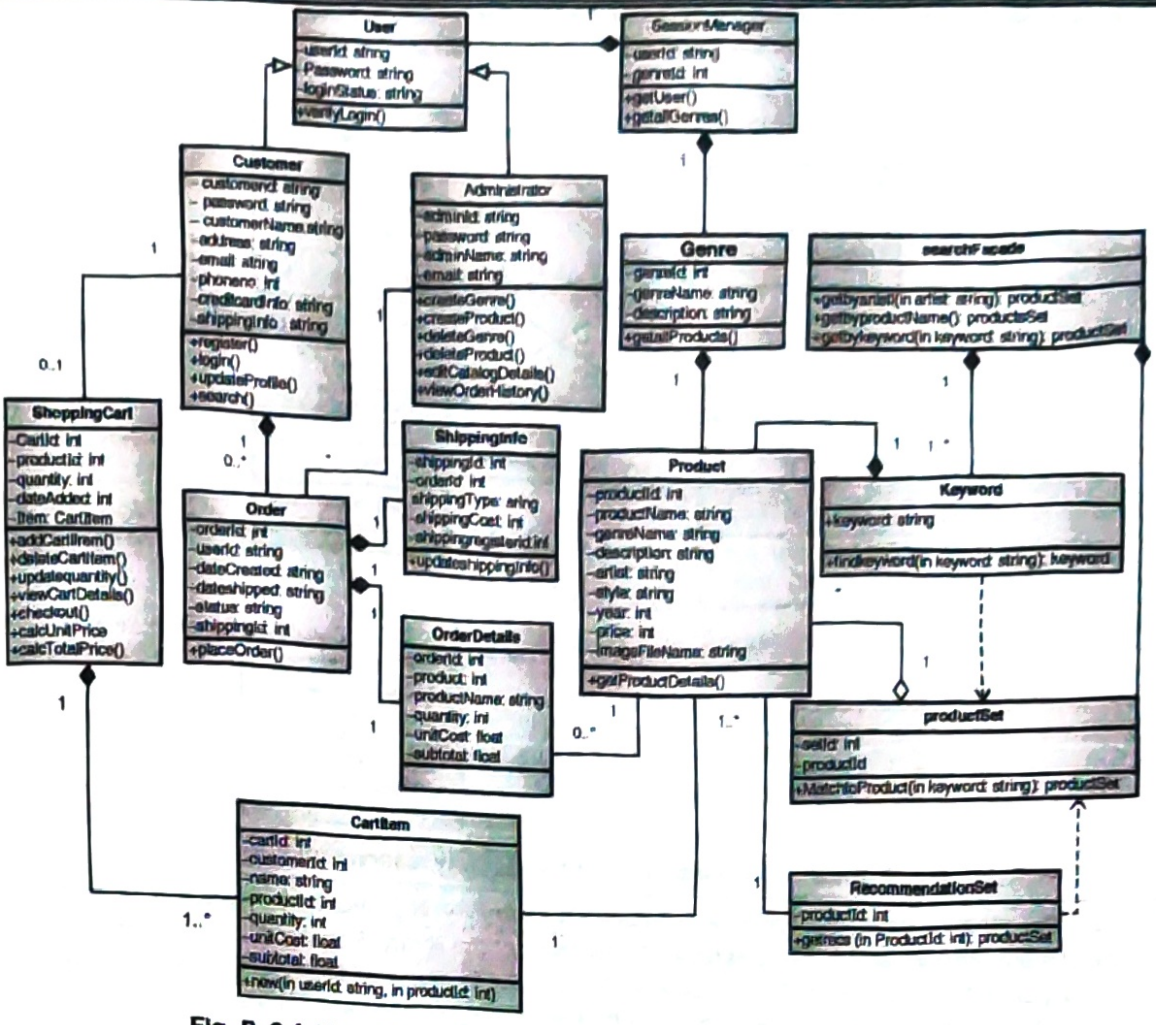

Fig. P. 2.4.1(a) : Class diagram for Online E-commerce Music Store

Ex. 2.4.2 : Draw a class diagram for engineering college.

Soln. :

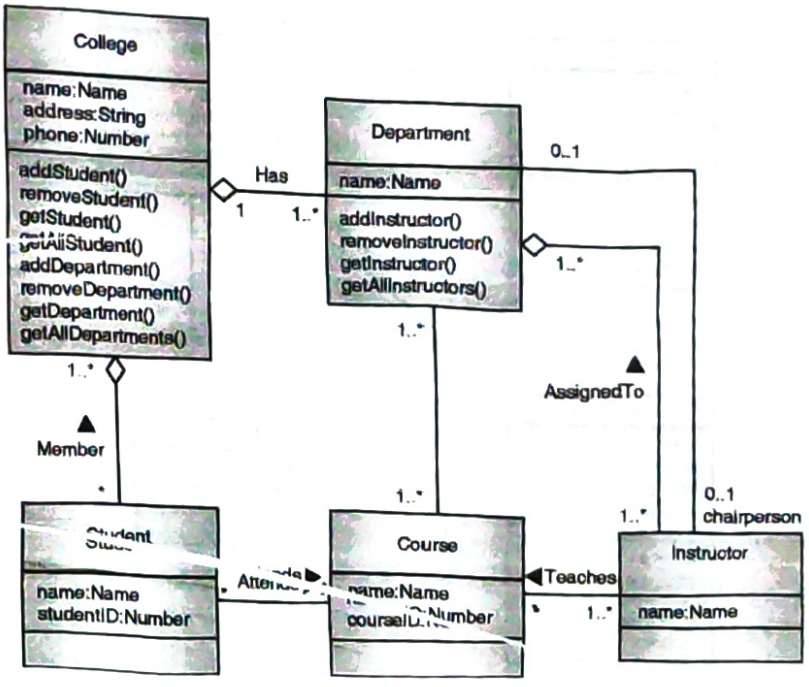

Fig. P. 2.4.2 : Class diagram for engineering college

Ex. 2.4.3 : Construct the class diagram for the Railway reservation system. **(10 Marks)**

Soln. :

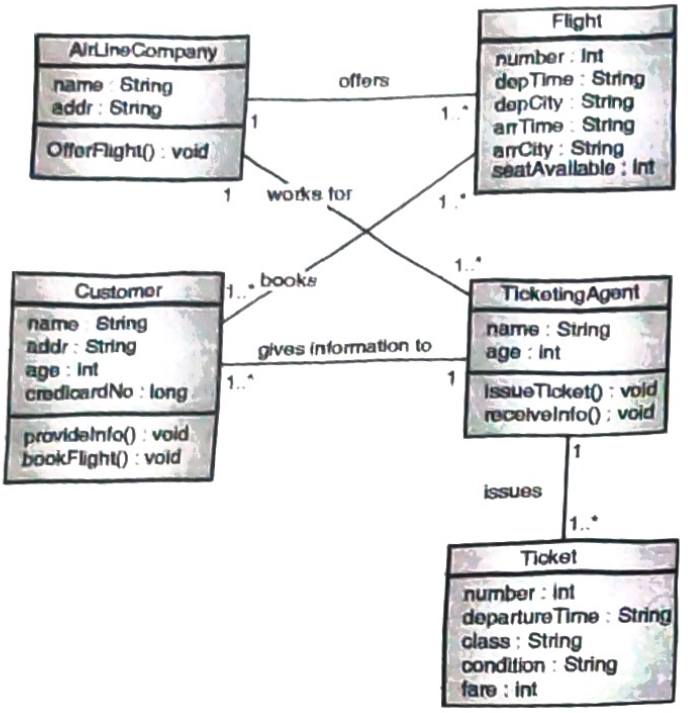

Fig. P. 2.4.3 : Class diagram for the Railway reservation system

Ex. 2.4.4 : Draw the class diagram for following scenario :A library lends books and magazines to members, who are registered in the system. Also it handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books and magazines are removed when they are Old date or in poor condition. A member can reserve a book or magazine that is not currently available in the library, so that when it's returned or purchased by the library, that person is notified. The library can easily create, update and delete information about the titles, members, loans and reservations in the system. **(10 Marks)**

Soln. :

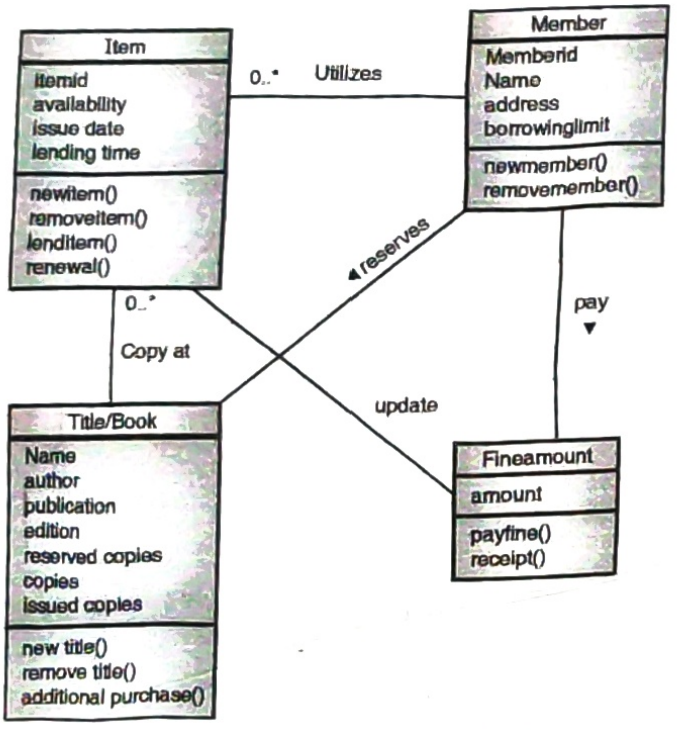

Fig. P.2.4.4 : Class diagram for LMS

Ex. 2.4.5 : Draw a class diagram for online shopping system. Assume the scope. SPPU - April 18, 5 Marks

Soln. :

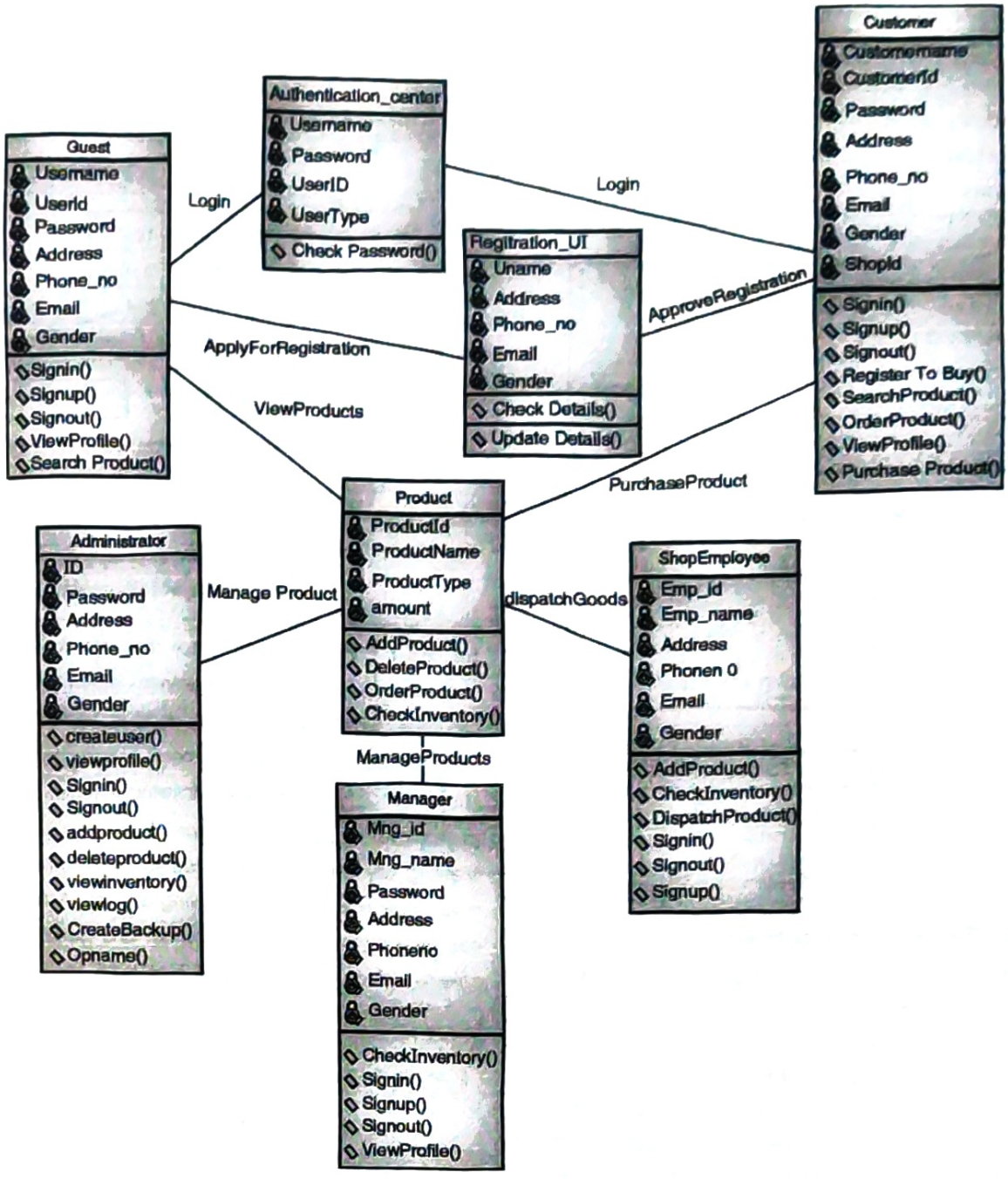

Fig. P. 2.4.5 : Class diagram for online shopping system

2.5 Object Diagram

- Object diagram provide a formal graphic notation for modeling object, classes and their relationship to one another. Object diagrams are useful for both abstract modeling and for designing actual programs.
- There are two types of object diagram : Class diagrams and instance diagrams.
- Class diagram is a schema, pattern or template for describing many possible instances of data, A class diagram describes object classes.

- An instance diagram describes how a particular set of objects are related to each other. An instance diagram describes object instances. Instance diagram are useful for documenting test cases (especially scenarios) and discussing example. a given class diagram describes infinite set of instance diagrams.

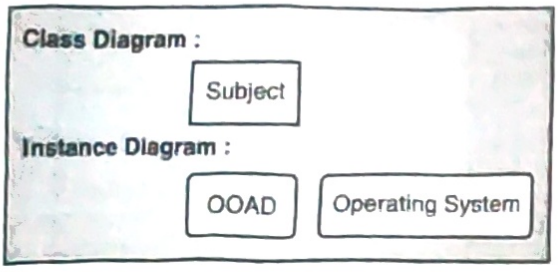

Fig. 2.5.1 : Class and instance diagram

The Relationship between Classes and Objects

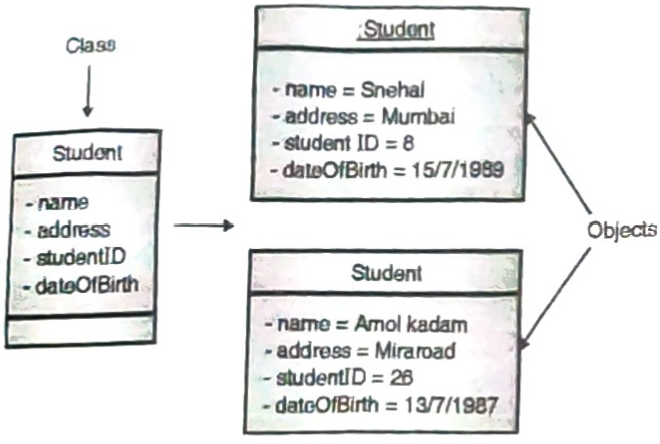

Fig. 2.5.2 : The relationship between classes and objects

- At the class level, the Student class attributes indicate that the Students have names, addresses, studentIDs, and a date of birth. At the object level, the attributes indicate the values for the name, address, studentID, and date of birth. Only the class instance (objects) should be able to change the value of the attributes. The state of an object is defined by the value of its attributes and the existence of links to other objects.
- The object diagram is a special kind of class diagram. An object is an instance of a class. This essentially means that an object represents the state of a class at a given point of time while the system is running. The object diagram captures the state of different classes in the system and their relationships or associations at a given point of time.

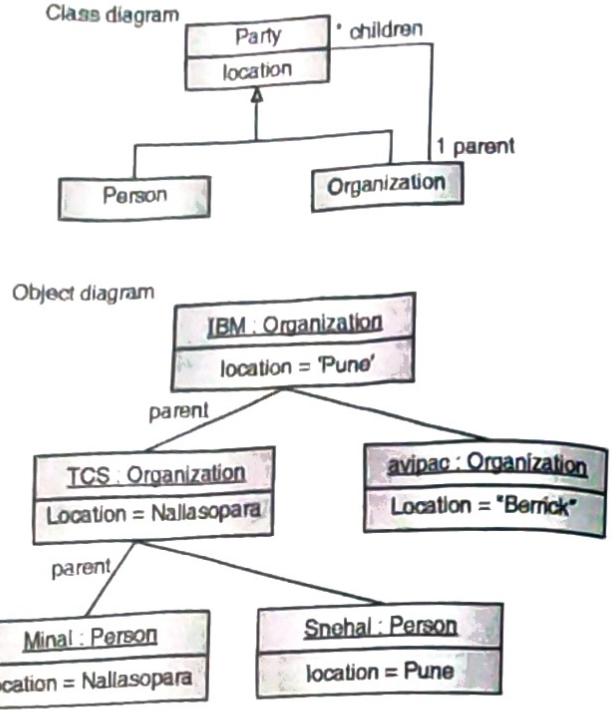

Fig. 2.5.3 : Object diagram

Ex. 2.5.1 : Prepare an object diagram for the following scenario: A graphical document editor that supports grouping, which is a concept used in variety of graphical editors/ assume that a document is composed of several sheets. Each sheet contains drawing objects, including text, geometrical objects, and groups. A group is simply a set of drawing objects, possibly including other groups. A group must contain at least two drawing objects. A drawing object can be a direct member of at most one group. Geometrical objects include circles, ellipses, rectangle, lines and squares **(12 Marks)**

Soln. :

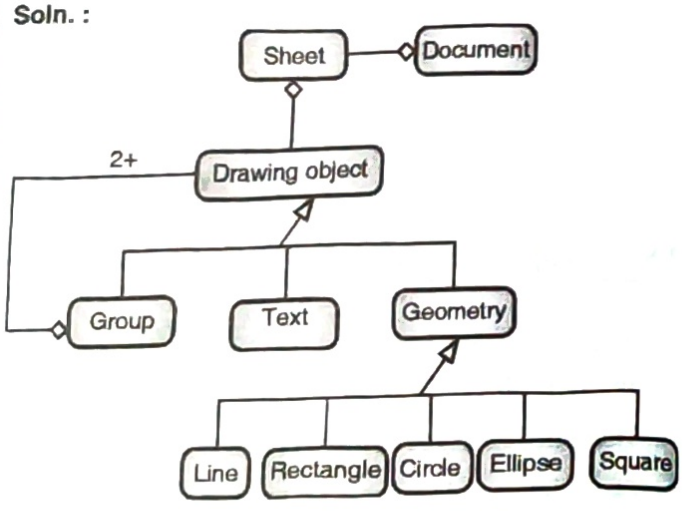

Fig. P. 2.5.1: Object diagram

Ex. 2.5.2 : Prepare an object model to describe undirected graphs. An undirected graph consists of set vertices and a set of edges. Edges connect pairs of vertices. Your, model should capture only the structure of graphs (i.e. connectivity), and need, not be concerned with geometrical details such as location of vertices or lengths of edges. A typical graph is shown in Fig. P.2.5.2. **(10 Marks)**

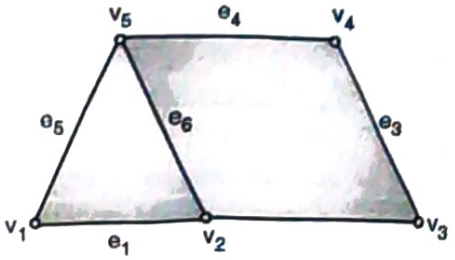

Fig. P. 2.5.2 : Sample undirected graph

Soln. :

Undirected graph is aggregation of vertex and graph. And edge should have at least two vertices.

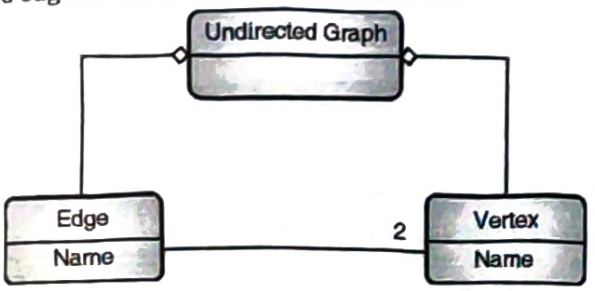

Fig. P. 2.5.2(a) : Object diagram

2.6 Package Diagram

University Question

Q. Different elements used in package diagram.

SPPU : April 18, 5 Marks

Q. Explain Package Diagram with UML notations.

SPPU : April 19, 5 Marks

- Package diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams provide a visualization of the namespaces. Package diagram is used to simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.
- A package is rendered as a tabbed folder - a rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.

Fig. 2.6.1

- **Package org.hibernate :** The members of the package may be shown within the boundaries of the package. In this case the name of the package should be placed on the tab.

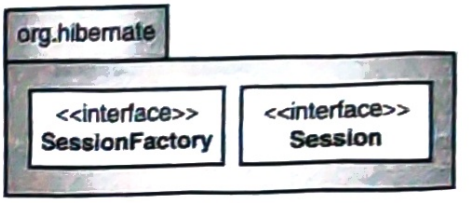

Fig. 2.6.2

- Package org.hibernate contains SessionFactory and Session.

Purpose of Package Diagrams

- Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.
- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- A package is a collection of logically related UML elements.
- Packages are depicted as file folders and can be used on any of the UML diagrams.

Key Elements of Package Diagram

- Packages are used to organize a large set of model elements:
  - Visibility
  - Import
  - Access

Table 2.6.1 : Key Elements of Package Diagram

Construct	Description	Syntax
Package	A grouping of model elements.	Name
Import	A dependency indicating that the public contents of the target package are added to the namespace of the source package.	<<import>>
Access	A dependency indicating that the public contents of the target package are available in the namespace of the source package.	<<access>>

- **<<import>>** : one package imports the functionality of other package.
- **<<access>>** : one package requires help from functions of other package.
- **Example** : Package Diagram for Order Processing subsystem.

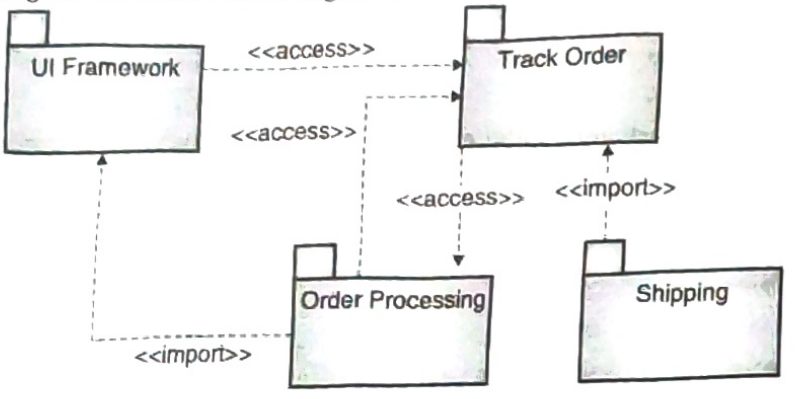

Fig. 2.6.3 : Package diagram for order processing subsystem

2.7 Component Diagram

University Question

Q. Define Component. **SPPU : Dec. 19, 2 Marks**

- A component diagram shows the organization and dependencies among a set of components. You use component diagrams to model the static implementation view of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents. Component diagrams are essentially class diagrams that focus on a system's components.
- We can model different types of components based on their use and applicability in the system. Components that we can model in system can be simple executable components or a library component that represents system and application libraries used in the system. We can also have file components that represents the source code file of the file implementation. Components are connected by dashed lines showing the dependency relationships between them.
- Use component diagrams when you are dividing your system into components and want to show their interrelationships through interfaces or the breakdown of components into a lower-level structure.
- With component diagram, the staff responsible for compiling and deploying the system will know which code libraries exist and which executable files will be created when the code is compiled. Developers will know which code libraries exist and what the relationships are between them. The component dependencies will let those who are responsible for compiling know in which order the components need to be compiled

What is a Component ?

- A component is a physical module of code. Components can include both source code libraries and runtime files. For example, if you are using C++, each of you .cpp and .h files is a separate component. The .exe file that you create after the code is compiled is also a component.
- The Component view is concerned with the physical structure of the system. A component is simply a file associated with the system. It may be a source code file, an executable file, or a DLL file. In Rose, there are various icons you can use to distinguish the different types of components.
- Before you generate code, you map each of your files to the appropriate component(s). In C++, for example, each class is mapped to two components- one representing the .cpp files for that class and one representing the .h file. In Java, you map each class to a single component, representing the .java file for that class.
- Once the components are created, they are added to a Component diagram and relationships are drawn between them. The only type of relationship between components is a dependency.

2.7.1 Types of Components / Elements of Component Diagram

1. Component

The Component icon represents a software module with a well-defined interface. In the component specification, you specify the type of component in the Stereotype field (e.g., ActiveX, Applet, Application, DLL, and Executables).

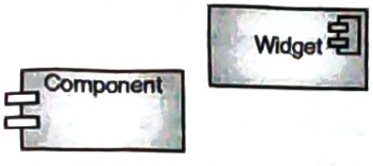

Fig. 2.7.1

2. Subprogram Specification and Body

These icons represent a subprogram's visible specification and the implementation body. A subprogram is typically a collection of subroutines. Subprograms do not contain class definitions.

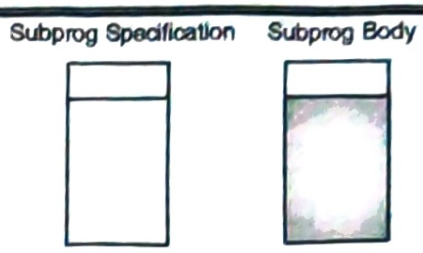

Fig. 2.7.2

3. Main Program

The Main Program icon represents the main program. A main program is the file that contains the root of a program.

Fig. 2.7.3

4. Package Specification and Body

A package is the implementation of a class. A package specification is a header file, which contains function prototype information for the class. In C++, package specifications are the .h files. In Java, you use the Package Specification icon to represent the .java files. A package body contains the code for the operations of the class. In C++, package bodies are the .cpp files.

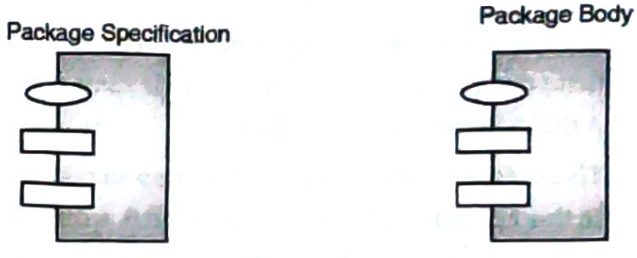

Fig. 2.7.4

5. Task Specification

These icons represent packages that have independent threads of control. An executable file is commonly represented as a task specification with a .exe extension.

Task Specification

Fig. 2.7.5

6. **Database**

This icon represents a database, which may contain one or more schemas. On a Component diagram, a database is shown with the following icon.

Database

Fig. 2.7.6

7. **Dependencies**

- The only type of relationship that exists between components is a component dependency. A component dependency suggests that one component depends on another. A component dependency is drawn as a dashed arrow between the components.

Fig. 2.7.7

- In this example, Component A depends upon Component B. In other words, there is some class in A that depends on some class in B.
- These dependencies have compilation implications. In this example, because A depends on B, A cannot be compiled until B has been compiled. Someone reading this diagram will know that B should be compiled first, followed by A.
- Fig. 2.7.8 shows an example component diagram. In this example, a sales till can connect to a sales server component, using a sales message interface. Because the network is unreliable, a message queue component is set up so the till can talk to the server when the network is up and talks to a queue when the network is down; the queue will then talk to the server when the network becomes available. As a result, the message queue both supplies the sales message interface to talk with the till and requires that interface to talk with the server. The server is broken down into two major components. The transaction processor realizes the sales message interface and the accounting driver talks to the accounting system.

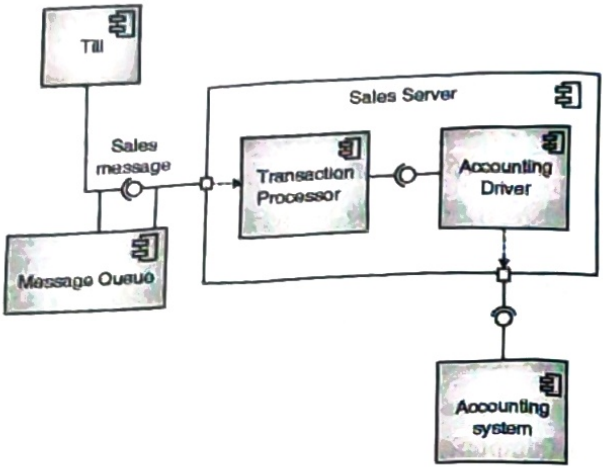

Fig. 2.7.8 : A component diagram

2.7.2 Interface

University Question

Q. Define interface. Explain provided interface and required interface with an example.

SPPU : April 18, Dec. 19, 5 Marks

- The interface is a named set of public features. It separates the specification of functionality from its implementation by a class diagram or a subsystem. An interface symbol cannot be instantiated. It declares a contract that may be realized by zero or more classifiers such as a class or a subsystem. Anything that realizes an interface accepts the functionalities of the interface and agrees to abide by the contract defined by the interface.
- There are two types of interfaces,

1. Provided interfaces
2. Required interfaces

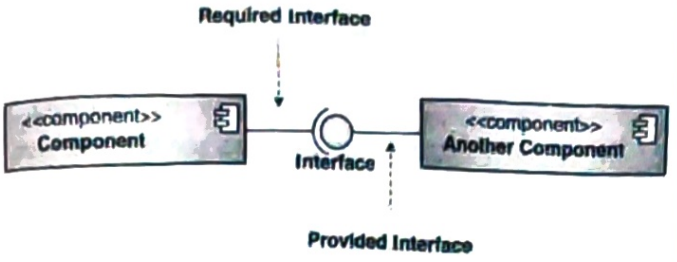

Fig. 2.7.9

1. **Provided interface** symbols with a complete circle at their end represent an interface that the component provides - this "lollipop" symbol is shorthand for a realization relationship of an interface classifier.
2. **Required interface** symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires.

2.8 Composite Structure Diagrams

- A composite structure diagram is a diagram that shows the internal structure of a classifier, including its interaction points to other parts of the system. It shows the configuration and relationship of parts, that together, perform the behavior of the containing classifier.
- A composite structure diagram also provides these benefits:
  - Helps users understand the current state of their system
  - Breaks down the internal structure of multiple classes, interfaces, or components, and their interactions
  - Details runtime architectures and usage patterns not found in static diagrams
  - Provides users with information to optimize and troubleshoot their system
- Composite structure diagram describes the way classes can be displayed as composite elements exposing interfaces and containing ports and parts.

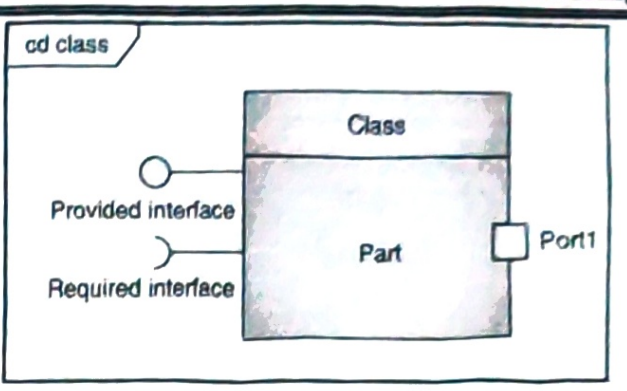

Fig. 2.8.1

2.8.1 Part

- A part is an element that represents a set of one or more instances which are owned by a containing classifier instance. So for example, if a diagram instance owned a set of graphical elements, then the graphical elements could be represented as parts; if it were useful to do so, to model some kind of relationship between them. Note that a part can be removed from its parent before the parent is deleted, so that the part isn't deleted at the same time.
- A part is shown as an unadorned rectangle contained within the body of a class or component element.

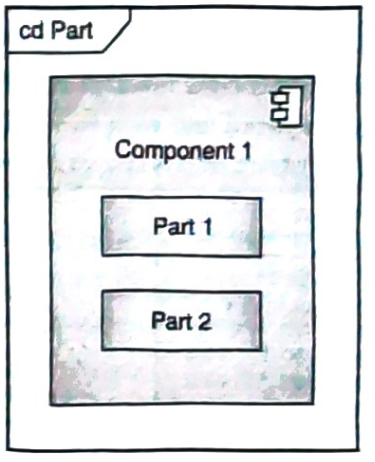

Fig. 2.8.2

2.8.2 Port

- A port is a typed element that represents an externally visible part of a containing classifier instance. Ports define the interaction between a classifier and its environment. A port can appear on the boundary of a contained part, a class or a composite structure. A port may specify the services a classifier provides as well as the services that it requires of its environment.

- A port is shown as a named rectangle on the boundary edge of its owning classifier.

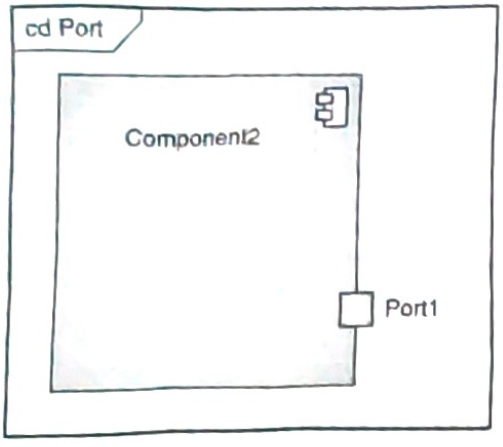

Fig. 2.8.3

2.8.3 Interfaces

- An interface is similar to a class but with a number of restrictions. All interface operations are public and abstract, and do not provide any default implementation. All interface attributes must be constants. However, while a class may only inherit from a single super-class, it may implement multiple interfaces.
- An interface, when standing alone in a diagram, is either shown as a class element rectangle with the «interface» keyword and with its name italicized to denote it is abstract, or it is shown as a circle.

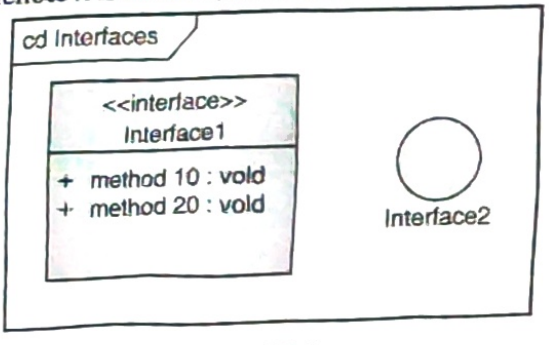

Fig. 2.8.4

- Note that the circle notation does not show the interface operations. When interfaces are shown as being owned by classes, they are referred to as exposed interfaces. An exposed interface can be defined as either provided or required. A provided interface is an affirmation that the containing classifier supplies the operations defined by the named interface element and is defined by drawing a realization link between the class and the interface. A required interface is a statement that the classifier is able to communicate with some other classifier which provides operations defined by the named interface element and is defined by drawing a dependency link between the class and the interface.
- A provided interface is shown as a "ball on a stick" attached to the edge of a classifier element. A required interface is shown as a "cup on a stick" attached to the edge of a classifier element.

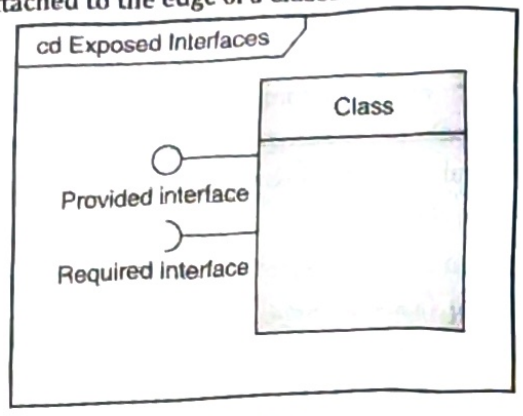

Fig. 2.8.5

2.8.4 Delegate

- A delegate connector is used for defining the internal workings of a component's external ports and interfaces. A delegate connector is shown as an arrow with a «delegate» keyword. It connects an external contract of a component as shown by its ports to the internal realization of the behavior of the component's part.

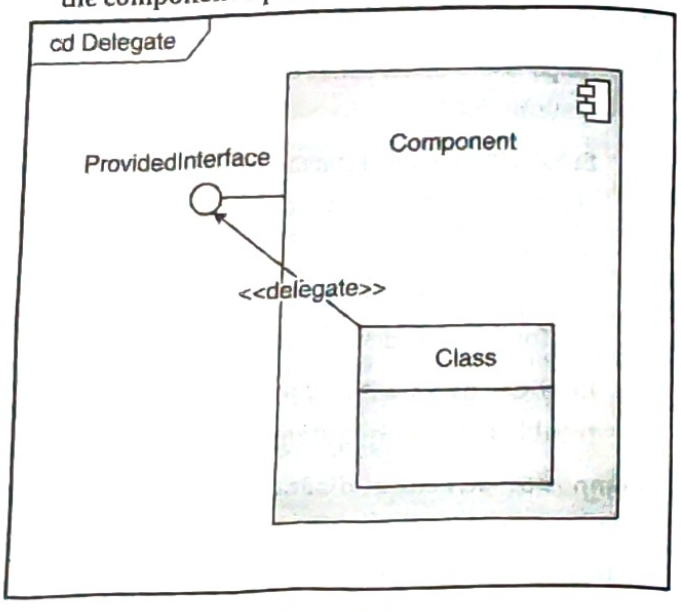

Fig. 2.8.6

2.9 Deployment Diagram

- The deployment diagram provides a different perspective of the application, captures the configuration of the runtime elements of the application.
- A deployment diagram is a diagram that shows the configuration of runtime processing nodes and the components that live on them. Graphically it is a collection of vertices and arcs. It is also called as process diagram.
- Deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.
- Deployment diagrams are not only important for visualizing, specifying and documenting embedded client/server and distributed system but also managing executable systems through forward and reverse engineering.
- Deployment diagram is an implementation diagram that shows the structure of a run-time system. From it, you can learn about the physical relationships among software and hardware components and the distribution of components to processing nodes.
- We use deployment diagram to show the structure of the run-time system and communicate how the hardware and software elements that make up an application will be configured and deployed.

2.9.1 Elements of a Deployment Diagram

University Question

Q. Explain the different elements of development diagram. **SPPU : April 18, 2 Marks**

1. Processor

Processor is a piece of hardware capable of executing programs. The processor is presented as shaded cube with a name of the object. A processor can have a list of processes on it.

Fig. 2.9.1

2. Device

A piece of hardware incapable of executing program is called as device. Device will also have name on a cube.

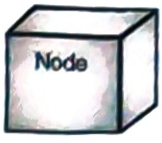

Fig. 2.9.2

3. Connection

- Similar to relation/association used in class diagrams to define the inter-connection between nodes. It normally indicates direct hardware coupling such as Bluetooth connection or fibre optic connection. Connections are normally bidirectional shown in the diagram. Some connections can be uni-directional. They are shown by a line with an arrow.

Fig. 2.9.3

- Deployment diagrams may contain notes and constraints. They may also contain components each of which must live on some node. Deployment diagrams may also contain packages or subsystems both of which are used to group elements of our model into larger chunks.

2.9.2 When to use Deployment Diagram

We use deployment diagrams to model the static deployment view of the system. When we model the static deployment view of the system, we will typically use deployment diagrams in one of the three ways :

1. To model embedded systems

- An embedded system is a software intensive collection of hardware that interfaces with the physical world. Developing an embedded system is far more complex than a software problem. When we model such a system we have to take into account its interface with the real world and that means, reasoning about unusual devices as well as nodes. Hardware is the major part in embedded system. Deployment diagram should help both the groups to visualize the system engineering aspect of the project.

- To model an embedded system
  - Identify the devices and nodes that are unique to our system.
  - By using the UML's extensibility mechanisms define system- specific stereotypes with appropriate icons for unusual devices. If stereotyping is not possible, use processors for software component and devices for hardware component.
  - Model the relationships among these processor and devices in deployment diagram.

2. To model client/server systems

- A client server system is a common architecture focused on making a clear separation of concerns between the system's user interface (which lives on the client) and the systems persistent data (which lives on the server). Client server systems are one end of the continuum of distributed systems and require you to make decisions about the network connectivity of clients to servers and about the physical distribution of your systems software components across the nodes. We use the deployment diagrams to visualize, specify and document our decisions about the topology of our client server system and how its software components are distributed across the client and server.
- In client server system, there are many variations. The client can be thin client or thick client. Thin client means client has limited amount of computational capacity and it just manage the user interface and visualization of information. Thick client has good amount of computational capacity and typically carries some of system logic and business rules. The choice between thin client and thick client is an architectural decision that is influenced by number of technical, economic and political factors.
- We use deployment diagrams to visualize, specify and document our decision about the topology of our client/server system and how its software components are distributed across the client and server.
- To model client/server system
  - Identify the nodes that represent yours systems client and server processors.
  - Highlight special devices significant to the system. For e.g. we will have to model special devices, such as credit card readers, display devices other than monitors, because their placement in the systems hardware topology are likely to be architecturally significant.
  - Provide visual cues for these processor and devices via stereotyping.
  - Model the topology of these nodes in deployment diagram.
  - Similarly, specify the relationship between the components in your system's implementation view and nodes in your systems deployment view.

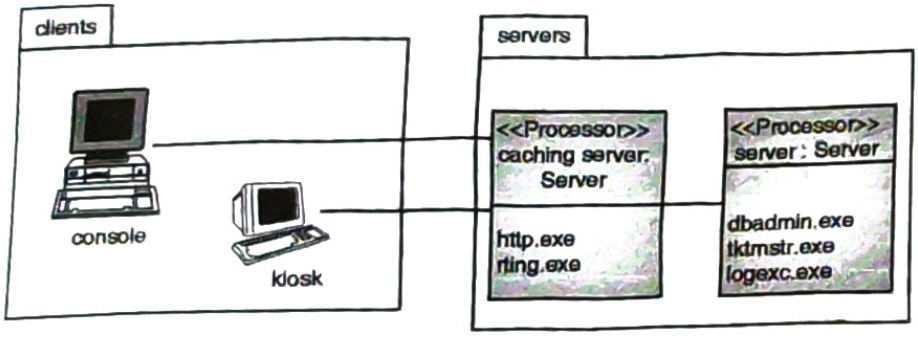

Fig. 2.9.4

For example :

Fig. 2.9.4 shows the topology of human resource system which is based on client- server architecture. The client and server split explicitly by using the packages named client and server. The client package contains two nodes (console and Kiosk) both of them are stereotyped and are visually distinguishable. The server package contain two kinds of nodes (catching sever and server) both of these have been adorned with some of the components that reside on each.

3. **To model fully distributed systems**

- At the other end of the continuum of distributed systems are those that are widely, if not globally, distributed. Typically encompassing multiple levels of server such systems are often hosts to multiple versions of software components, some of which may even migrate from node to node. Crafting such systems requires us to make decisions that enable the continuous change in the systems topology. We can use deployment diagrams to visualize the systems current topology and distribution of components to reason about the impact of changes on that topology.
- We can use the UML's deployment diagrams to reasons about the topology of such fully distributed system. When we document fully distributed system using deployment diagrams, we want to expand on the details of the system's networking devices, each of which we can represent as a stereotyped node.
- Visualizing, specifying and documenting the topology of fully distributed systems such as these are valuable activities for the systems administrators, who must keep tabs on an enterprise's computing assets.
- To model a fully distributed system
  - Identify the systems devices and processors as for simpler client /server system.
  - Model the communication devices to the level of detail sufficient to make assessments about the performance of the system network or the impact of changes to the network.
  - Use packages for logical grouping of nodes.
  - Model these devices and processor using deployment diagrams , use tools that discover the topology of your system by walking your systems network
  - If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behaviour you are interested in and expand on these use cases, with interaction diagram.

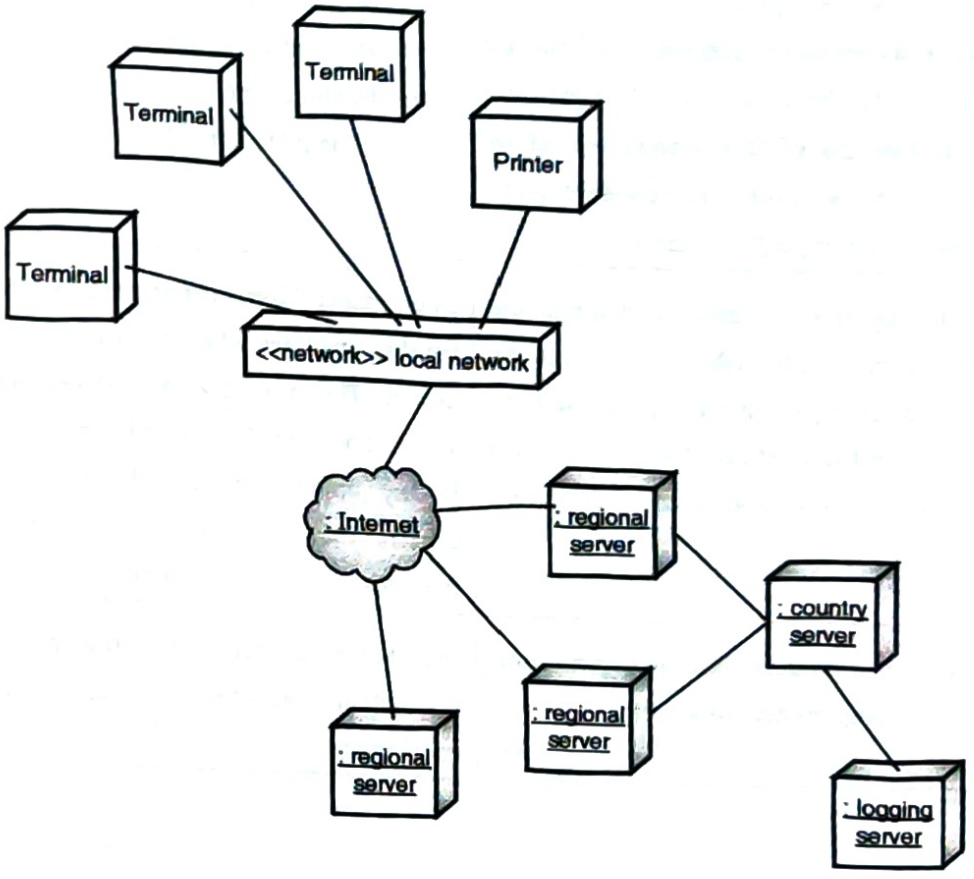

Fig. 2.9.5

- When modeling fully distributed system, we can also rectify the network itself as node i.e. we can rectify Local Area Network or Wide Area Network as node. In such case, we can use nodes attribute and operation to capture properties about network.

Difference between component diagram and deployment diagram

University Question

Q. Explain the difference between component diagram and deployment diagram in UML.

SPPU : May 18, Dec. 19, 5 Marks

Table 2.9.1

Sr. No.	Component Diagram	Deployment diagram
1.	Component diagram depends upon classes , interfaces which are part of class diagram.	Deployment diagram depends on Component which are part of component diagram.
2.	Component diagram shows you how different elements of your system have been grouped together (into assemblies / dllsetc) - and the link between these components. Example, if "Utility.dll" is a component and say it is deployed on the Client Machine (hardware). Then, the Component Diagram of this system will show Utility and its link with other components in the system (say Customer / SQL Packages).	A Deployment diagram takes you one step further and describes on which hardware elements do these components reside Example: the Deployment Diagram will show the hardware configuration - DB Server / Web Server / Client Machine .and Utility component will be placed into the Client Machine Node.
3.	Component Diagrams are used to diagram the high level components which you will be developing. For example , in my application, I use COM to separate the modules. Each COM object runs in its own executable and is a modeled as a component.	Deployment Diagrams are used to diagram the actual runtime modules which are required to complete the system. This include your modules and any hardware / software which is required to complete your system. In my application, the Deployment diagram will contain the names of all the servers and which modules they will be running.
4.	A component diagram describes the organization of the physical components in a system.	Deployment diagrams depict the physical resources in a system including nodes, components, and connections.

Ex. 2.9.1 : Draw a Deployment Diagram for Inventory Management System.

Soln. :

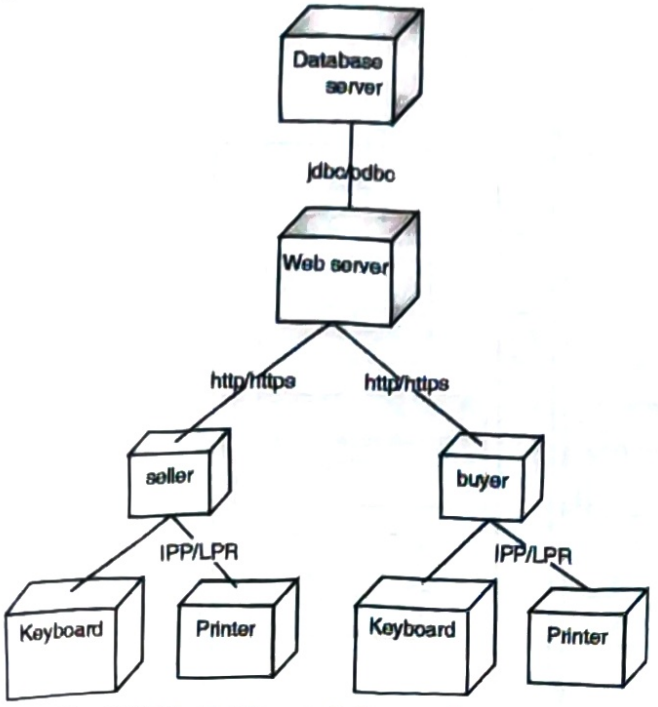

Fig. P.2.9.1 : Deployment diagram for Inventory Management System

1. Processor

Here we have 2 processors which handle the online inventory management.

i. **Web Server :** This server will handle the entire HTTP request from the user like login and will simultaneously pass the request to the database server. It will handle all the html pages and images in the website and it uses JDBC/ODBC drivers to communicate with database server.

ii. **Database Server :** The database or db server will store the entire password corresponding to usernames and also all the transactions i.e. what a user has purchased and what is corresponding delivery date of the product.

2. Devices

1. In online shopping, devices are the clients which will access the websites and make transactions on the website. Also the printers will also be included in the devices and each client will have its printer. To access printer's protocols like IPP (Internet Printing Protocol)/LPR (Line Printer Remote Protocol) is used and for accessing the website HTTP protocol is used or Client Server Protocol is used.

2. **Connection :** The connection will be bidirectional as this is client server architecture so for every request of the client there will be a response from the server also the server will communicate while user makes his transactions.

Ex. 2.9.2 : Draw Deployment diagram for the airline reservation system.

Soln. :

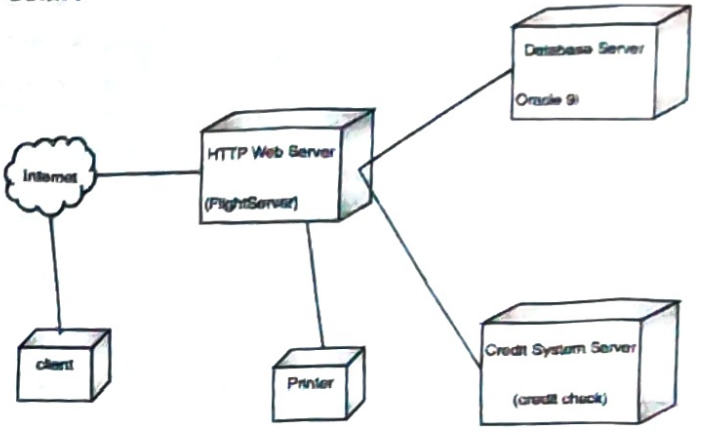

Fig. P.2.9.2 : Deployment diagram for airline reservation system

Ex. 2.9.3 : Draw a deployment diagram for the Order Processing system.

Soln. :

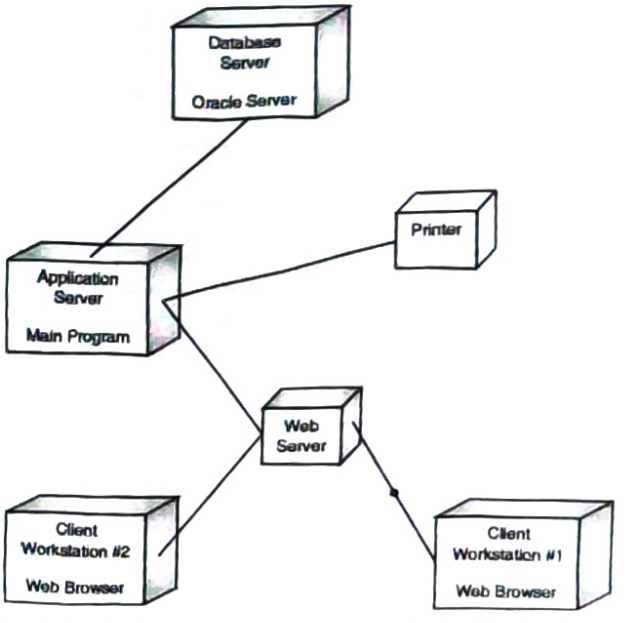

Fig. P.2.9.3 : Deployment diagram for order processing system

Ex. 2.9.4 : Draw a deployment diagram for ATM.

Soln. : The deployment architecture of an ATM system typically consists of three major nodes: the ATM stations, the consortium computer, and the bank computer. The topology is a simple star, the consortium communicates with all the ATM stations and with the entire bank computer. The architecture uses the station code and the bank code to distinguish the phone lines to the consortium computer.

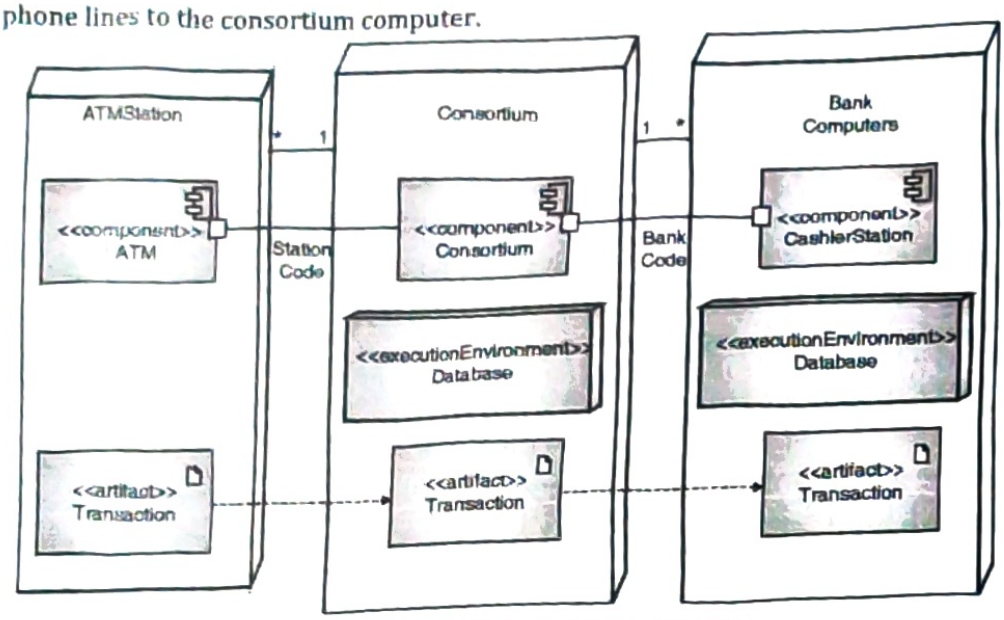

Fig. P.2.9.4 : Deployment diagram for ATM

Ex. 2.9.5 : Draw Deployment Diagram for Java Online Web Application. Users invoke the web client through a Java enabled web browser such as IE or Firefox with the Java JVM plug-in installed. Through the Java Applet, user can select product item(s) they would like to order online by using the online Shopping Cart. As the online transaction involves the credit card and personal information, users communicate with the web host using SSL "Provided Interface" of the Java Servlet Component and it will then update the status to the Database Component through the "Provided Interface" JDBC.

Soln. :

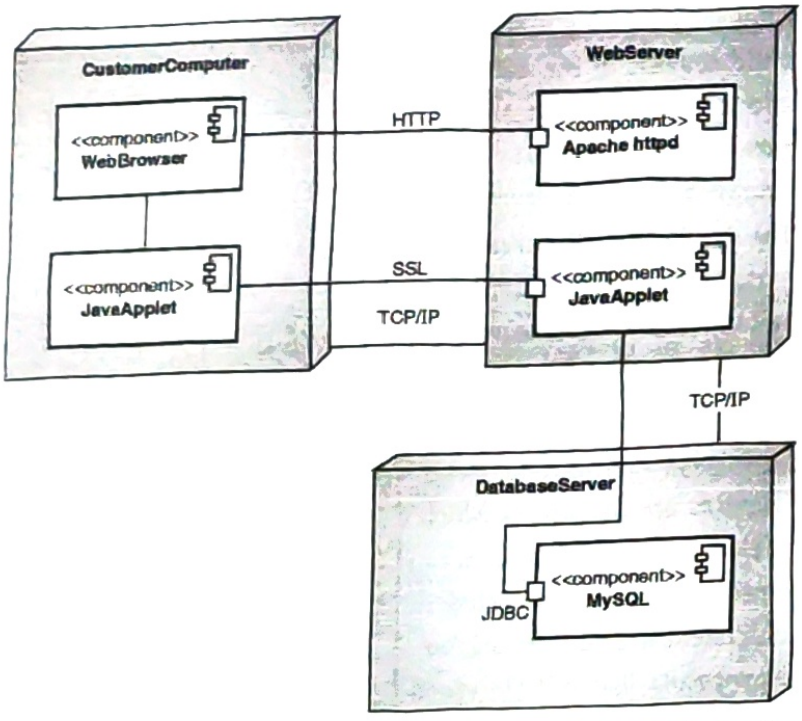

Fig. P.2.9.5 : Deployment diagram for a Typical Java Online Web Application

Ex. 2.9.6 : Draw a deployment diagram for engineering college.

Soln. :

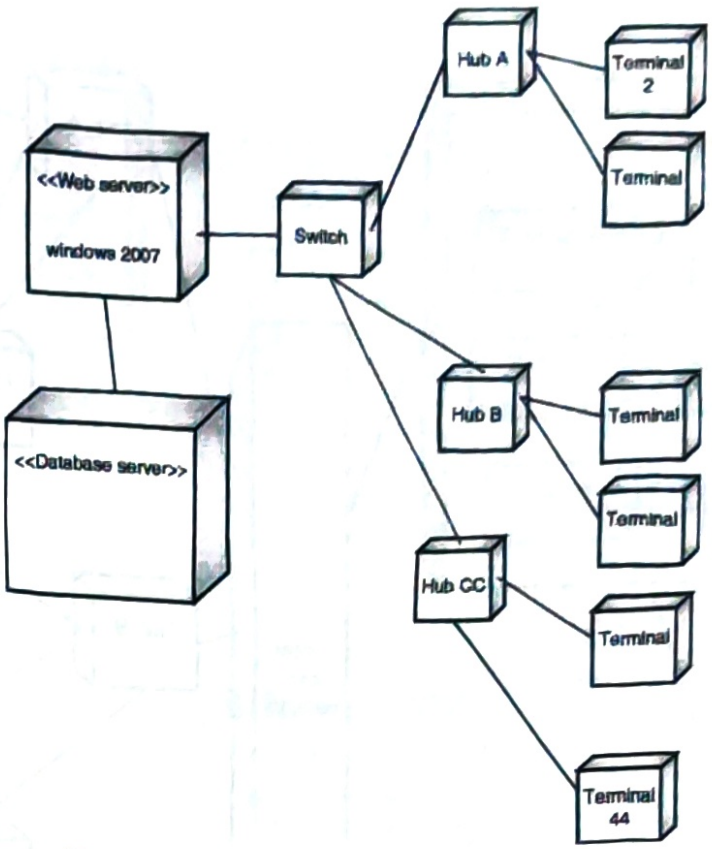

Fig. P.2.9.6 : Deployment diagram for engg. college

Ex. 2.9.7 : You are appointed as a consultant for the Intranet development of your college. Construct the component diagram and deployment diagram.

Soln. :

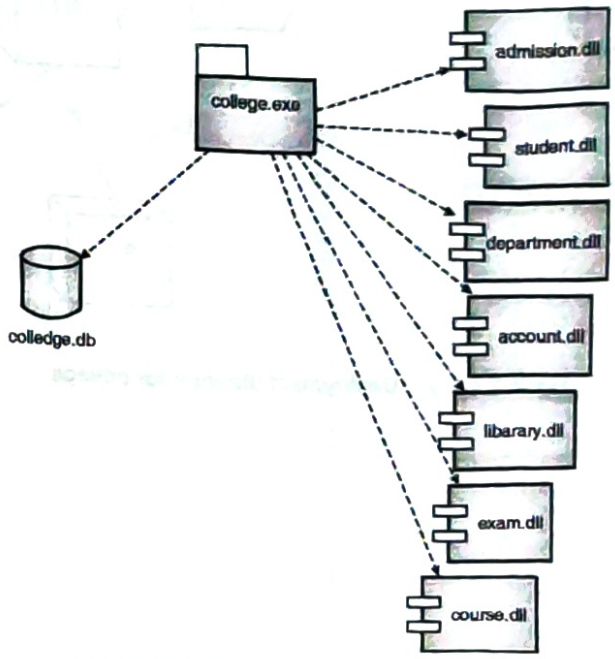

Fig. P.2.9.7 : Component diagram for college

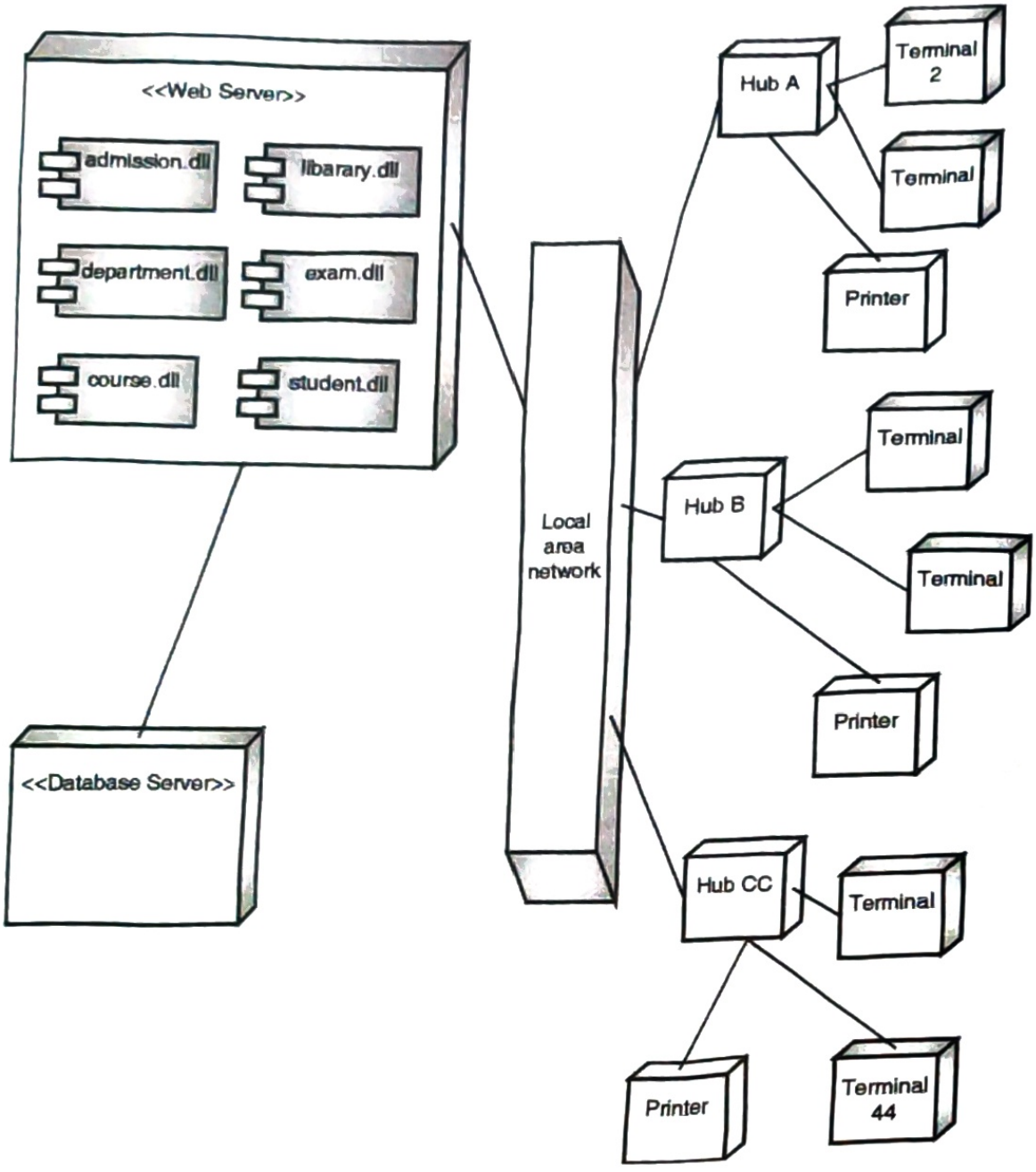

Fig. P.2.9.7(a) : Deployment diagram for college

Ex. 2.9.8 : Draw a deployment diagram for any enterprise application.

Soln. :

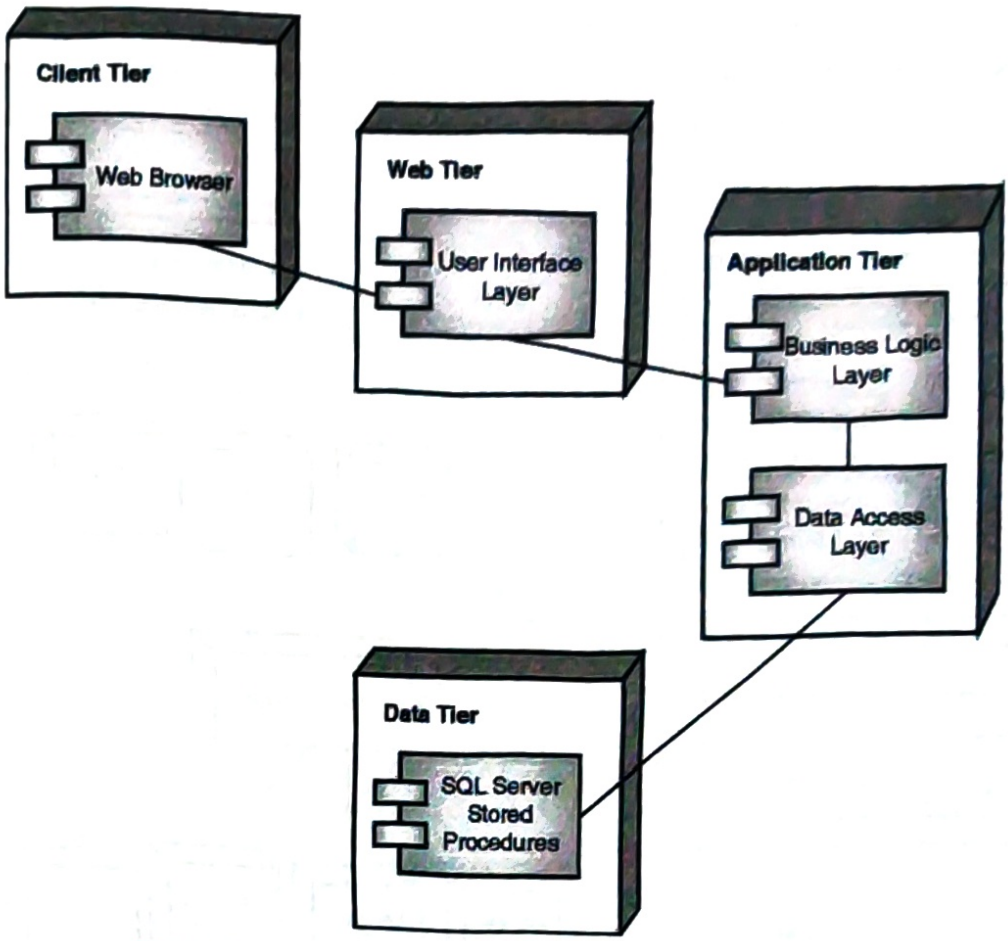

Fig. P.2.9.8 : Deployment diagram for any enterprise application

Ex. 2.9.9 : Draw a deployment diagram for checking E-mail scenario.

Soln. :

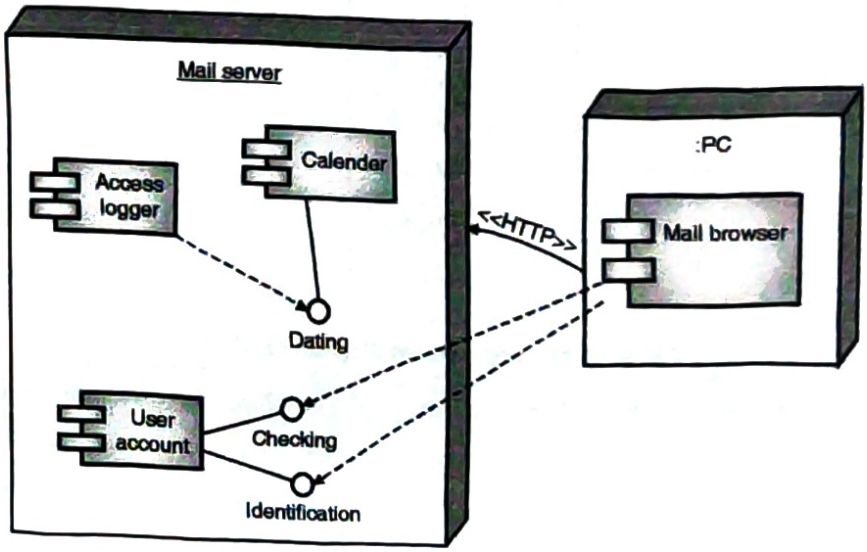

Fig. P.2.9.9 : Deployment diagram for E- mail

Ex. 2.9.10 : Draw a deployment diagram for the below scenario. Consider a university database that keeps track of student and their majors, transcripts and registration and the university courses. Several sections of each course are offered and each section is related to the instructor who is teaching. It also keeps track of sponsored research projects of faculty and graduate students of academic departments of the particular college. The database also keeps track of the research grants and contracts awarded to the University. A grant is related to one principal investigator and to all researchers it supports.

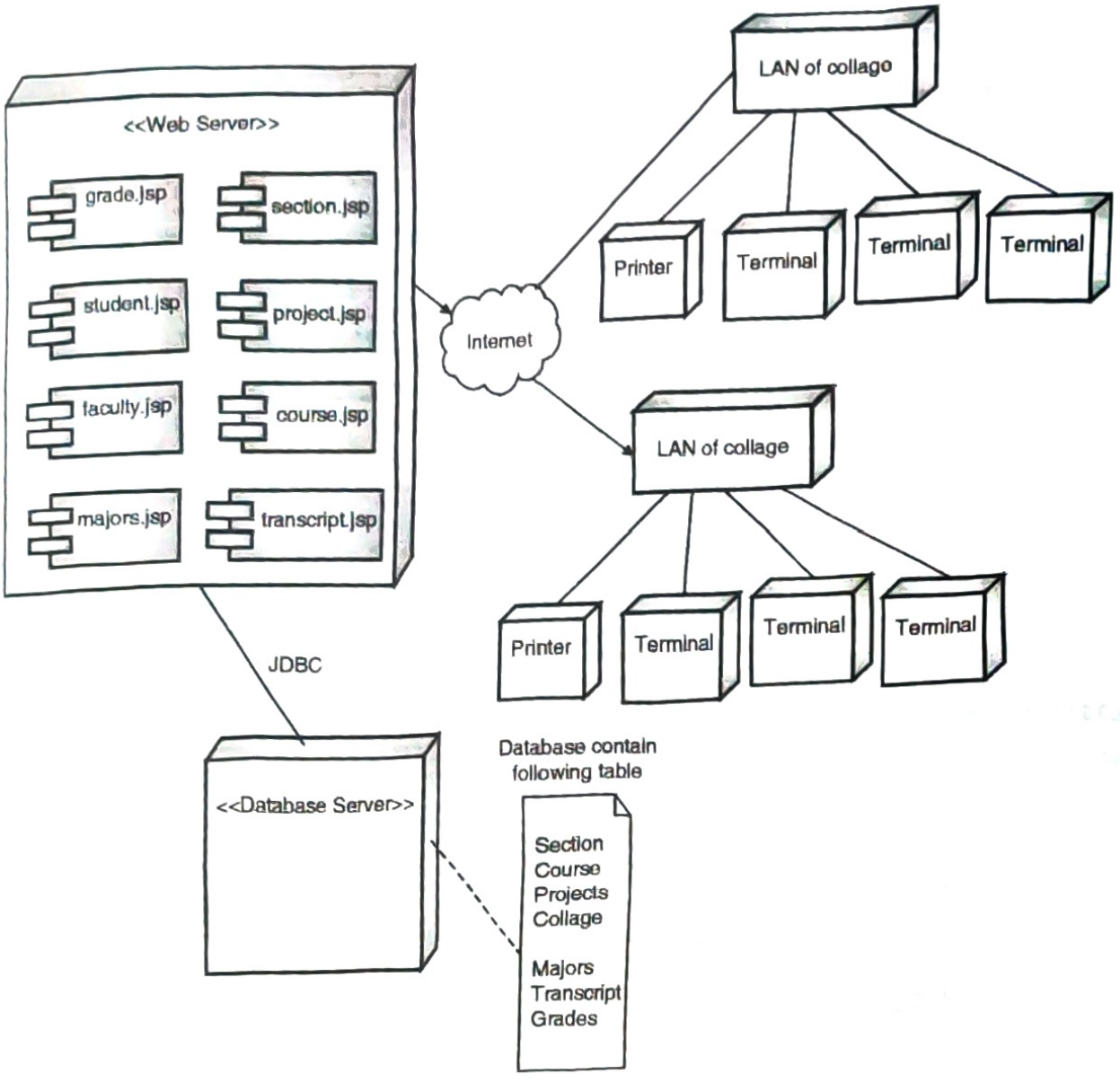

Fig. P.2.9.10 : Deployment diagram for UMS

Ex. 2.9.11 : Draw a deployment diagram for J2EE web application with load balancing and clustering which shows specific service instances involved. **SPPU - April 18, 3 Marks**

Soln. :

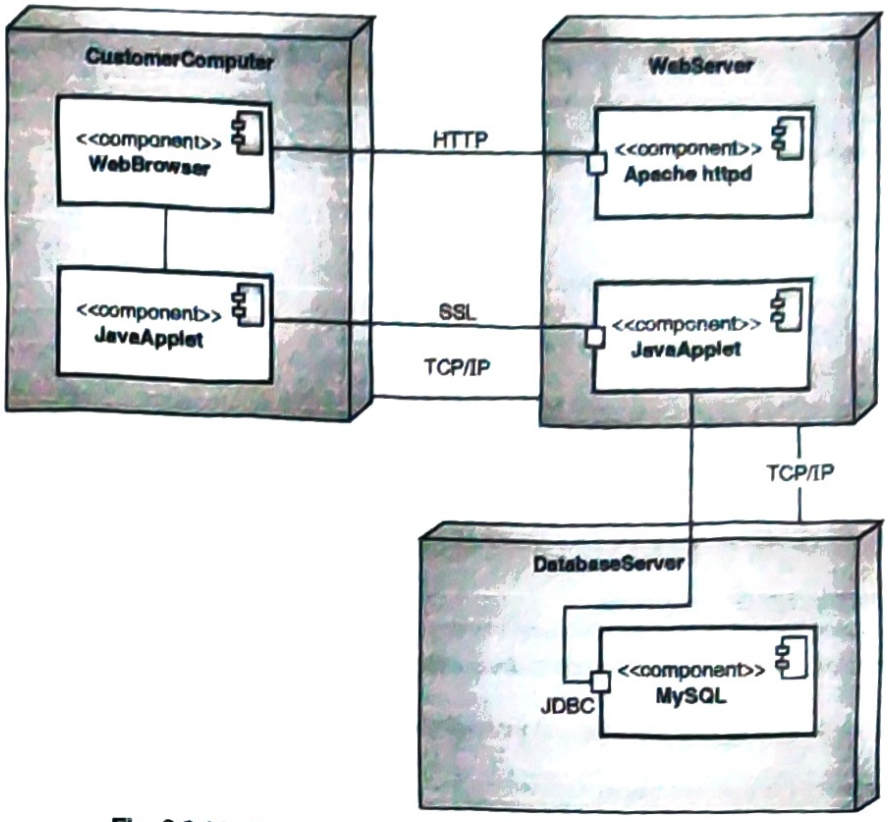

Fig. 2.9.11 : Deployment diagram for J2EE web application

Ex. 2.9.12 : Describe and draw Deployment Diagram for ATM System. **SPPU- April 19, 5 Marks**

Soln. : The deployment architecture of an ATM system typically consists of three major nodes : the ATM stations, the consortium computer, and the bank computer. The topology is a simple star, the consortium communicates with all the ATM stations and with the entire bank computer. The architecture uses the station code and the bank code to distinguish the phone lines to the consortium computer.

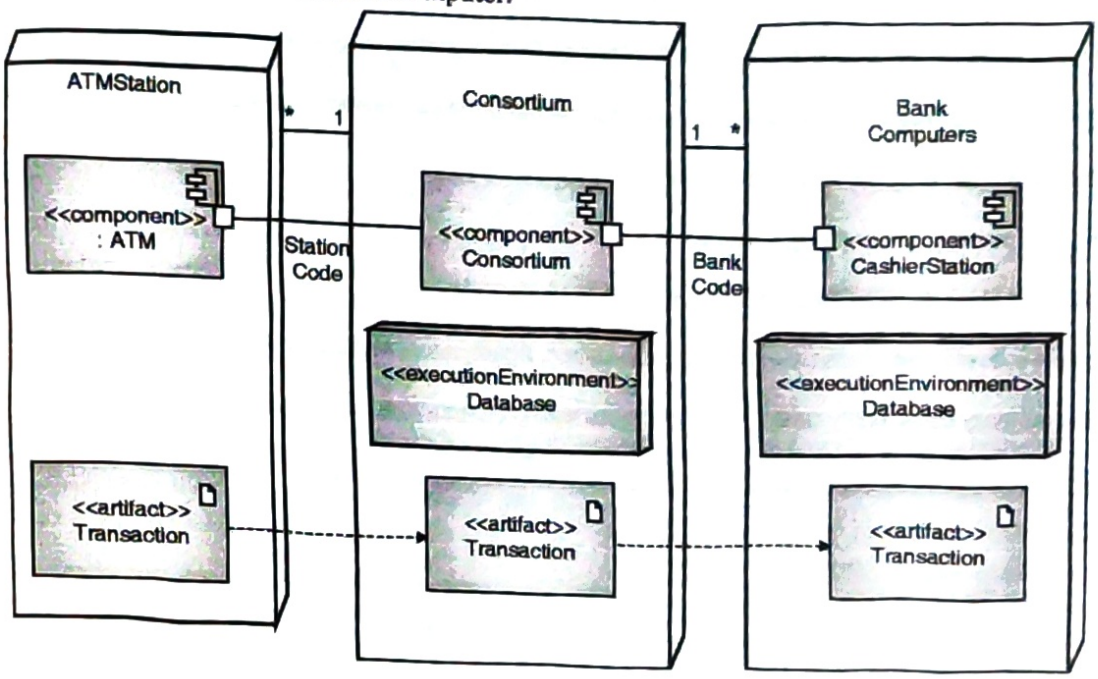

Fig. P.2.9.12 : Deployment diagram for ATM

Review Questions

Q.1 How can you identify association, generalization and aggregation relationship ? Illustrate.

Q.2 Explain following terms:

a) Association b) Aggregation

c) Multiplicity d) Generalization.

Q.3 Explain the following with suitable examples: Multiplicity, Generalization, Aggregation, and Extend.

Q.4 Explain with example modeling an association as a class.

Q.5 How can you identify classes and objects for particulars system? Illustrate different steps.

Q.6 Explain approaches for identifying classes.

Q.7 Explain boundary class, entity class, and control class with UML notations.

Q.8 Draw data structure (class diagram) for Hospital management system.

Q.9 Draw class diagram for Lift control system.

Q.10 Draw class diagram for computerizing the library of your college

Q.11 Explain in detail the process of developing class diagram.

Q.12 How can you identify association, generalization and aggregation relationship ? Illustrate.

Q.13 How can you identify classes and objects for particulars system? Illustrate different steps.

Q.14 Explain the following with suitable examples: Multiplicity, Generalization, Aggregation, Extend.

Q.15 What is static modeling ? Explain different steps that are performed in contracting a static model.

Q.16 Construct the class diagram for cellular phone.

Q.17 Explain approaches for identifying classes.

Q.18 Construct the class diagram for Cellular phone.

Q.19 Explain the following with suitable examples: Association, Aggregation, Multiplicity, Generalization, and Navigability.

Q.20 Explain boundary class, entity class and control class with UML notations.

❑❑❑

3 Dynamic Modeling

Syllabus

Activity diagram : Different Types of nodes, Control flow, Activity Partition, Exception handler, Interruptible activity region, Input and output parameters, Pins.

Interaction diagram : Sequence diagram, Interaction Overview diagram, State machine diagram, Advanced State Machine diagram, Communication diagram, Timing diagram.

3.1 Activity Diagram

- Activity diagrams represent the dynamics of the system. The activity diagram focuses on representing various activities or chunk of processing and their sequence of activation. The activities in general may not correspond to methods of classes.
- Activity diagrams are used to show the workflow of a system i.e. they show the flow of control from activity to activity in the system, what activities can be done in parallel, and any alternate paths through the flow.
- Activity diagram are similar to flow chart. The main difference is that activity diagrams can represent parallel activity and synchronization aspects involved in different activities.
- Activity diagrams may be created to represent the flow across use cases or they may be created to represent the flow within a particular use case. Activity diagrams may be created to show the workflow for an operation.
- Activity diagrams are a technique to describe procedural logic, business process, and work flow. Activity diagram describes the behaviour of a system in terms of activities.
- Activities are modeling elements that represent the execution of a set of operations. The completion of these operations triggers a transition to another activity.
- Activity diagrams are normally employed in business process modeling. This is carried out during requirements analysis and specification stage. Activity diagrams can be very useful to understand complex processing activities involving the roles played by many components. These diagrams can be used to develop interaction diagrams which help to allocate activities to classes.
- Activity diagrams make it possible to represent the three structural components of all procedural programming languages :
  - Sequences
  - Selections
  - Iterations
- These structural components equip the activity diagrams with the ability to model all aspects of behaviour in a system, from business procedures to operations of the classes. Hence, they are utilised within enterprise modeling, analysis, and design phases.