4.3 SECURITY

The System shall provide appropriate facilities to ensure that only authorised users have access to the information stored in the System.

Review Questions

Q. 1 What is SRS ? Explain need & benefits of SRS.

Q. 2 Explain the issues of requirement gathering ?

Q. 3 What are the techniques of requirement gathering ?

Q. 4 What is Requirements engineering?

Q. 5 Describe the requirements engineering tasks?

Q. 6 What is the need of Requirements' Validation ?

Q. 8 Explain Requirement Management.

Q. 9 List the various stakeholders involved in requirement analysis ?

Q. 10 Write an SRS for Maharashtra State Electricity Board. Assume the functional and non-functional requirements.

□□□

UNIT I

CHAPTER 4

Object-oriented Design using UML

Syllabus

Class diagram, Object diagram, Use case diagram, Sequence diagram, Collaboration diagram, State chart diagram, Activity diagram, Component diagram, Deployment diagram

4.1 Introduction

- Unified Modeling Language (UML) begins by constructing a model which is a simplification of reality.
- Model is an abstraction of some underlying problem.
- Model is a complete description of a system from any particular perspective constructed to understand the system before building the new or before modifying the existing system.

Static and Dynamic Models

- A *static model* is the snapshot of system's parameters at rest or at a specific point of time. For example, a customer could have more than one account at a time. Static models have stability and there is no change of data in future. The UML *class diagram* is a type of a static model.
- A *dynamic model* is a collection of behaviours of the system. It represents how objects interact with each other in order to perform any task. The UML *interactive* (*sequence and collaborative*) and *activity diagrams* are types of a dynamic model.

Benefits of a Model

1. Clarity

Models make it easier to. We can very easily express complex ideas using models as it allows visual examination of the whole system possible.

2. Complexity

Models reduce complexity by separating unimportant aspects from those that are important.

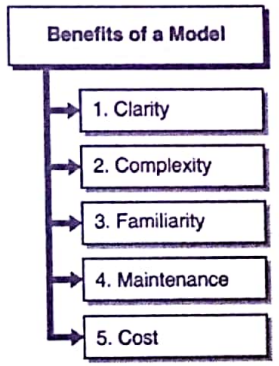


Fig. C4.1 : Benefits of a Model

➜ **3. Familiarity**

Models are the representation of how the information is actually represented so that you may understand the system and feel more comfortable to work.

➜ **4. Maintenance**

Visual identification and confirmation about the locations to be changed will reduce errors. Also manipulation (changing) of a model is more easier than manipulating a real system.

➜ **5. Cost :**

The cost of building and modifying a model is much lower as compared to the same task performed on a real system.

4.2 UML

- UML is a set of notations and conventions used to model an application to describe system's parameters and behaviours..
- The UML is an object-oriented methodology and technique because it is generally applicable to object-oriented problem solving.
- UML is not defined as 'data modelling' technique, but as an "object modelling" technique. Instead of entities, it models "object classes".
- Because of confluence in ideas, techniques, personalities, and politics; UML became a standard notation for representing the structure of data in the object-oriented community. It was developed when the three great personalities of the object-oriented community - James Rumbaugh, Grady Booch, and Ivar Jacobson agreed to adopt a standard notation for UML.
- UML was originally developed at Rational Software but is now administered by Object Management Group (OMG) i.e. responsible for creating and maintaining the UML language specifications.

OMG defines UML as, "a graphical language" for :

- **Visualizing :** There are some aspects of software system that you can not understand unless you build models.
- **Specifying :** models are precise, unambiguous, and complete.
- **Constructing :** allows direct connection to a variety of programming languages.
- **Documenting :** artifacts of a software system.

4.2.1 Features of UML

➜ **1. Syntax**

UML is just a language that tells what model elements and diagrams are available and the rules associated with them. It doesn't tell you what diagrams to create.

➜ **2. Comprehensive**

It can be used to model anything that may fill the user's requirement.

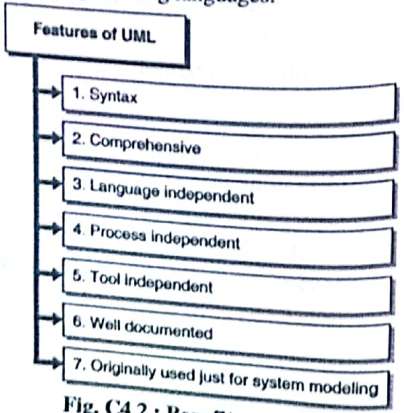


Fig. C4.2 : Benefits of a Model

➜ **3. Language independent**

It does not matter what high-level language is to be used in the code. Mapping into code is the responsibility of the case tool that you use.

➜ **4. Process independent**

The process by which the models are created is separate from the definition of the language.

➜ **5. Tool independent**

UML gives freedom for tool vendors to be creative and add value to visual modeling with UML.

➜ **6. Well documented**

The UML notation guide is available as a reference to all the syntax available in UML language.

➜ **7. Originally used just for system modeling**

Some user defined extensions are becoming more widely used now, for example for business modeling and web-based modeling.

4.2.2 Need of UML

- **Software construction needs a plan :** Software development is a very complex job and once a particular structure is in place, it is very difficult job to make any changes. UML helps in constructing a visual model of the real system.

  Example :A building architect creates a drawing of the building from more than one direction, ensuring that the structure and dimensions of the building are appropriate. Only when this visual model of the proposed work is approved, then the real construction of the building i.e. laying of bricks is begun.
- **Visualize in multiple dimensions and levels of detail :** As in the above example, UML, allows software to be visualized from multiple dimensions so that a computer system can be completely understood before construction begins.
- UML is also used to produce several models at increasing levels of detail. These increasing levels of detail can be added to each part of the software as it is constructed until final stage.
- **Appropriate for both new and legacy systems :** UML can be applicable for both new system developments and improvements in existing systems where only those parts that need modifications are modeled.
- **Documents software assets :** Undocumented or poorly documented software has a very low value as a company asset. Documentation saves the company from losses occurred by depending on key personnel who may leave at any time and thus improves understanding of the company's critical business processes.
- **Unified and universal :** OMG has made UML as a de-facto (actual) standard modeling language in the software industry. UML is called as a universal language because it can be applied in many areas of software development.
- **Inherent traceability :** The Use Case driven approach of UML modeling ensures that all levels of model trace back to elements of original functional requirements.
- **Incremental development and re-development :** UML models can be applicable in incremental development environment. It is not only possible to develop only those parts of the model that are required to satisfy the new requirements, but also possible to demonstrate that the code needed to fulfil these requirements is in place.
- **Parallel development of large systems :** Large complex UML models can be decomposed into simple models that can be developed independently by different people or groups simultaneously at the same time.

4.2.3 UML Advantages

- UML modelling is effective for large, complex software systems and also for modeling middleware.
- It is easy to learn and provides advanced features for expert analysts, designers and architects.
- It can specify the system in an implementation-independent manner.
- It allows re-usability i.e. 10-20% of the constructs are used 80-90% of the times.
- Structural UML modeling represents a skeleton that can be refined and extended with additional structure and behavior.
- Use case modeling specifies the functional requirements of the system in an object-oriented manner.
- Allows modelling of any type of application m running on any type of combination of hardware, operating system, programming language, and network.
- Increases efficiency and thus reduces cost and time-to-market.
- UML Profiles (i.e. subsets of UML designed for specific purposes) help to model Transactional, Real-time, and Fault-Tolerant systems in a natural way.

4.2.4 UML Diagrams

The most common 8 UML diagrams are :

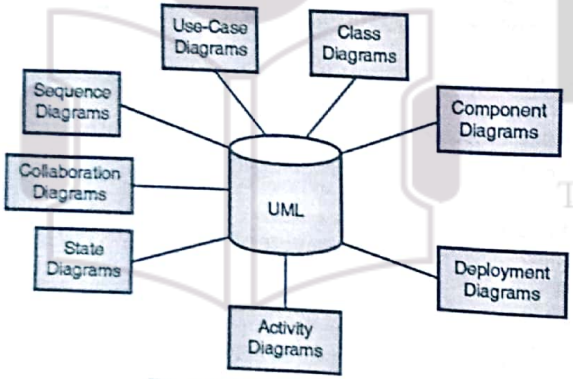


Fig. 4.2.1 : UML diagrams

Table 4.2.1 : Classification of Key UML Diagrams

1.	System Requirements	1. Use Case Diagram
2.	System Structure	2. Class Diagram 3. Interactive/Communication/Collaborative Diagram 4. Component Diagram 5. Deployment Diagram
3.	System Behavior	6. Interactive Sequence Diagram 7. State Diagram 8. Activity Diagram

Syllabus Topic : Use-case Diagrams

4.3 Use-case Diagrams

- Use case diagrams describe what a system does from an external observer's viewpoint. The focus is on **what** a system does rather than **how**.
- Use case diagrams are based on scenarios. A ***scenario*** is an example of ***what*** happens when someone interacts with the system i.e. it describes the course of events that may take place in the system.
- A Use Case Diagram shows relationships between ***actors, use cases*** and their ***associations*** (also called as interactions or communications) in a ***system***.
- Actors are represented using *stick figures,* Use cases by *ovals* and Communications by *lines* that link actors to use cases.
- A use case diagram is a visual representation of different *scenarios* showing the *association* between an *actor* (primary elements/business roles) and a *use case* (business processes that forms the system).

Example : A use case scenario for a medical clinic:

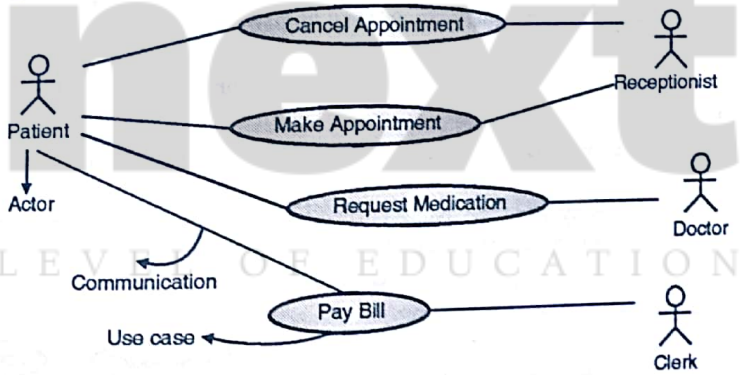


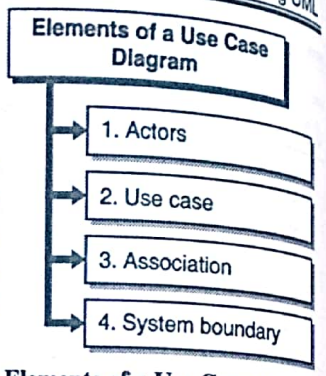
Fig. 4.3.1 : Use case diagram for a medical clinic

A patient calls-up in a clinic to get an appointment for monthly check-up. The receptionist finds out the nearest empty time slot in the appointment book and schedules the appointment for that time slot.

4.3.1 Need of Use Case Diagrams

1. **Determines requirements :** New use cases often generate new requirements as the system is analyzed and so on, the design takes shape. It is helpful in project planning, documentation and development of system's requirements
2. **Communicates with clients :** Notations in use case diagrams help developers to easily communicate and discuss with clients.
3. **Generates test cases :** Collection of scenarios in a use case leads to set of test cases for those particular scenarios.
4. **Represents complete functionalities :** Use cases represent complete functionality of the system or say, business process rules.
5. Discovers classes and relationships among subsystems of the system.

4.3.2 Elements of a Use Case Diagram



Use case diagram constitutes of following elements :

1. Actors

- An actor in a use case diagram is any entity that performs certain roles in a given system. These may be the business roles of users who perform certain functionalities in a given system. If some entity does not make any affect on a certain piece of functionality then, it makes no sense to represent it as an actor.

Fig. C4.3 : Elements of a Use Case Diagram

- An ***actor*** is who or what initiates the events involved in that task.
- Identify an actor in the proposed system by searching for business terms that represent certain roles.
- An actor is shown as a stick figure in a use case diagram portrayed outside the system boundary.

Example : In the statement "students visit the library to borrow books. Librarian updates the catalogue" - *student* and *librarian* are the roles that are identified as actors in the system.

Fig. 4.3.2 : Actors in a Library Use-case diagram

2. Use case

- A use case in a use case diagram is a visual representation of distinct business functionality in a given system.
- Identify the use cases by listing the unique business functions in your problem statement. Each of these business functions can be classified as a potential use case.
- A use case is shown as an ellipse in a use case diagram.

Example : In the same above statement about library, there are two functions - "Borrow book" and "Update catalog" that are identified as use-cases in the system.

Fig. 4.3.3 : Use cases in a Library Use case diagram

3. Association

- An association is an interaction between an actor and a use case.
- Associations between actors and use cases are indicated in use case diagrams by solid lines.
- Associations are shown by lines connecting use cases and actors to one another, with an arrowhead (arrowheads are very rarely used) on one end of the line. The arrowhead is used only when you need to indicate the direction of initial invocation or to indicate the primary actor within the use case.

Example : In the same above statement about library, there is an association between actor 'student' and use-case 'borrow book'.

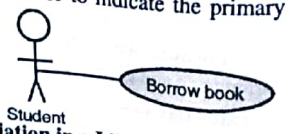


Fig. 4.3.4 : Example of an Association in a Library use case diagram

4. System boundary

- A system boundary of a use case diagram defines the limits of the system.

- A system boundary defines the scope of what a system will be. A system cannot have infinite use cases.
- The system boundary is shown as a rectangle spanning all the use cases in the system.
- System boundary of a system encloses the use cases of the system inside a rectangle and actors of the system outside the rectangle.

Example : In the same above statement about library, there is an association between actor 'student' and use-case 'borrow book'.

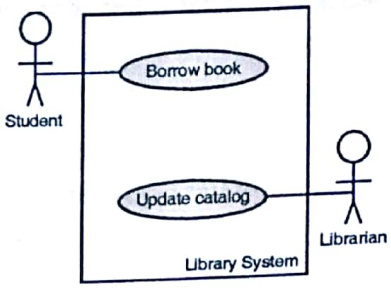


Fig. 4.3.5 : A Use case diagram showing the system boundary of a Library System

- But it is not that the system boundary represents the entire system always as in the above case - For large and complex systems, each of the modules may be enclosed in different system boundaries. And then the entire system can span all these modules finally into one system boundary.

Example :

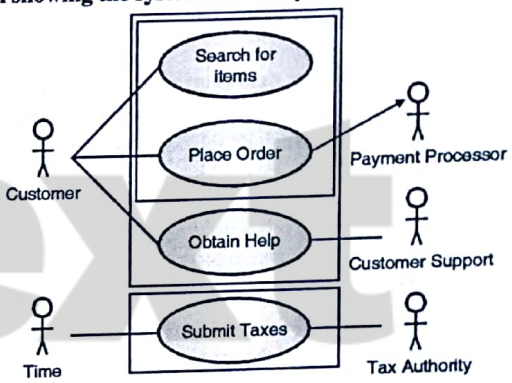


Fig. 4.3.6 : Example of System boundary boxes in use case diagram

- In the Fig. 4.3.6, Time is an external effect outside of the control of the system. When a certain time is reached that event is detected by the real-time clock that is inside the system and the use case starts automatically. It would be too difficult to hardcode the system always to start at a particular time, so normally an actor (i.e. Time) through another use case would be able to turn the automation on and off and set the time.

Example : Use-case diagram for Courseware Management System

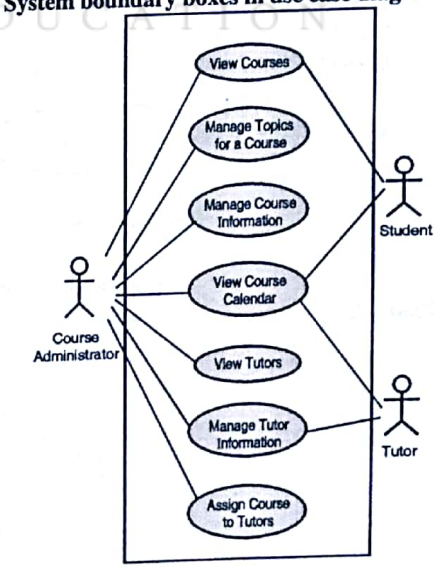


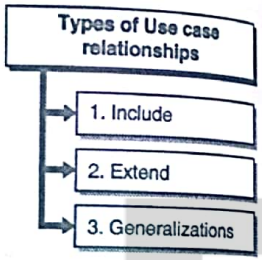
Fig. 4.3.7 : Use case diagram for Courseware Management System

4.3.3 Use Cases Relationships

- Use cases share different types of relationships. A relationship between two use cases is usually a dependency between those two use cases.
- Reusing the same use case in different types of relationships reduces the overall effort required in re-defining the use cases in a system.

Types of Use case relationships

Use-case diagrams support 3 types of relationships – *include, extend* and *generalization.*



1. Include

(Relationship from parent use case to inclusion use case)

- In this relationship, one use case references another use case.
- When one use case is using the functionality of another use case in a diagram, this relationship between the use cases is named as an include relationship.

Fig. C4.4 : Types of Use case relationships

- In an include relationship, a use case includes the functionality described in another use case as a part of its business process flow.
- An include relationship is depicted with a *directed arrow* having a *dotted shaft.*
  - The *base* of the arrowhead is the *parent use case.*
  - The *tip* of the arrowhead points to the *child use case.*
  - The *stereotype* "<<include>>" identifies the relationship as an include relationship.

Example : In library management system, the use case "Check for reservation" is contained within the "Borrow book" use case i.e. whenever the "Borrow book" use case executes, the scenario (business steps) defined in the "Check for reservation" use case is also executed.

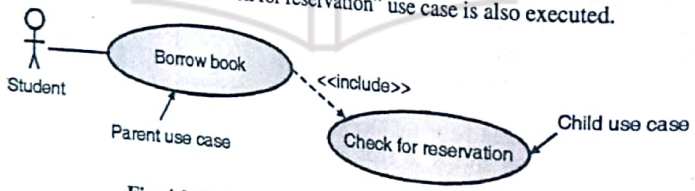


Fig. 4.3.8 : Example of <<include>> relationship

Example : Use case diagram for calculating in a Calculator depicting 'include' relationship. (Input is of the form "variable = expression")

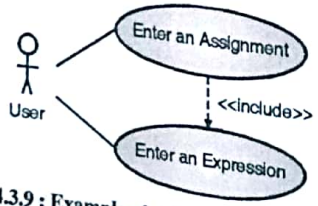


Fig. 4.3.9 : Example of <<include>> relationship

Example : Use case diagram for a Library Management System depicting 'include' relationship.

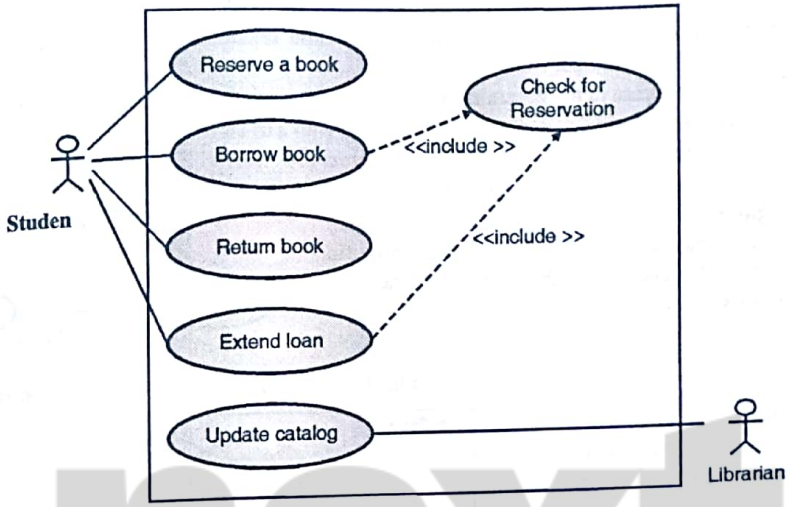


Fig. 4.3.10 : Use case diagram depicting 'include' relationship

2. Extend

(Relationship from extension use case to parent use case)

- Adds new behaviours or actions to a use case .
- This is used to add a use case that is similar to another use case but does a bit more or is more specialized.
- It expands the common behavior to fit the special circumstances.
- In an extend relationship, the child use case (extension 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. (Extend relationship shows pointing of arrow just opposite to include relationship)
  - The *tip* of the arrowhead points to the *parent use case.*
  - The *base* of the arrowhead is the *child use case.*
  - The *stereotype* "<<extend>>" identifies the relationship as an extend relationship.

Example : The *extend* relationship between the "Distribute Fee Schedule" (parent) and "Distribute information to students" (extended) use cases. The "Distribute information to students" use case adds to the functionality of the "Distribute Fee Schedule" use case. The "Distribute information to students" use case is a specialized version of the generic "Distribute Fee Schedule" use case.

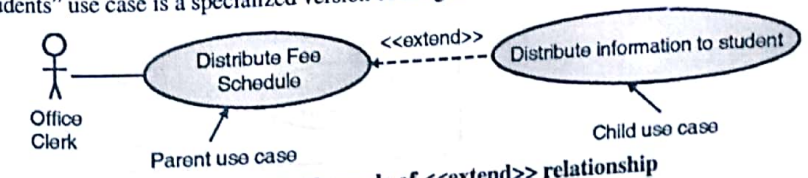


Fig. 4.3.11 : Example of <<extend>> relationship

3. Generalizations

(Relationship from a general use case to more specific use case)

- A generalization relationship is also a parent-child relationship between use cases like the above two relationships.
- The child use case in the generalization relationship is an enhancement of the parent use case.
- Generalization is depicted with a directed arrow with a triangle arrowhead from child to parent.
  - The *tip* of the arrowhead points to the *parent use case.*
  - The *base* of the arrowhead is the *child use case.*

Example : The *generalization* relationship between "transaction" (parent or more specific usecase) and "deposit", "withdrawal" and "transfer" (generalized) use cases.

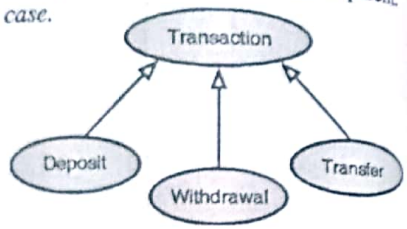


Fig. 4.3.12 : Example of a generalization relationship

Example : Use case diagram of ATM System depicting all the three relationships –include, extend and generalization.

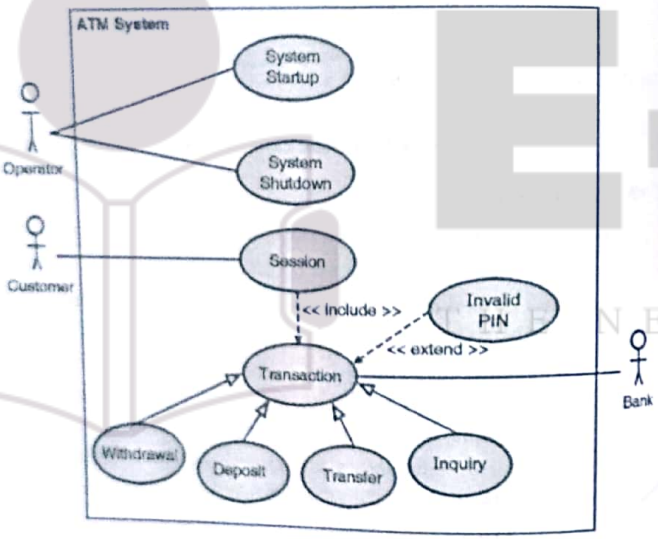


Fig. 4.3.13

Example : Use case diagram of Courseware Mgmt. System depicting all the three relationships – include, extend and generalization.

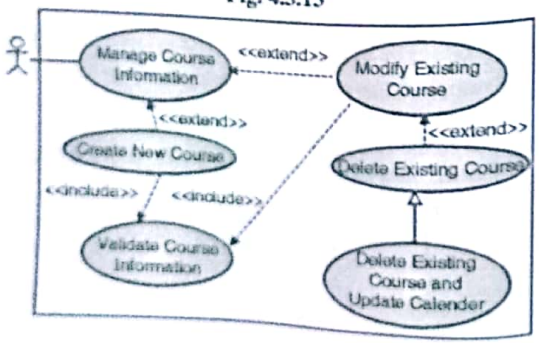


Fig. 4.3.14

Example : Use case diagram of Purchase Order System depicting all the three relationships – *include, extend and generalization.*

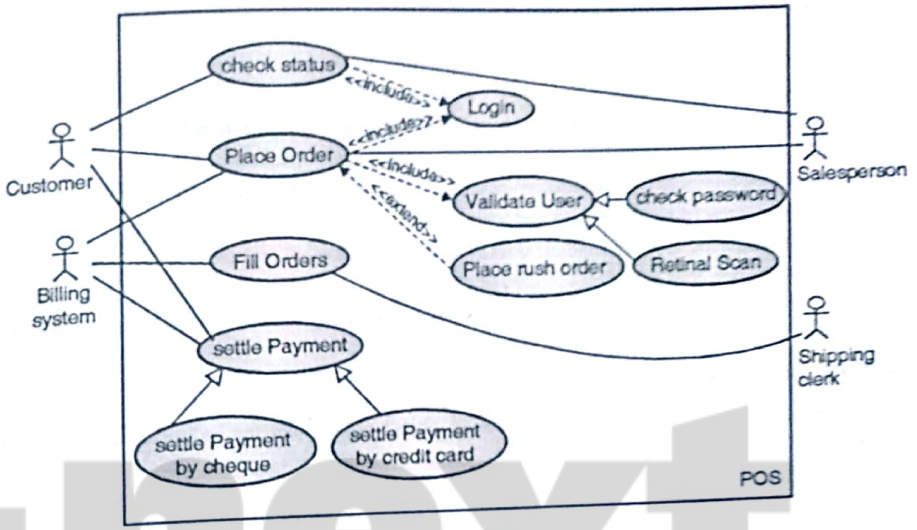


Fig. 4.3.15

Example : Use case diagram for Medical Clinic depicting all the three relationships –include, extend and generalization.

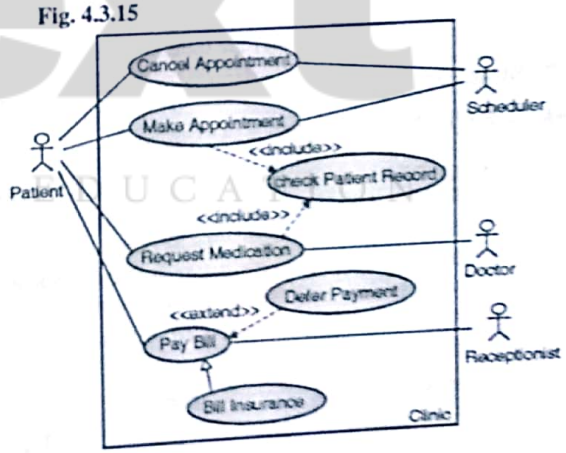


Fig. 4.3.16

Syllabus Topic : Class Diagram

4.4 Class Diagram

- This type of model includes classes, objects, attributes, operations and packages. We will see how to identify and design all these things.
- Here we will see the brief overview and various examples of class diagrams.

4.4.1 Identifying Classes and Objects

First, examine the problem statement (use cases) and then you can determine the classes by identifying each noun and entering it into a simple table. Classes mark themselves in one of the following ways:

- External entities (other systems, devices, and people) that produce or consume information to be used by a computer based system.
- Things (reports, displays, letters, signals) that are part of the information domain for the problem.
- Occurrences or Events that occur within the context of system operation.
- Roles (manager, engineer or salesperson) played by people who interact with the system.
- Organizational units (division, group and team) that is relevant to an application.
- Places (manufacturing floor or loading dock) that establish the context of the problem.
- Structures (Sensors, four wheeled vehicles or computers) that define a class of objects.
- To draw any type of UML diagrams, first we have to identify the classes and objects. And this is possible by studying the classification theory.

Classification Theory

- Classification is about identifying a class of an object rather than the individual objects within a system.
- Classification is the process of checking if an object belongs to a category or a class.
- Classes are important mechanism for classifying objects. The chief role of a class is to define the attributes, methods and its instances (objects).
- Classes are important as they are the conceptual building blocks for designing systems.

Strategies (Approaches) for identifying objects (Classes)

4.4.1(A) Noun Phrase Approach

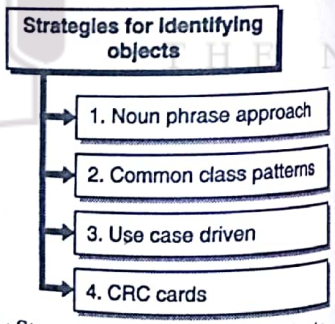


Fig. C4.5 : Strategies for identifying objects

- This approach was proposed by Rebecca Wirfs-Brock, Brian Wilkerson and Lauran Wiener.
- According to this approach, *Nouns are considered as candidate classes and verbs as its methods.* Nouns are converted into singular if they are in plural form.
- Nouns (Candidate classes) are divided into 3 categories :
  1. Relevant classes
  2. Irrelevant (can be skipped) classes
  3. Fuzzy classes
- Irrelevant classes do not have any purpose. They are unnecessary, so it is safe to slip it.
- Candidate classes are selected only from other 2 categories either by adding or removing some classes as necessary.

Guidelines for selecting 'tentative classes' in an application

- Circle or underline the nouns and noun phrases that occur in the requirements document(s); these become candidate classes (objects).

Example

- The system will keep track of membership information
- The system will manage inventory
- The system will facilitate the selling of bicycles

- All classes must make sense in the application domain; avoid computer implementation classes – place them to the design stage.

Guidelines for selecting 'candidate classes' from Relevant and Fuzzy Classes

You can challenge the initially selected candidate classes based on the following class rules.

- **Redundant classes :** Two classes cannot have same information. If more than one word is used to define the same data, select the one that is more meaningful in context of the system.
- **Adjective classes :** An adjective suggests a different kind of object, different use of same object or it could be utterly irrelevant. If the use of adjective signals that the behaviors of the object is different then make new class.

  Example : Adult members behave differently than Teenage members, so the two should be classified as different classes.
- **Attribute classes :** Tentative objects that are used only as values should be defined or restated as attributes and not as class.

  Example : Client Status and Demographic of Client are not classes but attributes of the Client class.
- **Irrelevant classes :** Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose. Simply eliminate the candidate class.

The process of identifying relevant classes and eliminating irrelevant classes is an incremental approach. You can move back and forth among these steps as often as you like. This can be shown as in Fig. 4.4.1

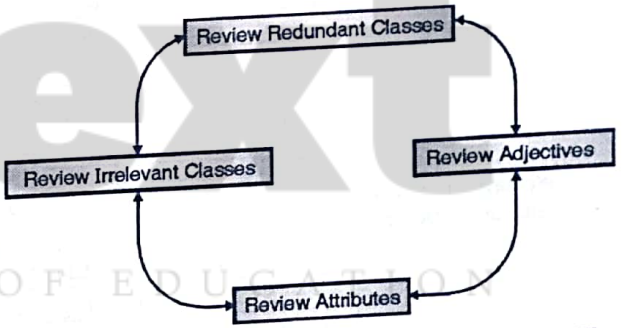


Fig. 4.19 : Moving back and forth among these classes (steps) as often as you like

Disadvantages

- Depends on the completeness and correctness of the requirements document which is rarely possible.
- Large volumes of text on system documentation might lead to too many candidate classes.

Example : Easy Money bank ATM system's requirements.

- The bank client must be able to deposit an amount to and withdraw an account from his or her accounts using the touch screen at the Easy Money ATM Machine. Each transaction must be recorded, and the client must be able to review all transactions performed against a given account. Recorded transactions must include the date, time, transaction type, amount and account balance after the transaction.
- An Easy Money bank client can have two types of accounts : checking account and savings account. For each account, one related savings account can exist.
- Access to the Easy Money bank accounts is provided by ATM Card having PIN code consisting of four integers between 0 and 9.

- One PIN code allows access to all accounts held by a bank client.
- No receipts will be provided for any account transactions.
- The bank application operates for a single banking institution only.
- Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw cash from a related savings account if the requested withdrawal amount on the checking account is less than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified with a message.

Solution : Initial study produces the following noun phrases :

Account	Client's Account
Account Balance	Four Digits
Amount	Message
ATM Card	Password
ATM Machine	PIN
Bank	PIN Code
Bank Client	Receipts
Card	Record
Cash	Savings
Checking	Savings Account
Checking Account	System
Client	Transaction

We have to select candidate classes from relevant and fuzzy class category, so it's safe to eliminate the irrelevant classes. Strike out the eliminated classes.

Account	Client's Account
Account Balance	Four Digits (irrelevant class)
Amount	Message
ATM Card	Password
ATM Machine	PIN
Bank	PIN Code
Bank Client	Receipts
Card	Record
Cash	Savings
Checking	Savings Account
Checking Account	System
Client	Transaction

Now, according to the redundant class guideline, eliminate the classes that describe the same idea. We have to select class i.e. most meaningful. Thus, reviewed list of candidate classes is :

Account	Client's Account (Account, Client's Account = Account)
Account Balance	Four Digits (irrelevant class)
Amount	Message

Account	Client's Account (Account, Client's Account = Account)
ATM Card	Password
ATM Machine	PIN (PIN, PIN Code = PIN Code)
Bank	PIN Code
Bank Client	Receipts
Card (ATM Card, Card = ATM Card)	Record
Cash	Savings (Savings, Savings Account = Savings Account)
Checking (Checking, Checking Account = Checking Account)	Savings Account
Checking Account	System
Client (Client, Bank Client = Bank Client)	Transaction

Now, according to the adjective class guideline, create new classes if the classes behave differently with their adjective. But, in this example, there are no adjectives with any of the classes. Thus, there is no change. Now, according to the attribute class guideline, eliminate the nouns that are attributes, not classes.

Account	Client's Account (Account, Client's Account = Account)
Account Balance (An attribute of the Account class)	Four Digits (irrelevant class)
Amount (A value, not a class)	Message (A value, not a class)
ATM Card	Password (An attribute of the Bank Client class)
ATM Machine	PIN (PIN, PIN Code = PIN Code)
Bank	PIN Code (An attribute of the Bank Client class)
Bank Client	Receipts
Card (ATM Card, Card = ATM Card)	Record
Cash	Savings (Savings, Savings Account = Savings Account)
Checking (Checking, Checking Account=Checking Account)	Savings Account
Checking Account	System
Client (Client, Bank Client = Bank Client)	Transaction

Now, reviewing the class purpose, each class must have a purpose; else eliminate that class which adds no purpose to the system. Cash and Receipts do not add purpose to the system. The final candidate classes are :

Account
(It is an abstract class that defines the common behaviors that can be inherited by checking or savings account)
ATM Card
(Provides a client with a key to access his account)
ATM Machine
(Provides interface to EasyMoney Bank)
Bank
(Bank clients belong to the bank)
Bank Client
(A client is an individual who has an account in the bank)
Checking Account
(Provides more specialized withdrawal service to the client's account)
Savings Account
(Models a client's savings account)
Transaction
(Keeps track of transaction, time, date, type, amount and balance)

4.4.1(B) Common Class Pattern Approach

The common class patterns approach is based on knowledge of common classes that has been proposed by researchers.

The patterns used for finding the candidate class and object using this approach are :

Pattern Name	Description	Example
Concept class	Principles or ideas. Are non-tangible	Performance
Events class	Things that happen at a given date and time, or as steps in an ordered sequence. These are associated with attributes such as : who, what, when, where, how or why.	Landing, Request, Interrupt, Order
Organization class	Formally organized collections of people, resources or facilities having a defined mission.	Departments
People class	Different roles that users play in interacting with the application. Two categories of people class : 1) People who use the system. 2) People who do not use the system but about whom information is kept by the system.	Teachers, Students, Employees
Places class	Areas set aside for people or things. Physical locations that the system must keep information about.	Travel Office, Buildings
Tangible things and devices class	Physical objects that are tangible and devices with which the application interacts	Cars, Sensors

Disadvantages

- Loosely bound to user requirements.

- Naming misinterpretations are possible.

Example : Take the same above Easy Money Bank ATM system. We can classify the nouns based on common class pattern approach as follows :

Pattern Name	Classes
Concept class	Not Applicable
Events class	Account, Checking Account, Savings Account, Transaction
Organization class	Bank
People class	Bank Client
Places class	Not Applicable
Tangible things and devices class	ATM Machine

4.4.1(C) Use Case Driven Approach

- Use-case modeling is a problem-driven (function-driven) approach in which the designer first considers the problem (functions or say use-cases) rather than considering the relationship between objects.
- This approach helps to understand the behavior of the system's objects.
- Use cases are used to model the scenarios in the system and specify which external actors interact with the scenarios. The scenarios are described through sequence of steps.
- The designer then examines the steps of each scenario to find out what objects are needed for the scenario to occur.
- At least one scenario must be prepared for each different use-case instance. Each scenario shows different sequence of interaction between actors and the system.
- Designer walks through each scenario to identify the objects, responsibilities of each object and how these objects interact with each other.

Example : Consider the same above case study of Easy Money Bank.

Assume a use case scenario, say 'Invalid PIN'

Use-case name : Invalid PIN

Step 1 : Insert ATM Card
Step 2 : Request PIN
Step 3 : Insert PIN
Step 4 : Verify PIN
Step 5 : If invalid PIN, display bad message
Step 6 : Eject ATM Card

Based on these activities, we find that the classes that interact with each other are : Bank Client, ATM Machine and Bank Operator.

Disadvantage

Relies on the completeness of use case scenarios.

4.4.1(D) CRC Approach

- A Class Responsibility Collaborator (CRC) model was developed by Beck and Cunningham in 1989, Wilkinson in 1995, Ambler in 1995.

- CRC is an effective way to identify classes based on the analysis of how objects collaborate to perform business functions (use cases).
- CRC Card Modeling is a simple but powerful object-oriented hands-on analysis technique.
- It was first developed as a teaching tool, then used as a collaborative technique to involve users, analysts and programmers in the design process
- It is a collection of standard index cards that is divided into 3 sections : *class name, responsibilities and collaborators.*

Class Name	
Responsibilities (attributes and operations)	Collaborators (relationships)

Fig. 4.4.2 : Three sections of CRC (4" × 6" card)

1. A ***class name*** represents a collection of *similar objects*. It appears in upper left hand corner.

Example : In a university system, classes would represent students, professors, and seminars. The name of the class appears across the top of a CRC card and is typically a singular noun or singular noun phrase, such as *Student*, *Professor*, and *Seminar*.

2. A ***responsibility*** is something that a class *knows* or *does*.

Example : Students have names, addresses, and phone numbers. These are the things a student *knows*. Students also enroll in seminars, drop seminars, and request transcripts. These are the things a student *does*. The things a class knows and does constitute its responsibilities. Also, a class can change values of the things it knows, but it is unable to change the values of what other classes know.

3. A ***collaborator*** is another class that a class interacts with to fulfil its responsibilities. Sometimes a class has a responsibility to fulfil, but not have enough information to do it. For example, students enrol in seminars. To do this, a student needs to know if a spot is available in the seminar and, if so, he then needs to be added to the seminar. However, students only have information about themselves (their names and so forth), and not about seminars. What the student needs to do is collaborate/interact with the card labelled *Seminar* to sign up for a seminar. Therefore, *Seminar* is included in the list of collaborators of *Student*.

Student	
Name Address Phone Number Enroll in Seminar Drop a Seminar Request transcripts	Seminar

Fig: 4.4.3 : Student CRC card

Creating CRC Models

The process of creating CRC model consists of three steps :

Step 1: Identify classes' responsibilities (and identify classes)

- Classes are identified and grouped by common attributes, which also provide candidates for superclass.
- Once you have identified potential classes (e.g. by finding nouns in the requirements specification), you use one card for each class.
- The class names are written on the CRC cards. The card also notes sub and superclass to show the class structure.

Step 2: Assign responsibilities

- On this card, you write down the 'responsibilities' of that class. These are the potential methods.
- Responsibilities are distributed. They should be as general as possible and placed as high as possible in the inheritance hierarchy.

Step 3: Identify collaborators

- Once you agree that the classes & responsibilities are OK, you find out for each class, which other classes it need to collaborate with and write this on the card.

  Collaboration is done either 1) to request for information or 2) to request to perform a task

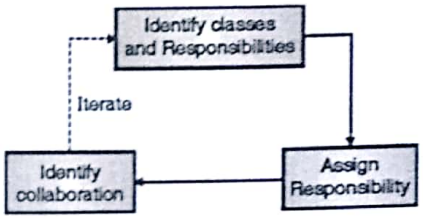


Fig. 4.4.4: CRC Process

Move the Cards around

To help everyone to understand the system, the cards must be placed on the table in an intelligent manner so that two cards that collaborate with each other must be placed close together, whereas two cards that don't collaborate must be placed far to each other.

In this manner, the more two cards collaborate; the closer they should be on the table. By placing the cards that collaborate with one another close together, it is easier to understand the relationships between classes.

customer	
Name Address Phone Number Place order Knows order History	order

order	
Order Date Delivery Date Order Number Order Items Make Total Applicable Taxes	Items

Fig. 4.4.5: CRC Cards of 'Purchase Order'

Example : Few CRC Cards of Purchase Order.

Example : CRC cards for University Enrollment.

Student	
Student Name Address Phone Number Enroll in Degree	Degree Course

Course	
Course Code Course Name	Degree

Degree	
Degree Code Degree Name	Study Program

StudyProgram	
Years Semesters	

Fig. 4.4.6 : CRC Cards for 'University Enrollment'

Example : CRC cards for Seminar Enrollment.

Student	
Student Name Address Phone Number Email-Id Marks Received Validating Information	Enrollment

Enrollment	
Student Name marks Grade Seminar Topic	Seminar

Seminar	
Name Seminar Number Fees Waiting list Enrolled Student Instructor Add Student Drop Student	Student Professor

Professor	
Professor Name Address Phone number Email-Id Salary Provide Information Instructing seminars	Seminar

Fig. 4.4.7 : CRC Cards for 'Seminar Enrollment'

We need to identify the responsibilities because

- Responsibilities identify problems that are to be solved.
- A responsibility serves as a handle for discussing potential solutions.
- Once the system's responsibilities are understood, we can start identifying the attributes of the system's classes. Note that, responsibilities include *attributes* and *methods* of the class.

4.4.2 Identifying Attributes

Attributes correspond to nouns followed by preposition phrases or to adjectives or adverbs.

- Omit derived attributes - They should be expressed as a method.

Example : In the same Easy Money ATM system example, we will state the attributes of Account, ATM Machine, Bank Client and Transaction classes.

Account	ATMMachine	BankClient	Transaction
number balance	address state	firstname lastname pinnumber cardnumber accountnumber	transID transDate transTime transType amount post Balance

Fig. 4.4.8

4.4.3 Defining Methods

- Methods are defined as services that the objects must provide.
- Methods are the events that occur between objects of the class.
- An event is considered to be an action that transmits information; therefore these are the operations that the objects must perform.
- Methods also can be derived from the use case scenario testing.

Example : In the same EasyMoney ATM system example, we will state the methods of Account, ATM Machine, BankClient and Transaction classes.

Account	ATMMachine	BankClient	Transaction
number balance deposit() withdraw() createTransaction()	address state	fristname lastname pinnumber cardnumber verifyPassword()	transID transDate transTime transType amount postBalance

Fig. 4.4.9

4.4.4 Finalizing the Object (Class) Definition

Table 4.4.1 : Various ways of naming a Class

Description	Example
Use a noun or noun phrase	my latest books
Should be singular and not plural	my latest book
Avoids possessives	a latest book
Doesn't contain irrelevant adjectives	a book
Uses initial capital letters	A Book
Doesn't have spaces between words	ABook
Doesn't contain articles (a, an, the) pronouns	Book

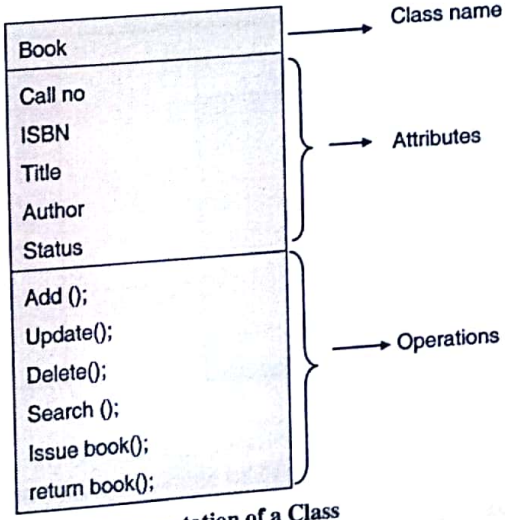


Fig. 4.4.10 : Representation of a Class

Example: Class Diagram of Library Management System

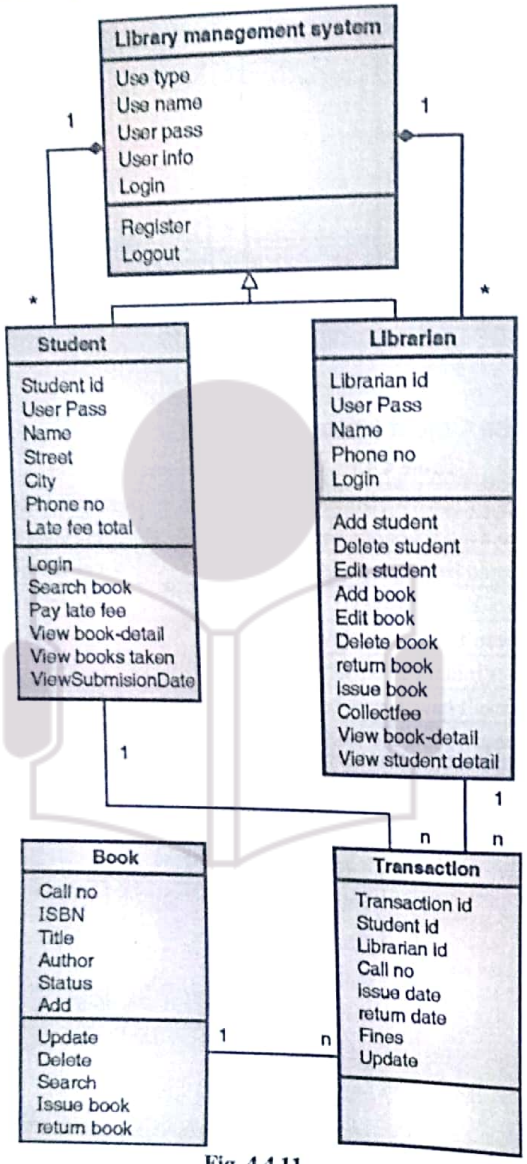


Fig. 4.4.11

Example : Draw class diagram for Private course management system where it has various courses. Each course consists of various topics. Course has course calendar and It is managed by course administrator. Course administrator can manage some tutors for each course. Many students can be admitted in each course.

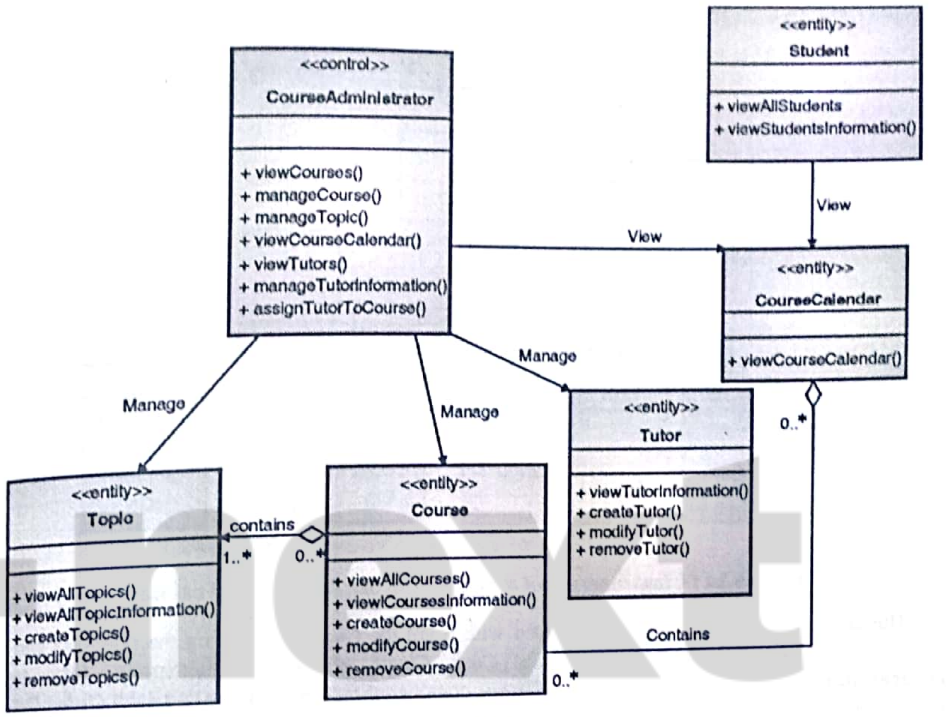


Fig. 4.4.12

Example: Draw class diagram for online shopping.

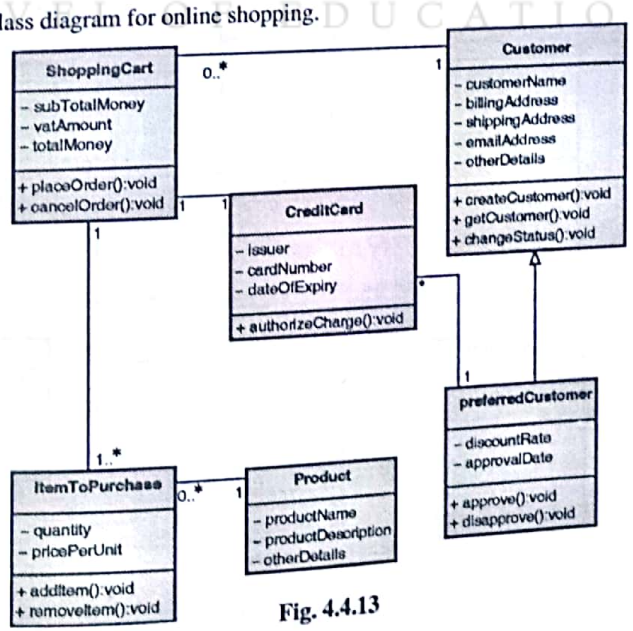


Fig. 4.4.13

Example : The class diagram of a customer order from a retail catalog.

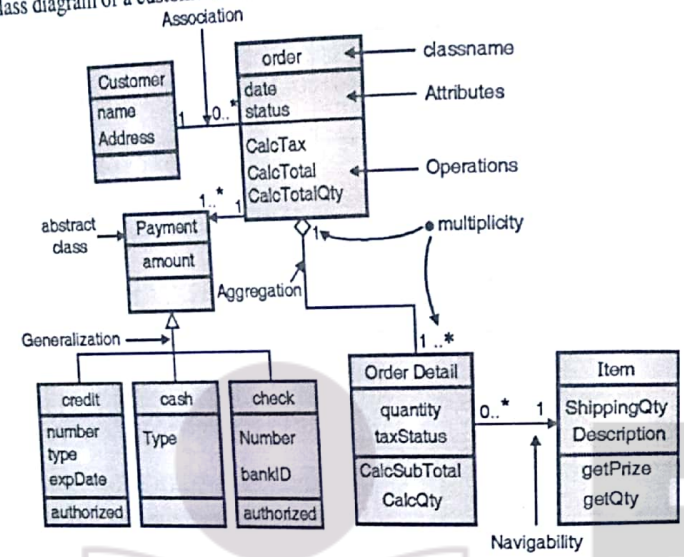


Fig. 4.4.14 : Class diagram of a customer order from a retail catalog

The central class is the *Order*. **Associated** with it are the *Customer* making the purchase and the *Payment*. A *Payment* is one of three kinds: *Cash*, *Check*, or *Credit* demonstrating **inheritance (generalization)**. The *Order* contains *Order Details* demonstrating **aggregation** and each *Order Details* is associated with *Item*.

Example : Draw class diagram for Seminars given by students from different courses.

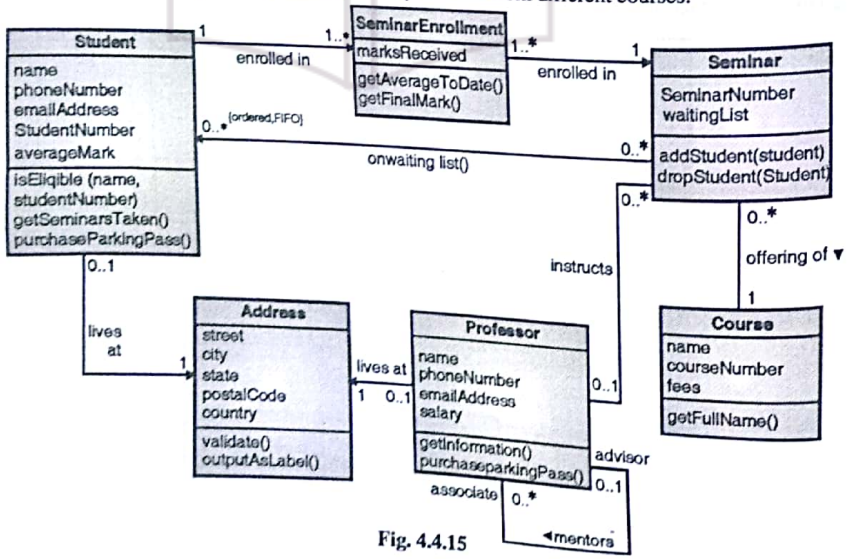


Fig. 4.4.15

Example : Class diagram for Hospital management where it has various departments. If each department consists of various doctors out of which, one of the doctors is the head of the department. Patient takes treatment from Doctors where treatments are based on Laboratory Result. Patient can be an Indoor Patient or Outdoor Patient. Hospital has various operation theatres used for various operations by various doctors on indoor patient.

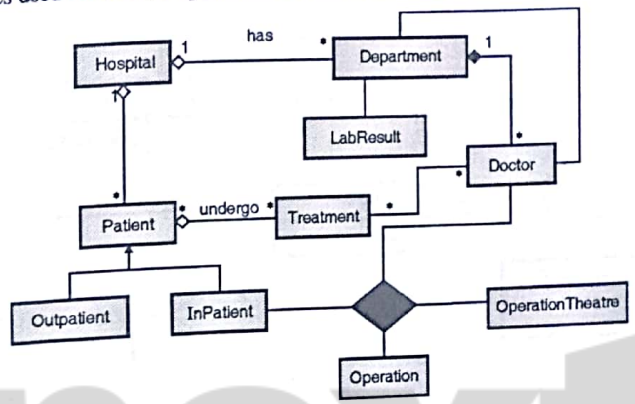


Fig. 4.4.16

Example: Class diagram for word processor.

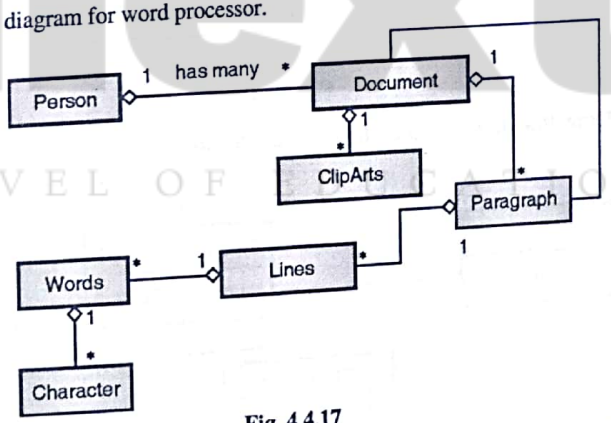


Fig. 4.4.17

Syllabus Topic : Object Diagram

4.5 Object Diagram

Here we will see just the various examples.

Example : Draw the object diagram for Hospital management system.

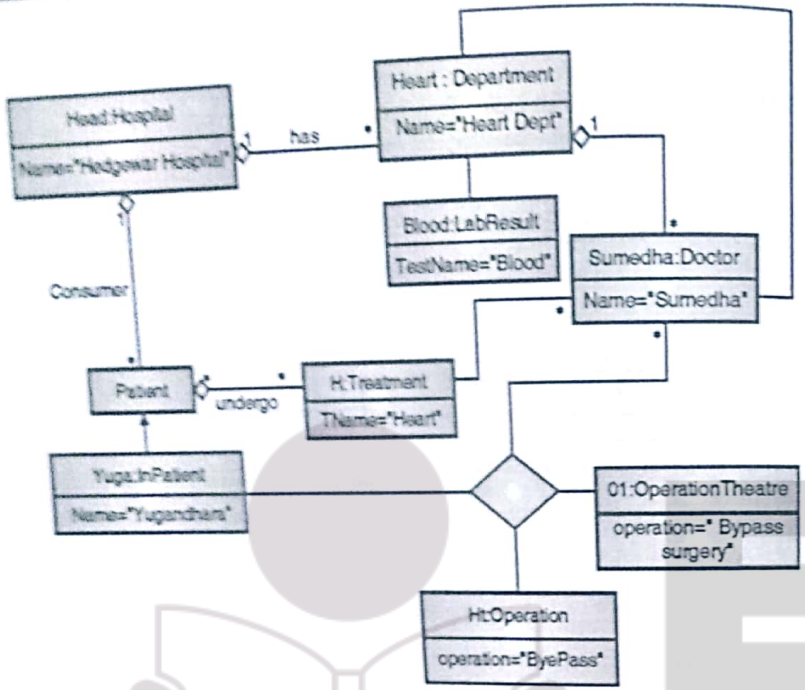


Fig. 4.5.1

Example : Draw the object diagram for word processor.

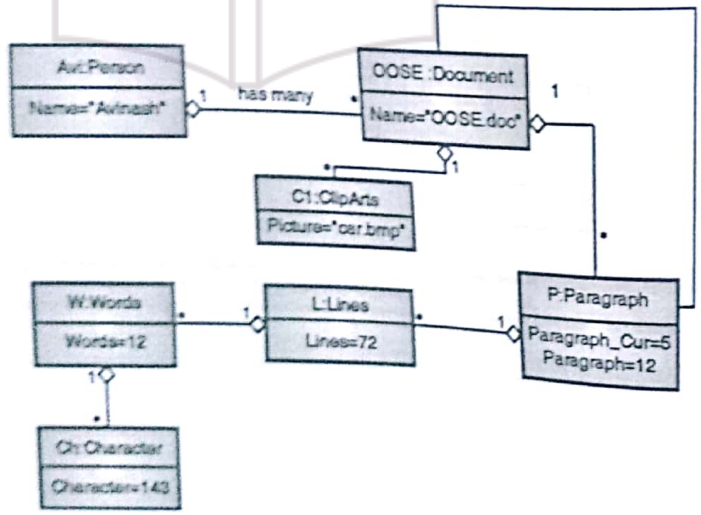


Fig. 4.5.2

Example : Draw the Object diagram for car rental system.

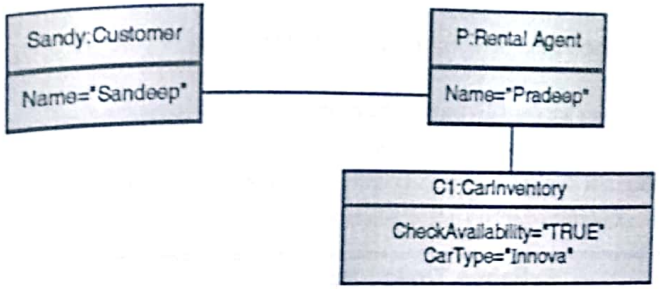


Fig. 4.5.3

4.6 Packages

- You draw a class model on a single page for many small and medium sized problems. But it is very difficult to understand the entirely large model, thus, it is better to partition the larger class models so that it is easily understood. These partitions can then be put into packages.
- A package is a group of elements (classes, associations, generalizations and smaller packages) with a common theme. Packages are given a representative name.
- Packages form a tree with increasing abstraction towards the root, which is the top level package or the entire application.

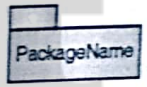


Fig. 4.6.1 : Notation for a package

Example : Package diagram for Seminar enrolment by students from different courses.

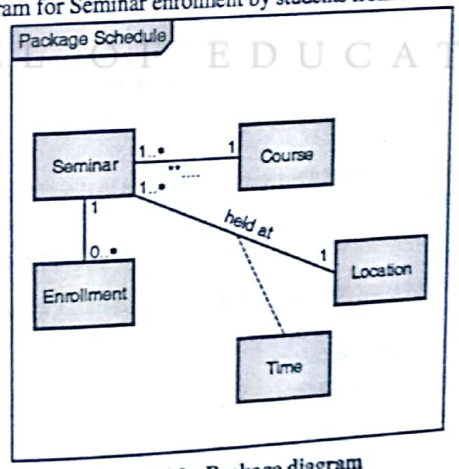


Fig. 4.6.2 : Package diagram

4.7 Interaction Diagram

- Describes the collaboration between groups of objects. The collaboration is required to get the job done.

- Used to model the behavior of several objects in a use case. This is possible using a Sequence diagram.
- Demonstrates collaboration between the different object. This is possible using a Collaboration diagram.
- The communications are partially ordered based on time or in a sequence.

UML supports two kinds of Interaction diagrams:

1. **Sequence Diagram** displays the time sequence of the objects participating in the interaction.
2. **Collaboration Diagram** displays interaction around the objects and their links to one another.

Syllabus Topic : Sequence Diagram

4.7.1 Sequence Diagram

- They represent how objects interact with each other to perform the behaviours as stated in a use case scenario.
- A sequence diagram shows a series of messages exchanged by a selected set of objects with emphasis on the chronological course of events.
- The focus is less on messages themselves and more on the **order** in which messages are exchanged.
- Sequence diagrams communicate what messages are sent between a system's objects as well as the order in which they occur.
- Sequence Diagrams are used primarily to design, document and validate the architecture, interfaces and logic of the system by describing the sequence of actions that need to be performed to complete a task or scenario.
- UML sequence diagram is a dynamic modelling technique because they provide a dynamic view of the system behavior which can be difficult to extract from static diagrams or specifications.
- Is used primarily to show the interactions between objects in the sequential order of their occurrence.

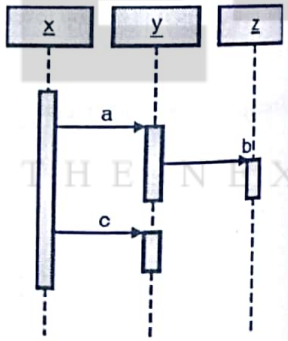


Fig. 4.7.1 : Sequence diagram

- They are used to validate and describe the logic of a ***usage scenario*** which may be :
  - part of a use case or
  - an entire pass through a use case or
  - a pass through several use cases.

Scenario

- A scenario is a sequence of events that occur during one particular execution (use case) of a system.
- A scenario can include all events in the system or can only include only those events that are generated by a certain object in that system.
- A scenario is usually structured with numbered steps in a sequence and calls to sub-sequences.

Example : Use case Scenario for phone call

1. Caller lifts receiver
   - 1.1 Dial tone begins
   - 1.2 Caller dials a digit
   - 1.3 Dial tone ends
   - 1.4 Caller dials remaining digits
2. Ringing tone starts at caller side and receiver also hears the phone ring
3. Receiver answers the phone
4. Ringing stops on caller and receiver side
   - 4.1 Phone is connected. Caller and Receiver talk to each other
5. Receiver hangs up.
6. Phone connection is broken on both sides.

Designing a sequence diagram

Two dimensions of Sequence diagram :

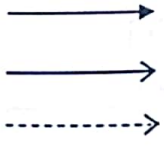
1. The *vertical dimension* (top to bottom) of the sequence diagram shows the ***time*** sequence of messages as they occur.
2. The *horizontal dimension* shows the ***object*** instances that the messages are sent to.

Diagram Elements in a Sequence diagram

- **Actor :** Represents an external person or entity that interacts with the system.

user

- **Object :** Represents an object in the system or one of its components.
- **Lifelines :** Each vertical *dotted line* is a lifeline representing the time limit that an object exists. A lifeline has a rectangle containing its object name. If its name is caller then the lifeline represents the classifier which owns the sequence diagram.
- **Messages:** Messages are displayed as arrows. Messages can be complete, lost or found; synchronous or asynchronous; call or signal. The synchronous message is denoted by the *solid* synchronous or asynchronous; call or signal. The synchronous message is denoted by the *solid arrowhead*, asynchronous message *by line arrowhead*, and the return message by *dashed line*. Each message is labeled with a message name.



Synchronous Message

Asynchronous Message

Return Message

- **Activation bar :** A message (an arrow) goes from the sender to the top of the *activation bar* of the receiver's lifeline. A thin rectangle running down the lifeline denotes the *execution occurrence*, or *activation bar* of a focus of control. The *activation bar* represents the duration of execution of the message.

Example : Sequence diagram for a Phone Call.

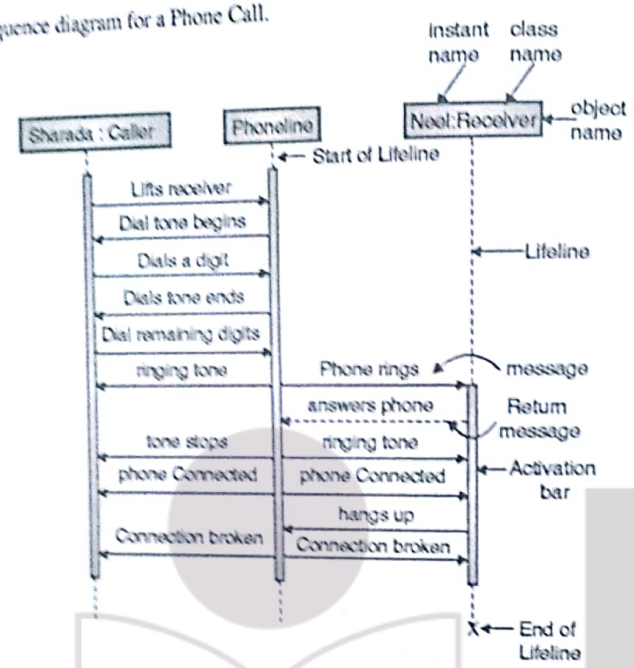


Fig. 4.7.2 : Sequence diagram for a Phone Call

Example : Simple Sequence diagram of Courseware Management System.

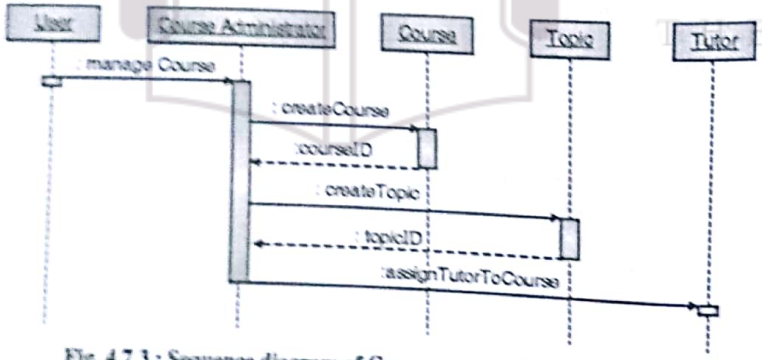


Fig. 4.7.3 : Sequence diagram of Courseware Management System

- **Self call** (Self-delegation) **:** It is a message that object sends to itself.
- **Lifeline Start and End :** A lifeline may be *created* or *destroyed* during the timescale represented by a sequence diagram.
  - To destroy, the lifeline is terminated by a stop symbol, represented as a *cross*.
  - And to create, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation.

- **Part Decomposition :** An object can have more than one lifeline coming from it. This allows for *inter-object* and *intra-object* messages to be displayed on the same diagram.

Example : Sequence diagram showing 'Part Decomposition'.

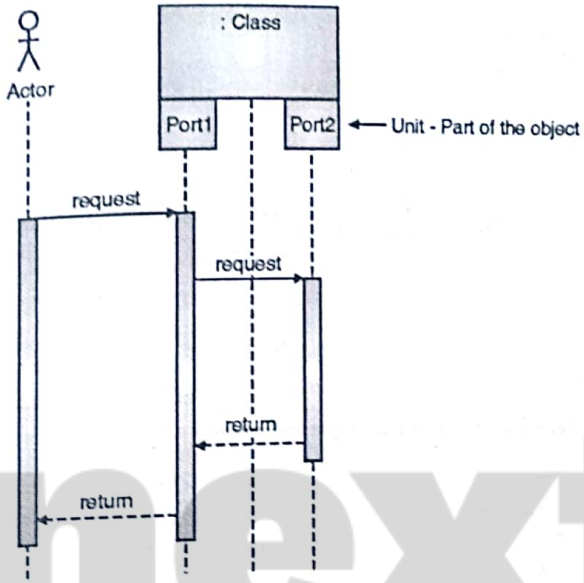


Fig. 4.7.4 : Sequence diagram showing 'Part Decomposition'

- **State Invariant:** A state invariant is a constraint placed on a lifeline that must be true at run-time. It is shown as a rectangle with semi-circular ends.

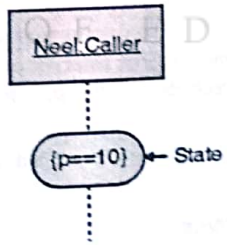


Fig. 4.7.5 : Showing state invariant

- **Condition:** The message is sent only if the condition is true.
  syntax: [expression] message-label
- **Iteration:** The message is sent many times to possibly multiple receiver objects.
  syntax: * [expression] message-label

Example : Sequence diagram for making a hotel reservation.

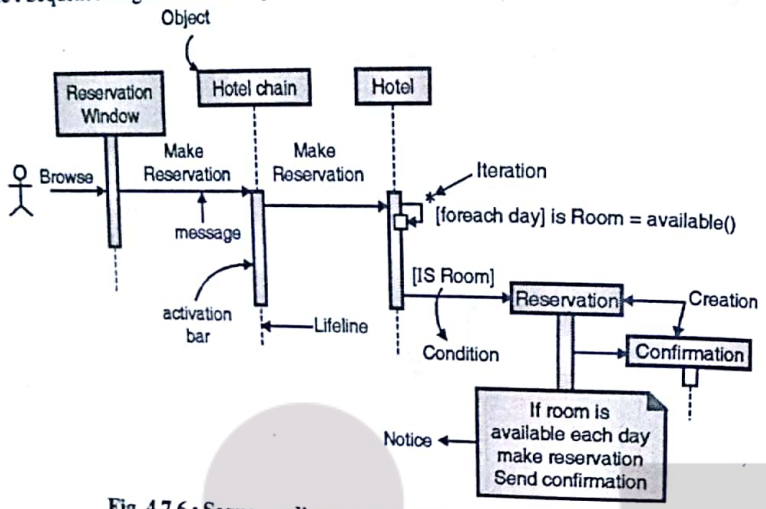


Fig. 4.7.6 : Sequence diagram for making a Hotel Reservation

- The object *(user interface)* of the class *Reservation window* initiates the messaging.
- The *Reservation window* sends a *make Reservation()* message to a *Hotel Chain*. The *Hotel Chain* then sends a make Reservation() message to a *Hotel*. If the *Hotel* has available rooms, then it makes a *Reservation* and a *Confirmation*.
- Every vertical *dotted line* is called as the **lifeline** that describes the time duration that an object exists for. Each *arrow* is called as a **message call** and it goes from the sender to the top of the **activation bar** on the receiver's lifeline. The *activation bar* represents the time duration of message execution.
- In our diagram, the *Hotel* issues a **self call** to determine if a room is available. If so, then the *Hotel* creates a *Reservation* and a *Confirmation*. The **asterisk** on the self call means **iteration** (to make sure there is available room for each day of the stay in the hotel). The expression in square brackets, [ ], is a **condition**.
- The diagram has a clarifying **note**, which is text inside a dog-eared rectangle. *Notes can be put into any kind of UML diagram.*

Advantages of Sequence diagrams

1. **Helps you to discover architectural, interface and logic problems early in SDLC process**

   This is specially true for systems involving interaction of components that are being implemented in parallel by different teams. In the phone call example, each task is implemented by a separate team. Having a set of sequence diagrams describing how the interfaces are actually used and what messages/actions are expected at different times gives each team a consistent implementation plan.

2. **Collaboration tool**

   - Sequence diagrams are valuable collaboration tools during design meetings because they allow you to discuss the design in concrete terms - you can see the interactions between entities, various state transitions and alternate cases on paper.

3. **Sequence diagram editor**

   - It become easy to edit your sequence diagrams - you can make the corrections during the meeting itself and instantly see the result of the changes as you make them.

4. **Documentation**

   - Sequence diagrams can be used to document the dynamic view of the system design at various levels of abstraction which is often difficult with static diagrams.

Syllabus Topic : Collaboration Diagram

4.7.2 Collaboration Diagram

- While sequence diagrams focus on the *interactions of objects over time*; the collaboration diagrams focus on the *relationships between objects.*
- Collaboration defines the communication pattern performed by set of instances.
- Collaboration diagram is almost similar to sequence diagram but only difference is that of *viewpoint* - Sequence diagram cannot illustrate the relation between objects where as collaboration diagram does.
- The collaboration diagram models how the interactions utilize the structure of participating objects and their relationships. It represents the relationship and interaction between software objects.
- Collaboration diagram illustrates messages being sent between classes and objects (instances). A diagram is created for each system operation that relates to the current development cycle (iteration).

Use of Collaboration diagram

- Collaboration diagrams are used in the systems where structure is important to concentrate on the effects of the instances.
- This shows interaction between classes but focuses on structural relationships between objects.

Designing a Collaboration diagram

- Wherever a message is to be sent, there must be a *link* i.e. an association must exist between the classes. Also, the association must be navigable in the required direction.
- *Actors* can be shown as in a use case diagram.
- Each ***object*** is shown as rectangle, which is labelled with either class or object name or both. If both class name and object name are used, Class names are preceded by colons (:) i.e. *objectName: className* (which was underlined in earlier versions; not now days).

Example: 'cat is a pet' where, cat is object and pet is class.

- Objects are linked through message paths. ***Links*** between objects are shown same like associations in the class model.
- *Messages* are attached to the message paths. Messages have a sequence number to denote time of occurrence. Each message in a collaboration diagram has a *sequence number*. The top-level message is numbered 1. Messages at the same level (sent during the same call) can have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur such as 1.1, 1.2...But it is easy to read simple numbering in collaboration diagrams than the decimal numbering. In a collaboration diagram numbering the message indicates the sequence.

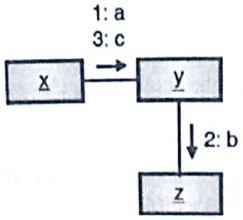


Fig. 4.7.7 : Collaboration diagram

Table 4.7.1 : Notations Used in Collaboration Diagram

Sr. No.	Notation	Use
1.		Actor :
2.	Alok: staff	Instance of class. Labeling as [instance name][: instance type]
3.	College:Staff	Multi Objects Labeling as [name][:type]
4.		Message Directions

Example : Collaboration diagram for making a Hotel Reservation.

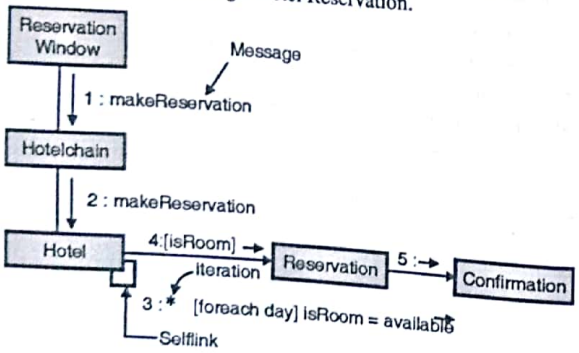


Fig. 4.7.8 : Collaboration diagram for making Hotel Reservation (simple numbering)

Example : Collaboration diagram for a Phone Call.

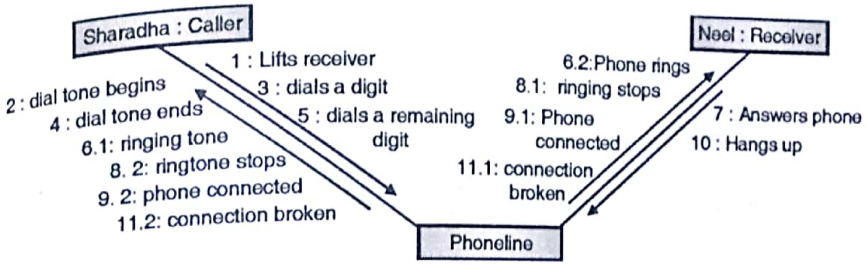


Fig. 4.7.9 : Collaboration diagram for a Phone Call (decimal numbering)

Example : Collaboration diagram for Courseware Management System.

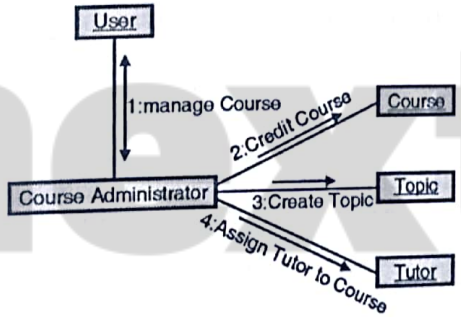


Fig. 4.7.10 : Collaboration diagram for Courseware Management System

Example : Collaboration diagram for Railway Reservation Systems :

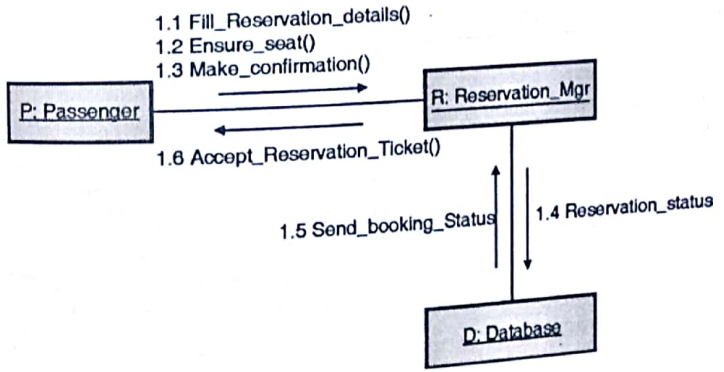


Fig. 4.7.11

Example : Collaboration diagram for E-Product Purchasing :

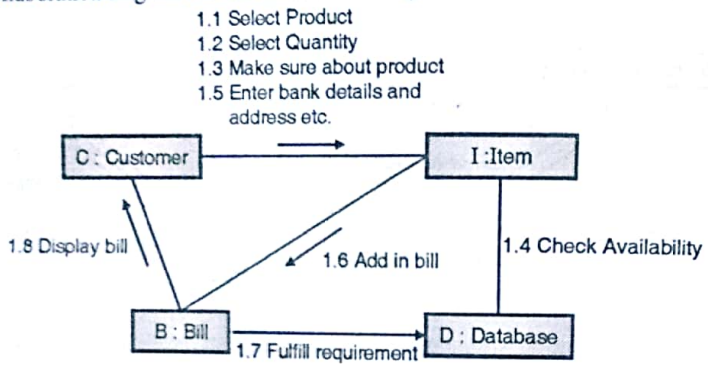


Fig. 4.7.12

Similarities and Differences in Sequence and Collaboration diagram

Similarities	1. Sequence & Collaboration diagrams together form the interaction diagrams.	
	2. These diagrams describe the same information, and can be transformed into one another.	
Differences	Sequence diagrams focus on the order in which the messages are sent.	Collaboration diagrams focus on relationships between objects.
	Sequence diagram is useful for describing the procedural flow through many objects, and for finding race conditions in concurrent systems.	Collaboration diagram is useful for visualising the way several objects collaborate to get a job done, and for comparing a dynamic model with a static model.

Syllabus Topic : State-chart Diagram

4.8 State-chart Diagram

- A **state** is a mode or condition of being.
- A state diagram is a dynamic model showing changes of state that an object goes through during its lifetime in response to events.
- State diagrams (also called State Chart diagrams) are used to help the developer better understand any complex functionality of specialized areas of the system.
- In short, State diagrams depict the dynamic behaviour of the entire system, or a sub-system, or even a single object in a system.

Designing a State Diagram

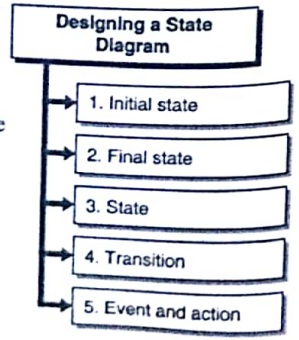


Fig. C4.6 : Designing a State Diagram

1. Initial state

The filled in circle shows the starting point or first activity of the diagram. This is also called as a **"pseudo state,"** where the state has no variables and no activities.

2. Final state

The filled circle with a border (bull's eye symbol) is the end of the state diagram. This is also a pseudo state because it does not have any variable or action described. A state diagram can have zero or more final states.

3. State

Represents the state of object at an instant of time. State is a recognizable situation and exists over an interval of time. This is denoted by a rounded rectangle and compartments within to describe state-name, variables, and activities.

state-name
variable
activities

4. Transition

A transition is a change from one state to other is indicated by an arrow. The *event* and *action* causing the transition are written beside the arrow, separated by a slash.

Event[Condition]/Action

- Transitions that occur when the state completed an activity are called **trigger less** transitions.
- If an event has to occur after the completion of some event or action, that event or action is called the guard condition (depicted by square brackets around the description of the event/action in the form of a Boolean expression). The transition then takes place after the guard condition occurs.

5. Event and action

A trigger that causes a transition to occur and changes the state is called as an event or action. An event occurs at a particular time has no duration. As described above, an event/action is written above a transition that it causes.

Example : State diagram for a phone call

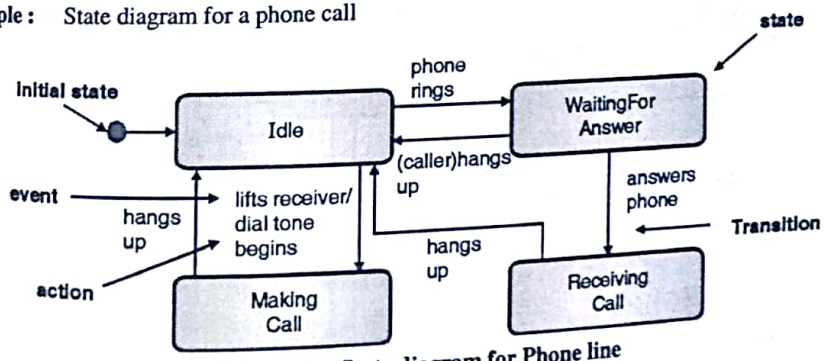


Fig. 4.8.1 : State diagram for Phone line

History states

A flow may require that the object go into wait state, and on the occurrence of a certain event, go back to the state it was in. This is shown with the help of a letter H enclosed within a circle.

Example : Showing History States.

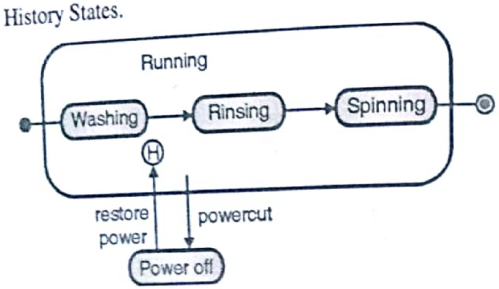
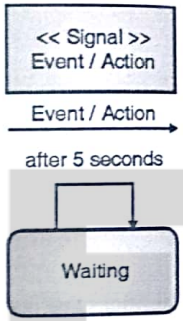


Fig. 4.8.2 : showing History States

- **Signal :** When an event produces a message or a trigger to be sent to a state that in-turn causes the state transition; then, that message is called as a signal. This is represented by the icon as <<Signal>> which is written above the event/action.
- **Self-transitions :** A state can have a transition that returns to itself, as in the diagram. This is most useful when an effect is associated with the transition.



Example : State diagram for login part of an online banking system.

- Logging in consists of entering a valid social security number (SSN) and personal id number, then submitting the information for validation.
- Logging in can be factored into four non-overlapping states: *Getting SSN, Getting PIN, Validating, and Rejecting*. From each state comes a complete set of **transitions** that determine the subsequent state.

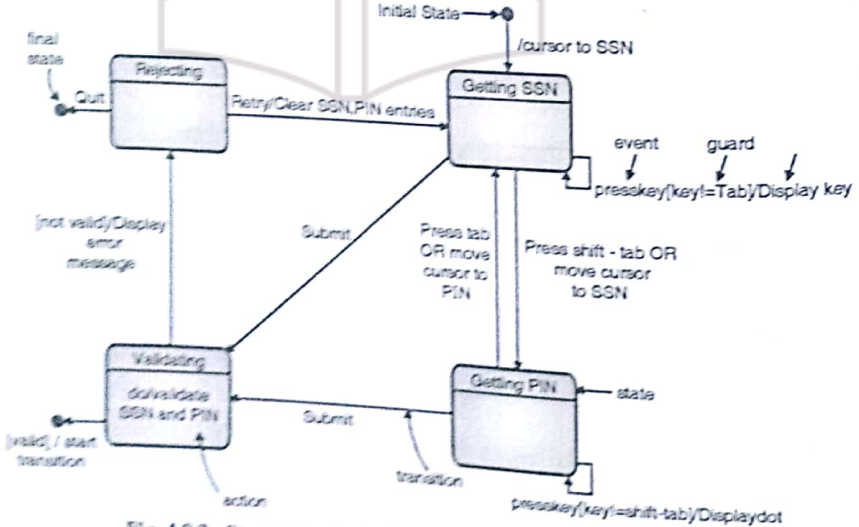


Fig. 4.8.3 : State diagram for login part of an online banking system

- **Actions inside the state:** Actions execute a function, assign a value to a data variable or initiate another transition. Specify these internal actions one per line. These internal actions (transitions) are processed without causing a state change.
- It takes the form: action-label / action-expression.
- The *action-label* may be any of the following:
  - **entry:** Executes the associated *action-expression* upon state entry.
  - eg. **entry / count:= 0; sum:= 0**
  - **exit:** Executes the associated *action-expression* upon state exit.
  - eg. **exit / ring bell.**
  - **do:** Executes the associated *action-expression* upon state entry and continues until state exit (or action completion).
  - eg. **do / display flashing light.**
  - **include**: The *action-expression* must name a finite automaton. The named automaton is a placeholder for a nested state diagram.
  - eg: **include / Order Processing**
  - *action-expression* - description of a computation.

Example : A state diagram for a temperature controller.

A temperature controller has been interface with a manufacturing unit in a factory which maintains the moderate temperature of 50%. If the temperature goes beyond or below the moderate temperature, the unit activities are cooling or heating units respectively and indicate through different signals.

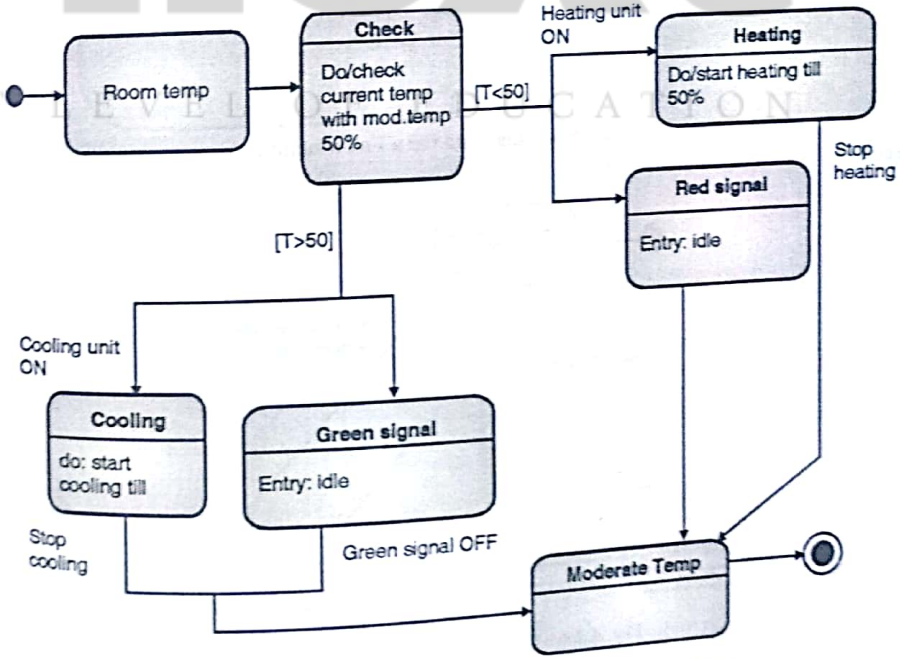


Fig. 4.8.4 : A state diagram for a temperature controller

Example : State diagram for a Fax Machine.

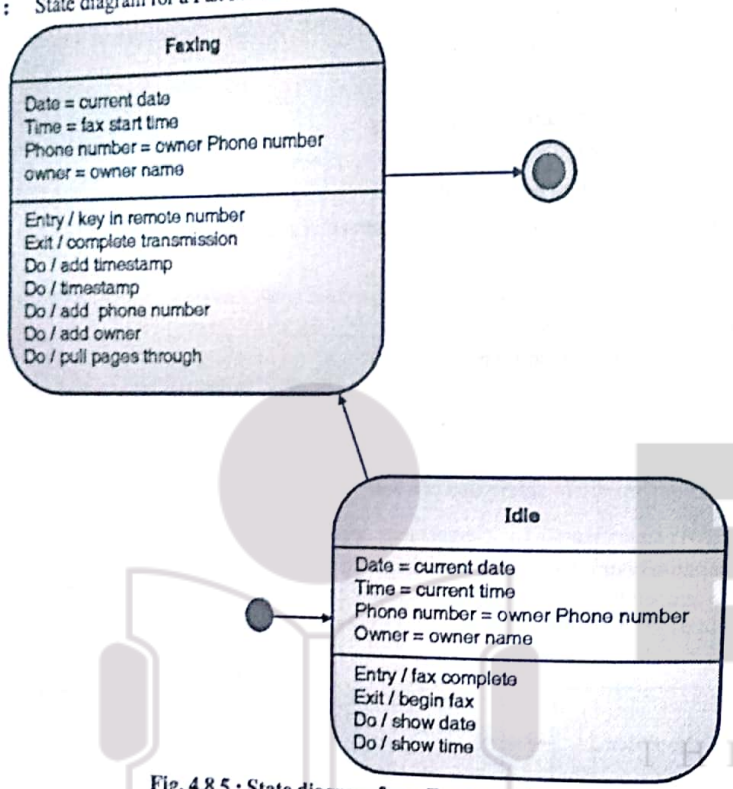


Fig. 4.8.5 : State diagram for a Fax Machine

Example : State diagram for ATM.

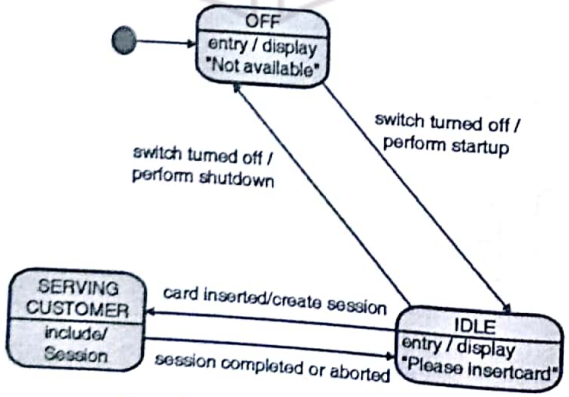


Fig. 4.8.6 : State diagram for ATM

Example : State diagram for Digital Watch.

A simple digital watch has a display and two buttons to set it, the A button and B button. The watch has two modes of operations display time and set time. In the display time mode, hours and minutes are displayed, separated by flashing colon. The set time mode has two sub modes, set hours and set minutes. The A button is used to select modes. Each time it is pressed the mode advances in sequence display, set hours, set minutes, display etc. Within the sub modes, the B button is used to advance the hour or minutes once, each time it is pressed. Button must be released before they can generate another event. Prepare a state diagram for digital watch.

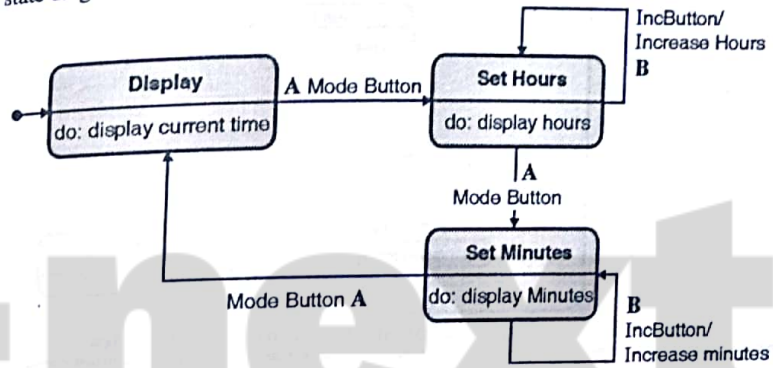


Fig. 4.8.7 : State diagram for Digital Watch

- **Choice Pseudo-State:** A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The Fig. 4.8.8 shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.

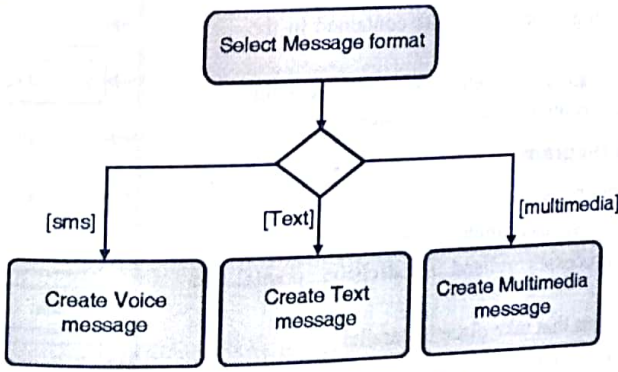


Fig. 4.8.8 : Choice pseudo-State

- **Nested States :** Nested states represents substates. The substate inherits the transitions of its superstate. Even though it affects the main state, a sub state is not shown as a part of the main state. Hence, it is depicted as contained within the main state flow.

Example : State diagram of phone line showing nested states in the main state *'Active'*.

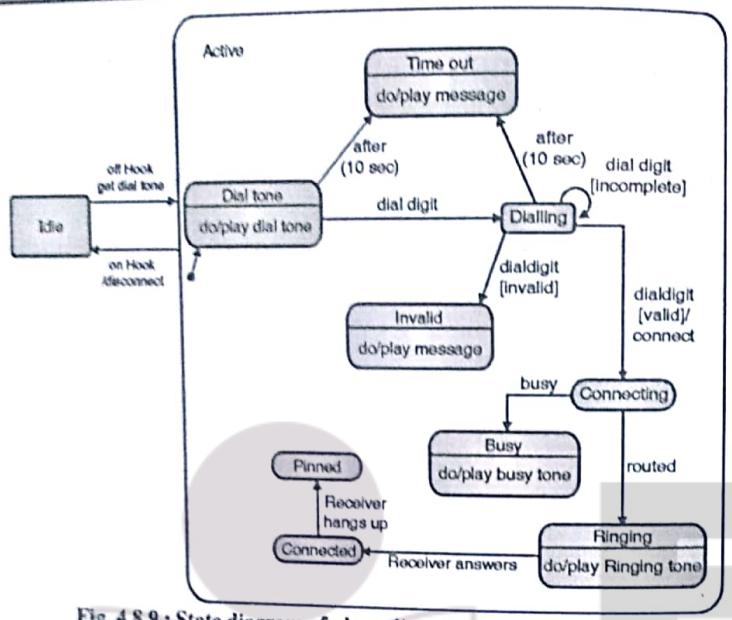


Fig. 4.8.9 : State diagram of phone line showing nested states

Syllabus Topic : Activity Diagram

4.9 Activity Diagram

- Activity diagram displays the sequence of activities.
- Activity diagrams show the workflow from a start point to the finish point detailing the number of decision paths that exist in the progression of events contained in the activity.
- They also describe situations, where parallel processing may occur in the execution of some activities.

Use of Activity diagram

- Describes Business rules
- Describes Complex series of multiple use cases
- Describes the processes related to decision points and alternate flows
- Describes operations that take place in parallel
- Describes software flows and logic control structures

Designing an Activity diagram

1. Initial node

The filled in circle is the starting point of the diagram. This indicates the beginning of the sequence of activities. An initial node makes it easier to read the diagram.

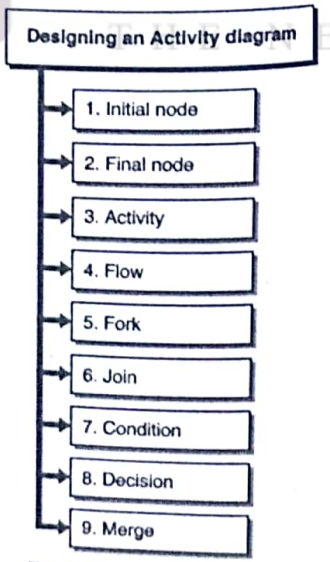


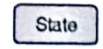
Fig. C4.7 : Designing an Activity diagram

2. Final node

The filled circle with a border is the ending point. An activity diagram can have zero or more activity final nodes. The final state ends the sequence of activities.

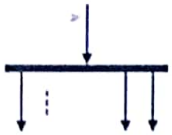
3. Activity

The rounded rectangles represent activities that occur. An activity may be physical, such as *Inspect Forms*, or electronic, such as *Display Student Screen*. It describes the action state of the system.



4. Flow

The solid arrows in the diagram represent the flow of activity. The outgoing arrow attached to an activity symbol indicates the transition i.e. triggered by the completion of the activity.

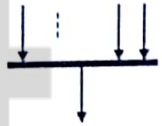


5. Fork

A thick black bar with one flow entering into it and several leaving it. This denotes the *beginning of parallel activity*. Forks have *One Entry Transition*.

6. Join

A thick black bar with several flows entering it and one leaving it. All flows going into the join must reach it before processing may continue. This denotes the *end of parallel activity*. Joins have *One Exit Transition*. Is exactly opposite to 'Fork'.

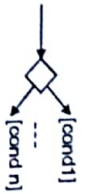


7. Condition

Text such as *[Incorrectpassword]* on a flow, defining a guard which must evaluate to true in order to traverse the node.

8. Decision

A *diamond with one flow entering and several leaving*. The flows leaving include conditions although some modelers will not indicate the conditions if it is obvious.



9 Merge

A *diamond with several flows entering and one leaving*. The implication is that one or more incoming flows must reach this point until processing continues, based on any guards on the outgoing flow. Is exactly opposite to 'Decision'.

Example : Showing Decision, Condition and Merge.

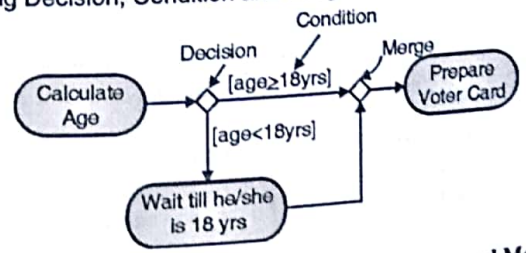


Fig. 4.9.1 : Example showing Decision, Condition and Merge

Partition (swimlanes) : Objects are generally organized into partitions, also called swimlanes, indicating who/what is performing the activities.

Example : Activity diagram for Aerobics Training Enrolment.

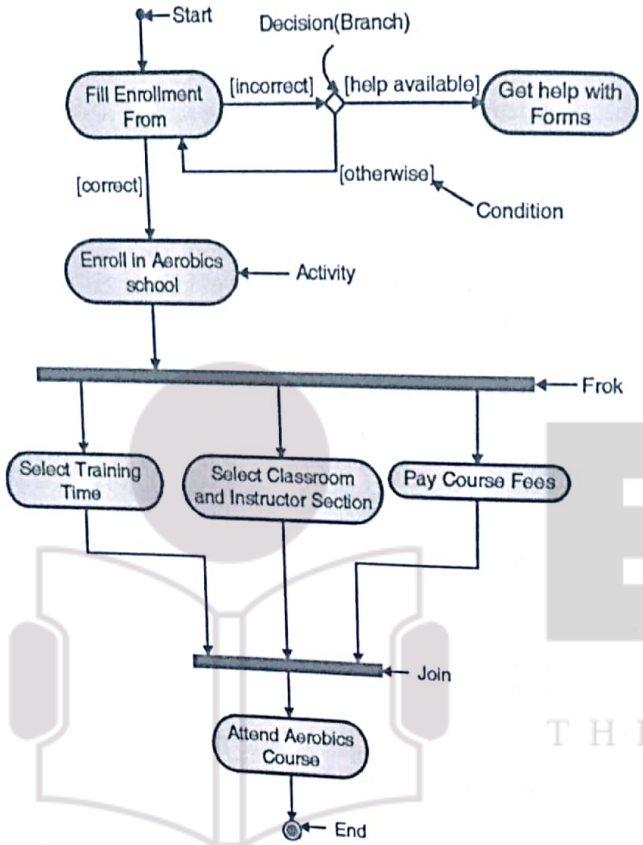


Fig. 4.9.2 : Activity diagram of Aerobics Training Enrolment

- After the 'Enrol in Aerobics School' activity completes, three activities occur: select training time, select classroom and instructor section, pay course fees. The **fork** (complex transition) indicates that these three activities can occur in any order.
- Then the **join** bar indicates that the 'Attend Aerobics Course' activity does not begin until after all the three activities (select training time, select classroom and instructor section, pay course fees) are complete.
- The two outgoing transitions of the 'Fill Enrolment Form' activity reflect a **decision** about whether the form is filled [correct]. The [incorrect] transition leads to a second decision, represented by the decision diamond, about whether to take help or fill the form again.

Example : Activity diagram for Login.

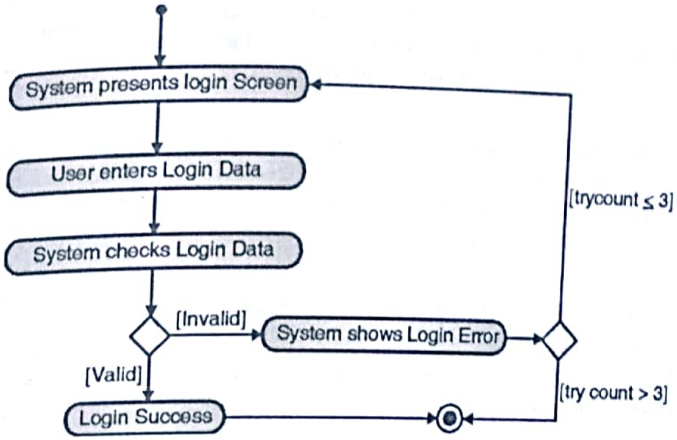


Fig. 4.9.3 : Activity diagram for Login

Example : Activity diagram for Customer process in Online buying of Toys.

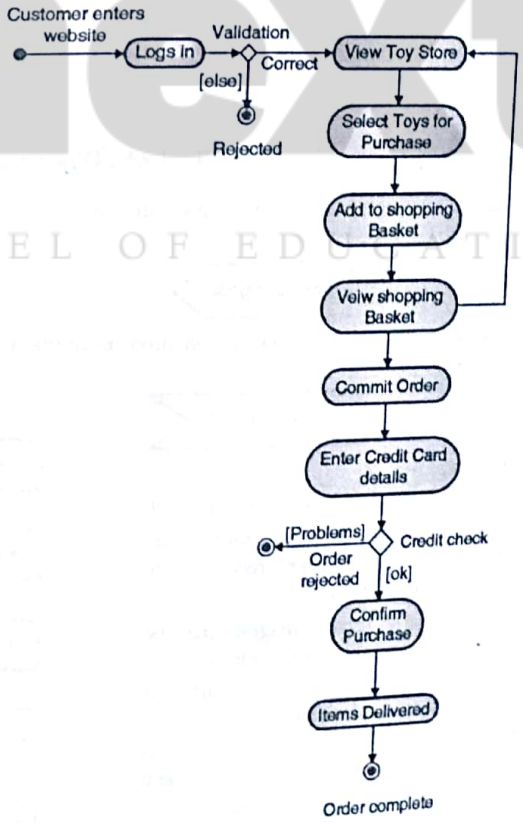
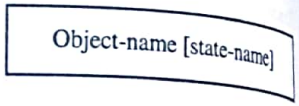


Fig. 4.9.4 : Activity diagram for Customer process in Online buying of Toys

- **Object State :** An object (indicated with a simple rectangle) in activity diagram represents instances of a specified object in a specified action state. If you specify the object name, the UML suite automatically encloses it within square brackets represented as;



- **Object flow :** The dashed arrows are used to represent the flow of objects. This shows how object states are used by action states. If an *action state generates an object* and it is used by the subsequent action state, then you can omit the transition (solid arrow) between the two states. You can use only dashed arrows. And these arrows are not labeled.

Example : Objects in states and object flows

In this example, completing the '*take order*' activity creates an '*Order*' object in the *[taken]* state.

The '*fill order*' activity takes an '*Order*' in the *[taken]* state and creates an '*Order*' in the *[filled]* state.

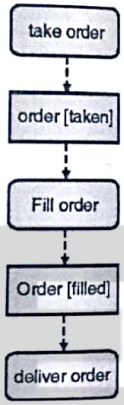
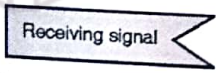


Fig. 4.9.5 : Object states and object flows

- **Signal sending :** A signal sending symbol shows a transition sending a signal. Do not label such signal sending transitions.

- **Signal receipt :** A signal receipt symbol shows a transition receiving a signal. Do not label such signal receiving transitions.



Example : Signal sending and receipt

In this example, the transition from '*take order*' to '*fill order*' sends an '*orderTaken*' signal to the '*Order*' object. The transition from '*fill order*' to '*deliverOrder*' receives an '*orderFilled*' signal from the '*Order*' object.

- **Swimlanes :** The contents of an activity diagram may be organized into partitions using solid vertical (or horizontal) lines also called as swimlanes. Each swimlane represents an organizational unit of some kind. A swimlane is a way to group activities performed by the same actor. You label each swimlane with the name of the class or department responsible for the activity states in the swimlane.

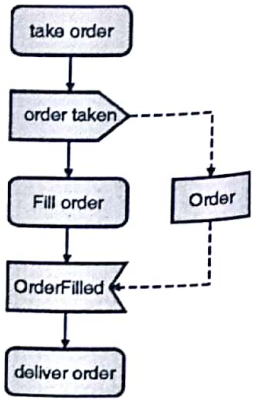


Fig. 4.9.6 : Signal sending and receipt

Example : Showing Swimlanes

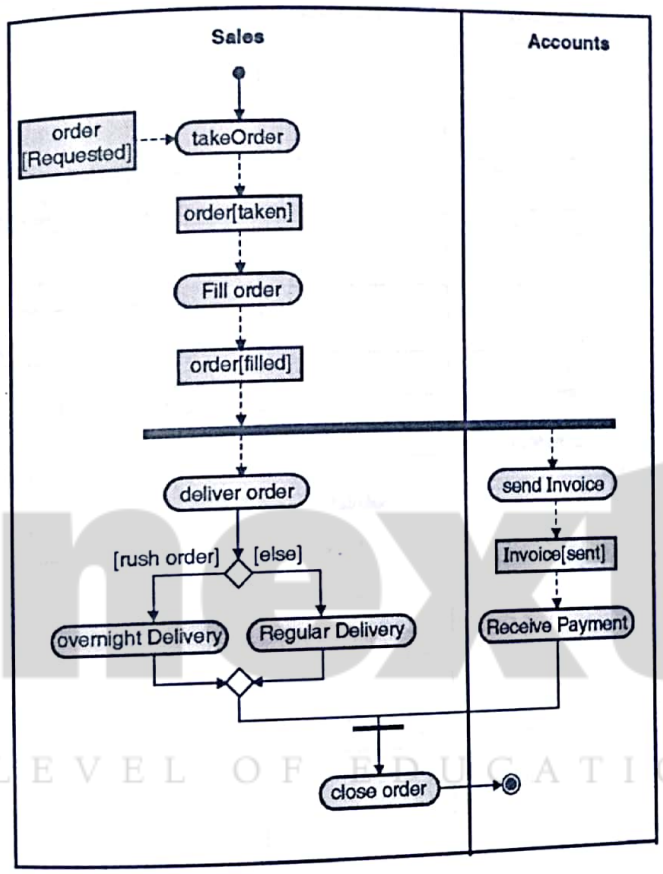


Fig. 4.9.7 : Showing Swimlanes

Example : Activity diagram to withdraw money from a bank account through ATM.

- The three involved classes of the activity diagram are *Customer, ATM,* and *Bank.*

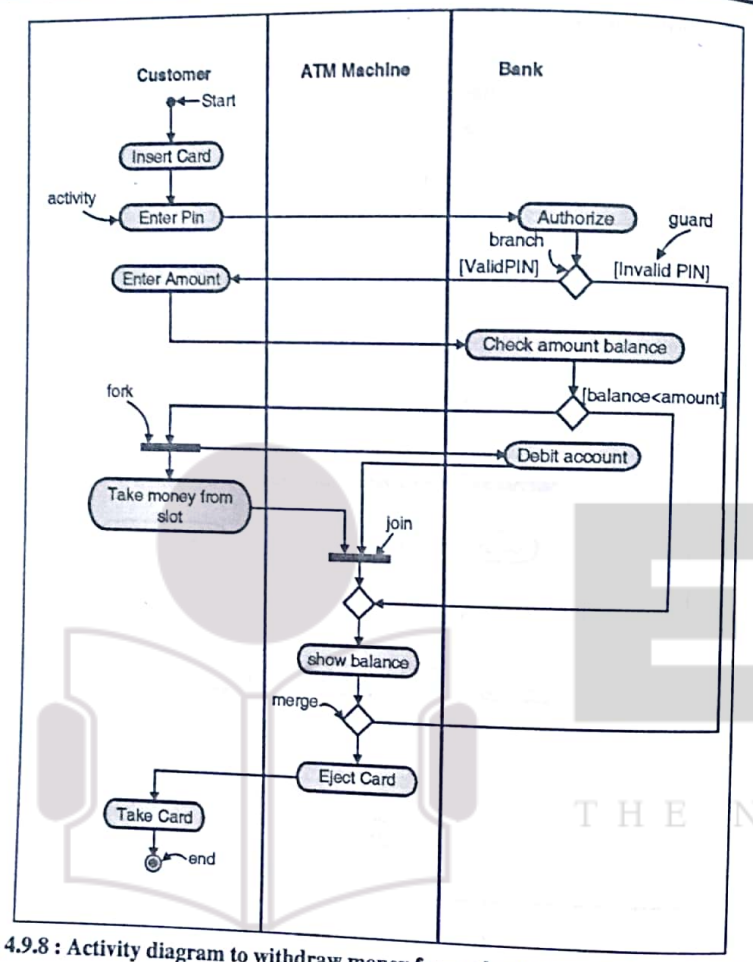


Fig. 4.9.8 : Activity diagram to withdraw money from a bank account through ATM.

4.10 Implementation Diagram

- Architecture model includes implementation diagram which shows the implementation phase of system development and its architecture and architecture. It describes the physical and logical structure.
- Physical architecture deals with
  - Detail description of the system and decompositions in terms of hardware and software.
  - Physical location of the classes and objects in which processes, programs and computers.
  - Dependency between different code files and connection hardware device.
- Logical architecture deals with
  - Functionality of the system
  - Functionality of the system deliver

  - Relationship among classes
  - Class and object collaboration to deliver the functionality.
- In short, Implementation deals with not only the code of the program but also with its implementation structure.

Examples of Implementation diagram :

1. *Component diagram* 2. *Deployment diagram*

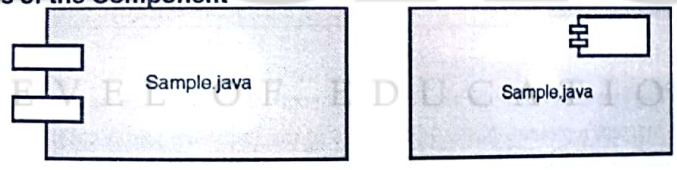
Syllabus Topic : Component Diagram

4.10.1 Component Diagram

Component

- A component is a distributable unit of software.
- A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.
- A component is a subtype of Class which provides attributes and operations and participates in Associations and Generalizations.
- A Component may form the abstraction for a set of realizing Classifiers that realize its behaviour.
- In addition, because a Class itself is a subtype of an Encapsulated Classifier, a Component may optionally have an internal structure and own a set of Ports that formalize its interaction points.
- Components are concerned with the physical aspect of the system. Physical aspects are the elements like executable, libraries, files, and documents etc. which reside in the node.

Notations of the Component



Notation 1 Notation 2

Notation 1 is somewhat older notation used by UML.

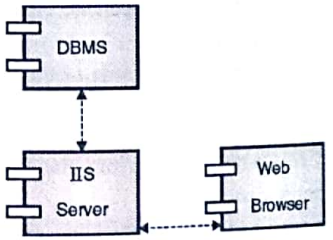


Fig. 4.10.1 : Component diagram of a 3-tier architecture of an application

- Every component has different name and properties of each component is different from each other.

- We use simple string to assign the name to a Component diagram. Sometimes the pathnames or the property parameters are also added to component name and in that case, the extendable name of component is used.

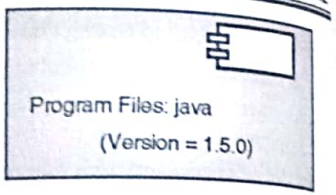


Fig. 4.10.2 : Extendable Notation

Similarities between Class and Component

- There are lot of similarities between a class and a component such as:
- Both have a distinct name
- Both realize a set of interfaces
- Both participate in dependency, generalisation and association relationship (remember class diagram)
- Both may have instances
- Both may be nested
- Both may participate in interaction.

Differences between Class and Component

Components	Classes
Components represent physical aspects related with world of bits	Classes represent logical abstraction
Physical packaging of logical components is at different level of abstraction.	Classes may not
Components only have operations that are reachable only through their interfaces.	Classes may have attributes and operations directly.

Component Diagram

- Component diagram shows how the physical components of a system are organised.
- Component diagram represents pieces of software in the implementation environment. Where the class and packages diagrams models the implementation view.
- According to the definition of the component, component is related with the physical aspects. Component diagrams are used to visualise the organisation and relationships among components in the system. These diagrams are also used to make executable system.
- Component diagram is somewhat different from the other diagrams which are used in UML. If we consider a simple class diagram, class contains the name of the class, attributes and the operation (function) respective to that class. In general class provide overall functionality. Component diagram never provide the functionality. But the component are those functionality description make by the component diagram. This is the futuristic importance of the component diagram.
- Here the components are files, libraries, packages etc.
- Component diagram can also be described as a static implementation of view of a system. Static implementation represents the organisation of the component at a particular movement.

Use of Component Diagram

- Visualize the components of a system.
- Describe the organization and relationships of the components.
- Construct executables by using forward and reverse engineering.

Designing Component Diagram

- Component diagrams are used to describe the physical artifacts of a system. We know that this artifact includes files, executables, libraries etc.
- So the purpose of this diagram is different, Component diagrams are used during the implementation phase of an application. But it is prepared well in advance to visualize the implementation details.
- Initially the system is designed using different UML diagrams and then when the artifacts are ready, component diagrams are used to get an idea of the implementation.
- This diagram is very important since without it the application cannot be implemented efficiently.
- A well prepared component diagram is also important for other aspects like application performance, maintenance etc.
- So before drawing a component diagram the following artifacts are to be identified clearly:
  1. What kinds of Files are used in the related system.
  2. Which Libraries and other artifacts relevant to the application.
  3. Relationships among the artifacts (files and libraries).
- Rules to draw the component diagram:
  1. Use a meaningful name to identify the component for which the diagram is to be drawn.
  2. Prepare a mental layout before producing using tools.
  3. Use notes for clarifying important points.
- Component diagram with following object oriented modeling language is used for management of these files which represent the work product components.

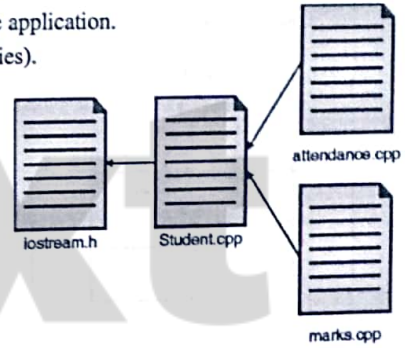


Fig. 4.10.3 : Modeling Source Code

Example : An employee of a company stays in a flat or bungalow. It may be owned by the company, employee or may be rented by either the company or the employee. The owner who lends houses to company may themselves be company employee. The Tenant in a flat has to pay additional water and electricity charges.

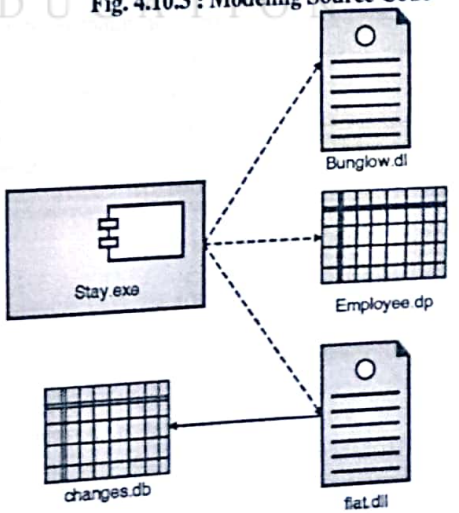
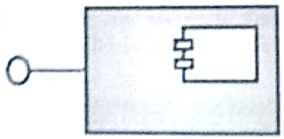
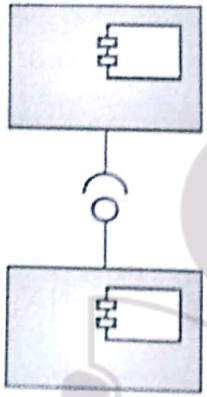


Fig. 4.10.4 : Modeling Physical Databases

Other Notations used in Component Diagram

Notation	Use
	Components implement interfaces
	Assembly Connector

Advantages of Component Diagram

- Models the real software in the implementation environment.
- They make known software configuration issue through the dependency relationships.
- It provides the accurate picture of existing system prior to making changes or development.
- They can reveal bottlenecks in an implementation without forcing us to read all of the code.

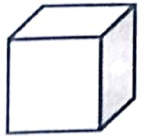
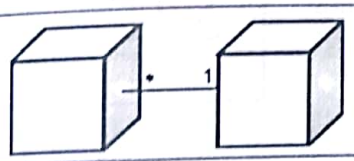
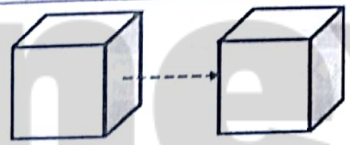
Syllabus Topic : Deployment Diagram

4.10.2 Deployment Diagram

- Deployment diagram show the configuration of run-time processing elements and the software components, processes and objects that live in them.
- Deployment diagram models the hardware of the implementation environment.
- A deployment diagram is a graph of nodes connected by communication association. In general the deployment diagram describes the run time architecture of processors, devices and the software components. Each of these node in the deployment diagram typically represents one of the hardware part like disk drive, client machine, server machine, processor, networking structure etc.
- Here, node means the components or logical elements which are used at the run time. Basically that are class, objects or collaborations which are executes at run time.
- That node connects for each other for communication purpose.

- Connection between nodes shows the system interaction path.
- Collaboration diagram models the topology of the hardware. It also models the embedded system. It put client-server model and distributed application model hardware.

Table 4.10.1 : Notations used in Deployment Diagram

Sr. No.	Notation	Use
1.		Node: A node is drawn as a three dimensional cube with names inside it.
2.		Association between two nodes: Communication association connects to each other.
3.		Dependency

Advantages of Deployment Diagram :

1. Models the hardware platform for the system.
2. Identifies hardware capabilities that affect performance planning and software configuration.

Example : Database server (any kind of) can connect with client machine through Internet

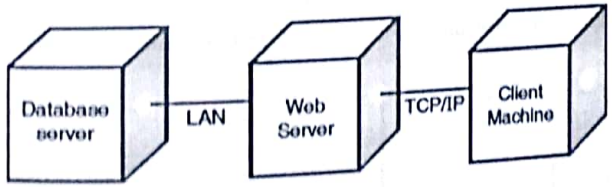


Fig. 4.10.5

Example : Online Shopping for checking the products

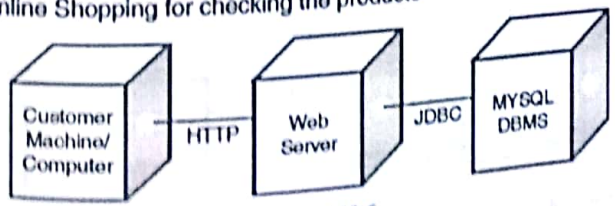


Fig. 4.10.6

1. We can combine the component diagram with deployment diagram as follows.

Example

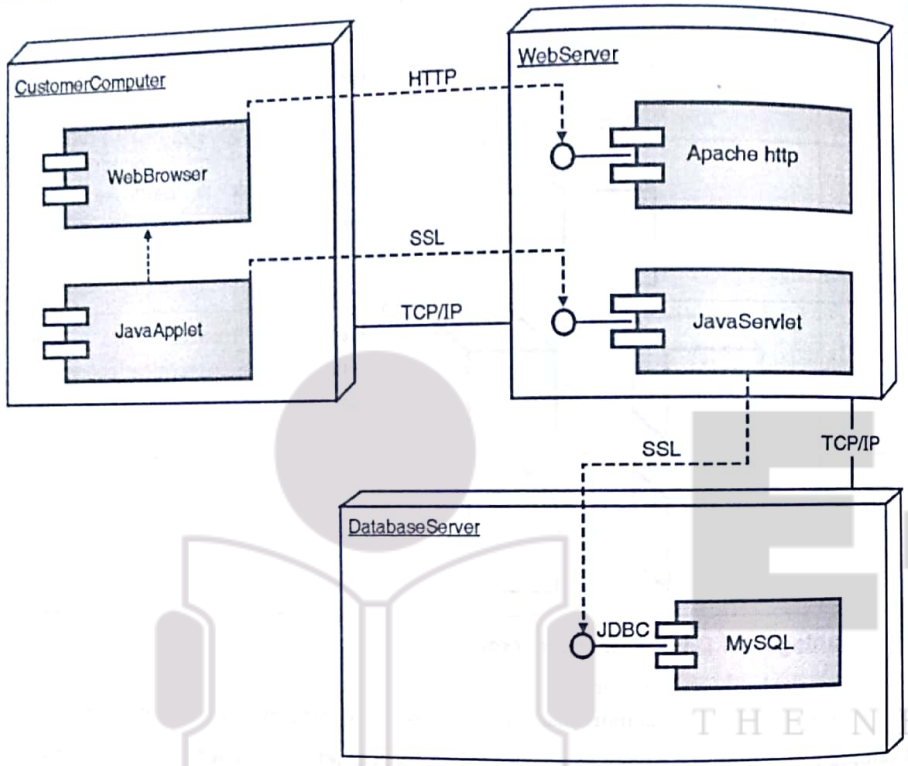


Fig. 4.10.7

Example : A simple deployment diagram of 3-tier architecture of an application.

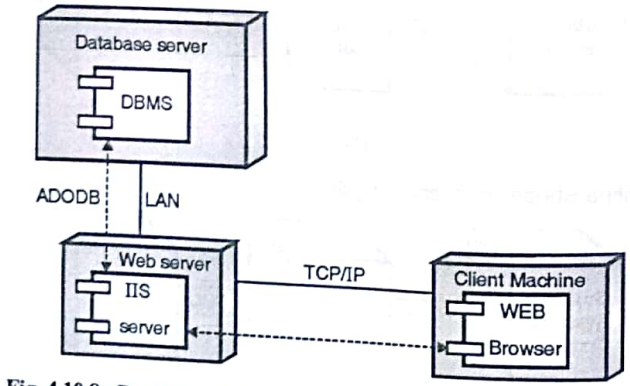


Fig. 4.10.8 : Deployment diagram encompassing component diagram

Review Questions

Q. 1 "UML is mainly used for software system". State 'True' or 'False' with explanation

Q. 2 What is classification theory?

Q. 3 List and explain the stereotypes used for modeling and interactions among objects.

Q. 4 Prepare use case diagram for bus reservation system.

Q. 5 Draw a class diagram for "mobile company". They have different distributors at different areas. Different facilities are provided such as incoming calls free, sending E-mails, songs can be seen etc. (Make necessary assumptions).

Q. 6 An employee of a company stays in a flat or bungalow. It may be owned by the company, employee or may be rented by either the company or the employee. The owner who let houses to company may themselves be company employee. The Tenant in a flat has to pay additional water and electricity charges. Draw component and deployment diagram for the system.

Q. 7 Draw a collaboration diagram for Railway reservation system. The passenger is required to fill a reservation from giving details of his journey. The counter clerk ensures whether the place is available and prepares a booking statement.

❑❑❑