Index

Sr. No.	Name of the experiment	Date	Page No.
1	Experiment-1 Software Requirement Specification Document		
2	Experiment-2 Use Case Diagram		
3	Experiment-3 Activity Diagram		
4	Experiment 4 Class Diagram		
5	Experiment 5 Sequence Diagram		
6	Experiment 6 Collaboration Diagram		
7	Experiment 7 State Chart Diagram		
8	Experiment 8 Component Diagram		
9	Experiment 9 Entity relationship diagram		
10	Experiment 10 Dataflow Diagram		

Experiment-1 Software Requirement Specification Document

Experiment Name:Prepare a SRS document in line with the IEEE recommended standards.

Description:

A Software Requirements Specification (SRS) is a document that describes the nature of a project, software or application. In simple words, SRS document is a manual of a project provided it is prepared before you kick-start a project/application. This document is also known by the names SRS report, software document. A software document is primarily prepared for a project, software or any kind of application.

There are a set of guidelines to be followed while preparing the software requirement specification document. This includes the purpose, scope, functional and nonfunctional requirements, software and hardware requirements of the project. In addition to this, it also contains the information about environmental conditions required, safety and security requirements, software quality attributes of the project etc.

What is a Software Requirements Specification document?

A Software requirements specification document describes the intended purpose, requirements and nature of a software to be developed. It also includes the yield and cost of the software.

In this document, SRS for flight management is produced

Table of Contents for a SRS Document

1. Introduction

1.1 Purpose

1.2 Document Conventions

1.3 Intended Audience and Reading Suggestions

1.4 Project Scope

1.5 References

2. Overall Description

2.1 Product Perspective

2.2 Product Features

2.3 User Classes and Characteristics

2.4 Operating Environment

2.5 Design and Implementation Constraints

2.6 Assumptions and Dependencies

3. System Features

3.1 Functional Requirements

4. External Interface Requirements

4.1 User Interfaces

4.2 Hardware Interfaces

4.3 Software Interfaces

4.4 Communications Interfaces

5. Nonfunctional Requirements

5.1 Performance Requirements

5.2 Safety Requirements

5.3 Security Requirements

5.4 Software Quality Attributes

1. INTRODUCTION

1.1.PURPOSE

The purpose of this document is to build an online system to manage flights and passengers to ease the flight management.

1.2.DOCUMENT CONVENTIONS

This document uses the following conventions.

DB	Database
DDB	Distributed Database
ER	Entity Relationship

1.3.INTENDED AUDIENCE AND READING SUGGESTIONS

This project is a prototype for the flight management system and it is restricted within the college premises. This has been implemented under the guidance of college professors. This project is useful for the flight management team and as well as to the passengers.

1.4.PROJECT SCOPE

The purpose of the online flight management system is to ease flight management and to create a convenient and easy-to-use application for passengers, trying to buy airline tickets. The system is based on a relational database with its flight management and reservation functions. We will have a database server supporting hundreds of major cities around the world as well as thousands of flights by various airline companies. Above all, we hope to provide a comfortable user experience along with the best pricing available.

1.5.REFERENCES

- Fundamentals of database systems by ramez elmarsi and shamkant b.navathe

2. OVERALL DESCRIPTION

2.1.PRODUCT PERSPECTIVE

A distributed airline database system stores the following information.

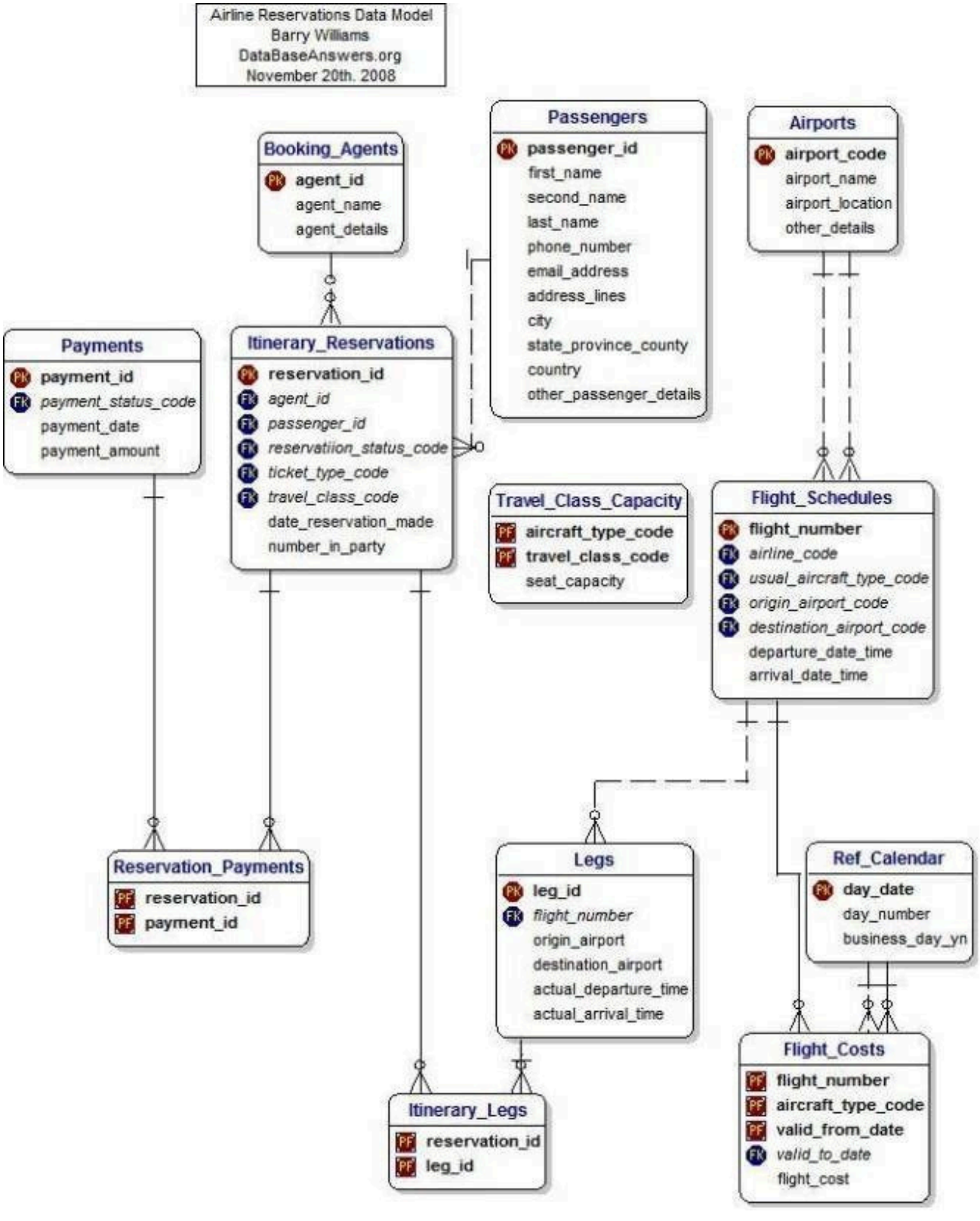
- Flight details:
  It includes the originating flight terminal and destination terminal, along with the stops in between, the number of seats booked/available seats between two destinations etc.
- Customer description:
  It includes customer code, name, address and phone number. This information may be used for keeping the records of the customer for any emergency or for any other kind of information.
- Reservation description:
  It includes customer details, code number, flight number, date of booking, date of travel.

2.2.PRODUCT FEATURES

The major features of airline database system as shown in below entity–relationship model (ER model)

The diagram shows the layout of airline database system – entity–relationship model

2.3.USER CLASS and CHARACTERISTICS



Users of the system should be able to retrieve flight information between two given cities with the given date/time of travel from the database. A route from city A to city B is a sequence of connecting flights from A to B such that: a) there are at most two connecting stops, excluding the starting city and destination city of the trip, b) the connecting time is between one to two hours. The system will support two types of user privileges, Customer, and Employee. Customers will have access to customer functions, and the employees will have access to both customer and flight management functions. The customer should be able to do the following functions:

- Make a new reservation
  - •One-way
  - •Round-Trip
  - •Multi-city
  - •Flexible Date/time
  - •Confirmation
- Cancel an existing reservation
- View his itinerary

\

The Employee should have following management functionalities:

- CUSTOMER FUNCTIONS.
  - Get all customers who have seats reserved on a given flight.
  - Get all flights for a given airport.
  - View flight schedule.
  - Get all flights whose arrival and departure times are on time/delayed.
  - Calculate total sales for a given flight.

- ADMINISTRATIVE
  - Add/Delete a flight
  - Add a new airport
  - Update fare for flights.
  - Add a new flight leg instance.
  - Update departure/arrival times for flight leg instances.

Each flight has a limited number of available seats. There are a number of flights which depart from or arrive at different cities on different dates and time.

2.4. OPERATING ENVIRONMENT

Operating environment for the airline management system is as listed below.

- distributed database
- client/server system
- Operating system: Windows.
- database: sql+ database
- platform: vb.net/Java/PHP

2.5. DESIGN and IMPLEMENTATION CONSTRAINTS

1.The global schema, fragmentation schema, and allocation schema.
2.SQL commands for above queries/applications
3.How the response for application 1 and 2 will be generated. Assuming these are global queries. Explain how various fragments will be combined to do so.
4.Implement the database at least using a centralized database management system.

2.6.ASSUMPTION DEPENDENCIES

Let us assume that this is a distributed airline management system and it is used in the following application:

- A request for booking/cancellation of a flight from any source to any destination, giving connected flights in case no direct flight between the specified Source-Destination pair exist.
- Calculation of high fliers (most frequent fliers) and calculating appropriate reward points for these fliers.

Assuming both the transactions are single transactions, we have designed a distributed database that is geographically dispersed at four cities Delhi, Mumbai, Chennai, and Kolkatta as shown in fig. below.

3. SYSTEM FEATURES

- DESCRIPTION and PRIORITY

The airline reservation system maintains information on flights, classes of seats, personal preferences, prices, and bookings. Of course, this project has a high priority because it is very difficult to travel across countries without prior reservations.

- STIMULUS/RESPONSE SEQUENCES
  - Search for Airline Flights for two Travel cities
  - Displays a detailed list of available flights and make a "Reservation" or Book a ticket on a particular flight. Cancel an existing Reservation.

- FUNCTIONAL REQUIREMENTS

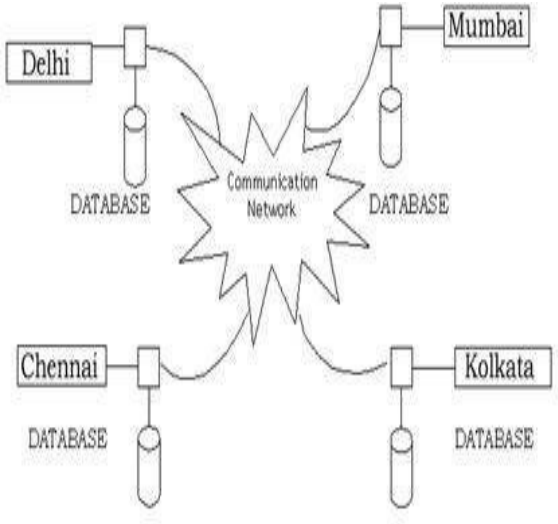
Other system features include:

DISTRIBUTED DATABASE:

Distributed database implies that a single application should be able to operate transparently on data that is spread across a variety of different databases and connected by a communication network as shown in below figure.

Distributed database located in four different cities

CLIENT/SERVER SYSTEM



The term client/server refers primarily to an architecture or logical division of responsibilities, the client is the application (also known as the front-end), and the server is the DBMS (also known as the back-end).

A client/server system is a distributed system in which,

- Some sites are client sites and others are server sites.
- All the data resides at the server sites.
- All applications execute at the client sites.

4. EXTERNAL INTERFACE REQUIREMENTS

4.1.USERINTERFACES

- Front-end software: Vb.net version
- Back-end software: SQL+

4.2.HARDWARE INTERFACES

- Windows.
- A browser which supports CGI, HTML & Javascript.

4.3.SOFTWARE INTERFACES

Following are the software used for the flight management online application.

Software used	Description
Operating system	We have chosen Windows operating system for its best support and user-friendliness.
Database	To save the flight records, passengers records we have chosen SQL+ database.
VB.Net	To implement the project we have chosen Vb.Net language for its more interactive support.

4.4.COMMUNICATION INTERFACES

This project supports all types of web browsers. We are using simple electronic forms for the reservation forms, ticket booking etc.

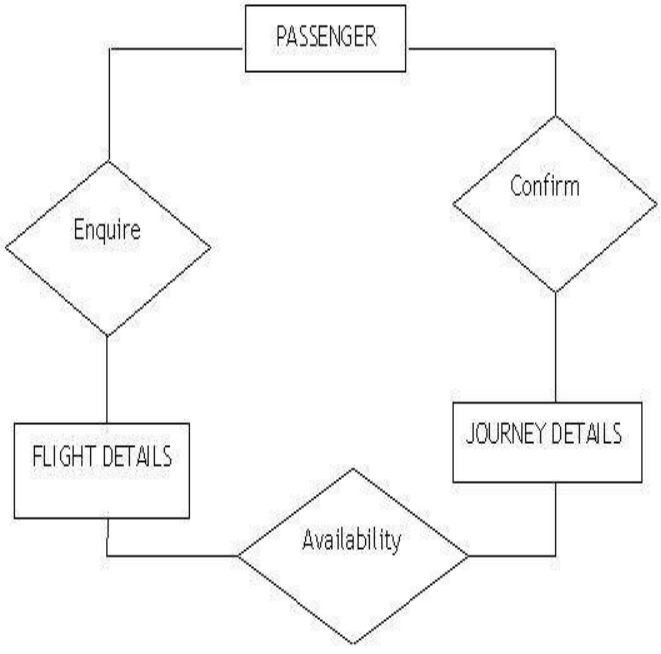
5. NONFUNCTIONAL REQUIREMENTS

5.1.PERFORMANCE REQUIREMENTS

The steps involved to perform the implementation of airline database are as listed below.

A)E-R DIAGRAM

The E-R Diagram constitutes a technique for representing the logical structure of a database in a pictorial manner. This analysis is then used to organize data as a relation, normalizing relation and finally obtaining a relation database.



the diagram shows the ER diagram of airline database

- ENTITIES: Which specify distinct real-world items in an application.
- PROPERTIES/ATTRIBUTES: Which specify properties of an entity and relationships.
- RELATIONSHIPS: Which connect entities and represent meaningful dependencies between them.

B)NORMALIZATION:

The basic objective of normalization is to reduce redundancy which means that information is to be stored only once. Storing information several times leads to wastage of storage space and increase in the total size of the data stored. If a database is not properly designed it can give rise to modification anomalies. Modification anomalies arise when data is added to, changed or deleted from a database table. Similarly, in traditional databases as well as improperly designed relational databases, data redundancy can be a problem. These can be eliminated by normalizing a database. Normalization is the process of breaking down a table into smaller tables. So that each table deals with a single theme. There are three different kinds of modifications of anomalies and formulated the first, second and third normal forms (3NF) is considered sufficient for most practical purposes. It should be considered only after a thorough analysis and complete understanding of its implications.

5.2.SAFETY REQUIREMENTS

If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was backed up to archival storage (typically tape) and reconstructs a more current state by reapplying or redoing the operations of committed transactions from the backed up log, up to the time of failure.

5.3.SECURITY REQUIREMENTS

Security systems need database storage just like many other applications. However, the special requirements of the security market mean that vendors must choose their database partner carefully.

5.4.SOFTWAREQUALITYATTRIBUTES

- AVAILABILITY: The flight should be available on the specified date and specified time as many customers are doing advance reservations.
- CORRECTNESS: The flight should reach start from correct start terminal and should reach the correct destination.
- MAINTAINABILITY: The administrators and flight in chargers should maintain correct schedules of flights.
- USABILITY: The flight schedules should satisfy a maximum number of customers needs.

Experiment- 2 Use case diagram

Name of the experiment: Draw the use case diagram and specify the role of each of the actors

Hardware Requirements: Pentium 4 processor (2.4 GHz), 128 Mb RAM, Standard keyboard n mouse, colored monitor.

Software Requirements: STAR UML, .

Outcome: Can produce the requirements in Use Case diagram.

Objective: Prepare a Requirement doccument in by using Use Case Diagram.

Description: According to the UML specification a use case diagram is “a diagram that shows the relationships among actors and use cases within a system.” Use case diagrams are often used to:

Provide an overview of all or part of the usage requirements for a system or organization in the form of an essential model or a business model

Communicate the scope of a development project

Model your analysis of your usage requirements in the form of a system use case model

Use case models should be developed from the point of view of your project stakeholders and not from the (often technical) point of view of developers.

Guidelines

1. Use Cases

A use case describes a sequence of actions that provide a measurable value to an actor. A use case is drawn as a horizontal ellipse on a UML use case diagram.

1. Use Case Names Begin With a Strong Verb
2. Name Use Cases Using Domain Terminology
3. Place Your Primary Use Cases In The Top-Left Corner Of The Diagram
4. Imply Timing Considerations By Stacking Use Cases.

2. Actors

An actor is a person, organization, or external system that plays a role in one or more interactions with your system (actors are typically drawn as stick figures on UML Use Case diagrams).

1. Place Your Primary Actor(S) In The Top-Left Corner Of The Diagram
2. Draw Actors To The Outside Of A Use Case Diagram
3. Name Actors With Singular, Business-Relevant Nouns
4. Associate Each Actor With One Or More Use Cases
5. Actors Model Roles, Not Positions
6. Use <<system>> to Indicate System Actors
7. Actors Don't Interact With One Another
8. Introduce an Actor Called "Time" to Initiate Scheduled Events

3. Relationships

There are several types of relationships that may appear on a use case diagram:

An association between an actor and a use case An association between two use cases
A generalization between two actors
A generalization between two use cases

Associations are depicted as lines connecting two modeling elements with an optional open-headed arrowhead on one end of the line indicating the direction of the initial invocation of the relationship. Generalizations are depicted as a close-headed arrow with the arrow pointing towards the more general modeling element.

1. Indicate An Association Between An Actor And A Use Case If The Actor Appears Within The Use Case Logic
2. Avoid Arrowheads On Actor-Use Case Relationships
3. Apply <<include>> When You Know Exactly When To Invoke The Use Case
4. Apply <<extend>> When A Use Case May Be Invoked Across Several Use Case Steps
5. Introduce <<extend>> associations sparingly
6. Generalize Use DCoawsnelosa Wdedh ebny Baa Slaikn Rgalem C(boanladkriatmio1n5 0R9e@skuclctistm I.ned S Business Logic
7. Do Not Apply <<uses>>, <<includes>>, or <<extends>>
8. Avoid More Than Two Levels Of Use Case Associations
9. Place An Included Use Case To The Right Of The Invoking Use Case
10. Place An Extending Use Case Below The Parent Use Case
    Apply the "Is Like" Rule to Use Case Generalization
11. Place an Inheriting Use Case Below The Base Use Case
    Apply the "Is Like" Rule to Actor Inheritance
12. Place an Inheriting Actor Below the Parent Actor
13.
14.

4. System Boundary Boxes

The rectangle around the use cases is called the system boundary box and as the name suggests it indicates the scope of your system – the use cases inside the rectangle represent the functionality that you intend to implement.

1. Indicate Release Scope with a System Boundary Box.
2. Avoid Meaningless System Boundary Boxes.

Creating Use Case Diagrams

we start by identifying as many actors as possible. You should ask how the actors interact with the system to identify an initial set of use cases. Then, on the diagram, you connect the actors with the use cases with which they are involved. If actor supplies information, initiates the use case, or receives any information as a result of the use case, then there should be an association between them.

Conclusion: The Use case diagram was made successfully by following the steps describedabove.

Experiment-3 Activity Diagram

ExperimentName:Drawtheactivitydiagram

Outcome : Can produce the activity diagram for requirements modeling.

Objective: To Draw a sample activity diagram for real project or system.

Description:

HardwareRequirements:

Pentium 4 processor (2.4 GHz), 128 Mb RAM, Standard keyboard n mouse, colored monitor.

SoftwareRequirements:

STAR UML

Theory:
Activity diagrams are typically used for business process modeling, for modeling the logic captured by a single usecaseor usage scenario, or for modeling the detailed logic of a business rule. Although UML activity diagrams could potentially model the internal logic of a complex operation it would be far better to simply rewrite the operation so that it is simple enough that you don't require an activity diagram. In many ways UML activity diagrams are the object- oriented equivalent of flow charts and data flow diagrams (DFDs) from structured development.

Let's start by describing the basic notation :

- Initial node. The filled in circle is the starting point of the diagram. An initial node isn't required although it does make it significantly easier to read the diagram.
- Activity final node. The filled circle with a border is the ending point. An activity diagram can have zero or more activity final nodes.
- Activity. The rounded rectangles represent activities that occur. An activity may be physical, such as *Inspect Forms*, or electronic, such as *Display Create Student Screen*.
- Flow/edge. The arrows on the diagram. Although there is a subtle difference between flows and edges,never a practical purpose for the difference although.

- Fork. A black bar with one flow going into it and several leaving it. This denotes the beginning of parallel activity.

- Join. A 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 processing.

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

- 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.

- 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.

- Partition. If figure is organized into three partitions, it is also called swimlanes, indicating who/what is performing the activities (either the *Applicant*, *Registrar*, or *System*).

- Sub-activity indicator. The rake in the bottom corner of an activity, such as in the *Apply to University* activity, indicates that the activity is described by a more finely detailed activity diagram.

- Flow final. The circle with the X through it. This indicates that the process stops at this point.

- Activities

An activity, also known as an activity state, on a UML Activity diagram typically represents the invocation of an operation, a step in a business process, or an entire business process.

- Question “Black Hole” Activities. A black hole activity is one that has transitions into it but none out, typically indicating that you have either missed one or more transitions.

- Question “Miracle” Activities. A miracle activity is one that has transitions out of it but none into it, something that should be true only of start points.

- Decision Points

A decision point is modeled as a diamond on a UML Activity diagram.

- Decision Points Should Reflect the Previous Activity. In figure1 we see that there is no label on the decision point, unlike traditional flowcharts which would include text describing the actual decision being made, we need to imply that the decision concerns whether the person was enrolled in the university based on the activity that the decision point follows. The guards, depicted using the format *[description]*, on the transitions leaving the decision point also help to describe the decision point.

- Avoid Superfluous Decision Points. The *Fill Out Enrollment Forms* activity in FIGURE1 includes an implied decision point, a check to see that the forms are filled out properly, which simplified the diagram by avoiding an additional diamond.

- Guards

A guard is a condition that must be true in order to traverse a transition.

- Each Transition Leaving a Decision Point Must Have a Guard

- Guards Should Not Overlap. For example guards such as x <0, x = 0, and x > 0 are consistent whereas guard such as x <= 0 and x >= 0 are not consistent because they overlap – it isn’t clear what should happen when x is 0.

- Guards on Decision Points Must Form a Complete Set. For example, guards such as x < 0 and x >0 are not complete because it isn’t clear what happens when x is

- Exit Transition Guards and Activity Invariants Must Form a Complete Set. An activity invariant is a condition that is always true when your system is processing an activity.

- Apply a [Otherwise] Guard for "Fall Through" Logic.

- Guards Are Optional. It is very common for a transition to not include a guard, even when an activity includes several exit transitions.

5. Parallel Activities

It is possible to show that activities can occur in parallel,

- A Fork Should Have a Corresponding Join. In general, for every start (fork) there is an end (join). In UML 2 it is not required to have a join, but it usually makes sense.

- Forks Have One Entry Transition.
- Joins Have One Exit Transition

- Avoid Superfluous Forks.

- Swimlane Guidelines

A swimlane is a way to group activities performed by the same actor on an activity diagram or to group activities in a single thread.

- Have Less Than Five Swimlanes.
- Consider Swimareas For Complex Diagrams.
- Swimareas Suggest The Need to Reorganize Into Smaller Activity Diagrams.

- Consider Horizontal Swimlanes for Business Processes. In FIGURE 3 you see that the swimlanes are drawn horizontally, going against common convention of drawing them vertically.

7 Action-Object Guidelines

Activities act on objects, In the strict object-oriented sense of the term an action object is a system object, a software construct. In the looser, and much more useful for business application modeling, sense of the term an action object is any sort of item. For example in FIGURE 3 the *ExpenseForm* action object is likely a paper form.

- Place Shared Action Objects on Swimlane Separators
- When An Object Appears Several Time Apply State Names
- State Names Should Reflect the Lifecycle Stage of an Action Object
- Show Only Critical Inputs and Outputs
- Depict Action Objects As Smaller Than Activities

Experiment Number 4

Class Diagram

Experiment Name: Identify the classes. Classify them as weak and strong classes and draw the class diagram.

Outcome: Class diagram helps to understand the static structure of a system. It shows relationships between classes, objects, attributes, and operations.

Objective: Identify the classes. Classify them as weak and strong classes and draw the class diagram.

Description:

A class diagram models the static structure of a system. It shows relationships between classes, objects, attributes, and operations.

Classes

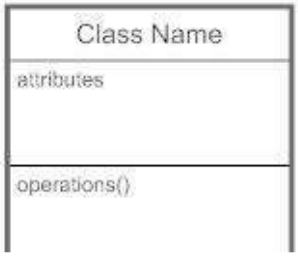
Classesrepresent an abstraction of entities with common characteristics. Associations represent the relationships between classes.

Illustrate classes with rectangles divided into compartments. Place the name of the class in the first partition (centered, bolded, and capitalized), list the attributes in the second partition (left-aligned, not bolded, and lowercase), and write operations into the third.

Class

Active Classes

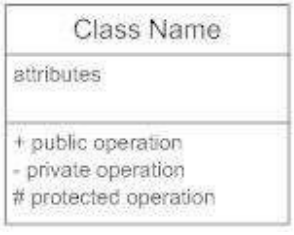
Active classes initiate and control the flow of activity, while passive classes store data and serve other classes. Illustrate active classes with a thicker border.



Active class

Visibility

Use visibility markers to signify who can access the information contained within a class. Private visibility, denoted with a - sign, hides information from anything outside the class partition. Public visibility, denoted with a + sign, allows all other classes to view the marked information. Protected visibility, denoted with a # sign, allows child classes to access information they inherited from a parent class.

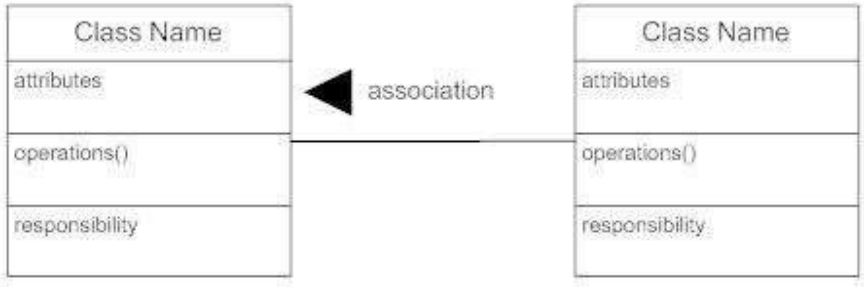


Visibility

Marker	Visibility
+	public
-	private
#	protected
~	package

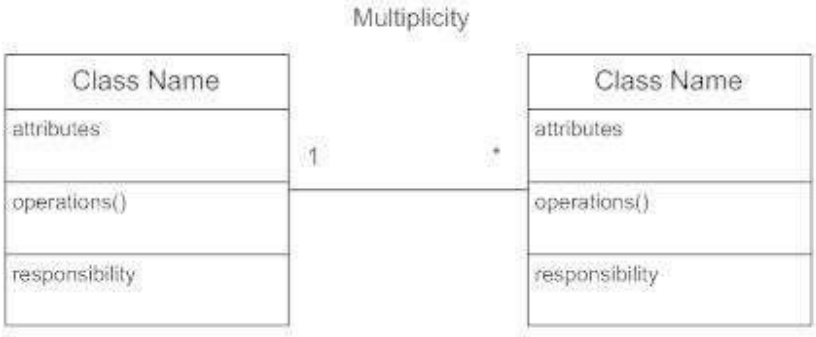
Associations

Associations represent static relationships between classes. Place association names above, on, or below the association line. Use a filled arrow to indicate the direction of the relationship. Place roles near the end of an association. Roles represent the way the two classes see each other.



Multiplicity (Cardinality)

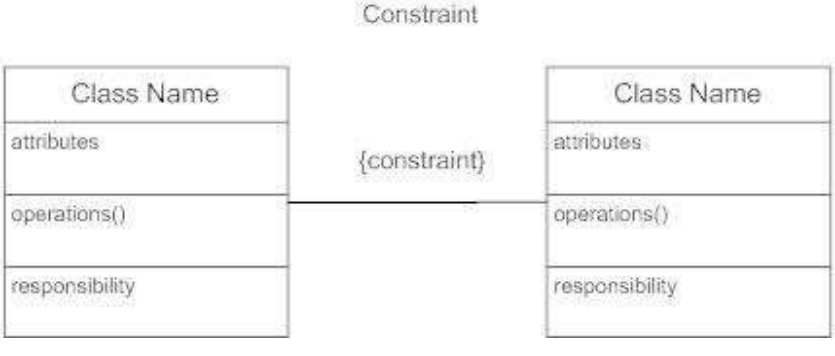
Place multiplicity notations near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for just one company.



Indicator	Meaning
0..1	Zero or one
1	One only
0..*	0 or more
1..* *	1 or more
n	Only n (where n > 1)
0..n	Zero to n (where n >1)
1..n	One to n (where n > 1)

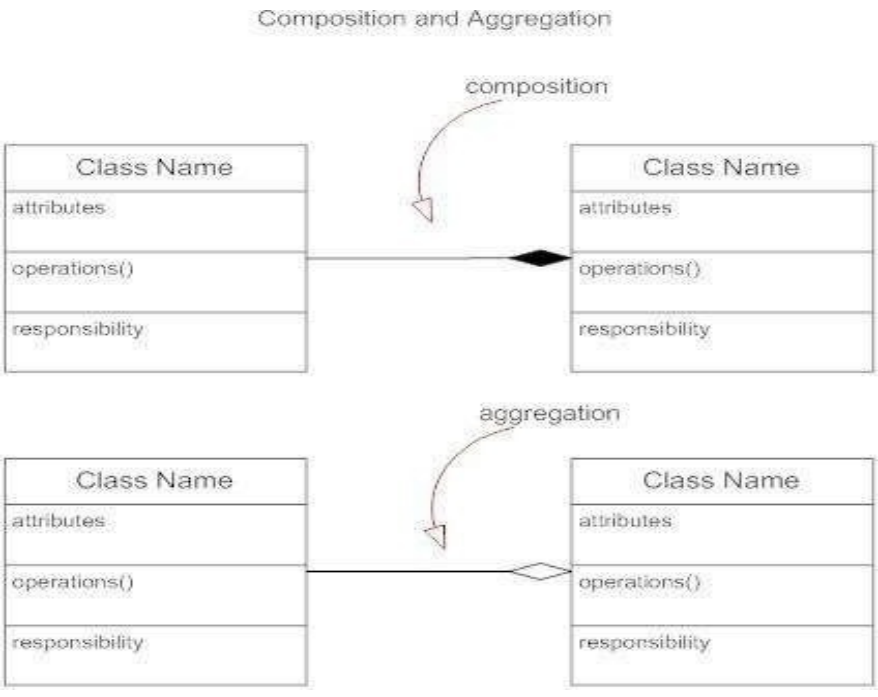
Constraint

Place constraints inside curly braces {}.



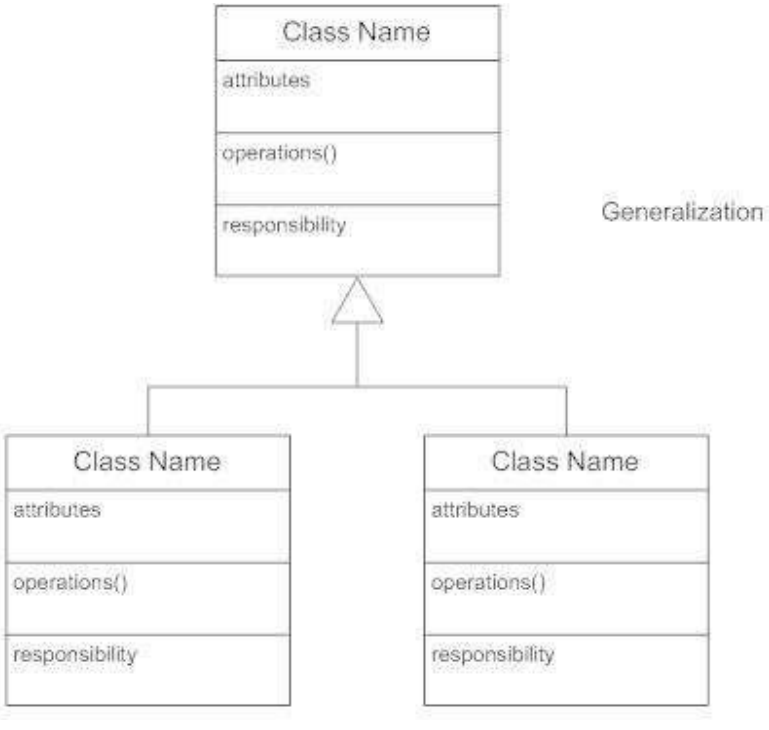
Composition and Aggregation

Composition is a special type of aggregation that denotes a strong ownership between Class A, the whole, and Class B, its part. Illustrate composition with a filled diamond. Use a hollow diamond to represent a simple aggregation relationship, in which the "whole" class plays a more important role than the "part" class, but the two classes are not dependent on each other. The diamond ends in both composition and aggregation relationships point toward the "whole" class (i.e., the aggregation).



Generalization

Generalization is another name for inheritance or an "is a" relationship. It refers to a relationship between two classes where one class is a specialized version of another. For example, Honda is a type of car. So the class Honda would have a generalization relationship with the class car.



In real life coding examples, the difference between inheritance and aggregation can be confusing. If you have an aggregation relationship, the aggregate (the whole) can access only the PUBLIC functions of the part class.

On the other hand, inheritance allows the inheriting class to access both the PUBLIC and PROTECTED functions of the superclass.

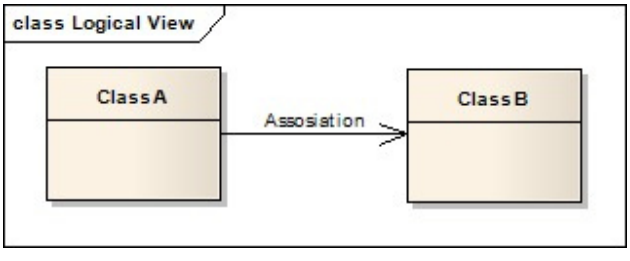
UML Class Diagram: Association, Aggregation and Composition

The UML Class diagram is used to visually describe the problem domain in terms of types of objects (classes) related to each other in different ways.

There are 3 primary inter-object relationships: A*ssociation*, A*ggregation*, and *Composition.* Using the right relationship line is important for placing implicit restrictions on the visibility and propagation of changes to the related classes, a matter which plays an important role in understanding and reducing system complexity.

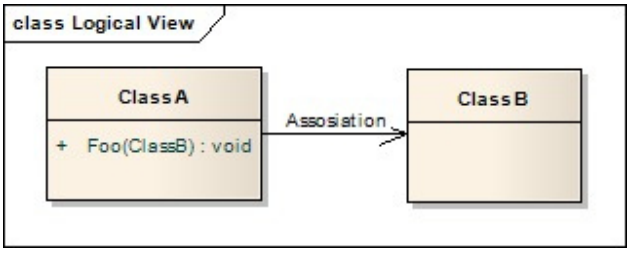
i)Association

The most abstract way to describe static relationship between classes is using the Association link, which simply states that there is some kind of a link or a dependency between two classes or more.



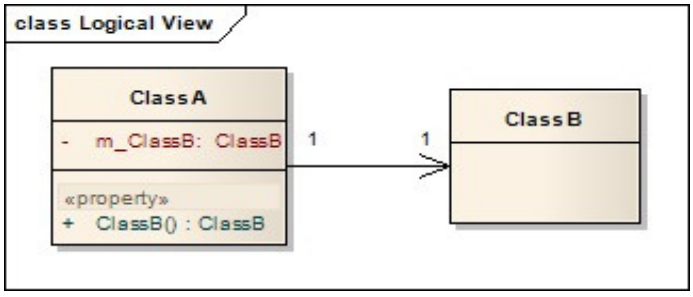
Weak Association

ClassA may be linked to ClassB in order to show that one of its methods includes parameter of ClassB instance, or returns instance of ClassB.



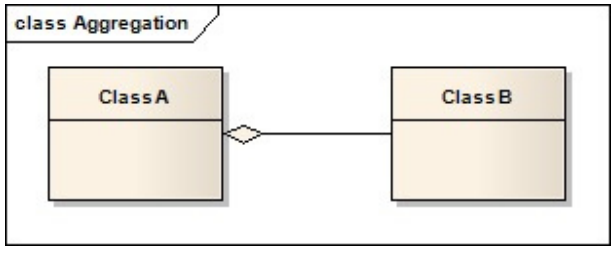
Strong Association

ClassA may also be linked to ClassB in order to show that it holds a reference to ClassB instance.

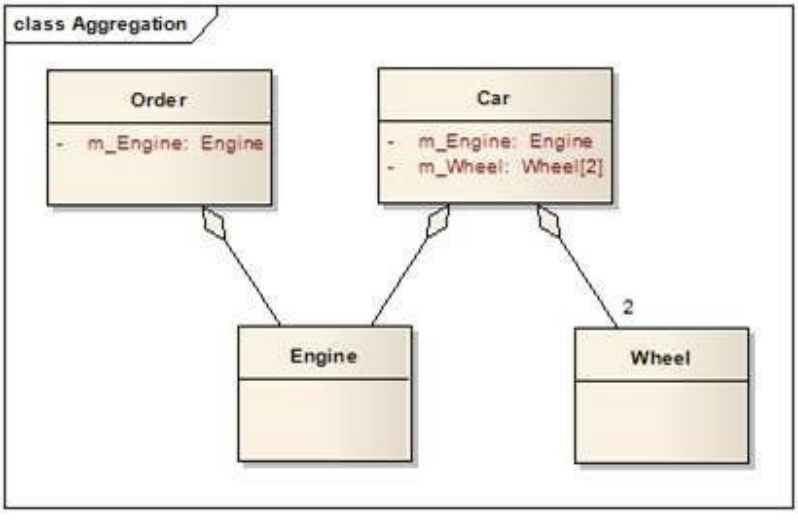


ii)Aggregation (Shared Association) (Weak Class)

In cases where there's a part-of relationship between ClassA (whole) and ClassB (part), we can be more specific and use the aggregation link instead of the association link, highlighting that the same ClassB instance can also be aggregated by other classes in the application (therefore aggregation is also known as shared association). Class B is weak Class.



It's important to note that the aggregation link doesn't state in any way that ClassA owns ClassB nor that there's a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that ClassA instance is not the exclusive container of ClassB instance, as in fact the same ClassB instance has another container/s.



Aggregation v.s. Association

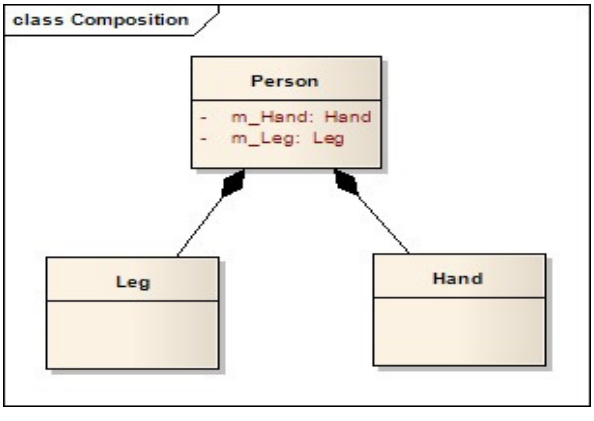
The association link can replace the aggregation link in every situation, while aggregation cannot replace association in situations where there's only a 'weak link'

between the classes, i.e. ClassA has method/s that contain parameter of ClassB, but ClassA doesn't hold reference to ClassB instance.

Martin Fowler suggest that the aggregation link should not be used at all because it has no added value and it disturb consistency, Quoting Jim Rumbaugh "Think of it as a modeling placebo".

iii)Composition (Not-Shared Association) (Strong Class)

We should be more specific and use the composition link in cases where in addition to the part-of relationship between ClassA and ClassB - there's a strong lifecycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result. Class Person is strong class.



The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts), meaning that the container object and its parts constitute a parent-child/s relationship.

Unlike association and aggregation, when using the composition relationship, the composed class cannot appear as a return type or parameter type of the composite class. Thus, changes to the composed class cannot propagate to the rest of the system. Consequently, usage of composition limits complexity growth as the system grows.

Clarification: It is possible for a class to be composed by more than one class. For example, ClassA may be composed by ClassB and ClassC. However, unlike aggregation, instances of ClassB and ClassC will never share the same ClassA instance. That would violate the *propagation of changes* principle. ClassB instance will have its own instance of ClassA, and ClassC instance will have its own instance of ClassA.

Output:

Experiment Number 5- Sequence Diagram

Experiment Name: Draw the sequence diagram for any two scenarios.

Description:

Sequence diagrams are a popular dynamic modeling solution in UML because they specifically focus on *lifelines*, or the processes and objects that live simultaneously, and the messages exchanged between them to perform a function before the lifeline ends.

What is a sequence diagram in UML?

To understand what a sequence diagram is, it's important to know the role of the Unified Modeling Language, better known as UML. UML is a modeling toolkit that guides the creation and notation of many types of diagrams, including behavior diagrams, interaction diagrams, and structure diagrams.

A sequence diagram is a type of interaction diagram because it describes how—and in what order—a group of objects works together. These diagrams are used by software developers and business professionals to understand requirements for a new system or to document an existing process. Sequence diagrams are sometimes known as event diagrams or event scenarios.

Note that there are two types of sequence diagrams: UML diagrams and code-based diagrams. The latter is sourced from programming code and will not be covered in this guide. Lucidchart's ~~UML diagramming software is~~ equipped with all the shapes and features you will need to model both.

Benefits of sequence diagrams

Sequence diagrams can be useful references for businesses and other organizations. Try drawing a sequence diagram to:

- Represent the details of a UML use case.
- Model the logic of a sophisticated procedure, function, or operation.
- See how objects and components interact with each other to complete a process.
- Plan and understand the detailed functionality of an existing or future scenario.

Use cases for sequence diagrams

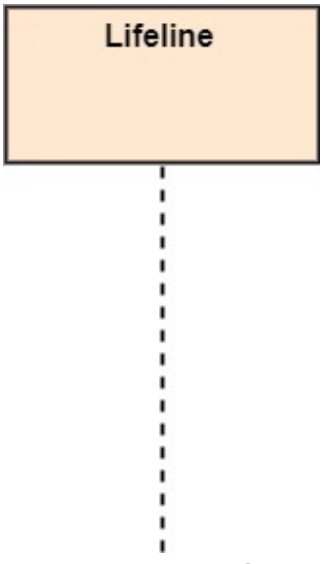
The following scenarios are ideal for using a sequence diagram:

- Usage scenario: A usage scenario is a diagram of how your system could potentially be used. It's a great way to make sure that you have worked through the logic of every usage scenario for the system.
- Method logic: Just as you might use a UML sequence diagram to explore the logic of a use case, you can use it to explore the logic of any function, procedure, or complex process.
- Service logic: If you consider a service to be a high-level method used by different clients, a sequence diagram is an ideal way to map that out.

Notations of a Sequence Diagram

Lifeline

An individual participant in the sequence diagram is represented by a lifeline. It is positioned at the top of the diagram.

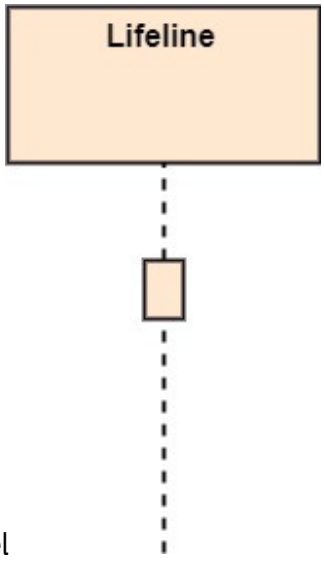


Actor

A role played by an entity that interacts with the subject is called as an actor. It is out of the scope of the system. It represents the role, which involves human users and external hardware or subjects. An actor may or may not represent a physical entity, but it purely depicts the role of an entity. Several distinct roles can be played by an actor or vice versa.

Activation

It is represented by a thin rectangle on the lifeline. It describes that time period in which an operation is performed by an element, such that the top and the bottom of the rectangle is associated with the



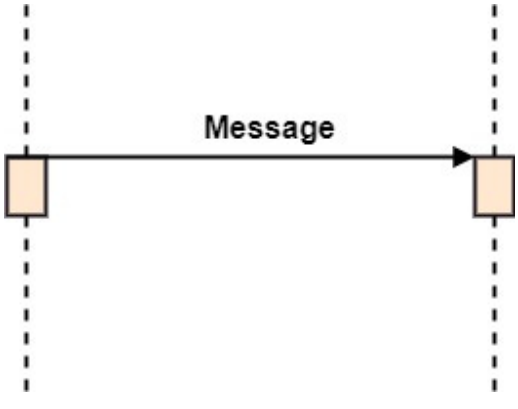
initiation and the completion time, each respectivel

Messages

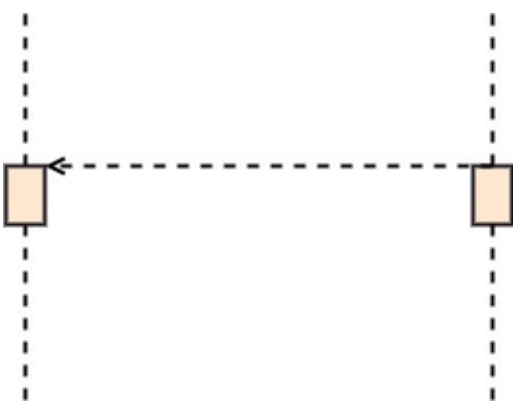
The messages depict the interaction between the objects and are represented by arrows. They are in the sequential order on the lifeline. The core of the sequence diagram is formed by messages and lifelines.

Following are types of messages enlisted below:

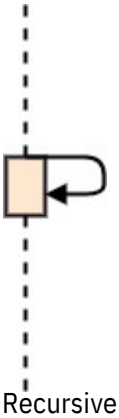
- Call Message: It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.



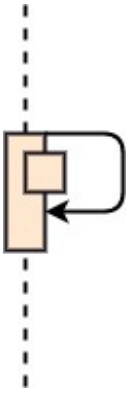
- Return Message:It defines a particular communication between the lifelines of interaction that represent the flow of information from the receiver of the corresponding caller message.



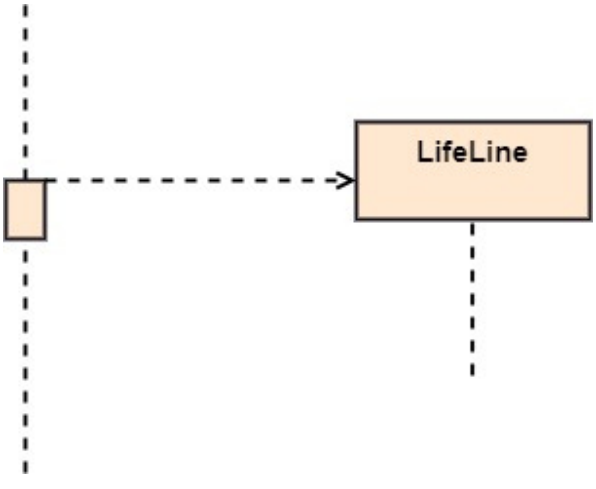
- Self Message: It describes a communication, particularly between the lifelines of an interaction that represents a message of the same lifeline, has been invoked.



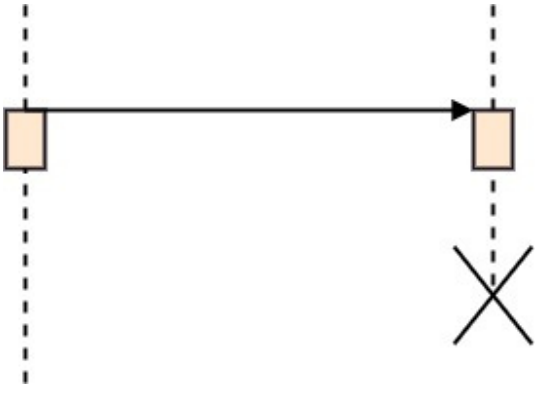
- Recursive Message: A self message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self message as it represents the recursive calls.



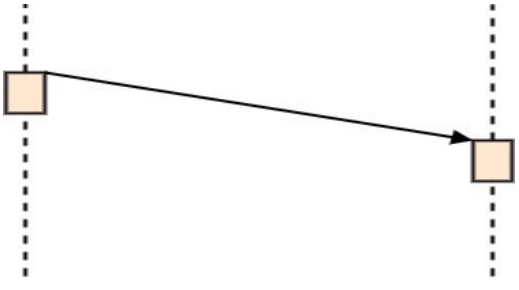
- Create Message: It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



- Destroy Message: It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.



- Duration Message: It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modeling a system.



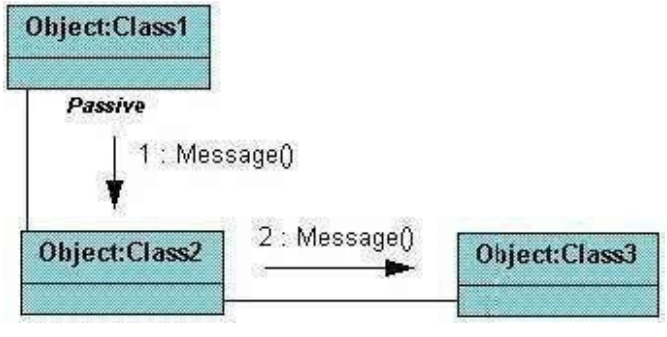
Experiment Number-6 Collboration Diagram

Experiment Name: Draw the collaboration diagram..

Description:

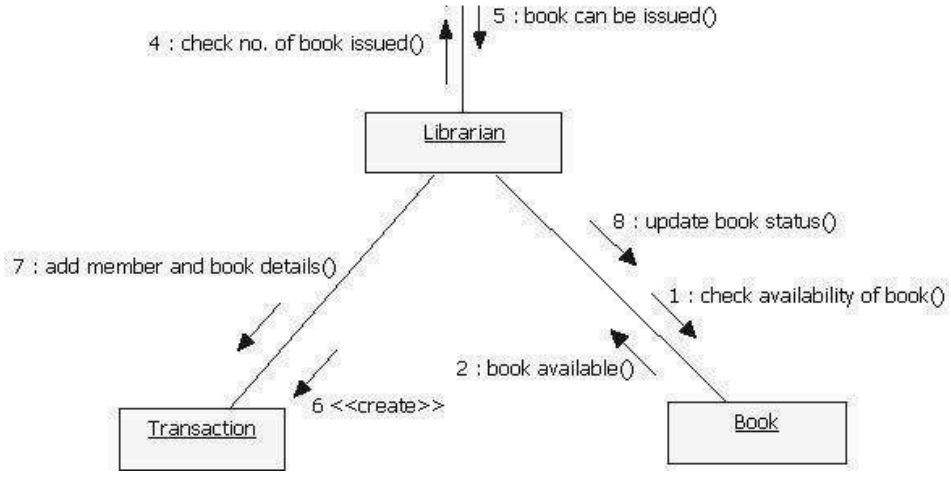
Collaboration diagrams ar relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in

UML. A simple 1, 2, 3... format can be used, as the example below shows, or for more detailed and complex diagrams a 1, 1.1 ,1.2, 1.2.1... scheme can be used.

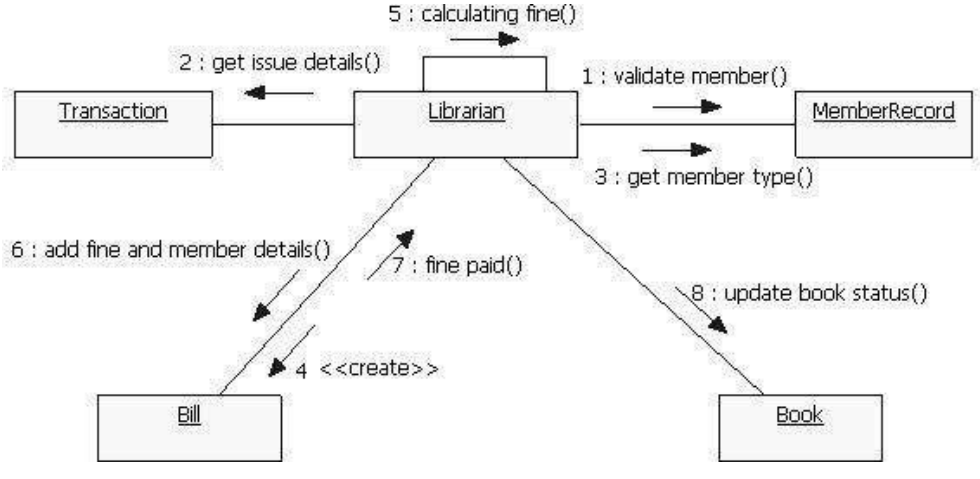


OUTPUT:

Collaboration diagram for issuing Book:



Collaboration diagram for returning Book:



Conclusion: The Collaboration diagram was made successfully by following the steps described above.

Experiment Number 7 State Chart diagram

Experiment Name: Draw the state chart diagram

Description:

A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events.

Activity, is a special kind of a Statechart diagram. As Statechart diagram defines the states, it is used to model the lifetime of an object.

Purpose of Statechart Diagrams

Statechart diagram is one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events. Statechart diagrams are useful to model the reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model lifetime of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system.

Following are the main purposes of using Statechart diagrams –

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

How to Draw a Statechart Diagram?

Statechart diagram is used to describe the states of different objects in its life cycle. Emphasis is placed on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as

the condition of objects when a particular event occurs.

Before drawing a Statechart diagram we should clarify the following points –

- Identify the important objects to be analyzed.
- Identify the states.
- Identify the events.

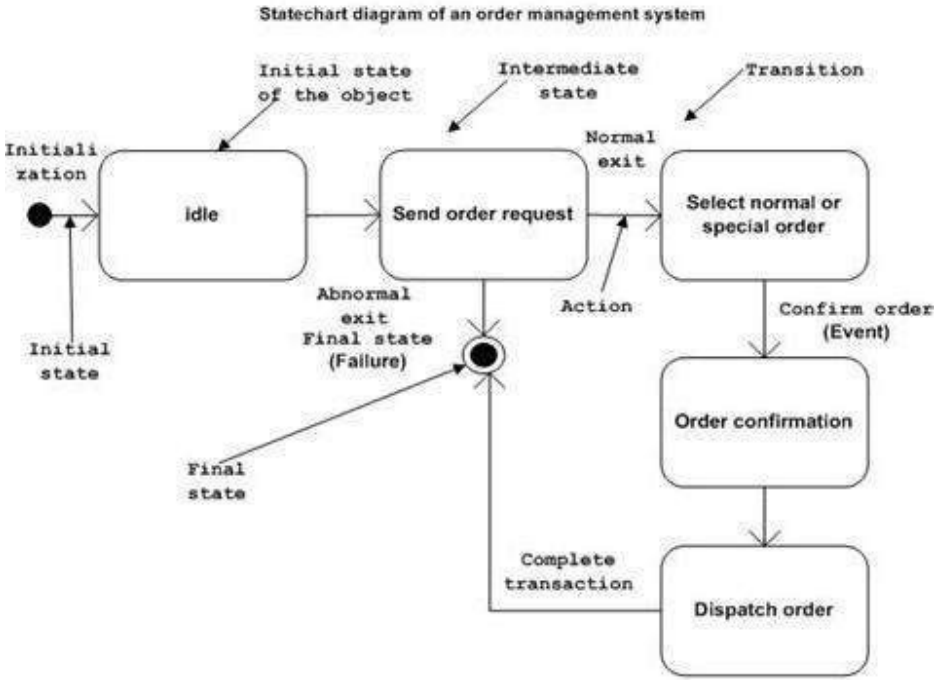
Following is an example of a Statechart diagram where the state of Order object is analyzed

The first state is an idle state from where the process starts. The next states are arrived for events like send request, confirm request, and dispatch order. These events are responsible for the state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exits. This abnormal exit may occur due to some problem in the system.

When the entire life cycle is complete, it is considered as a complete transaction

as shown in the following figure. The initial and final state of an object is also shown in the following figure.



WheretoUseStatechartDiagrams?

From the above discussion, we can define the practical applications of a Statechart diagram. Statechart diagrams are used to model the dynamic aspect of a system like other four diagrams discussed in this tutorial. However, it has some distinguishing characteristics for modeling the dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature.

Its specific purpose is to define the state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model the states and also the events operating on the system. When implementing a system, it is very important to clarify different states of an object during its life time and Statechart diagrams are used for this purpose. When these states and events are identified, they are used to model it and these models are used during the implementation of the system.

If we look into the practical implementation of Statechart diagram, then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behavior during its execution.

The main usage can be described as –

- To model the object states of a system.
- To model the reactive system. Reactive system consists of reactive objects.
- To identify the events responsible for state changes.
- Forward and reverse engineering.

Experiment 8

Experiment Name:Draw the component diagram.

Description: A component is something required to execute a stereotype function. Examples of stereotypes in components include executables, documents, database tables, files, and library files.

Components are wired together by using an *assembly connector* to connect the required interface of one component with the provided interface of another component. This illustrates the *service consumer - service provider* relationship between the two components.

An *assembly connector* is a "connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port."

When using a component diagram to show the internal structure of a component, the provided and required interfaces of the encompassing component can delegate to the corresponding interfaces of the contained components.

A *delegation connector* is a "connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the component's parts."[1]

The example above illustrates what a typical insurance policy administration system might look like. Each of the components depicted in the above diagram may have other component diagrams illustrating its internal structure.

component is represented by a rectangle with either the keyword "component" or a stereotype in the top right corner: a small rectangle with two even smaller rectangles jutting out on the left.

The lollipop, a small circle on a stick, represents an implemented or provided interface. The socket symbol is a semicircle on a stick that can fit around the lollipop. This socket is a dependency or needed interface.

The component diagram notation set now makes it one of the easiest UML diagrams to draw. Figure 1 shows a simple component diagram using the former UML 1.4 notation; the example shows a relationship between two components: an Order System component that uses the Inventory System component. As you can see, a component in UML 1.4 was drawn as a rectangle with two smaller rectangles obtruding from its left side.

Downloaded by Balak Ram (balakram1509@kccitm.edu.in)

Billing:- Any transaction shall be recorded in the form of a receipt and the same would be dispensed to the customer. The billing procedures are handled by the billing module that enable user to choose whether he wants the printed statement of the transaction or just the updation in his account.

Balance Enquiry:- Balance enquiry for any account linked to the card shall be facilitated.

Cancelling:- The customer shall abort a transaction with the press of a Cancel key. For example on entering a wrong depositing amount. In addition the user can also cancel the entire session by pressing the abort key and can start a fresh session all over again.

Map locating other machines:- The machine also has a facility of displaying the map that marks the locations of other ATM machines of the same bank in the entire city.

Mobile Bills Clearings:- The machine also allows the user to clear off his pending mobile bills there only, if the name of his operator is mentioned there in the list. The machine displays the list of the companies supported by that bank to the user.

EXPERIMENT 9

Name: Entity Relationship Diagram

Hardware Requirements: Pentium 4 processor (2.4 GHz), 128 Mb RAM, Standard keyboard n mouse, colored monitor.

Software Requirements: STAR UML

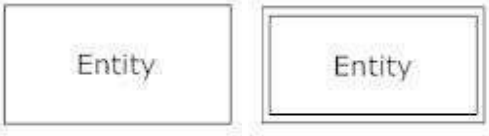
Objective: Prepare a Requirement document in by using Entity Relationship

Outcome: Able to visualize the requirements in Entity Relationship diagram.

Diagram. Description: An Entity Relationship (ER) Diagram is a type of flowchart that illustrates how "entities" such as people, objects or concepts relate to each other within a system. ER Diagrams are most often used to design or debug relational databases in the fields of software engineering, business information systems, education and research.

Components: An ER diagram is a means of visualizing how the information a system produces is related. There are five main components of an ERD:

Entities, which are represented by rectangles. An entity is an object or concept about which you want to store information. A weak entity is an entity that must defined by a foreign key relationship with another entity as it cannot be uniquely identified by its own attributes alone.



Actions, which are represented by diamond shapes, show how two entities share information in the database. In some cases, entities can be self-linked. For example, employees can supervise other employees.

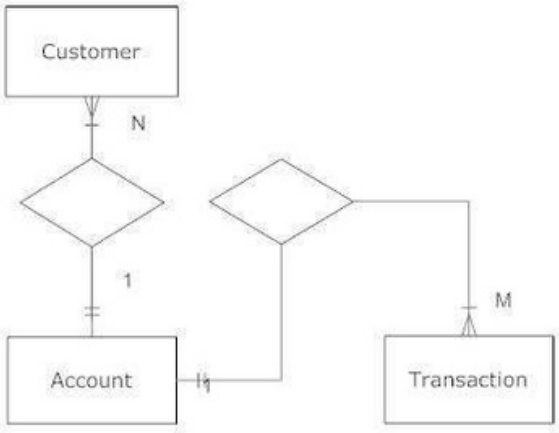
Attributes, which are represented by ovals. A key attribute is the unique, distinguishing characteristic of the entity. For example, an employee's social security number might be the employee's key attribute.

A multivalued attribute can have more than one value. For example, an employee entity can have multiple skill values.

A derived attribute is based on another attribute. For example, an employee's monthly salary is based on the employee's annual salary.

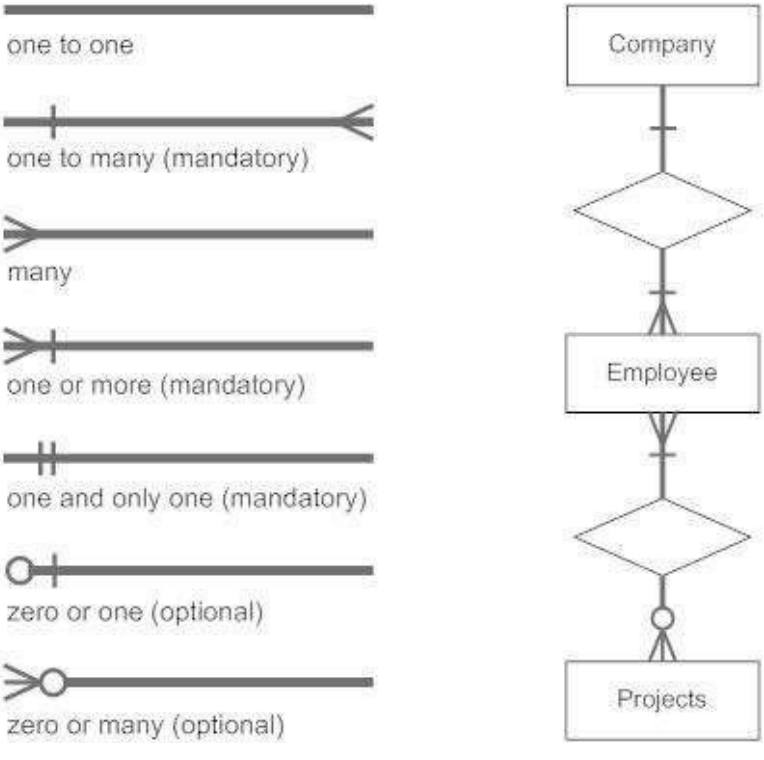
Connecting lines, solid lines that connect attributes to show the relationships of entities in the diagram.

Cardinality specifies how many instances of an entity relate to one instance of another entity. Ordinality is also closely linked to cardinality. While cardinality specifies the occurrences of a relationship, ordinality describes the relationship as either mandatory or optional. In other words, cardinality specifies the maximum number of relationships and ordinality specifies the absolute minimum number of relationships.



There are many notation styles that express cardinality.

Information Engineering Style



Chen Style

Ordinality - describes the minimum (optional vs mandatory)

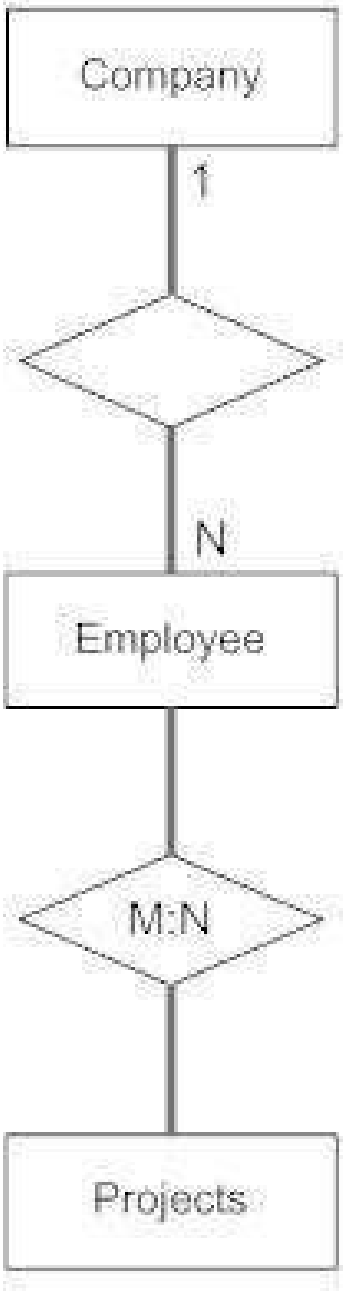
M:N

Cardinality - describes the maximum

1:N (n=0,1,2,3...)
one to zero or more

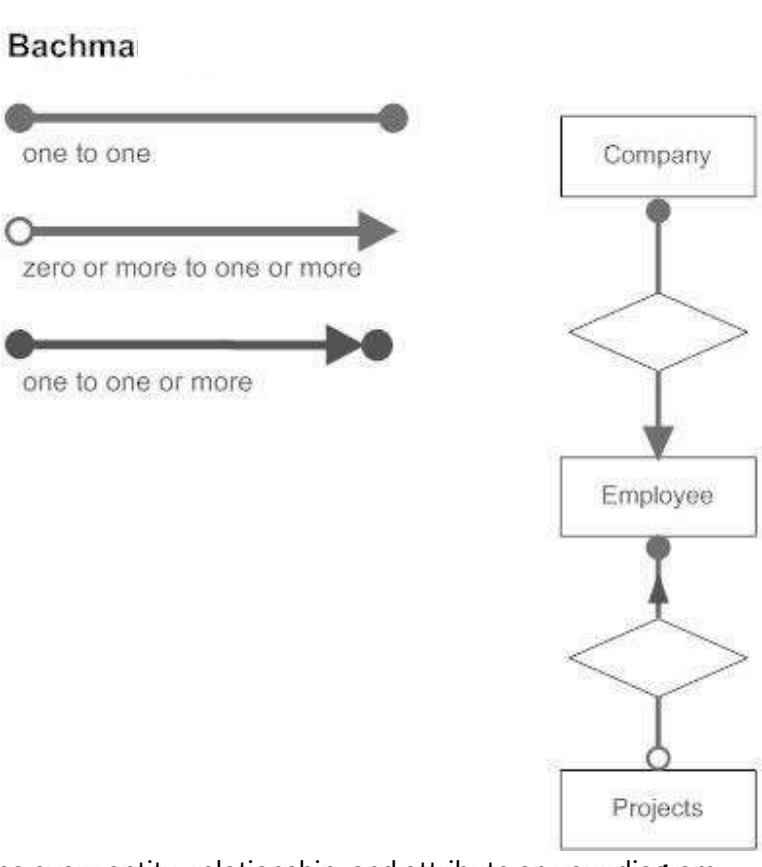
M:N (m and n=0,1,2,3...)
zero or more to zero or more
(many to many)

1:1
one to one



Tipsfor EffectiveERDiagrams

1. Make sure that each entity only appears once per diagram.



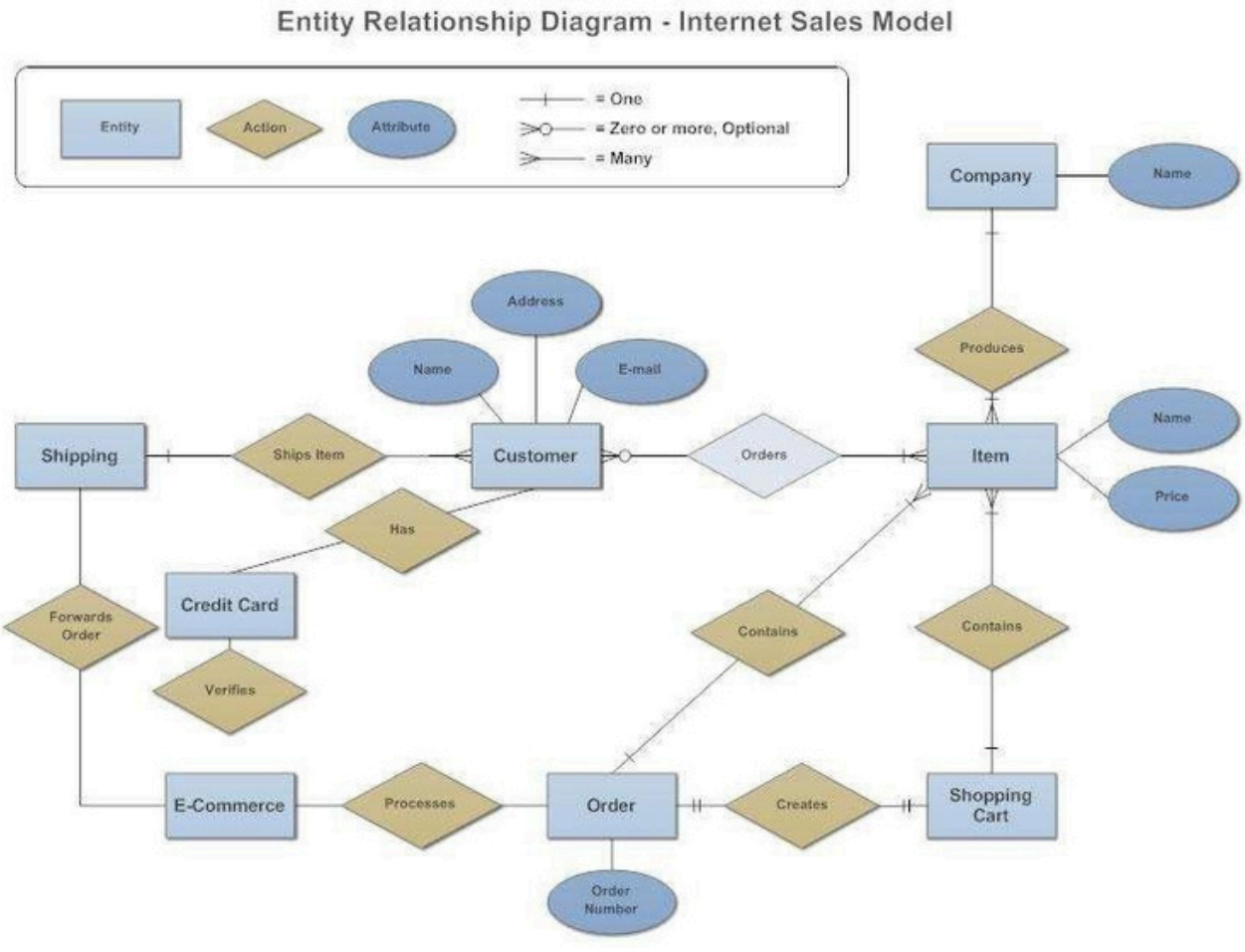
2. Name every entity, relationship, and attribute on your diagram.

3. Examine relationships between entities closely. Are they necessary? Are there any relationships missing? Eliminate any redundant relationships. Don't connect relationships to each other.

4. Use colors to highlight important portions of your diagram.

Experiment 10
Data Flow Diagram

3. Entity Relationship Diagram Example



Hardware Requirements: Pentium 4 processor (2.4 GHz), 128 Mb RAM, Standard keyboard n mouse, colored monitor.

Software Requirements: SmartDraw/MS Word, Windows XP.

Outcome: Can produce the requirements in Data Flow diagram.

Objective: Prepare a Requirement document in by using Data Flow Diagram.

Description: Data flow diagram is graphical representation of flow of data in an information system. It is capable of depicting incoming data flow, outgoing data flow and stored data. The DFD does not mention anything about how data flows through the system.

There is a prominent difference between DFD and Flowchart. The flowchart depicts flow of control in program modules. DFDs depict flow of data in the system at various levels. DFD does not contain any control or branch elements.

Components:

External Entity an outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system. They might be an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.

Process any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules. A short label is used to describe the process, such as “Submit payment.”

Data store files or repositories that hold information for later use, such as a database table or a membership form. Each data store receives a simple label, such as “Orders.”

Data flow the route that data takes between the external entities, processes and data stores. It portrays the interface between the other components and is shown with arrows, typically labeled with a short data name, like “Billing details.”

DFDrules

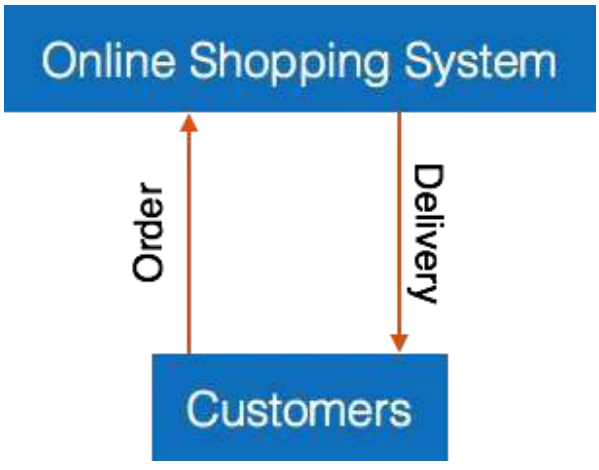
- Each process should have at least one input and an output.
- Each data store should have at least one data flow in and one data flow out.
- Data stored in a system must go through a process.
- All processes in a DFD go to another process or a data store.

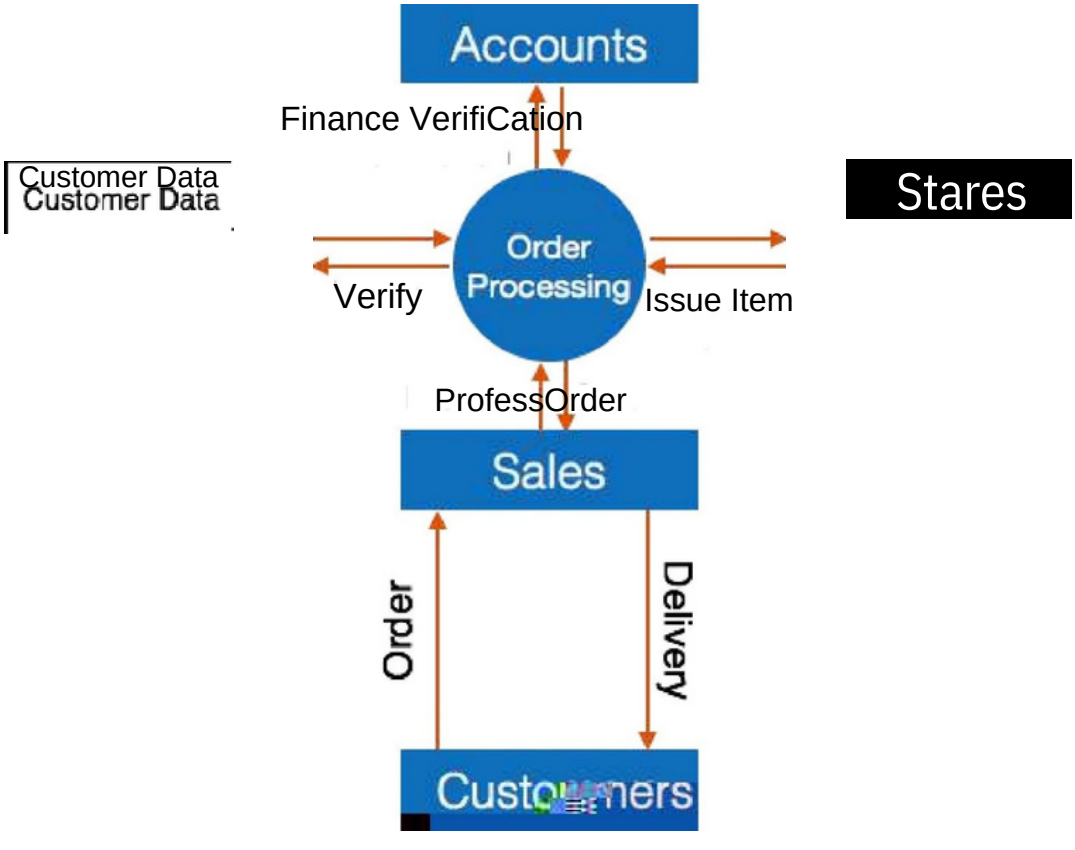
DFD Levels:

A data flow diagram can dive into progressively more detail by using levels and layers, zeroing in on a particular piece. DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond. 1.DFD Level 0 is also called a Context Diagram.

It’s a basic overview of the whole system or process being analyzed or modeled. It’s designed to be an at-a-glance view, showing the system as a single high-level process, with its relationship to external entities. It should be easily understood by a wide audience, including stakeholders, business analysts, data analysts and developers.

2.DFD Level 1 provides a more detailed breakout of pieces of the Context Level Diagram. You will highlight the main functions carried out by the system, as you break down the high-level process of the Context Diagram into its subprocesses.


Online Shopping System
Order
Delivery
Customers


Accounts
Finance VerifiCation
Customer Data
Customer Data
Order Processing
Verify
Issue Item
ProfessOrder
Sales
Order
Delivery
Customers

Stares