

Practical No. 01

Aim:- Identifying the Requirements from Problem Statements.

Introduction:-

Requirements identification is the first step of any software development project. Until the requirements of a client have been clearly identified, and verified, no other task (design, coding, testing) could begin. Usually business analysts having domain knowledge on the subject matter discuss with clients and decide what features are to be implemented.

In this experiment we will learn how to identify functional and non-functional requirements from a given problem statement. Functional and non-functional requirements are the primary components of a Software Requirements Specification.

Objectives:-

After completing this experiment you will be able to:

- Identify ambiguities, inconsistencies and incompleteness from a requirements specification
- Identify and state functional requirements
- Identify and state non-functional requirements

Requirements:-

Sommerville defines "requirement" [\[1\]](#) as a specification of what should be implemented. Requirements specify how the target system should behave. It specifies what to do, but not how to do. Requirements engineering refers to the process of understanding what a customer expects from the system to be developed, and to document them in a standard and easily readable and understandable format. This documentation will serve as reference for the subsequent design, implementation and verification of the system.

It is necessary and important that before we start planning, design and implementation of the software system for our client, we are clear about its requirements. If we don't have a clear vision of what is to be developed and what all features are expected, there would be serious problems, and customer dissatisfaction as well.

Characteristics of Requirements

Requirements gathered for any new system to be developed should exhibit the following three properties:

- **Unambiguity:** There should not be any ambiguity what a system to be developed should do. For example, consider you are developing a web application for your client. The client requires that enough number of people should be able to access the application simultaneously. What's the "enough number of people"? That could mean 10 to you, but, perhaps, 100 to the client. There's an ambiguity.

- **Consistency:** To illustrate this, consider the automation of a nuclear plant. Suppose one of the clients say that it the radiation level inside the plant exceeds R1, all reactors should be shut down. However, another person from the client side suggests that the threshold radiation level should be R2. Thus, there is an inconsistency between the two end users regarding what they consider as threshold level of radiation.
- **Completeness:** A particular requirement for a system should specify what the system should do and also what it should not. For example, consider a software to be developed for ATM. If a customer enters an amount greater than the maximum permissible withdrawal amount, the ATM should display an error message, and it should not dispense any cash.

Categorization of Requirements

Based on the target audience or subject matter, requirements can be classified into different types, as stated below:

- **User requirements:** They are written in natural language so that both customers can verify their requirements have been correctly identified
- **System requirements:** They are written involving technical terms and/or specifications, and are meant for the development or testing teams

Requirements can be classified into two groups based on what they describe:

- **Functional requirements (FRs):** These describe the functionality of a system -- how a system should react to a particular set of inputs and what should be the corresponding output.
- **Non-functional requirements (NFRs):** They are not directly related what functionalities are expected from the system. However, NFRs could typically define how the system should behave under certain situations. For example, a NFR could say that the system should work with 128MB RAM. Under such condition, a NFR could be more critical than a FR.

Non-functional requirements could be further classified into different types like:

- **Product requirements:** For example, a specification that the web application should use only plain HTML, and no frames
- **Performance requirements:** For example, the system should remain available 24x7
- **Organizational requirements:** The development process should comply to SEI CMM level 4

Identifying Functional Requirements

Given a problem statement, the functional requirements could be identified by focusing on the following points:

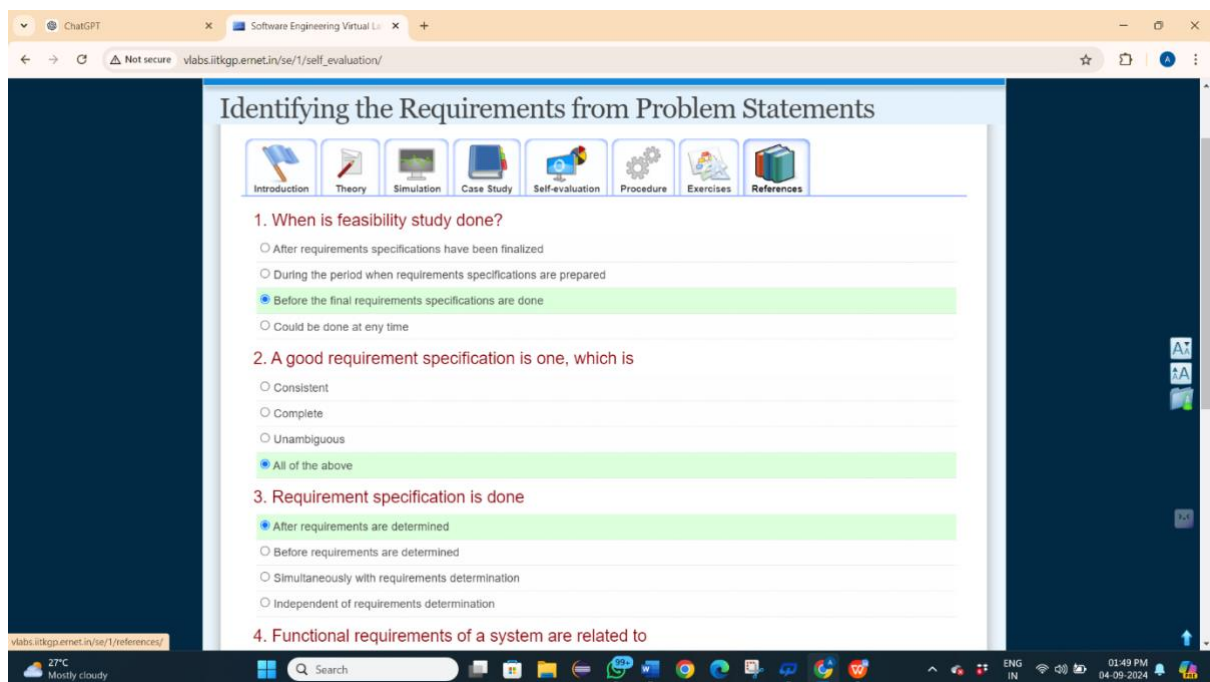
- Identify the high level functional requirements simply from the conceptual understanding of the problem. For example, a Library Management System, apart from anything else, should be able to issue and return books.

- Identify the cases where an end user gets some meaningful work done by using the system. For example, in a digital library a user might use the "Search Book" functionality to obtain information about the books of his interest.
- If we consider the system as a black box, there would be some inputs to it, and some output in return. This black box defines the functionalities of the system. For example, to search for a book, user gives title of the book as input and get the book details and location as the output.
- Any high level requirement identified could have different sub-requirements. For example, "Issue Book" module could behave differently for different class of users, or for a particular user who has issued the book thrice consecutively.

Preparing Software Requirements Specifications

Once all possible FRs and non-FRs have been identified, which are complete, consistent, and non-ambiguous, the Software Requirements Specification (SRS) is to be prepared. IEEE provides a template [\[iv\]](#), also available [here](#), which could be used for this purpose. The SRS is prepared by the service provider, and verified by its client. This document serves as a legal agreement between the client and the service provider. Once the concerned system has been developed and deployed, and a proposed feature was not found to be present in the system, the client can point this out from the SRS. Also, if after delivery, the client says a new feature is required, which was not mentioned in the SRS, the service provider can again point to the SRS. The scope of the current experiment, however, doesn't cover writing a SRS.

Self Evaluation:-



Simultaneously with requirements determination
Independent of requirements determination

4. Functional requirements of a system are related to

- ☒ Using the system (by users) to get some meaningful work done
- ☐ How the system functions under different constraints
- ☐ Whether they adhere to the organization policies

5. SRS refers to

- ☒ Software Requirements Specification
- ☐ System Resources Statement
- ☐ Statement of Reliability of System
- ☐ Standard Requirements Statement

6. The main objective behind preparing a SRS is to

- ☒ Let client and developers agree that they understand each other
- ☐ Formally note down the requirements
- ☐ Estimate the cost of development
- ☐ To judge whether the project could be undertaken

Submit Clear

MADE WITH **django** Sponsored by MHRD (NME-ICT) | Licensing Terms | Disclaimer
Copyright © 2010-2016 IIT Kharagpur

27°C Mostly cloudy 01:49 PM 04-09-2024

Exercise:-

Exercise 1

Limitations:

Submit

Following are the ambiguities

- ☐ None
- ☒ There's no specification when an auction gets over
- ☐ It doesn't say who are registered users
- ☐ No mention about what technology to be used for developing the application

Following are the inconsistencies

- ☐ None
- ☒ An item is said to be sold to the max bidder after auction is over; it can also be sold before the auction is over
- ☐ A registered user seems could be both buyer and seller

The problem statement is incomplete because

- ☐ None
- ☒ No mention of how a new user registers
- ☒ No mention of any dispute regarding the sold product
- ☒ No mention of what kind of products could be put on auction

Reset Submit

Result
Excellent!

27°C Mostly cloudy 01:54 PM 04-09-2024

Exercise 2

1. Identifying different functionalities to be obtained from a system

Limitations: This list is in no way complete; exercise #4 would address this again

[Submit](#)

Following functional requirements could be obtained from the requirements specifications

- ☒ Registration: New users have to register themselves online with the site and accept its terms & conditions ✓
- ☒ User Login: A user has to login into the site using his correct user ID & password ✓
- ☒ Upload Item for Auction: An authenticated user can upload an item into the site, which is to be put on auction subsequently ✓
- ☒ Auction Item: User puts an item already uploaded by him into the site on auction ✓
- ☐ Balance Check: Bidder should have enough bank balance to bid
- ☒ Bid for Item: Any registered & authenticated user of the system could place a bid for an item on auction ✓
- ☒ Win Auction: After the auction is over, the maximum bidder for the item owns the item post payment ✓
- ☒ Ship Item: Seller of the item ships the item to the auction owner after he (seller) receives the payment ✓
- ☐ Availability: The system should remain up & running before, during and after an auction
- ☒ Remove Item: Owner removes an item after uploading it, and doesn't put on auction ✓
- ☒ Remove auctioned item: System automatically removes an item from its inventory after it has been successfully auctioned ✓
- ☐ Site Support: Customer care for the website should provide 24x7 help over phone

[Reset](#) [Submit](#)

Result
Excellent!

Sponsored by MHRD (NME-ICT) | [Licensing Terms](#) | [Disclaimer](#) | [Except otherwise noted, content on this site is licensed under the CC-BY-NC-SA 4.0 license. See details.](#)

High UV Now

Search

ENG IN 01:55 PM 04-09-2024

Exercise 3

Learning Objectives:

1. Identifying characteristics that a system should have, but not done by the system itself

Limitations:

[Submit](#)

Following possible non-functional requirements could be identified from the requirements specifications

- ☐ The system provides option for online registration of new users
- ☒ The system should remain up & running throughout its working hours ✓
- ☐ System automatically removes an item from its database after it has been successfully auctioned
- ☒ Sessions of different users must not affect each other ✓
- ☐ Customer care for the website should provide 24x7 help over phone
- ☒ System should maintain privacy of their users and should not leak their information to third parties ✓
- ☒ System should be able to service 100 users simultaneously ✓
- ☒ System could remain unavailable for upto 2 hours for maintenance once in a quarter with 36 hour prior notice ✓

[Reset](#) [Submit](#)

Result
Excellent!

MADE WITH **django** | Sponsored by MHRD (NME-ICT) | [Licensing Terms](#) | [Disclaimer](#) | [Except otherwise noted, content on this site is licensed under the CC-BY-NC-SA 4.0 license. See details.](#)

Copyright © 2010-2016 IIT Kharagpur

High UV Now

Search

ENG IN 01:55 PM 04-09-2024

Practical No. 02

Aim:- E-R Modeling from the Problem Statements.

Introduction:-

Developing databases is a very important task to develop a system. Before going to form exact database tables and establishing relationships between them, we conceptually or logically can model our database using ER diagrams.

In this experiment we will learn how to find the entities, its attributes and how the relationships between the entities can be established for a system.

Objectives

After completing this experiment you will be able to:

- Identify entity sets, their attributes, and various relationships
- Represent the data model through ER diagram

Entity Relationship Model

Entity-Relationship model is used to represent a logical design of a database to be created. In ER model, real world objects (or concepts) are abstracted as entities, and different possible associations among them are modeled as relationships.

For example, student and school -- they are two entities. Students study in school. So, these two entities are associated with a relationship "Studies in".

As another example, consider a system where some job runs every night, which updates the database. Here, job and database could be two entities. They are associated with the relationship "Updates".

Entity Set and Relationship Set

An entity set is a collection of all similar entities. For example, "Student" is an entity set that abstracts all students. Ram, John are specific entities belonging to this set. Similarly, a "Relationship" set is a set of similar relationships.

Attributes of Entity

Attributes are the characteristics describing any entity belonging to an entity set. Any entity in a set can be described by zero or more attributes.

For example, any student has got a name, age, an address. At any given time a student can study only at one school. In the school he would have a roll number, and of course a grade in which he studies. These data are the attributes of the entity set Student.

Weak Entity

An entity set is said to be weak if it is dependent upon another entity set. A weak entity can't be uniquely identified only by its attributes. In other words, it doesn't have a super key.

Keys

One or more attribute(s) of an entity set can be used to define the following keys:

- **Super key:** One or more attributes, which when taken together, helps to uniquely identify an entity in an entity set. For example, a school can have any number of students. However, if we know grade and roll number, then we can uniquely identify a student in that school.
- **Candidate key:** It is a minimal subset of a super key. In other words, a super key might contain extraneous attributes, which do not help in identifying an object uniquely. When such attributes are removed, the key formed so is called a candidate key.
- **Primary key:** A database might have more than one candidate key. Any candidate key chosen for a particular implementation of the database is called a primary key.
- **Prime attribute:** Any attribute taking part in a super key

Entity Generalization and Specialization

Once we have identified the entity sets, we might find some similarities among them. For example, multiple person interacts with a banking system. Most of them are customers, and rest employees or other service providers. Here, customers, employees are persons, but with certain specializations. Or in other way, person is the generalized form of customer and employee entity sets.

ER model uses the "ISA" hierarchy to depict specialization (and thus, generalization).

Mapping Cardinalities

One of the main tasks of ER modeling is to associate different entity sets. Let's consider two entity sets E1 and E2 associated by a relationship set R. Based on the number of entities in E1 and E2 are associated with, we can have the following four type of mappings:

- **One to one:** An entity in E1 is related to at most a single entity in E2, and vice versa
- **One to many:** An entity in E1 could be related to zero or more entities in E2. Any entity in E2 could be related to at most a single entity in E1.
- **Many to one:** Zero or more number of entities in E1 could be associated to a single entity in E2. However, an entity in E2 could be related to at most one entity in E1.
- **Many to many:** Any number of entities could be related to any number of entities in E2, including zero, and vice versa.

ER Diagram

From a given problem statement we identify the possible entity sets, their attributes, and relationships among different entity sets. Once we have these information, we represent them pictorially, called an entity-relationship (ER) diagram.

Importance of ER modeling

Figure - 01 shows the different steps involved in implementation of a (relational) database.

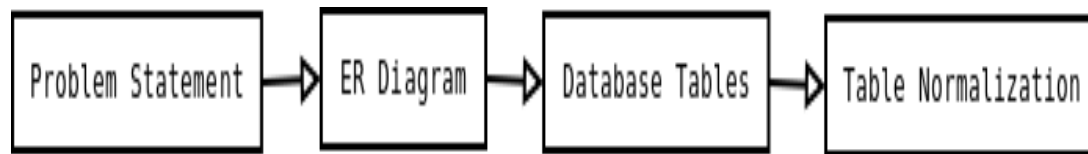




Figure - 01: Steps to implement a RDBMS

Given a problem statement, the first step is to identify the entities, attributes and relationships. We represent them using an ER diagram. Using this ER diagram, table structures are created, along with required constraints. Finally, these tables are normalized in order to remove redundancy and maintain data integrity. Thus, to have data stored efficiently, the ER diagram is to be drawn as much detailed and accurate as possible.

Graphical Notations for ER Diagram

Term	Notation	Remarks
Entity set		Name of the set is written inside the rectangle
Attribute		Name of the attribute is written inside the ellipse
Entity with attributes		Roll is the primary key; denoted with an underline
Weak entity set		
Relationship set		Name of the relationship is written inside the diamond
Related entity sets		

Term	Notation	Remarks
Relationship cardinality		A person can own zero or more cars but no two persons can own the same car
Relationship with weak entity set		

Self Evaluation:-

Software Engineering Virtual Lab

Not secure vlabs.itkgp.emet.in/se/4/self_evaluation/

E-R Modeling from the Problem Statements

Introduction Theory Simulation Case Study Self-evaluation Procedure Exercises References

- Entity Relationship model is used to represent the database at a
 - ☒ Logical level
 - ☐ Physical level
- In ER model an entity is usually related to another entity. The relationship is indicated by a
 - ☐ Rectangle
 - ☐ Ellipse
 - ☐ Triangle
 - ☒ Diamond
- An entity set could be related to itself
 - ☒ True
 - ☐ False
- A prime attributes is indicated by
 - ☒ Underline inside an ellipse
 - ☐ Double ellipse
 - ☐ Dotted ellipse
 - ☐ Dotted rectangle
- Students study in schools -- this is a
 - ☐ 1:1 mapping
 - ☐ 1:N mapping
 - ☒ M:1 mapping
 - ☐ M:N mapping

Submit Clear

25°C Partly cloudy 08:38 PM 04-09-2024

Exercise:-**Exercise 1**

Select 1

From the following problem statement identify the possible entity sets, their attributes, and relationships.

SE VLabs Inc. is a young company with a few departments spread across the country. As of now, the company has a strength of 200+ employees.

Each employee works in a department. While joining, a person has to provide a lot of personal and professional details including name, address, phone #, mail address, date of birth, and so on. Once all these information are furnished, a unique ID is generated for each employee. He is then assigned a department in which he will work.

There are around ten departments in the company. Unfortunately, two departments were given same names. However, departments too have ID's, which are unique.

Note: Try to use the features of the interface provided to capture as much details as possible.


Table #1: Add entities

Entity	Weak	Add
<input type="text"/>	<input type="checkbox"/>	

Table #2: Add attributes for each entity

Entity	Attribute	Primary Key	Add
employee	<input type="text"/>	<input type="checkbox"/>	


Table #3: Define relationship between two entities

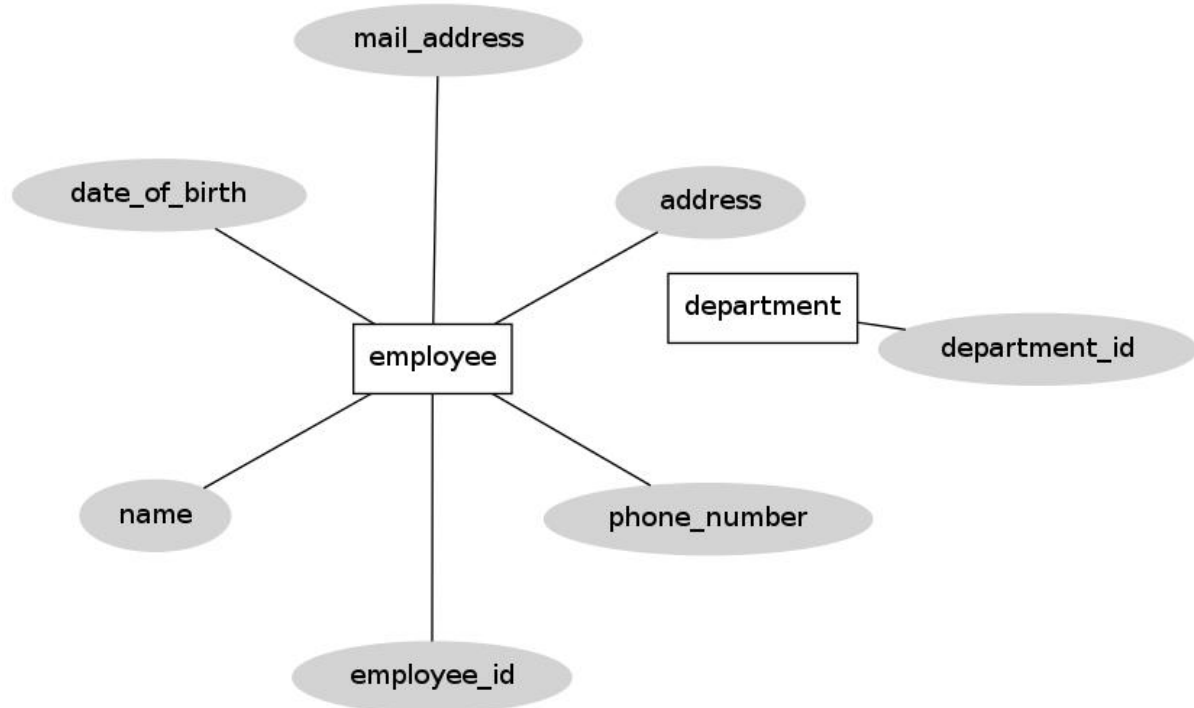
Entity	Relation	Entity	Constraint	Add
employee	<input type="text"/>	department	Many To One	

Table #4: Entities and their attributes

Entity	Attributes	Weak
employee	<ul style="list-style-type: none"> employee_id name address phone_number mail_address date_of_birth 	No
department	<ul style="list-style-type: none"> department_id 	No

Table #5: Relationships between entities

Entity	Relation	Entity	Constraint Type	Remove
employee	works_in	department	Many To One	



Exercise 2

Select 2

Draw an ER diagram for the following problem:

The latest cab services agency in the city has approached you to develop a Cab Management System for them. They would be using this software to efficiently manage and track different cabs that are operated by them.

Cabs are solely owned by the agency. They hire people in contracts to drive the cabs. A cab can be uniquely identified by, like any other vehicle in the country, its license plate. A few different categories of cars are available from different manufacturers. And a few of them are AC cars.

Cab drivers are given a identification card while joining. The ID card contains his name, permanent address, phone number, date of joining, duration of contract. Also, an unique alphanumeric code is assigned to each number.

The agency provides service from 8 AM to 8 PM. Whenever any passenger books a cab, an available cab is allocated for him. The booking receipt given to the passenger contains the car #, source and destination places. Once he reaches the destination, he signs on a duplicate copy of the receipt and gives back to the driver. Driver must submit this duplicate copy signed by the passenger at the agency for confirmation.

To evaluate their quality of service, the agency also wants a (optional) customer satisfaction survey, where passengers can provide feedback about their journey through the agency's website.

Note: This exercise is adapted from [ii]

 Submit

Table #4: Entities and their attributes

Entity	Attributes	Weak
cab	<ul style="list-style-type: none"> LicensePlate Manufacturer Category IsAC 	No
driver	<ul style="list-style-type: none"> DriverID Name Address Phone DateOfJoining ContractDuration 	No
passenger	<ul style="list-style-type: none"> PassengerID Name Phone 	No
booking	<ul style="list-style-type: none"> BookingID Source Destination BookingTime PassengerSignature 	No
feedback	<ul style="list-style-type: none"> FeedbackID PassengerID Rating Comments 	No
receipt	<ul style="list-style-type: none"> ReceiptID BookingID SignedCopy 	No

Table #5: Relationships between entities

Entity	Relation	Entity	Constraint Type	Remove
cab	is_driven_by	driver	One To Many	
cab	is_assigned_to	booking	One To One	
passenger	gives	feedback	One To Many	
passenger	makes	booking	One To Many	
driver	handles	booking	One To Many	
booking	generates	receipt	One To One	

Practical No. 03

Aim:- Modeling UML Use Case Diagrams and Capturing Use Case Scenarios

Introduction:-

Objectives

After completing this experiment you will be able to:

- How to identify different actors and use cases from a given problem statement
- How to associate use cases with different types of relationships
- How to draw a use-case diagram

Use case diagrams

Use case diagrams belong to the category of behavioural diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform, one or more actors, and dependencies among them.

Actor:- An actor can be defined as [\[1\]](#) an object or set of objects, external to the system, which interacts with the system to get some meaningful work done. Actors could be human, devices, or even other systems.

For example, consider the case where a customer *withdraws cash* from an ATM. Here, customer is a human actor.

Actors can be classified as below [\[2\]](#), [\[i\]](#):

- **Primary actor:** They are principal users of the system, who fulfill their goal by availing some service from the system. For example, a customer uses an ATM to withdraw cash when he needs it. A customer is the primary actor here.
- **Supporting actor:** They render some kind of service to the system. "Bank representatives", who replenishes the stock of cash, is such an example. It may be noted that replenishing stock of cash in an ATM is not the prime functionality of an ATM.

In a use case diagram primary actors are usually drawn on the top left side of the diagram.

Use Case

A use case is simply [\[1\]](#) a functionality provided by a system.

Continuing with the example of the ATM, *withdraw cash* is a functionality that the ATM provides. Therefore, this is a use case. Other possible use cases includes, *check balance*, *change PIN*, and so on.

Use cases include both successful and unsuccessful scenarios of user interactions with the system. For example, authentication of a customer by the ATM would fail if he enters wrong PIN. In such case, an error message is displayed on the screen of the ATM.

Subject:- Subject is simply [iii] the system under consideration. Use cases apply to a subject. For example, an ATM is a subject, having multiple use cases, and multiple actors interact with it. However, one should be careful of external systems interacting with the subject as actors.

Graphical Representation

An actor is represented by a stick figure and name of the actor is written below it. A use case is depicted by an ellipse and name of the use case is written inside it. The subject is shown by drawing a rectangle. Label for the system could be put inside it. Use cases are drawn inside the rectangle, and actors are drawn outside the rectangle, as shown in figure - 01.

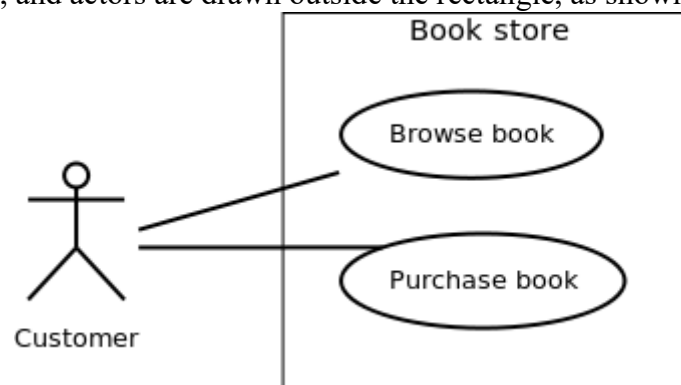


Figure - 01: A use case diagram for a book store

Association between Actors and Use Cases

A use case is triggered by an actor. Actors and use cases are connected through binary associations indicating that the two communicates through message passing.

An actor must be associated with at least one use case. Similarly, a given use case must be associated with at least one actor. Association among the actors are usually not shown. However, one can depict the class hierarchy among actors.

Use Case Relationships

Three types of relationships exist among use cases:

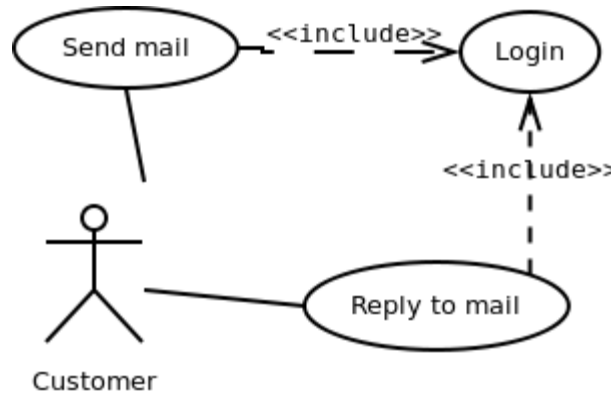
- Include relationship
- Extend relationship
- Use case generalization

Include Relationship

Include relationships are used to depict common behaviour that are shared by multiple use cases. This could be considered analogous to writing functions in a program in order to avoid repetition of writing the same code. Such a function would be called from different points within the program.

Example

For example, consider an email application. A user can send a new mail, reply to an email he has received, or forward an email. However, in each of these three cases, the user must be logged in to perform those actions. Thus, we could have a *login* use case, which is included by *compose mail*, *reply*, and *forward email* use cases. The relationship is shown in figure - 02.



Notation

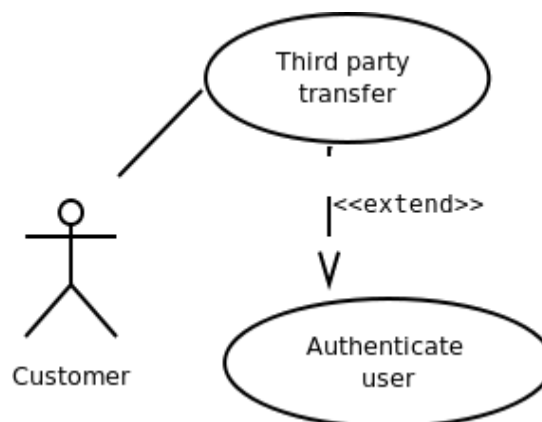
Include relationship is depicted by a dashed arrow with a «include» stereotype from the including use case to the included use case.

Extend Relationship

Use case extensions are used to depict any variation to an existing use case. They are used to specify the changes required when any assumption made by the existing use case becomes false [\[iv, v\]](#).

Example

Let's consider an online bookstore. The system allows an authenticated user to buy selected book(s). While the order is being placed, the system also allows to specify any special shipping instructions [\[vii\]](#), for example, call the customer before delivery. This *Shipping Instructions* step is optional, and not a part of the main *Place Order* use case. Figure - 03 depicts such relationship.



Notation

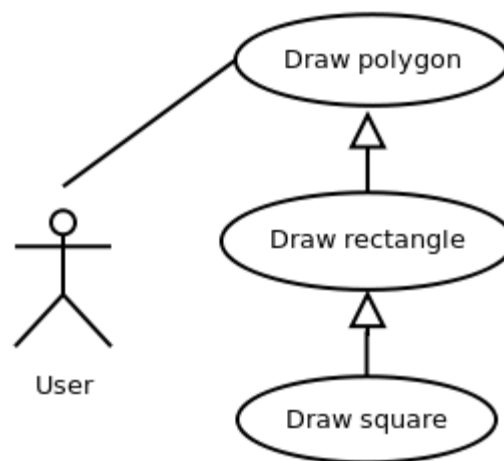
Extend relationship is depicted by a dashed arrow with a «extend» stereotype from the extending use case to the extended use case.

Generalization Relationship

Generalization relationships are used to represent the inheritance between use cases. A derived use case specializes some functionality it has already inherited from the base use case.

Example

To illustrate this, consider a graphical application that allows users to draw polygons. We could have a use case *draw polygon*. Now, rectangle is a particular instance of polygon having four sides at right angles to each other. So, the use case *draw rectangle* inherits the properties of the use case *draw polygon* and overrides its drawing method. This is an example of generalization relationship. Similarly, a generalization relationship exists between *draw rectangle* and *draw square* use cases. The relationship has been illustrated in figure - 04.



Notation

Generalization relationship is depicted by a solid arrow from the specialized (derived) use case to the more generalized (base) use case.

Identifying Actors

Given a problem statement, the actors could be identified by asking the following questions [\[2\]](#):

- Who gets most of the benefits from the system? (The answer would lead to the identification of the primary actor)
- Who keeps the system working? (This will help to identify a list of potential users)
- What other software / hardware does the system interact with?
- Any interface (interaction) between the concerned system and any other system?

Identifying Use cases

Once the primary and secondary actors have been identified, we have to find out their goals i.e. what are the functionality they can obtain from the system. Any use case name should start with a verb like, "Check balance".

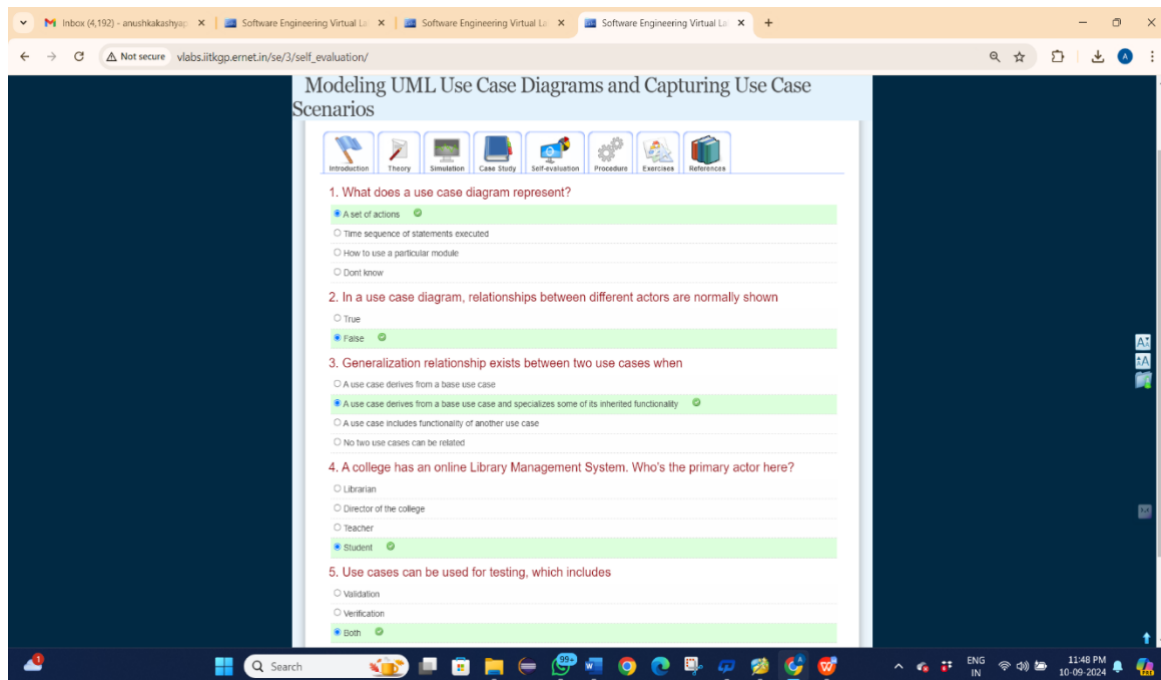
Guidelines for drawing Use Case diagrams

Following general guidelines could be kept in mind while trying to draw a use case diagram [\[1\]](#):

- Determine the system boundary
- Ensure that individual actors have well-defined purpose
- Use cases identified should let some meaningful work done by the actors
- Associate the actors and use cases -- there shouldn't be any actor or use case floating without any connection
- Use include relationship to encapsulate common behaviour among use cases , if any

Also look at [\[ix\]](#) for further tips.

Self Evaluation:-



Practical No. 04

Aim:- Modelling Data Flow Diagrams

Introduction:-

Information Systems (IS) help in managing and updating the vast business-related information. Before designing such an IS, it is helpful to identify the various stakeholders, and the information that they would be exchanging with the system. An IS, however, is a large software comprised of several modules, which, in turn, share the process the available data. These data are often stored in databases for further references. A Data Flow Diagram (DFD) is used to pictorially represent the functionalities of the ISs by focusing on the sources and destinations of the data flowing in the system.

Objectives:-

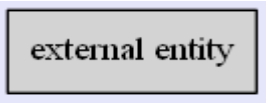


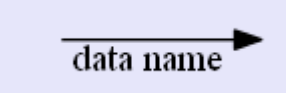
After completing this experiment you will be able to:

- Identify external entities and functionalities of any system
- Identify the flow of data across the system
- Represent the flow with Data Flow Diagrams

Data Flow Diagram

DFD provides the functional overview of a system. The graphical representation easily overcomes any gap between 'user and system analyst' and 'analyst and system designer' in understanding a system. Starting from an overview of the system it explores detailed design of a system through a hierarchy. DFD shows the external entities from which data flows into the process and also the other flows of data within a system. It also includes the transformations of data flow by the process and the data stores to read or write a data.

Graphical notations for Data Flow Diagram

Term	Notation	Remarks
External entity		Name of the external entity is written inside the rectangle
Process		Name of the process is written inside the circle
Data store		A left-right open rectangle is denoted as data store; name of the data store is written inside the shape
Data flow		Data flow is represented by a directed arc with its data name

Explanation of Symbols used in DFD:-

- Process:** Processes are represented by circle. The name of the process is written into the circle. The name of the process is usually given in such a way that represents the functionality of the process. More detailed functionalities can be shown in the next Level if it is required. Usually it is better to keep the number of processes less than 7 [\[1\]](#). If we see that the number of processes becomes more than 7 then we should combine some the processes to a single one to reduce the number of processes and further decompose it to the next level [\[2\]](#).
- External entity:** External entities are only appear in context diagram [\[2\]](#). External entities are represented by a rectangle and the name of the external entity is written into the shape. These send data to be processed and again receive the processed data.
- Data store:** Data stores are represented by a left-right open rectangle. Name of the data store is written in between two horizontal lines of the open rectangle. Data stores are used as repositories from which data can be flown in or flown out to or from a process.
- Data flow:** Data flows are shown as a directed edge between two components of a Data Flow Diagram. Data can flow from external entity to process, data store to process, in between two processes and vice-versa.

Context diagram and leveling DFD:-

We start with a broad overview of a system represented in level 0 diagram. It is known as context diagram of the system. The entire system is shown as single process and also the interactions of external entities with the system are represented in context diagram. Further we split the process in next levels into several numbers of processes to represent the detailed functionalities performed by the system. Data stores may appear in higher level DFDs.

Numbering of processes : If process 'p' in context diagram is split into 3 processes 'p1', 'p2' and 'p3' in next level then these are labeled as 0.1, 0.2 and 0.3 in level 1 respectively. Let the process 'p3' is again split into three processes 'p31', 'p32' and 'p33' in level 2, so, these are labeled as 0.3.1, 0.3.2 and 0.3.3 respectively and so on.

Balancing DFD: The data that flow into the process and the data that flow out to the process need to be match when the process is split into in the next level[2]. This is known as balancing a DFD.

See simulation[ii] and case study[iii] of the experiment to understand data flow diagram in more real context.

Note :

1. External entities only appear in context diagram[2] i.e, only at level 0.
2. Keep number of processes at each level less than 7[i].
3. Data flow is not possible in between two external entities and in between two data stores[i].
4. Data cannot flow from an External entity to a data store and vice-versa[i].

Case Study

1 : A Library Information System for SE VLabs Institute

The SE VLabs Institute has been recently setup to provide state-of-the-art research facilities in the field of Software Engineering. Apart from research scholars (students) and professors, it also includes quite a large number of employees who work on different projects undertaken by the institution.

As the size and capacity of the institute is increasing with the time, it has been proposed to develop a Library Information System (LIS) for the benefit of students and employees of the institute. LIS will enable the members to borrow a book (or return it) with ease while sitting at his desk/chamber. The system also enables a member to extend the date of his borrowing if no other booking for that particular book has been made. For the library staff, this system aids them to easily handle day-to-day book transactions. The librarian, who has administrative privileges and complete control over the system, can enter a new record into the system when a new book has been purchased, or remove a record in case any book is taken off the shelf.

Any non-member is free to use this system to browse/search books online. However, issuing or returning books is restricted to valid users (members) of LIS only.

The final deliverable would be a web application, which should run only within the institute LAN. Although this reduces security risk of the software to a large extent, care should be taken no confidential information (eg., passwords) is stored in plain text.

Figure 1 shows the context-level DFD for LIS. The entire system is represented with a single circle (process). The external entities interacting with this system are members of LIS, library staff, librarian, and non-members of LIS. Two databases are used to keep track of member information and details of books in the library.

Let us focus on the external entity, Member. In order to issue or return books a member has to login to the system. The data flow labeled with "Login credentials" indicate the step when a member authenticates himself by providing required information (user ID, password). The system in turn verifies the user credentials using information stored in the members database. If all information are not provided correctly, the user is shown a login failure message. Otherwise, the user can continue with his operation. Note that a DFD does not show conditional flows. It can only summarize the information flowing in and out of the system.

The data flow with the label "Requested book details" identify the information that the user has to provide in order to issue a book. LIS checks with the books database whether the given book is available. After a book has been issued, the transaction details is provided to the member.

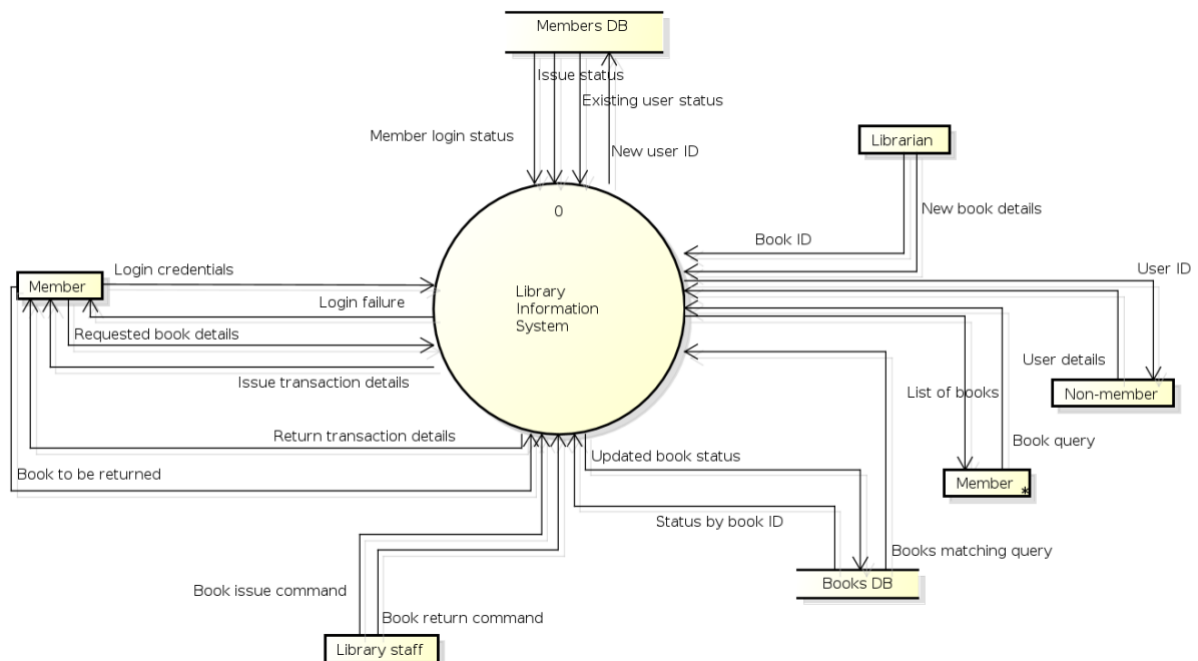


Figure 1: Context-level DFD for Library Information System

The level-1 DFD is shown in figure 2. Here, we split the top-level view of the system into multiple logical components. Each process has a name, and a dotted-decimal number in the form 1.x. For example, the process "Issue book" has the number 1.2, which indicates that in the level 1 DFD the concerned process is numbered 2. Other processes are numbered in a similar way.

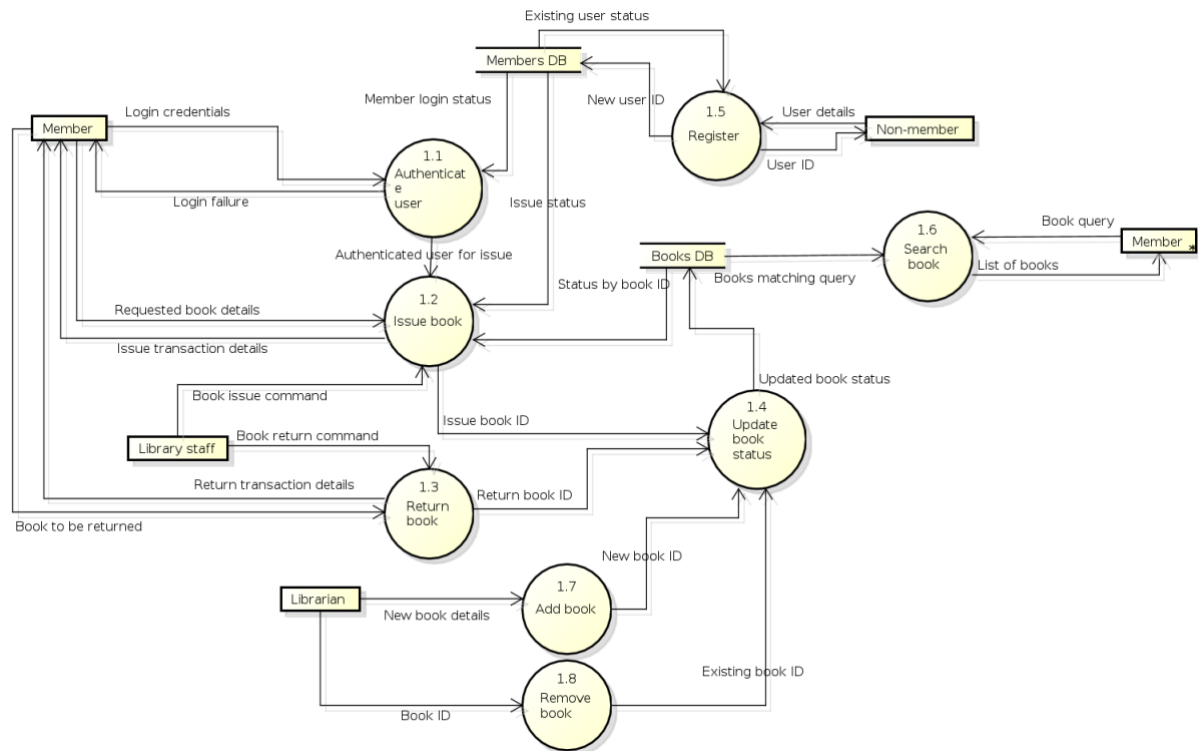


Figure 2: Level 1 DFD for Library Information System

Comparing figures 1 and 2 one might observe that the information flow in and out of LIS has been preserved. We observe in figure 2 that the sub-processes themselves exchange information among themselves. These information flows would be, in turn, preserved if we decompose the system into a level 2 DFD.

Finally, in order to eliminate intersecting lines and make the DFD complex, the Member external entity has been duplicated in figure 2. This is indicated by a * mark near the right-bottom corner of the entity box.

Self Evaluation:-

Software Engineering Virtual Lab

Modeling Data Flow Diagrams

1. A DFD represents

☒ Flow of data

☐ Flow of control

☐ Both the above

☐ Neither one

2. Which is not a component of a DFD?

☒ Decision

☐ Data flow

☐ Process

☐ Data store

3. How many processes are present in level-0 or context diagram?

☒ 1

☐ 2

☐ 3

4. External entities can appear in a DFD

☒ Only at level-0

☐ At any level

☐ Only at level-1

☐ Either at level-0 or at level-1

5. Data flow in a DFD is not possible in between

☒ External entity and data store

☐ Two processes

☐ Data store and process

☐ Process and external entity

Submit Clear

Exercise:-

Exercise 1

Software Engineering Virtual Lab

Modeling Data Flow Diagrams

Select 1

Draw a context level DFD to depict the typical user authentication process used by any system. An user gives two inputs -- user name and password.

Submit

Table #1: Add external entity

External Entity (EE)	Add
User	<input checked="" type="button" value="+Add"/>

Table #2: Add process

Process (P)	Level	Add
Authentication	0	<input checked="" type="button" value="+Add"/>

Table #3: Add data store

Data Store (DS)	Add
User Database	<input checked="" type="button" value="+Add"/>

Table #4: Add data flows

Type (PDS/EE)	From	Data Label	Type (PDS/EE)	To	Add
External entity	User	Username	Process	Authentication()	<input checked="" type="button" value="+Add"/>

Summary of the list of inputs taken

Table #5: List of external entities, data stores, and processes

External Entity	Data Store	Process
User	User Database	Authentication()

Table #6: List of data flows

From	Data Label	To	Remove
User	Username	Authentication()	<input checked="" type="button" value="+Add"/>

Result

Software Engineering Virtual Labs | Solution - Google Chrome

Not secure vlabs.iitkgp.ernet.in/se/show_solution/23/

Solution

Screenshot

```
graph TD
    User[User] -- "User ID" --> System((System))
    User -- "Password" --> System
    System -- "Authentication status" --> User
    System -- "User ID and password" --> Usersdb[Users db]
```

Results & Discussions

Exercise 2

Eclipse online with OffDocs x Installation of CISCO packet tr... x Software Engineering Virtual L... x please give solution of this pro... x

Not secure vlabs.iitkgp.ernet.in/se/8/exercise/

• User registration
• User login
• Profile update

Draw a Level 1 DFD to depict the above data flow and the corresponding processes. Should there be any data store in the DFD?

Submit

Table #1: Add external entity

External Entity (EE)	Add
ExtCrp	EE-Add

Table #2: Add process

Process (P)	Level	Add
User registration	1	Process-Add

Table #3: Add data store

Data Store (DS)	Add
User Database	DS-Add

Table #4: Add data flows

Type (PDS/EE)	From	Data Label	Type (PDS/EE)	To	Add
External entity	ExtCrp	Registration information	Process	User registration	Process-Add

Summary of the list of inputs taken

Table #5: List of external entities, data stores, and processes

External Entity	Data Store	Process
• User • ExtCrp	• User Database	• User login(t) • Profile update(t) • User registration(t)

Table #6: List of data flows

From	Data Label	To	Remove
ExtCrp			

Result

Practical No. 05

Aim:- Discuss Statechart and Activity Modeling

Introduction:-

Capturing the dynamic view of a system is very important for a developer to develop the logic for a system. State chart diagrams and activity diagrams are two popular UML diagram to visualize the dynamic behavior of an information system.

In this experiment, we will learn about the different components of activity diagram and state chart diagram and how these can be used to represent the dynamic nature of an information system.

Objectives:-

After completing this experiment you will be able to:

- Identify the distinct states a system have
- Identify the events causing transitions from one state to another
- Represent the above information pictorially using simple states
- Identify activities representing basic units of work, and represent their flow

Statechart Diagrams

In case of Object Oriented Analysis and Design, a system is often abstracted by one or more classes with some well defined behaviour and states. A *statechart diagram* is a pictorial representation of such a system, with all its states, and different events that lead transition from one state to another. To illustrate this, consider a computer. Some possible states that it could have are: running, shutdown, hibernate. A transition from running state to shutdown state occur when user presses the "Power off" switch, or clicks on the "Shut down" button as displayed by the OS. Here, clicking on the shutdown button, or pressing the power off switch act as external events causing the transition. Statechart diagrams are normally drawn to model the behaviour of a complex system. For simple systems this is optional.

Building Blocks of a Statechart Diagram

1. **State:-** A state is any "distinct" stage that an object (system) passes through in its lifetime. An object remains in a given state for finite time until "something" happens, which makes it to move to another state. All such states can be broadly categorized into following three types:
 - **Initial:** The state in which an object remain when created
 - **Final:** The state from which an object do not move to any other state [optional]
 - **Intermediate:** Any state, which is neither initial, nor final

As shown in figure-01, an initial state is represented by a circle filled with black. An intermediate state is depicted by a rectangle with rounded corners. A final state is represented by a unfilled circle with an inner black-filled circle.

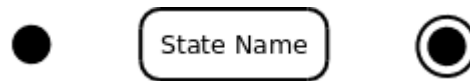


Figure-01: Representation of initial, intermediate, and final states of a statechart diagram

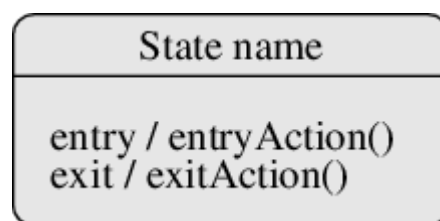
Intermediate states usually have two compartments, separated by a horizontal line, called the name compartment and internal transitions compartment [iv]. They are described below:

- **Name compartment:** Contains the name of the state, which is a short, simple, descriptive string
- **Internal transitions compartment:** Contains a list of internal activities performed as long as the system is in this state

The internal activities are indicated using the following syntax: action-label / action-expression. Action labels could be any condition indicator. There are, however, four special action labels:

- **Entry:** Indicates activity performed when the system enter this state
- **Exit:** Indicates activity performed when the system exits this state
- **Do:** indicate any activity that is performed while the system remain in this state or until the action expression results in a completed computation
- **Include:** Indicates invocation of a sub-machine

Any other action label identify the event (internal transition) as a result of which the corresponding action is triggered. Internal transition is almost similar to self transition, except that the former doesn't result in execution of entry and exit actions. That is, system doesn't exit or re-enter that state. Figure-02 shows the syntax for representing a typical (intermediate) state



States could again be either simple or composite (a state congaing other states). Here, however, we will deal only with simple states.

2. Transition:- Transition is movement from one state to another state in response to an external stimulus (or any internal event). A transition is represented by a solid arrow from the current state to the next state. It is labeled by: event [guard-condition]/[action-expression], where

- **Event** is the what is causing the concerned transition (mandatory) -- Written in past tense [\[iii\]](#)
- **Guard-condition** is (are) precondition(s), which must be true for the transition to happen [optional]
- **Action-expression** indicate action(s) to be performed as a result of the transition [optional]

It may be noted that if a transition is triggered with one or more guard-condition(s), which evaluate to false, the system will continue to stay in the present state. Also, not all transitions do result in a state change. For example, if a queue is full, any further attempt to append will fail until the delete method is invoked at least once. Thus, state of the queue doesn't change in this duration.

3. Action:- As mentioned in [\[ii\]](#), actions represents behaviour of the system. While the system is performing any action for the current event, it doesn't accept or process any new event. The order in which different actions are executed, is given below:

1. Exit actions of the present state
2. Actions specified for the transition
3. Entry actions of the next state

Figure-03 shows a typical statechart diagram with all it's syntaxes.

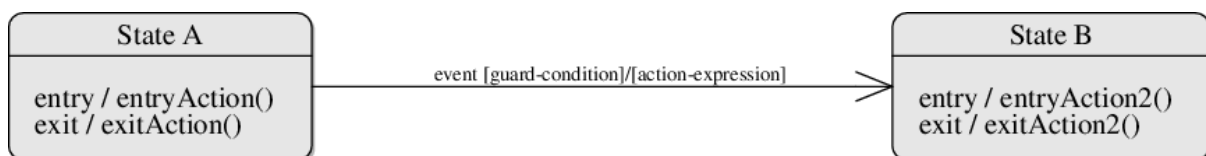


Figure-03: A statechart diagram showing transition from state A to B

Guidelines for drawing Statechart Diagrams

Following steps could be followed, as suggested in [\[i\]](#) to draw a statechart diagram:

- For the system to developed, identify the distinct states that it passes through
- Identify the events (and any precondition) that cause the state transitions. Often these would be the methods of a class as identified in a class diagram.
- Identify what activities are performed while the system remains in a given state

Activity Diagrams


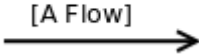
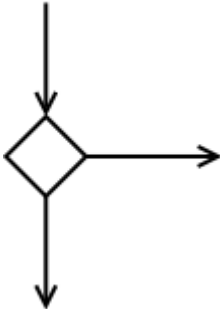
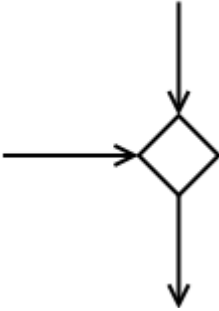
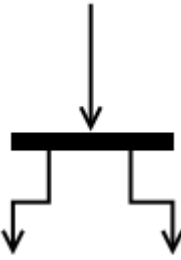
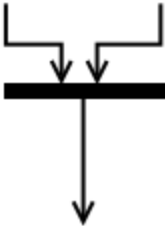
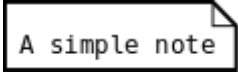
Activity diagrams fall under the category of behavioural diagrams in Unified Modeling Language. It is a high level diagram used to visually represent the flow of control in a system. It has similarities with traditional flow charts. However, it is more powerful than a simple flow chart since it can represent various other concepts like concurrent activities, their joining, and so on [\[vii, viii\]](#).

Activity diagrams, however, cannot depict the message passing among related objects. As such, it can't be directly translated into code. These kind of diagrams are suitable for confirming the logic to be implemented with the business users. These diagrams are typically used when the business logic is complex. In simple scenarios it can be avoided entirely [\[ix\]](#).

Components of an Activity Diagram

1. **Activity:-** An activity denotes a particular action taken in the logical flow of control. This could simply be invocation of a mathematical function, alter an object's properties and so on. An activity is represented with a rounded rectangle, as shown in table-01. A label inside the rectangle identifies the corresponding activity. There are two special type of activity nodes: initial and final. Initial node represents the starting point of a flow in an activity diagram. A final node represents the end point of all activities.
2. **Flow:-** A flow (also termed as edge, or transition) is represented with a directed arrow. This is used to depict transfer of control from one activity to another, or to other types of components, as we will see below. A flow is often accompanied with a label, called the guard condition, indicating the necessary condition for the transition to happen. The syntax to depict it is [guard condition].
3. **Decision:-** A decision node, represented with a diamond, is a point where a single flow enters and two or more flows leave. The control flow can follow only one of the outgoing paths. The outgoing edges often have guard conditions indicating true-false or if-then-else conditions.
4. **Merge:-** This is represented with a diamond shape, with two or more flows entering, and a single flow leaving out. A merge node represents the point where at least a single control should reach before further processing could continue.
5. **Fork:-** Fork is a point where parallel activities begin. A fork is graphically depicted with a black bar, with a single flow entering and multiple flows leaving out.
6. **Join:-** A join is depicted with a black bar, with multiple input flows, but a single output flow. Unlike a merge, in case of a join all of the incoming controls must be completed before any further progress could be made.
7. **Note:-** UML allows attaching a note to different components of a diagram to present some textual information. The information could simply be a comment or may be some constraint. A note can be attached to a decision point.
8. **Partition:-** Different components of an activity diagram can be logically grouped into different areas, called partitions or swimlanes. They often correspond to different units of an organization or different actors. The drawing area can be partitioned into multiple compartments using vertical (or horizontal) parallel lines. Partitions in an activity diagram are not mandatory.
- 9.

The following table shows commonly used components with a typical activity diagram.

Component	Graphical Notation
Activity	
Flow	
Decision	
Merge	
Fork	
Join	
Note	

A Simple Example

Figure-04 shows a simple activity diagram with two activities. The figure depicts two stages of a form submission. At first a form is filled up with relevant and correct information. Once it is verified that there is no error in the form, it is then submitted. The two other symbols shown in the figure are the initial node (dark filled circle), and final node (outer hollow circle with inner filled circle). It may be noted that there could be zero or more final node(s) in an activity diagram.

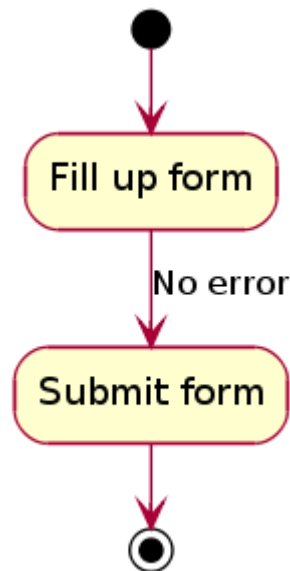


Figure-04: A simple activity diagram.

Guidelines for drawing an Activity Diagram

The following general guidelines could be followed to pictorially represent a complex logic.

- Identify tiny pieces of work being performed by the system
- Identify the next logical activity that should be performed
- Think about all those conditions that should be made, and all those constraints that should be satisfied, before one can move to the next activity
- Put non-trivial guard conditions on the edges to avoid confusion

Self Evaluation:-

Statechart and Activity Modeling

- What does an entry action of a state indicate?
 - ☒ Action performed after the system moves into the given state
 - ☐ Action performed before system moves into the given state
 - ☐ An optional action performed when system moves into the given state
 - ☐ None of the above
- What does the guard condition depicted over the transition between any two states indicate?
 - ☒ A condition that must be true for the transition to happen
 - ☐ A condition that must be false for the transition to occur
 - ☐ An indicator that this transition should not happen
 - ☐ An event that might happen as result of the transition
- A state can contain one or more sub-state(s) within it
 - ☒ True
 - ☐ False
- What does forking of several activities from a synchronization point indicate?
 - ☐ All those activities should get executed one after another
 - ☒ The activities can be performed in parallel
 - ☐ One or more activities could be skipped
- A decision point in an activity diagram is a control where
 - ☐ Multiple parallel activities merge
 - ☐ Decision is taken whether a transition should happen
 - ☒ A condition is checked and decided which activity should be performed next
 - ☐ There is no such control

Submit Clear

Exercise:-

Exercise 1

Statechart Diagram Activity Diagram

Table #1: Add states of the system

State Name	Add
	+ Add

Table #2: Internal activities of a state

State Name	Action Label	Action Expression	Add
State	Other	[custom label]	+ Add

Table #3: Add a note [optional]

State Name	Note	Position	Add
State		Select position	+ Add

Table #4: List of system states

State	Activities	Note (Position)	Remove
ON	entry / bulb is emitting light		
OFF	switch / bulb is not emitting light		

Table #5: Define state transitions

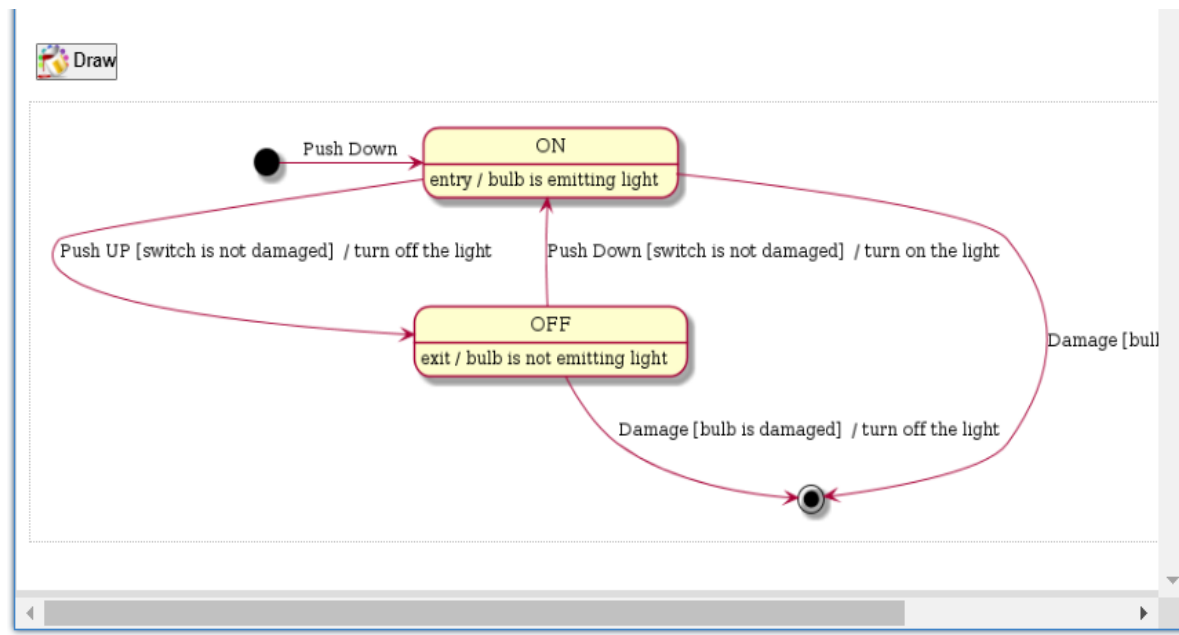
Current State	Next State	Event	Guard Condition	Action	Add
OFF	Final				+ Add

Table #6: List of state transitions

Current State	Event	Guard Condition	Action	Next State	Remove
Initial	Push Down	switch is not damaged	turn on the light	ON	
ON	Push Up	switch is not damaged	turn off the light	OFF	
OFF	Push Down	switch is not damaged	turn on the light	ON	
ON	Damage	bulb is damaged	turn off the light	Final	
OFF	Damage	bulb is damaged	turn off the light	Final	

Draw

Push Down ON



Exercise 2

The screenshot displays a web-based interface for creating statechart diagrams. The interface includes several tables for defining activities, transitions, and synchronization.

Table #1: Add activity

Activity Description	Add
	<input type="button" value="+ Add"/>

Table #2: Add simple sequential flows

Current Activity	Next Activity	Guard Condition	Add
<input type="text" value="Current activity"/>	<input type="text" value="Next activity"/>	<input type="text"/>	<input type="button" value="+ Add"/>

Table #3: List of simple transitions between activities

Current Activity	Next Activity	Guard Condition	Remove

Table #4: Specify parallel workflows (simple)

Type	Current Activities	Synchronization Bar	Following Activity(ies)	Add
Create parallel activities	<input type="text" value="Current activity"/>	<input type="text"/>	<input type="text" value="Parallel activity #1"/> <input type="text" value="Parallel activity #2"/> <input type="text" value="Parallel activity #3"/> <input type="text" value="Parallel activity #4"/> <input type="text" value="Parallel activity #5"/>	<input type="button" value="+ Add"/>
Merge parallel activities	<input type="text" value="Parallel activity #1"/> <input type="text" value="Parallel activity #2"/> <input type="text" value="Parallel activity #3"/> <input type="text" value="Parallel activity #4"/> <input type="text" value="Parallel activity #5"/>	<input type="text"/>	<input type="text" value="Next activity"/>	<input type="button" value="+ Add"/>

Table #5: List of parallel activities (simple)

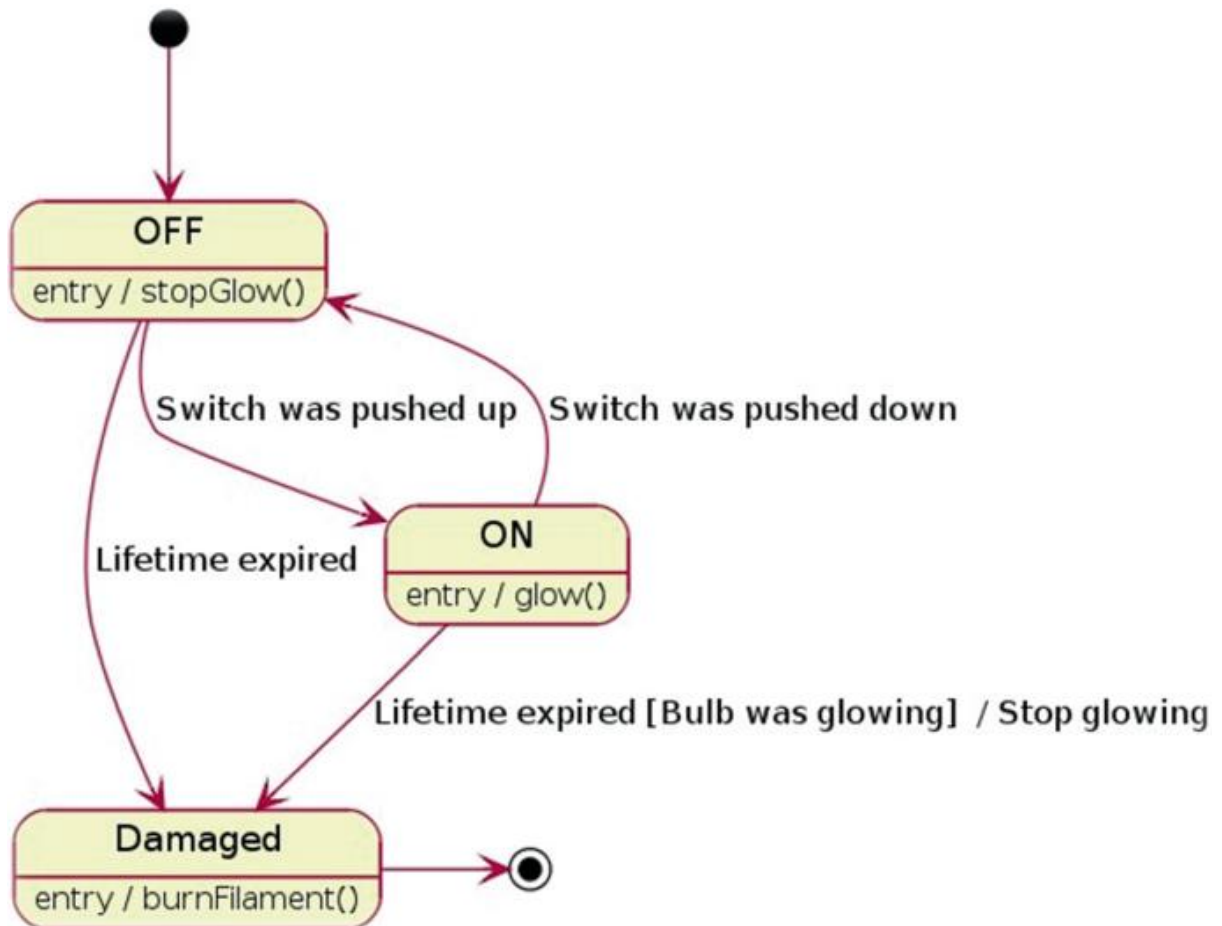
Current Activity	Link Point	Link To Activities	Merge Point	Merge To Activities	Remove

Table #6: Add conditional transitions from activity

Current Activity	Condition & Subsequent Action	Add
<input type="text" value="Current activity"/>	<div><div>if A condition THEN</div><div><input type="radio"/> Activity: Next activity <input type="radio"/> Fork: Synchronization bar <input type="radio"/> Merge: Synchronization bar Guard condition: <input type="text" value="True"/></div></div> <div><div>ELSE</div><div><input type="radio"/> Activity: Next activity <input type="radio"/> Fork: Synchronization bar <input type="radio"/> Merge: Synchronization bar Guard condition: <input type="text" value="False"/></div></div>	<input type="button" value="+ Add"/>

Table #7: List of conditional transitions from activity

Current Activity	Condition	IF TRUE: Guard Condition	IF FALSE: Guard Condition	Remove



Practical No. 06

Aim:- Modeling UML Class Diagrams and Sequence diagrams

Introduction:-

Classes are the structural units in object oriented system design approach, so it is essential to know all the relationships that exist between the classes, in a system. All objects in a system are also interacting to each other by means of passing messages from one object to another. Sequence diagram shows these interactions with time ordering of the messages.

In this Experiment, we will learn about the representation of class diagram and sequence diagram. We also learn about different relationships that exist among the classes, in a system. From the experiment of sequence diagram, we will learn about different types of messages passing in between the objects and time ordering of those messages, in a system.

Objectives:-

After completing this experiment you will be able to:

- Graphically represent a class, and associations among different classes
- Identify the logical sequence of activities undergoing in a system, and represent them pictorially

Structural and Behavioral aspects

Developing a software system in object oriented approach is very much dependent on understanding the problem. Some aspects and the respective models are used to describe problems and in context of those aspects the respective models give a clear idea regarding the problem to a designer. For developer, structural and behavioral aspects are two key aspects to see through a problem to design a solution for the same.

Class diagram

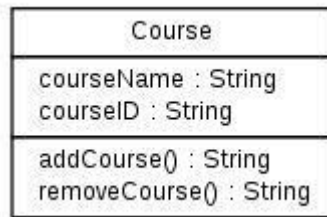
It is a graphical representation for describing a system in context of its static construction^[1].

Elements in class diagram:- Class diagram contains the system classes with its data members, operations and relationships between classes.

Class:- A set of objects containing similar data members and member functions is described by a class. In UML syntax, class is identified by solid outline rectangle with three compartments which contain

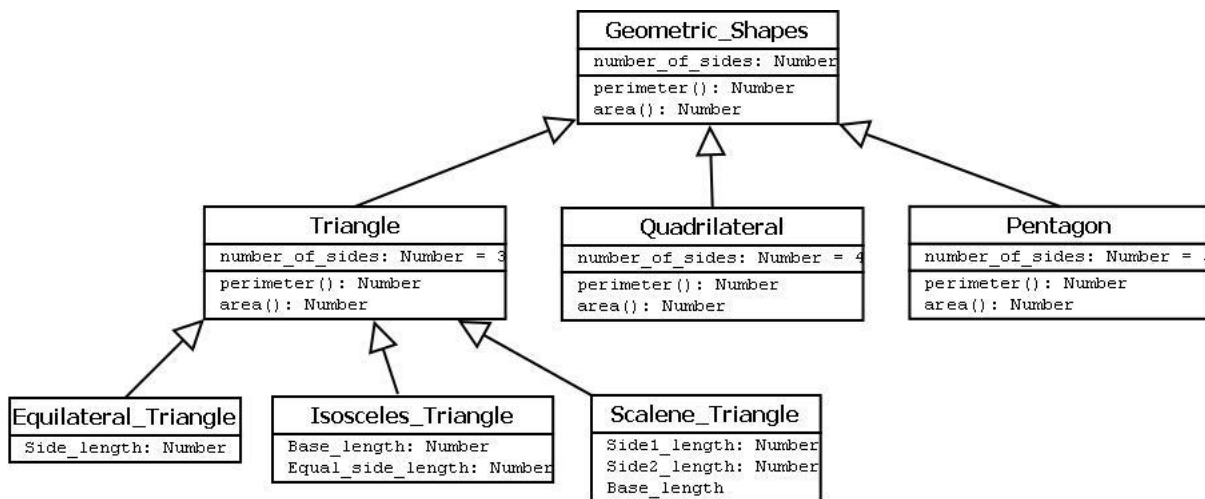
- **Class name:-** A class is uniquely identified in a system by its name. A textual string is taken as class name. It lies in the first compartment in class rectangle.
- **Attributes:-** Property shared by all instances of a class. It lies in the second compartment in class rectangle.
- **Operations:-** An execution of an action can be performed for any object of a class. It lies in the last compartment in class rectangle.

Example: To build a structural model for an Educational Organization, 'Course' can be treated as a class which contains attributes 'courseName' & 'courseID' with the operations 'addCourse()' & 'removeCourse()' allowed to be performed for any object to that class.



- **Generalization/Specialization:-** It describes how one class is derived from another class. Derived class inherits the properties of its parent class.

Example

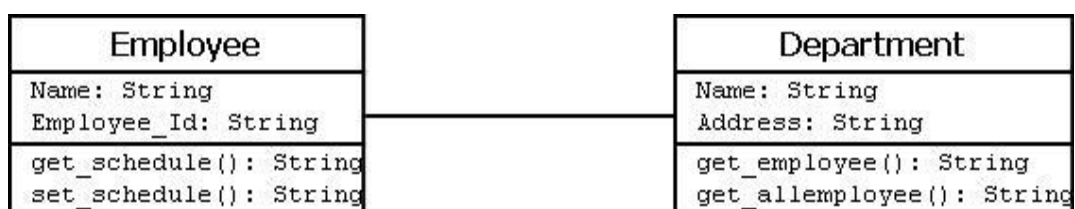


Relationships

Existing relationships in a system describe legitimate connections between the classes in that system.

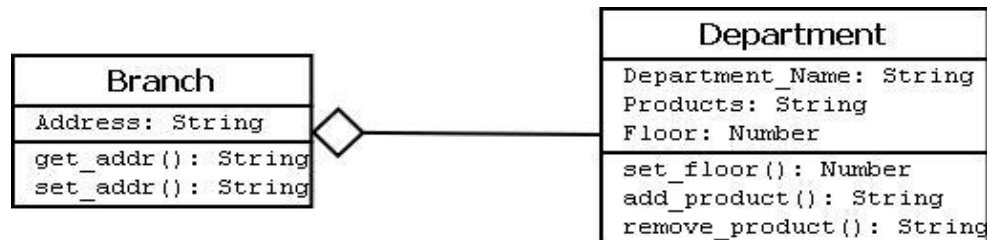
- **Association:-** It is an instance level relationship [\[i\]](#) that allows exchanging messages among the objects of both ends of association. A simple straight line connecting two class boxes represent an association. We can give a name to association and also at the both end we may indicate role names and multiplicity of the adjacent classes. Association may be uni-directional.

Example: In structure model for a system of an organization an employee (instance of 'Employee' class) is always assigned to a particular department (instance of 'Department' class) and the association can be shown by a line connecting the respective classes.



- **Aggregation:-** It is a special form of association which describes a part-whole relationship between a pair of classes. It means, in a relationship, when a class holds some instances of related class, then that relationship can be designed as an aggregation.

Example: For a supermarket in a city, each branch runs some of the departments they have. So, the relation among the classes 'Branch' and 'Department' can be designed as an aggregation. In UML, it can be shown as in the fig. below



- **Composition:-** It is a strong form of aggregation which describes that whole is completely owns its part. Life cycle of the part depends on the whole.

Example: Let consider a shopping mall has several branches in different locations in a city. The existence of branches completely depends on the shopping mall as if it is not exist any branch of it will no longer exists in the city. This relation can be described as composition and can be shown as below

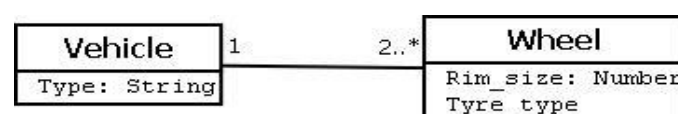


- **Multiplicity:-** It describes how many numbers of instances of one class is related to the number of instances of another class in an association.

Notation for different types of multiplicity:

Single instance	1
Zero or one instance	0..1
Zero or more instance	0..*
One or more instance	1..*
Particular range(two to six)	2..6

Example: One vehicle may have two or more wheels



Sequence diagram

It represents the behavioral aspects of a system. Sequence diagram shows the interactions between the objects by means of passing messages from one object to another with respect to time in a system.



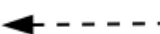
Elements in sequence diagram: Sequence diagram contains the objects of a system and their life-line bar and the messages passing between them.

Object:- Objects appear at the top portion of sequence diagram. Object is shown in a rectangle box. Name of object precedes a colon ':' and the class name, from which the object is instantiated. The whole string is underlined and appears in a rectangle box. Also, we may use only class name or only instance name. Objects which are created at the time of execution of use case and are involved in message passing, are appear in diagram, at the point of their creation.

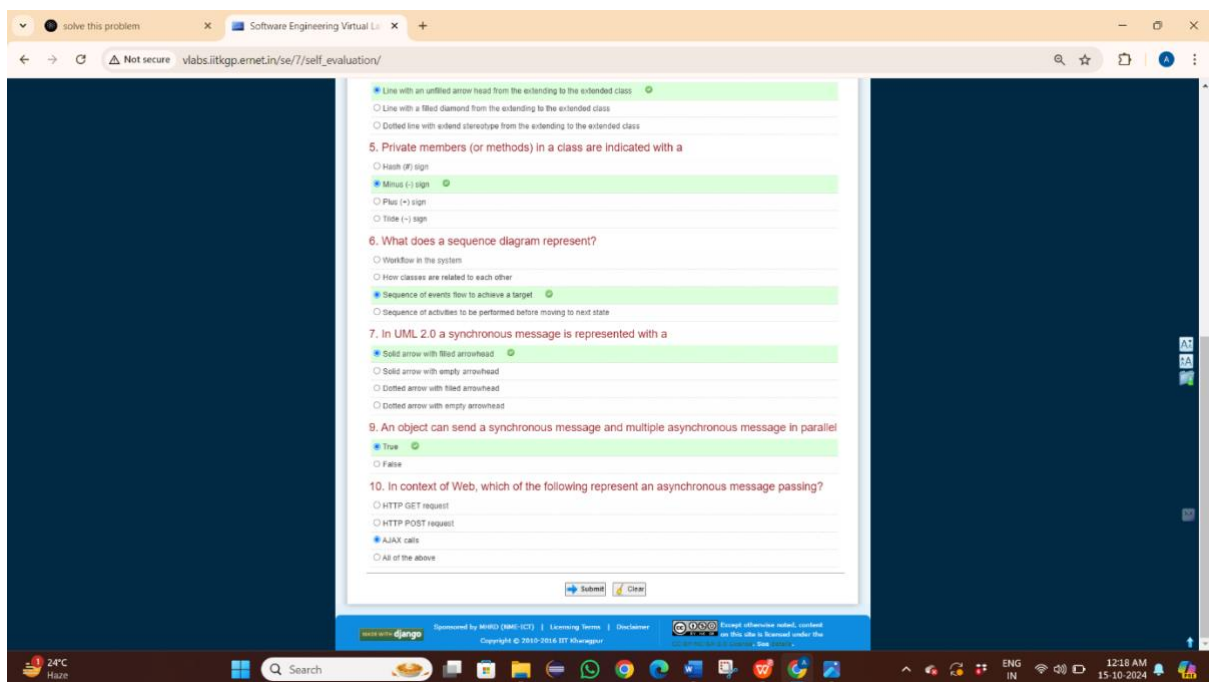
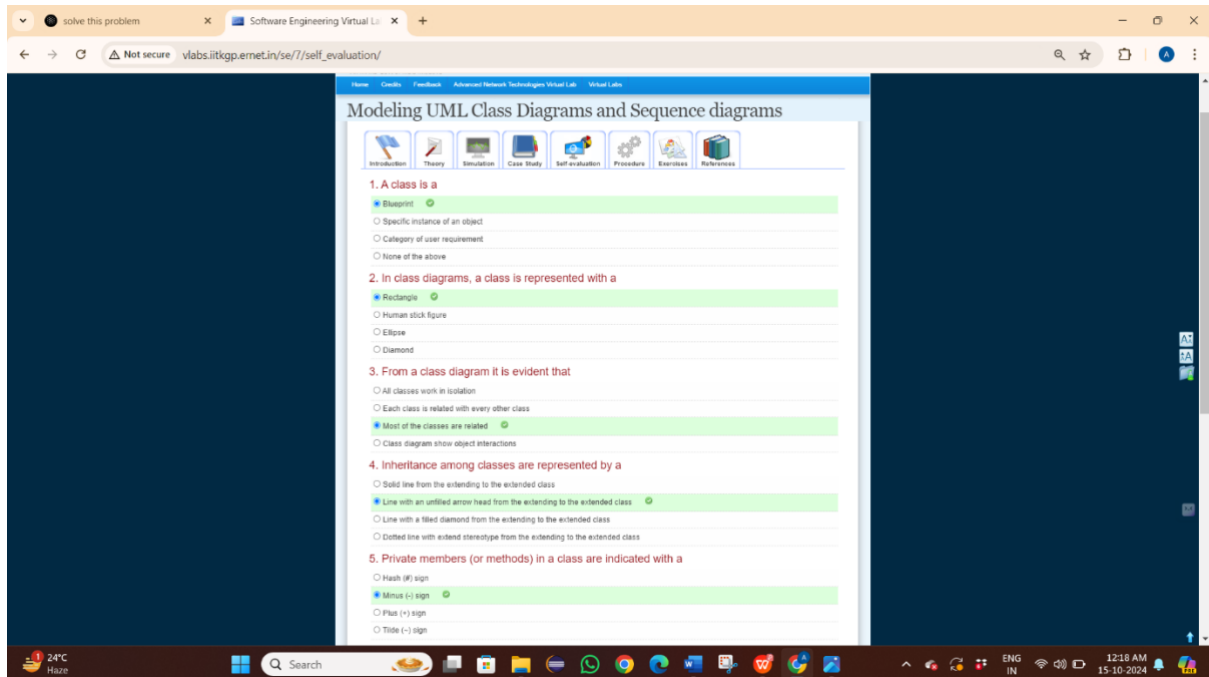
Life-line bar:- A down-ward vertical line from object-box is shown as the life-line of the object. A rectangle bar on life-line indicates that it is active at that point of time.

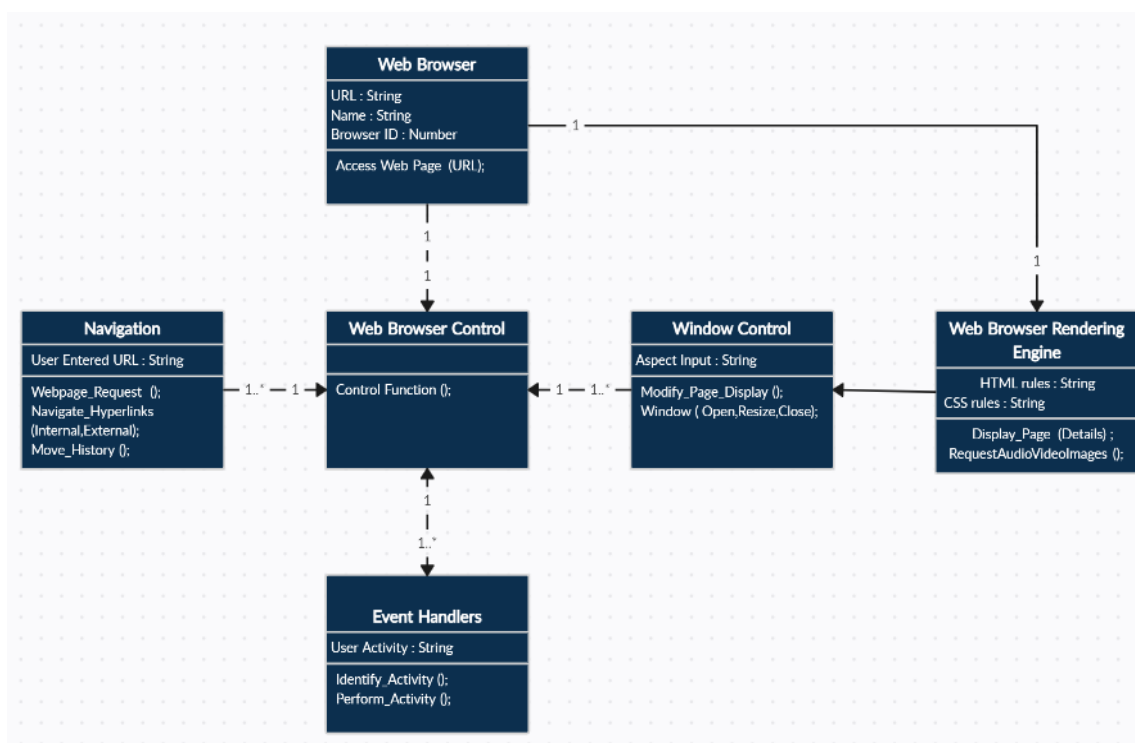
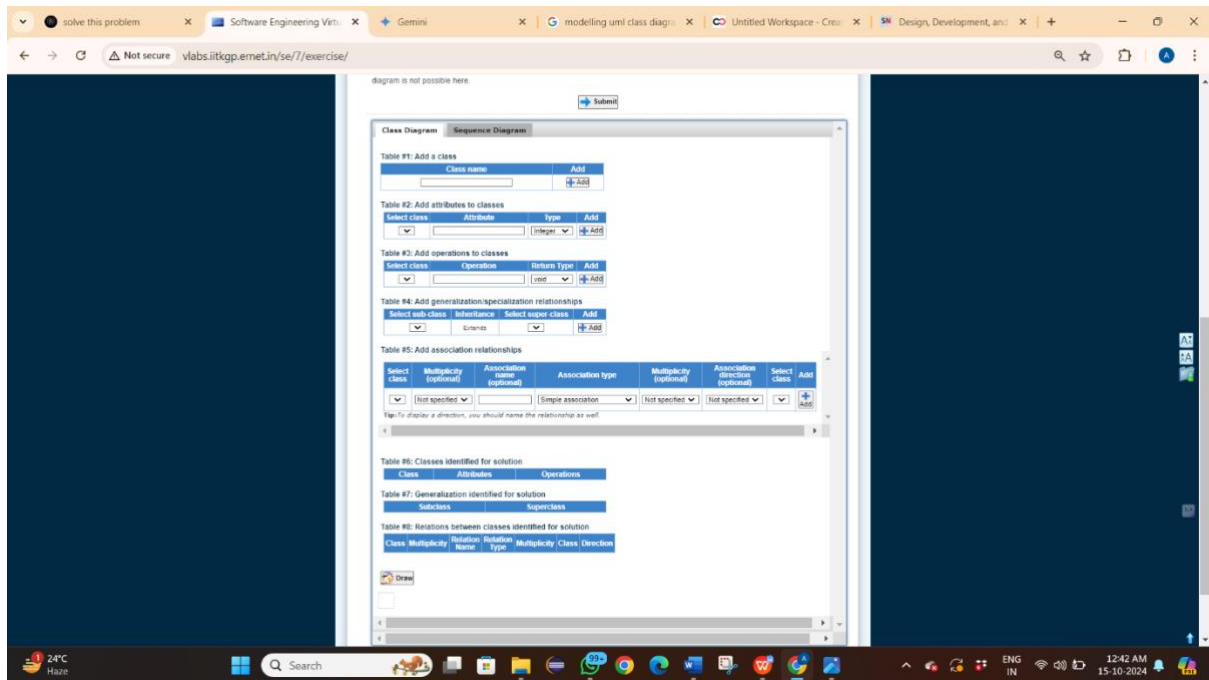
Messages:- Messages are shown as an arrow from the life-line of sender object to the life-line of receiver object and labeled with the message name. Chronological order of the messages passing throughout the objects' life-line show the sequence in which they occur. There may exist some different types of messages :

- **Synchronous messages:** Receiver start processing the message after receiving it and sender needs to wait until it is made [\[iii\]](#). A straight arrow with close and fill arrow-head from sender life-line bar to receiver end, represent a synchronous message [\[iii\]](#).
- **Asynchronous messages:** For asynchronous message sender needs not to wait for the receiver to process the message [\[iv\]](#). A function call that creates thread can be represented as an asynchronous message in sequence diagram [\[iv\]](#). A straight arrow with open arrow-head from sender life-line bar to receiver end, represent an asynchronous message [\[iii\]](#).
- **Return message:** For a function call when we need to return a value to the object, from which it was called, then we use return message. But, it is optional, and we are using it when we are going to model our system in much detail. A dashed arrow with open arrow-head from sender life-line bar to receiver end, represent that message.
- **Response message:** One object can send a message to self [\[iv\]](#). We use this message when we need to show the interaction between the same object.

Message Type	Notation
Synchronous message	
Asynchronous message	
Response message	

Self Evaluation:-



Exercise:-**Exercise 1****Class Diagram**

Sequence Diagram

The screenshot shows a web browser window with multiple tabs. The active tab displays a page titled "A web browser is a software that helps us access a resource (web page) available on the World Wide Web and identified by a URL. A web browser consists of different sub-components, which can be primarily categorized into browser rendering engine, and browser control." The page text describes the rendering engine and browser control components. Below the text, there is a "Submit" button and a "Class Diagram" tool interface. The tool interface includes a table for adding objects and a table for adding messages between objects.

Table #1: Add object

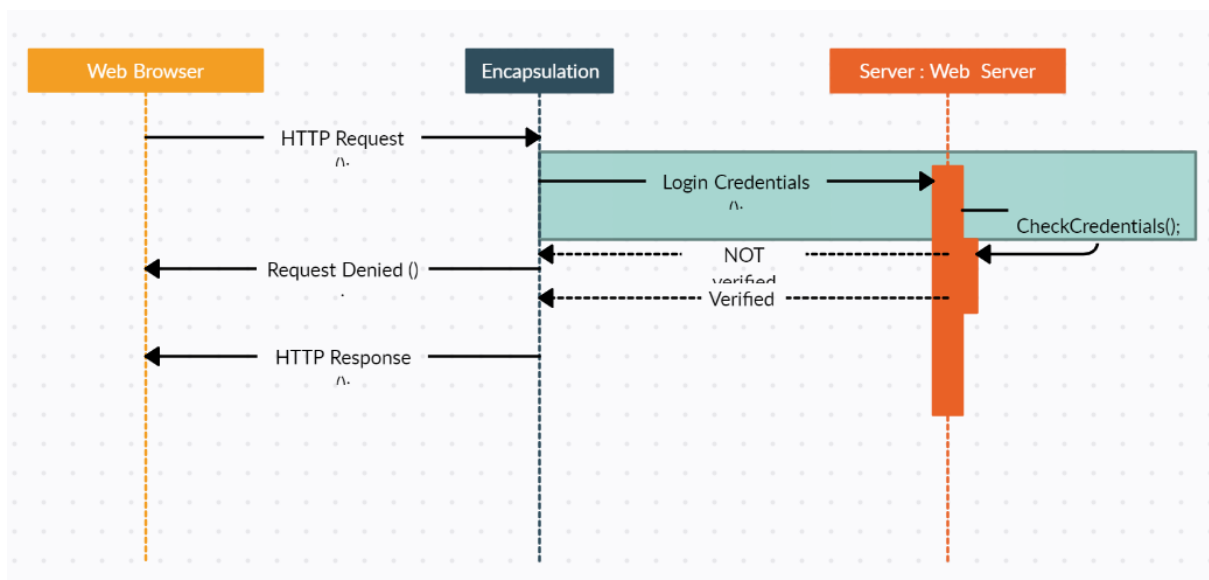
Object Name (Role)	Add
	<input type="button" value="Add"/>

Table #2: Add message passed between objects

Object (Sender)	Message Type	Message	Object (Receiver)	Add
<input type="text" value="Select object"/>	<input type="text" value="Select message type"/>	<input type="text" value=""/>	<input type="text" value="Select object"/>	<input type="button" value="Add"/>

Time sequence of events

It is possible to reorder the following events by dragging them and then placing it at appropriate position.



Practical No. 07

Aim:- Identifying Domain Classes from the Problem Statements.

Introduction:-

Same types of objects are typically implemented by class in object oriented programming. As the structural unit of the system can be represented through the classes, so, it is very important to identify the classes before start implementing all the logical flows of the system.

In this experiment we will learn how to identify the classes from a given problem statement.

Objectives:

After completing this experiment you will be able to:

- Understand the concept of domain classes
- Identify a list of potential domain classes from a given problem statement

Domain Class

In Object Oriented paradigm Domain Object Model has become subject of interest for its excellent problem comprehending capabilities towards the goal of designing a good software system. Domain Model, as a conceptual model gives proper understanding of problem description through its highly effective component – the Domain Classes. Domain classes are the abstraction of key entities, concepts or ideas presented in the problem statement [iv]. As stated in [v], domain classes are used for representing business activities during the analysis phase.

Below we discuss some techniques that can be used to identify the domain classes.

Traditional Techniques for Identification of Classes

Grammatical Approach Using Nouns

This object identification technique was proposed by Russell J. Abbot, and Grady Booch made the technique popular [1]. This technique involves grammatical analysis of the problem statement to identify list of potential classes. The logical steps are:

1. Obtain the user requirements (problem statement) as a simple, descriptive English text. This basically corresponds to the use-case diagram for the problem statement.
2. Identify and mark the nouns, pronouns and noun phrases from the above problem statements
3. List of potential classes is obtained based on the category of the nouns (details given later). For example, nouns that direct refer to any person, place, or entity in general, correspond to different objects. And so does singular proper nouns. On the other hand, plural nouns and common nouns are candidates that usually map into classes.

Advantages:- This is one of the simplest approaches that could be easily understood and applied by a larger section of the user base. The problem statement does not necessarily be in English, but in any other language.

Disadvantages:- The problem statement always may not help towards correct identification of a class. At times it could give us redundant classes. At times the problem statement may use abbreviations for large systems or concepts, and therefore, the identified class may actually point to an aggregate of classes. In other words, it may not find all the objects.

Using Generalization

In this approach, all potential objects are classified into different groups based on some common behaviour. Classes are derived from these groups.

Using Subclasses

Here, instead of identifying objects one goes for identification of classes based on some similar characteristics. These are the specialized classes. Common characteristics are taken from them to form the higher level generalized classes.

Steps to Identify Domain Classes from Problem Statement

We now present the steps to identify domain classes from a given problem statement. This approach is mostly based on the “Grammatical approach using nouns” discussed above, with some insights from [1].

1. Make a list of potential objects by finding out the nouns and noun phrases from narrative problem statement
2. Apply subject matter expertise (or domain knowledge) to identify additional classes
3. Filter out the redundant or irrelevant classes
4. Classify all potential objects based on categories. We follow the category table as described by Ross (table 5-3, pg 88, [1])

Categories	Explanation
People	Humans who carry out some function
Places	Areas set aside for people or things
Things	Physical objects
Organizations	Collection of people, resources, facilities and capabilities having a defined mission
Concepts	Principles or Ideas not tangible
Events	Things that happen (usually at a given date and time), or as a steps in an ordered sequence

5. Group the objects based on similar attributes. While grouping we should remember that
 - a. Different nouns (or noun phrases) can actually refer to the same thing (examples: house, home, abode)
 - b. Same nouns (or noun phrases) could refer to different things or concepts (example: I go to school every day / This school of thought agrees with the theory)
6. Give related names to each group to generate the final list of top level classes
7. Iterate over to refine the list of classes

Advanced Concepts

Identification of domain classes might not be a simple task for novices. It requires expertise and domain knowledge to identify business classes from plain English text. The concepts presented here have been kept simple in order to make a student familiarize with the subject. A lot of work has been done in this area, and various techniques have been proposed to identify domain classes. Interested readers may look at the following paper for an advanced treatment on this subject matter.

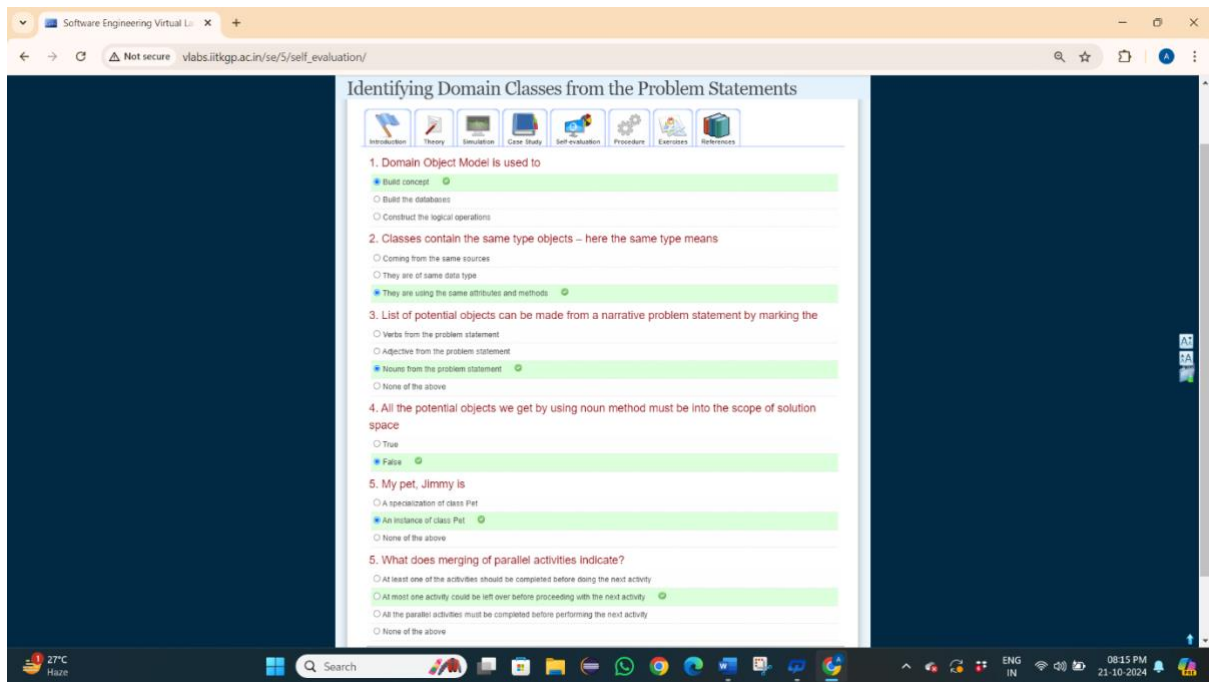
I. Y. Song, K. Yano, J. Trujillo, and S. Luján-Mora. "A Taxonomic Class Modeling Methodology for Object-Oriented Analysis", In Information Modeling Methods and Methodologies, Advanced Topics in Databases Series, Ed. (J Krostige, T. Halpin, K. Siau), Idea Group Publishing, 2004, pp. 216-240.

General Instructions

Follow are the steps to be followed in general to perform the experiments in Software Engineering Virtual Lab.

1. Read the theory about the experiment
2. View the simulation provided for a chosen, related problem
3. Take the self evaluation to judge your understanding (optional, but recommended)
4. Solve the given list of exercises

Self Evaluation:-



Exercise:-

Exercise 1:-

The latest cab services agency in the city has approached you to develop a Cab Management System for them. Following is the information they have given to implement the system.

Mr. Bose is the boss of this agency. Cabs are solely owned by the agency. They hire drivers to drive the cabs. Most of the cabs are without AC. However, a few comes with AC.

The agency provides service from 8 AM to 8 PM. Presently the service is limited only within Kolkata. Whenever any passenger books a cab, an available cab is allocated for him. A booking receipt is given to the passenger. He is then dropped to his home, office, or wherever he wants to go. In case the place is in too interior, the passenger is dropped at the nearest landmark.

Payments are made to the drivers by cheque drawn at the local branch of At Your Risk Bank. All kind of finances required for the business are dealt with this bank.

Recently Mr. Roy, neighbour of Mr. Bose, has given a proposal to book one of the cab in the morning everyday to drop his son to school, and drop him back to home later. Few other persons in the locality have also found the plan a good one. Hence, Mr. Bose is planning to introduce this "Drop to school" plan also very soon.

Learning Objectives:

1. Identifying potential classes (and their attributes) from a given problem statement
2. Use expert knowledge on the subject matter to identify other relevant classes

Practical No. 08

Aim:- Designing Test Suites.

Introduction:-

Development of a new software, like any other product, remains incomplete until it subjected to exhaustive tests. The primary objective of testing is not to verify that all desired features have been implemented correctly. However, it also includes verification of the software behavior in case of "bad inputs".

In this experiment, we discuss in brief about different types of testing, and provide the mechanisms to have hands-on experience on unit testing.

Objectives:

After completing this experiment you will be able to:

- Learn about different techniques of testing a software
- Design unit test cases to verify the functionality and locate bugs, if any

Software Testing

Testing software is an important part of the development life cycle of a software. It is an expensive activity. Hence, appropriate testing methods are necessary for ensuring the reliability of a program. According to the ANSI/IEEE 1059 standard, the definition of testing is the process of analyzing a software item, to detect the differences between existing and required conditions i.e. defects/errors/bugs and to evaluate the features of the software item.

The purpose of testing is to verify and validate a software and to find the defects present in a software. The purpose of finding those problems is to get them fixed.

- **Verification** is the checking or we can say the testing of software for consistency and conformance by evaluating the results against pre-specified requirements.
- **Validation** looks at the systems correctness, i.e. the process of checking that what has been specified is what the user actually wanted.
- **Defect** is a variance between the expected and actual result. The defect's ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

Standards for Software Test Documentation

IEEE 829-1998 is known as the 829 Standard for Software Test Documentation. It is an IEEE standard that specifies the form of a set of documents for use in software testing [\[1\]](#). There are other different standards discussed below.

- IEEE 1008, a standard for unit testing
- IEEE 1012, a standard for Software Verification and Validation

- IEEE 1028, a standard for software inspections
- IEEE 1044, a standard for the classification of software anomalies
- IEEE 1044-1, a guide to the classification of software anomalies
- IEEE 830, a guide for developing system requirements specifications
- IEEE 730, a standard for software quality assurance plans
- IEEE 1061, a standard for software quality metrics and methodology
- IEEE 12207, a standard for software life cycle processes and life cycle data
- BS 7925-1, a vocabulary of terms used in software testing
- BS 7925-2, a standard for software component testing

Testing Frameworks

Following are the different testing frameworks:

- JUnit - for Java unit test [\[ii\]](#)
- Selenium - is a suite of tools for automating web applications for software testing purposes, plugin for Firefox [\[iii\]](#)
- HP QC - is the HP Web-based test management tool. It familiarizes with the process of defining releases, specifying requirements, planning tests, executing tests, tracking defects, alerting on changes, and analyzing results. It also shows how to customize project [\[iv\]](#)
- IBM Rational - Rational software has a solution to support business sector for designing, implementing and testing software [\[v\]](#)

Need for Software Testing

There are many reasons for why we should test software, such as:

- Software testing identifies the software faults. The removal of faults helps reduce the number of system failures. Reducing failures improves the reliability and the quality of the systems.
- Software testing can also improve the other system qualities such as maintainability, usability, and testability.
- In order to meet the condition that the last few years of the 20th century systems had to be shown to be free from the 'millennium bug'.
- In order to meet the different legal requirements.
- In order to meet industry specific standards such as the Aerospace, Missile and Railway Signaling standards.

Test Cases and Test Suite

A test case describes an input descriptions and an expected output descriptions. Input are of two types: preconditions (circumstances that hold prior to test case execution) and the actual inputs that are identified by some testing methods. The set of test cases is called a test suite. We may have a test suite of all possible test cases.

Types of Software Testing

Testing is done in every stage of software development life cycle, but the testing done at each level of software development is different in nature and has different objectives. There are different types of testing, such as stress testing, volume testing, configuration testing, compatibility testing, recovery testing, maintenance testing, documentation testing, and usability testing. Software testing are mainly of following types [\[1\]](#)

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**

1. **Unit Testing:-** Unit testing is done at the lowest level. It tests the basic unit of software, that is the smallest testable piece of software. The individual component or unit of a program are tested in unit testing. Unit testing are of two types.

- **Black box testing:** This is also known as **functional testing** , where the test cases are designed based on input output values only. There are many types of Black Box Testing but following are the prominent ones.

- **Equivalence class partitioning:** In this approach, the domain of input values to a program is divided into a set of equivalence classes. e.g. Consider a software program that computes whether an integer number is even or not that is in the range of 0 to 10. Determine the equivalence class test suite. There are three equivalence classes for this program. - The set of negative integer - The integers in the range 0 to 10 - The integer larger than 10

- **Boundary value analysis :** In this approach, while designing the test cases, the values at boundaries of different equivalence classes are taken into consideration. e.g. In the above given example as in equivalence class partitioning, a boundary values based test suite is { 0, -1, 10, 11 }

- **White box testing:** It is also known as **structural testing**. In this testing, test cases are designed on the basis of examination of the code. This testing is performed based on the knowledge of how the system is implemented. It includes analyzing data flow, control flow, information flow, coding practices, exception and error handling within the system, to test the intended and unintended software behavior. White box testing can be performed to validate whether code implementation follows intended design, to validate implemented security functionality, and to uncover exploitable vulnerabilities. This testing requires access to the source code. Though white box testing can be performed any time in the life cycle after the code is developed, but it is a good practice to perform white box testing during the unit testing phase.

2. Integration Testing:- Integration testing is performed when two or more tested units are combined into a larger structure. The main objective of this testing is to check whether the different modules of a program interface with each other properly or not. This testing is mainly of two types:

- **Top-down approach**
- **Bottom-up approach**

In bottom-up approach, each subsystem is tested separately and then the full system is tested. But the top-down integration testing starts with the main routine and one or two subordinate routines in the system. After the top-level 'skeleton' has been tested, the immediately subroutines of the 'skeleton' are combined with it and tested.

3. System Testing:- System testing tends to affirm the end-to-end quality of the entire system. System testing is often based on the functional / requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability are also checked. There are three types of system testing

- **Alpha testing** is done by the developers who develop the software. This testing is also done by the client or an outsider with the presence of developer or we can say tester.
- **Beta testing** is done by very few number of end users before the delivery, where the change requests are fixed, if the user gives any feedback or reports any type of defect.
- **User Acceptance testing** is also another level of the system testing process where the system is tested for acceptability. This test evaluates the system's compliance with the client requirements and assess whether it is acceptable for software delivery

An error correction may introduce new errors. Therefore, after every round of error-fixing, another testing is carried out, i.e. called regression testing. Regression testing does not belong to either unit testing, integration testing, or system testing, instead, it is a separate dimension to these three forms of testing.

Regression Testing

The purpose of regression testing is to ensure that bug fixes and new functionality introduced in a software do not adversely affect the unmodified parts of the program [2]. Regression testing is an important activity at both testing and maintenance phases. When a piece of software is modified, it is necessary to ensure that the quality of the software is preserved. To this end, regression testing is to retest the software using the test cases selected from the original test suite.

Example

Write a program to calculate the square of a number in the range 1-100

```
#include <stdio.h>

int main()
{
    int n, res;
    printf("Enter a number: ");
    scanf("%d", &n);
    if (n >= 1 && n <= 100)
    {
        res = n * n;
        printf("\n Square of %d is %d\n", n, res);
    }
    else if (n <= 0 || n > 100)
        printf("Beyond the range");
    return 0;
}
```

Output

Inputs	Outputs
I1 : -2	O1 : Beyond the range
I2 : 0	O2 : Beyond the range
I3 : 1	O3 : Square of 1 is 1
I4 : 100	O4 : Square of 100 is 10000
I5 : 101	O5 : Beyond the range
I6 : 4	O6 : Square of 4 is 16
I7 : 62	O7 : Square of 62 is 3844

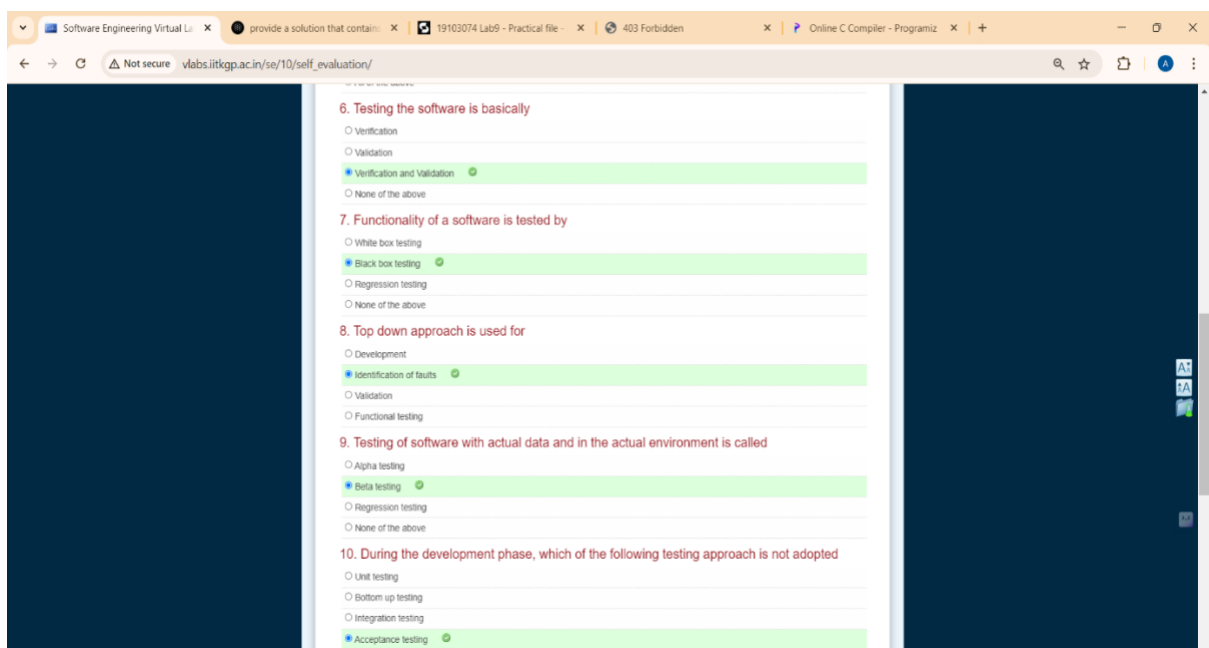
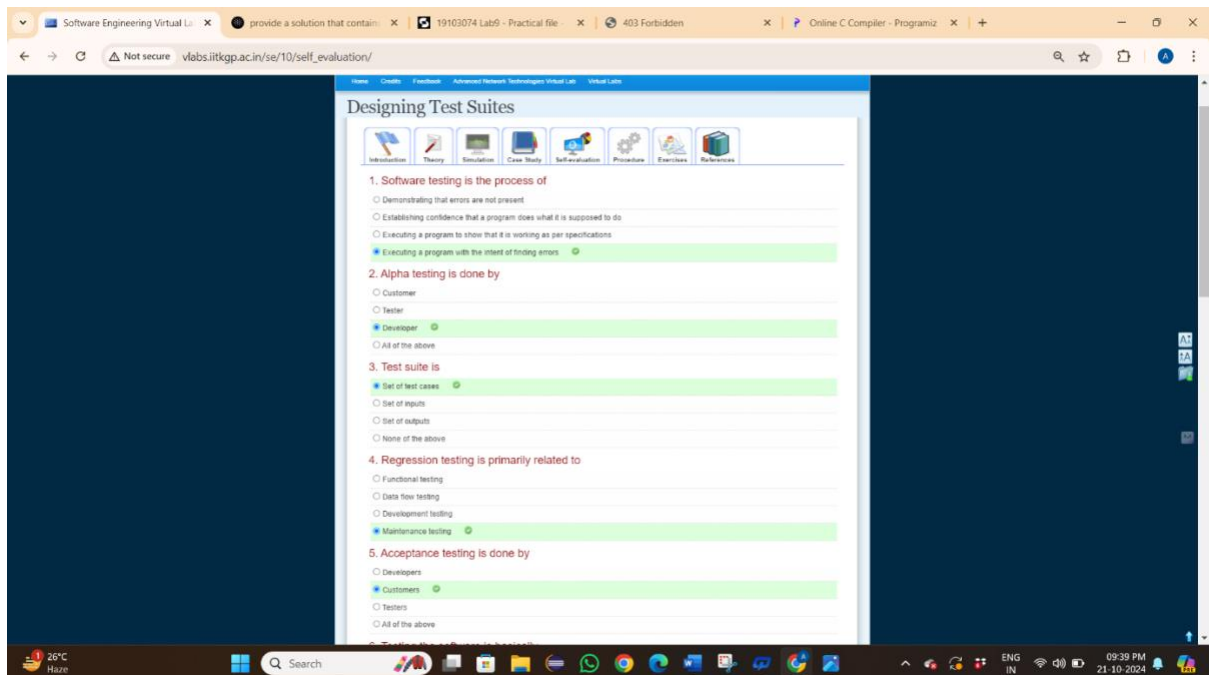
Test Cases

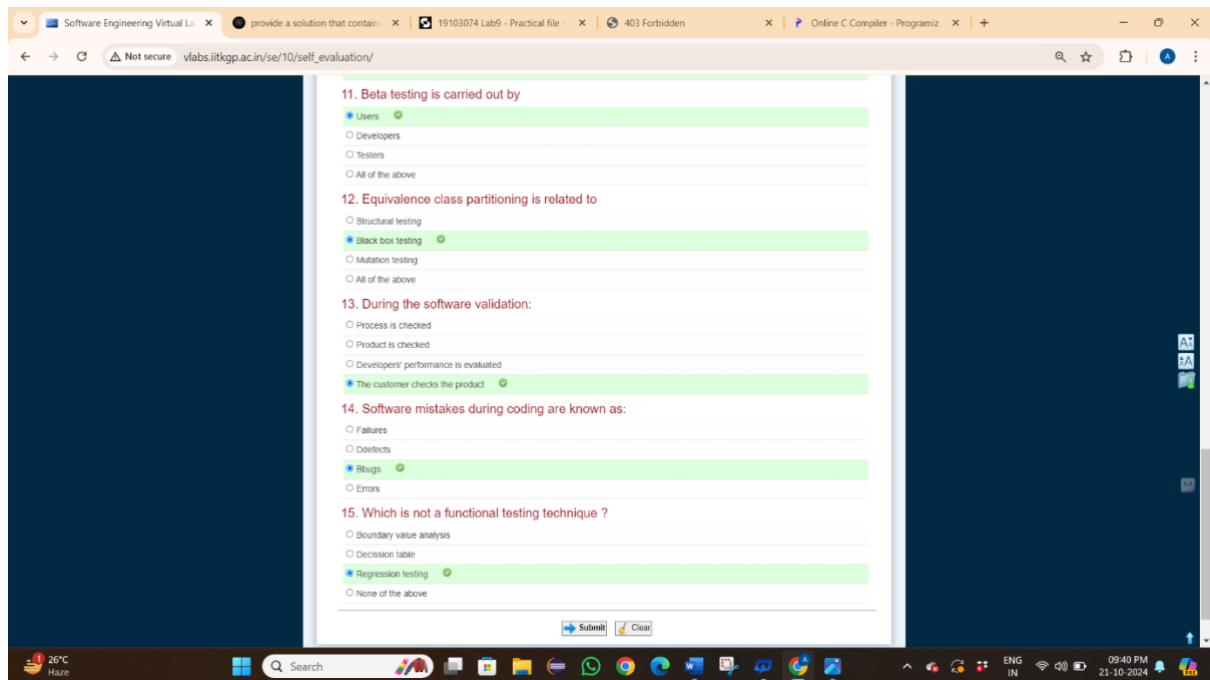
T1 : {I1 ,O1}
T2 : {I2 ,O2}
T3 : {I3, O3}
T4 : {I4, O4}
T5 : {I5, O5}
T6 : {I6, O6}
T7 : {I7, O7}

Some Remarks

A prevalent misconception among the beginners is that one should be concerned with testing only after coding ends. Testing is, in fact, not a phase towards the end. It is rather a continuous process. The efforts for testing should begin in the form of preparation of test cases after the requirements have been finalized. The Software Requirements Specification (SRS) document captures all features to be expected from the system. The requirements so identified here should serve as a basis towards preparation of the test cases. Test cases should be designed in such a way that all target features could be verified. However, testing a software is not only about proving that it works correctly. Successful testing should also point out the bugs present in the system, if any.

Self Evaluation:-





Exercise:-

Design a test suite for the following problem

The Absolute Beginners Inc. seems to have been fascinated by your work. Recently they have entrusted you with a task of writing a web-based mathematical software (using JavaScript). As part of this software, your team mate has written a small module, which computes area of simple geometric shapes. A portion of the module is shown below.

```
function square(side) { return side * side }

function rectangle(side1, side2) { return side1 * side1; }

function circle(radius) { return Math.PI * radius * radius; }

function right_triangle(base, height) { return 1 / 2 * base * height; }
```

Prepare a test suite that will

- Verify each of the above mentioned individual function is working correctly

Your task essentially is to verify whether each of the above function is returning correct values for given inputs. For example, a rectangle with length 10 unit and breadth 5 unit will have an area of 50 sq. unit. This can be verified from the output of the function call

```
rectangle(10, 5);
```

However, testing also attempts to point out possible bugs in the software. How would the above code behave for a call

```
rectangle(10, -5);
```

Modify the code to address this defect.

- In each function, return -1 if any given dimension is negative
- Modify the test suite such that it reflects desired performance for both correct and incorrect input(s)
- The code has another bug -- how would you identify it **from testing results**? Fix the bug and test it again.

Learning Objectives:

1. Get familiarized with unit testing
2. Verify implementation of functional requirements by writing test cases
3. Analyze results of testing to ascertain the current state of a project

Code

```

1 function square(side) { return side * side; }
2 function rectangle(side1, side2) { return side1 * side2; }
3 function circle(radius) { return 3.14 * radius * radius; }
4 function right_triangle(base, height) { return 1 / 2 * base * height; }
5
6
7

```

Position: Ln 3, Ch 38 Total: Ln 5, Ch 231

Create test suite

TS0: test

Default test suite

[\[Remove\]](#)

Add test cases

Summary

	A	B	C	D	E	F
1	Summary	Script	Expected Output	Actual Output	Manual Testing	Status
2	1	square(4)	16	16	<input type="checkbox"/> Yes	Pass
3	2	rectangle(2,3)	6	6	<input type="checkbox"/> Yes	Pass
4	3	circle(10)	314	314	<input type="checkbox"/> Yes	Pass
5	4	right_triangle(4,2)	4	4	<input type="checkbox"/> Yes	Pass
6					<input type="checkbox"/> Yes	No Run
7					<input type="checkbox"/> Yes	No Run
8					<input type="checkbox"/> Yes	No Run
9					<input type="checkbox"/> Yes	No Run
10					<input type="checkbox"/> Yes	No Run

Execute test suite

Test suite # TS0: Execution result

of test cases passed: 4

of test cases failed: 0

Total # of test cases: 4

Test suite status: **Passed**

[Refresh result](#)

