

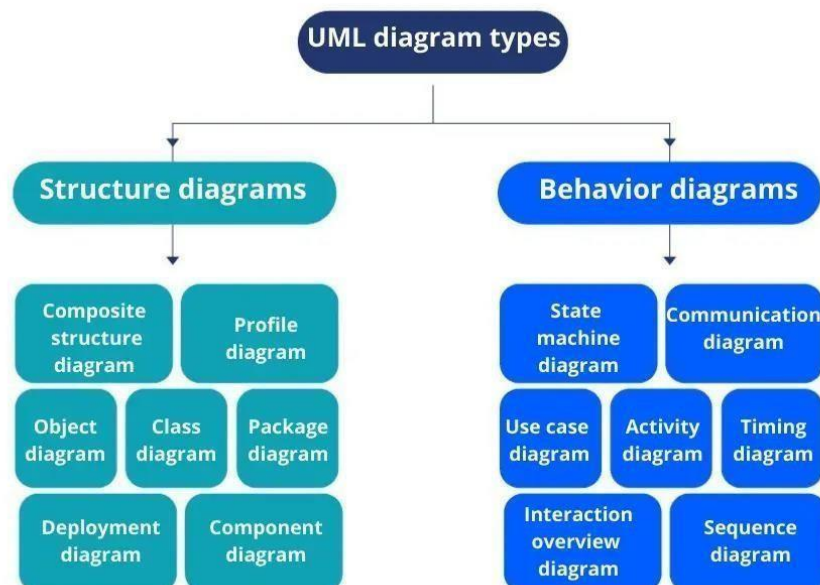
## INTRODUCTION TO UML

**Unified Modelling Language (UML)** is a general-purpose modelling language. The main aim of UML is to define a standard way to **visualize** the way a system has been designed. It is quite similar to blueprints used in other fields of engineering. UML is **not a programming language**, it is rather a visual language.

- We use UML diagrams to portray the **behaviour and structure** of a system.
- UML helps software engineers, businessmen, and system architects with modeling, design, and analysis.
- The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. It's been managed by OMG ever since.
- The International Organization for Standardization (ISO) published UML as an approved standard in 2005.

### need of UML :

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non-programmers about essential requirements, functionalities, and processes of the system.
- A lot of time is saved down the line when teams can visualize processes, user interactions, and the static structure of the system.



### Tools for creating UML Diagrams

There are several tools available for creating Unified Modelling Language (UML) diagrams, which are commonly used in software development to visually represent system architecture, design, and implementation. Here are some popular UML diagram creating tools:

- **Lucidchart:** Lucidchart is a web-based diagramming tool that supports UML diagrams. It's user-friendly and collaborative, allowing multiple users to work on diagrams in real-time.
- **Draw.io:** Draw.io is a free, web-based diagramming tool that supports various diagram types, including UML. It integrates with various cloud storage services and can be used offline.
- **Visual Paradigm:** Visual Paradigm provides a comprehensive suite of tools for software development, including UML diagramming. It offers both online and desktop versions and supports a wide range of UML diagrams.
- **StarUML:** StarUML is an open-source UML modelling tool with a user-friendly interface. It supports the standard UML 2.x diagrams and allows users to customize and extend its functionality through plugins.
- **Papyrus:** Papyrus is an open-source UML modelling tool that is part of the Eclipse Modelling Project. It provides a customizable environment for creating, editing, and visualizing UML diagrams.
- **PlantUML:** PlantUML is a text-based tool that allows you to create UML diagrams using a simple and human-readable syntax. It's often used in conjunction with other tools and supports a variety of diagram types.
- **Umbrello:** Umbrello is a Unified Modelling Language (UML) modelling tool and code generator. It can create diagrams of software and other systems in the industry standard UML format, and can also generate code from UML diagrams in a variety of programming languages. Features: Supported formats: XMI Several type of diagrams supported: use case, class, sequence, communication, state, activity, component, deployment, entity relationship.

**Umbrello** UML Modeller supports the following types:

- Class Diagram
- Sequence Diagram
- Collaboration Diagram
- Use Case Diagram
- State Diagram
- Activity Diagram
- Component Diagram
- Deployment Diagram
- Entity Relationship Diagram

**Steps to create UML Diagrams :**

- **Step 1: Identify the Purpose:** Determine the purpose of creating the UML diagram. Different types of UML diagrams serve various purposes, such as capturing requirements, designing system architecture, or documenting class relationships.
- **Step 2: Identify Elements and Relationships:** Identify the key elements (classes, objects, use cases, etc.) and their relationships that need to be represented in the diagram. This step involves understanding the structure and behavior of the system you are modelling.
- **Step 3: Select the Appropriate UML Diagram Type:** Choose the UML diagram type that best fits your modeling needs. Common types include Class Diagrams, Use Case Diagrams, Sequence Diagrams, Activity Diagrams, and more.
- **Step 4: Create a Rough Sketch:** Before using a UML modeling tool, it can be helpful to create a rough sketch on paper or a whiteboard. This can help you visualize the layout and connections between elements.
- **Step 5: Choose a UML Modeling Tool:** Select a UML modeling tool that suits your preferences and requirements. There are various tools available, both online and offline, that offer features for creating and editing UML diagrams.
- **Step 6: Create the Diagram:** Open the selected UML modeling tool and create a new project or diagram. Begin adding elements (e.g., classes, use cases, actors) to the diagram and connect them with appropriate relationships (e.g., associations, dependencies).
- **Step 7: Define Element Properties:** For each element in the diagram, specify relevant properties and attributes. This might include class attributes and methods, use case details, or any other information specific to the diagram type.
- **Step 8: Add Annotations and Comments:** Enhance the clarity of your diagram by adding annotations, comments, and explanatory notes. This helps anyone reviewing the diagram to understand the design decisions and logic behind it.
- **Step 9: Validate and Review:** Review the diagram for accuracy and completeness. Ensure that the relationships, constraints, and elements accurately represent the intended system or process. Validate your diagram against the requirements and make necessary adjustments.
- **Step 10: Refine and Iterate:** Refine the diagram based on feedback and additional insights. UML diagrams are often created iteratively as the understanding of the system evolves.
- **Step 11: Generate Documentation:** Some UML tools allow you to generate documentation directly from your diagrams. This can include class documentation, use case descriptions, and other relevant information.

## Steps to Create UML Diagrams

**Step 1**

Identify the Purpose

**Step 2**

Identify Elements and Relationships

**Step 3**

Select the Appropriate UML Diagram Type

**Step 4**

Create a Rough Sketch

**Step 5**

Choose a UML Modeling Tool

**Step 6**

Create the Diagram

**Step 7**

Define Element Properties

**Step 8**

Add Annotations and Comments

**Step 9**

Validate and Review

**Step 10**

Refine and Iterate

**Step 11**

Generate Documentation

## UML Diagrams

**Common Challenges in UML Modeling :**

1. **Time-Intensive:** UML modeling can be perceived as time-consuming, especially in fast-paced Agile environments where rapid development is emphasized. Teams may struggle to keep up with the need for frequent updates to UML diagrams.
2. **Over-Documentation:** Agile principles value working software over comprehensive documentation. There's a risk of over-documentation when using UML, as teams may spend too much time on detailed diagrams that do not directly contribute to delivering value.
3. **Changing Requirements:** Agile projects often face changing requirements, and UML diagrams may become quickly outdated. Keeping up with these changes and ensuring that UML models reflect the current system state can be challenging.
4. **Collaboration Issues:** Agile emphasizes collaboration among team members, and sometimes UML diagrams are seen as artifacts that only certain team members understand. Ensuring that everyone can contribute to and benefit from UML models can be a challenge.

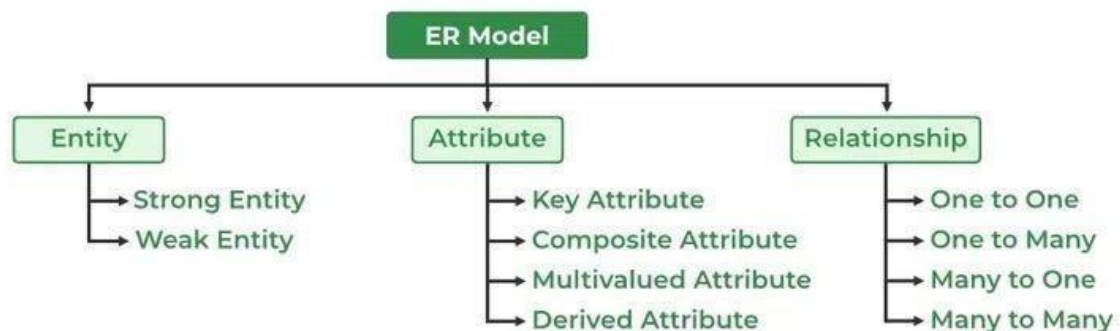
## EXPERIMENT-1

**AIM:** Demonstration of ER Diagram and to construct ER diagrams for given case studies.

### DESCRIPTION:

The **Entity Relational Model** is a model for identifying entities to be represented in the database and representation of how those entities are related. The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically.

- ER diagrams are used to represent the E-R model in a database, which makes them easy to convert into relations (tables).
- ER diagrams provide the purpose of real-world modeling of objects which makes them intently useful.
- ER diagrams require no technical knowledge and no hardware support.
- These diagrams are very easy to understand and easy to create even for a naive user.
- It gives a standard solution for visualizing the data logically.



### Components of ER Diagram :

ER Model consists of Entities, Attributes, and Relationships among Entities in a Database System.

- **Entity:** A definable thing—such as a person, object, concept or event—that can have data stored about it. Think of entities as nouns. Examples: a customer, student, car or product. Typically shown as a rectangle.










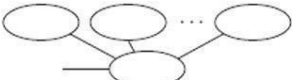



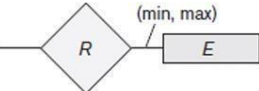
- **Entity type:** A group of definable things, such as students or athletes, whereas the entity would be the specific student or athlete. Other examples: customers, cars or products.
- **Entity categories:** Entities are categorized as strong, weak or associative.
- A **strong entity** can be defined solely by its own attributes, while a weak entity cannot. An associative entity associates entities (or elements) within an **entity set**.
- **Entity keys:** Refers to an attribute that uniquely defines an entity in an entity set. Entity keys can be super, candidate or primary.
- **Super key:** A set of attributes (one or more) that together define an entity in an entity set.
- **Candidate key:** A minimal super key, meaning it has the least possible number of attributes to still be a super key. An entity set may have more than one candidate key.
- **Primary key:** A candidate key chosen by the database designer to uniquely identify the entity set.
- **Foreign key:** Identifies the relationship between entities.

### Cardinality of Relationships:

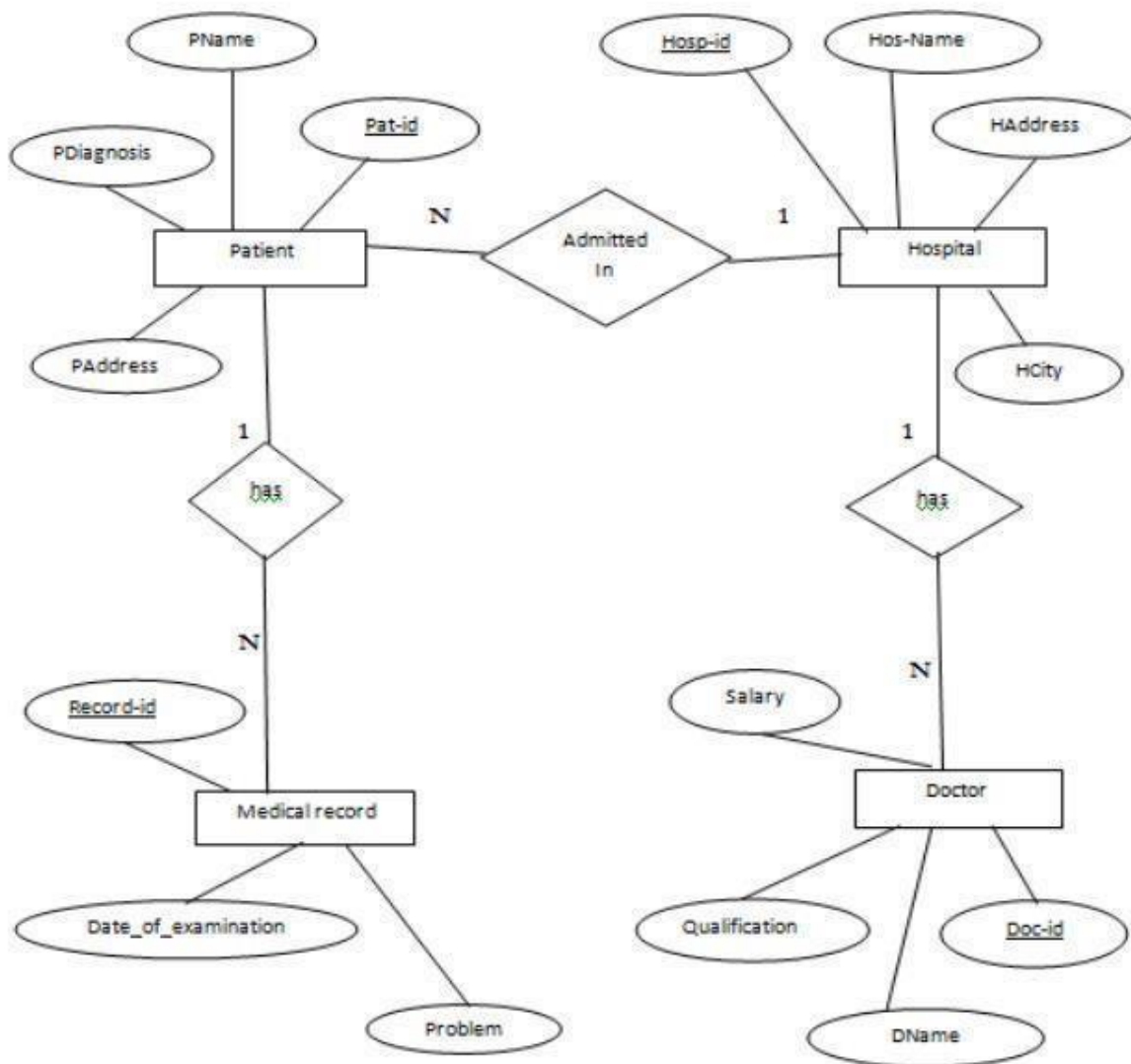
Cardinality is the number of entity instances to which another entity set can map under the relationship. This does not reflect a requirement that an entity has to participate in a relationship. Participation is another concept.

- **One-to-one:** X-Y is 1:1 when each entity in X is associated with at most one entity in Y, and each entity in Y is associated with at most one entity in X.
- **One-to-many:** X-Y is 1:M when each entity in X can be associated with many entities in Y, but each entity in Y is associated with at most one entity in X.
- **Many-to-many:** X:Y is M:M if each entity in X can be associated with many entities in Y, and each entity in Y is associated with many entities in X.

**ERD symbols and notations:** There are several notation systems:

Symbol	Meaning	Figure 7.14 Summary of the notation for ER diagrams.
	Entity	
	Weak Entity	
	Relationship	
	Identifying Relationship	
	Attribute	
	Key Attribute	
	Multivalued Attribute	
	Composite Attribute	
	Derived Attribute	
	Total Participation of $E_2$ in $R$	
	Cardinality Ratio 1: N $\Rightarrow$ 1 $E_1$ entity can be related to N $E_2$ entities	
	Structural Constraint (min, max) on Participation of $E$ in $R$	

## Hospital Management System:





## **EXPERIMENT-2**

**AIM:** To demonstrate Use Case Diagrams in Unified Modelling Language (UML).

### **DESCRIPTION:**

A Use Case Diagram is a type of behavioral diagram in the Unified Modeling Language (UML) that illustrates the interactions between users (actors) and a system to achieve specific goals. It provides a high-level overview of the functionalities provided by a system and the actors involved in those functionalities.

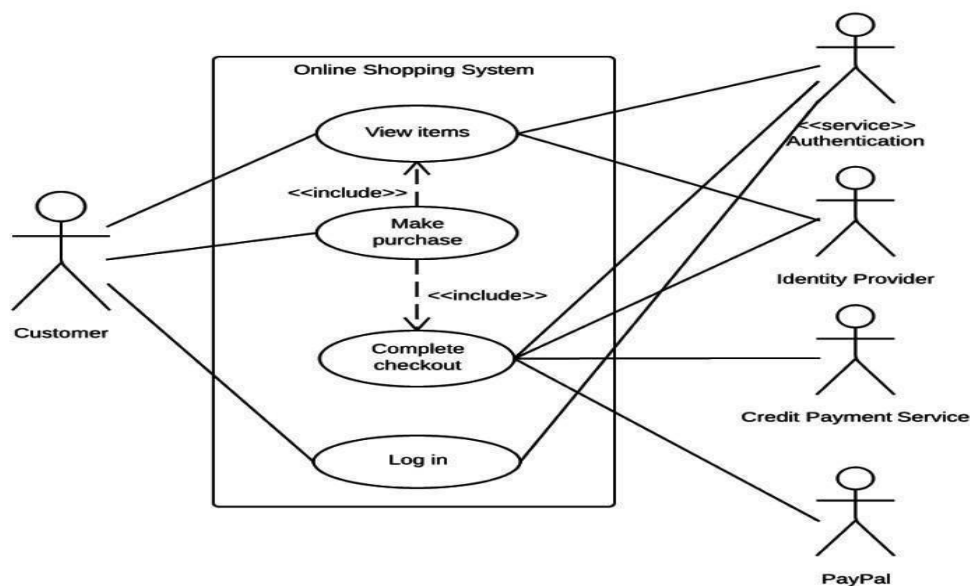
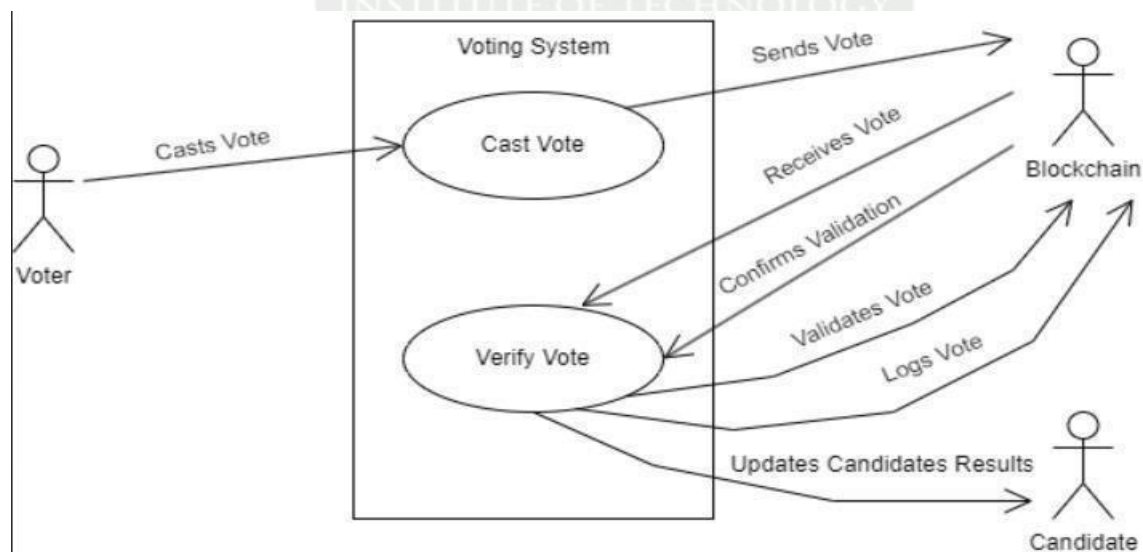
Here are the key elements of a Use Case Diagram:

- **Actors:** Actors represent the users or external systems interacting with the system. An actor can be a human user, another system, or even a hardware device. Actors are usually depicted as stick figures or blocks on the edges of the diagram.
- **Use Cases:** Use cases represent the specific functionalities or tasks that the system performs to achieve the goals of the actors. They describe the interaction between the system and its users to accomplish a particular goal. Use cases are typically represented as ovals or ellipses within the diagram.
- **Relationships:** Relationships between actors and use cases show which actors are involved in each use case. A solid line connecting an actor to a use case indicates that the actor participates in that use case.
- **System Boundary:** The system boundary, often represented as a rectangle, encloses all the use cases of the system. It defines the scope of the system being modeled.

Use Case Diagrams are beneficial for various reasons:

- **Understanding System Requirements:** They help stakeholders, including developers, designers, and clients, understand the system's functionalities and requirements from a user's perspective.
- **Communication:** Use Case Diagrams serve as a visual communication tool that bridges the gap between technical and non-technical stakeholders, ensuring everyone has a shared understanding of the system.
- **Validation:** They aid in validating system requirements by visualizing the interactions between users and the system, helping to identify potential gaps or ambiguities in the requirements.
- **System Design:** Use Case Diagrams can guide system design by identifying the key functionalities and interactions that need to be implemented, providing a foundation for more detailed design and development activities.

In summary, a Use Case Diagram is a valuable tool in the software development process, helping to visualize and clarify the system's functionalities, interactions, and requirements from a user-centric perspective.

**DIAGRAMS:****Online shopping system****Online voting system using blockchain:**

### **EXPERIMENT-3**

**AIM:** To demonstrate Data Flow Diagrams (DFD).

#### **DESCRIPTION:**

A Data Flow Diagram (DFD) is a visual representation that illustrates the flow of data within a system. It depicts how data moves between different components of a system, including external entities, processes, and data stores. DFDs provide a clear and concise overview of the system's data flow and interactions. They help in understanding the system's functionalities, requirements, and architecture.

DFDs are essential tools in systems analysis and design, aiding in requirement analysis, system design, implementation, and testing phases of the Software Development Lifecycle (SDLC). DFDs come in different levels, such as Level 0 (Context Diagram), Level 1, Level 2, etc., each providing varying levels of detail. The components of a DFD include External Entities, Processes, Data Stores, and Data Flows. External Entities represent sources or destinations of data outside the system. Processes depict the activities or functions performed by the system. Data Stores are repositories where data is stored, and Data Flows show the movement of data between components.

DFDs assist in identifying data sources, transformations, and sinks in a system. They help in identifying potential bottlenecks, redundancies, or inefficiencies in data flow and processes. DFDs are used across various industries and domains, including software development, business analysis, and system architecture. They aid in designing scalable, efficient, and robust systems by providing insights into data flow patterns and interactions. DFDs can be manually drawn or created using specialized software tools, making them versatile and adaptable to different project requirements.

Here are the key components of a Data Flow Diagram:

- **External Entities:** External entities represent external sources or destinations of data, such as users, systems, or organizations that interact with the system. They are usually depicted as squares on the boundaries of the diagram.
- **Processes:** Processes represent the activities or transformations that occur within the system. They describe how data is input, manipulated, and output by the system. Processes are typically depicted as circles or ovals within the diagram.
- **Data Stores:** Data stores represent repositories where data is stored within the system. They can be databases, files, or any other storage medium. Data stores are usually depicted as two parallel lines.
- **Data Flows:** Data flows represent the movement of data between external entities, processes, and data stores. They indicate the direction in which data is transferred.

and are depicted as arrows.

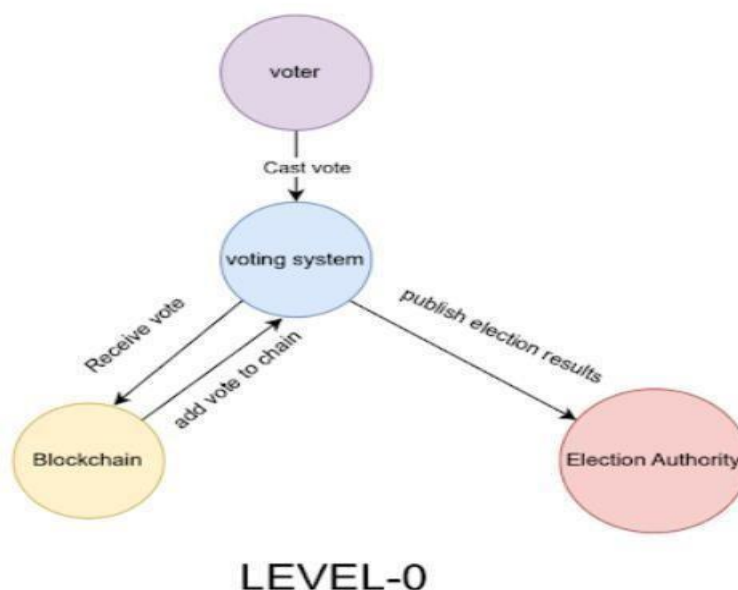
Types of Data Flow Diagrams:

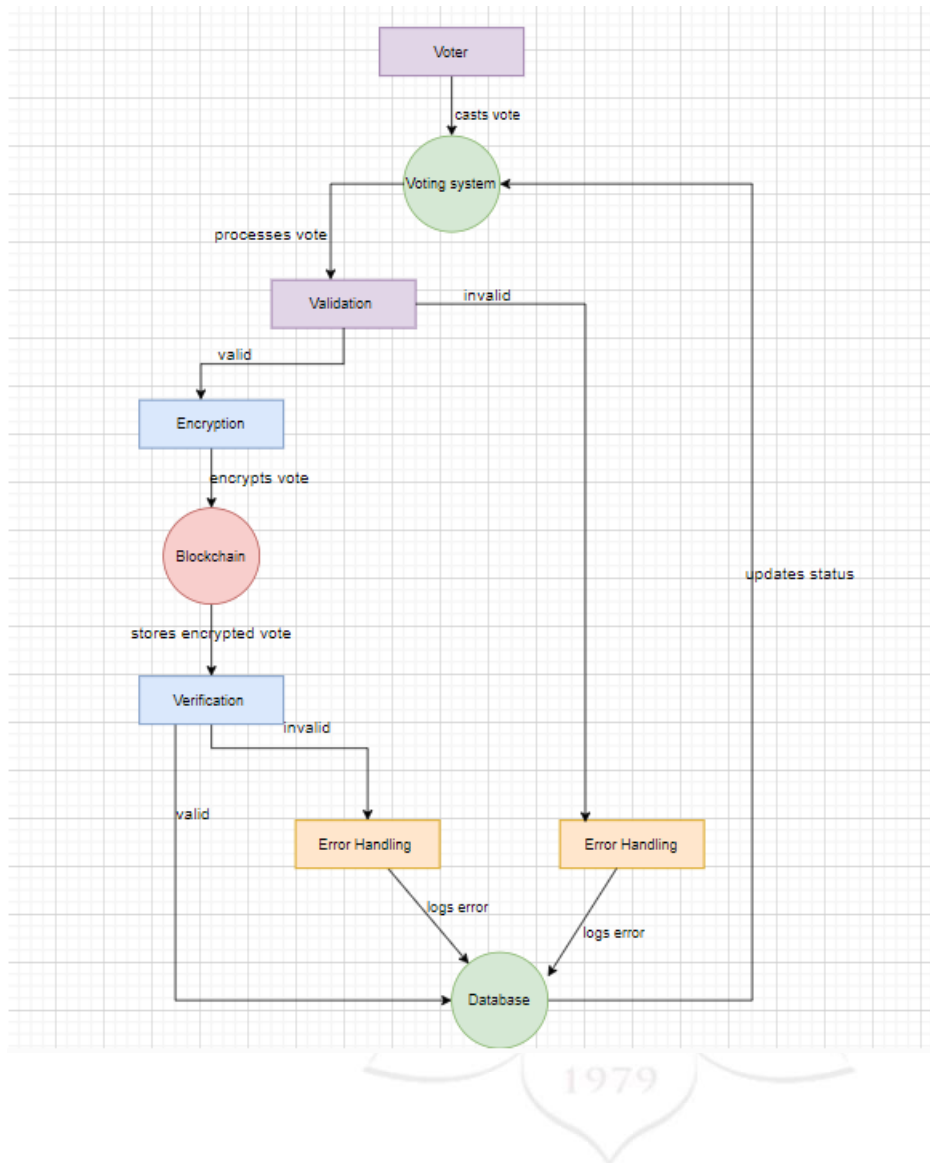
- Level 0 DFD (Context Diagram): This is the highest level of a DFD and provides a broad overview of the system. It shows the interactions between the system and external entities without detailing the internal processes.
- Level 1 DFD: This level provides a more detailed view of the system by breaking down the processes of the context diagram into sub-processes. It helps in understanding the internal workings of the system.
- Level 2 DFD, Level 3 DFD, etc: These levels further decompose the processes of the previous level into more detailed sub-processes, providing a deeper understanding of the system's functionalities and data flow.

Benefits of Data Flow Diagrams:

- Understanding and Analysis: DFDs help stakeholders understand the system's data flow and processes, facilitating analysis, and identifying areas for improvement.
- Communication: DFDs serve as a visual communication tool that helps in conveying complex system information in a clear and concise manner to both technical and non-technical stakeholders.
- System Design: DFDs aid in designing the system by providing a blueprint of the dataflow and processes, guiding the development team in implementing the system.

**DIAGRAMS:**





## **EXPERIMENT-4**

**AIM:** To demonstrate Class Diagrams in Unified Modelling Language (UML).

**DESCRIPTION:** Class Diagrams are one of the most widely used types of diagrams in the Unified Modeling Language (UML) to visualize the structure of a system by modeling its classes, their attributes, operations or methods, and the relationships between classes.

Key Components of Class Diagrams:

1. Classes:

- **Definition:** Classes represent the blueprint or template for creating objects in object-oriented programming. They encapsulate data (attributes) and behaviors (methods) related to a particular entity or concept.
- **Representation:** In a class diagram, classes are depicted as rectangles with three compartments: the top compartment contains the class name, the middle compartment contains attributes, and the bottom compartment contains methods.

2. Attributes:

- **Definition:** Attributes represent the properties or characteristics of a class. They describe the state of an object and are defined within a class.
- **Representation:** Attributes are listed in the middle compartment of a class rectangle in a class diagram.

3. Methods (Operations):

- **Definition:** Methods, also known as operations, represent the behaviors or actions that a class can perform. They define how objects interact and manipulate their data.
- **Representation:** Methods are listed in the bottom compartment of a class rectangle in a class diagram.

4. Relationships:

- **Association:** Represents a bi-directional relationship between two classes, indicating that instances of one class are linked to instances of another class.
- **Aggregation:** Represents a "whole-part" relationship between classes, where one class is a part of another class but can exist independently.



- **Composition:** Represents a stronger form of aggregation, where the part cannot exist without the whole.
- **Generalization (Inheritance):** Represents an "is-a" relationship between a superclass (parent) and subclass (child), indicating that the subclass inherits attributes and behaviors from the superclass.
- **Dependency:** Represents a weaker relationship where one class depends on another class, usually through method parameters or return types.

#### 5. Multiplicity:

- **Definition:** Multiplicity defines the number of instances of one class that can be associated with instances of another class in a relationship.
- **Representation:** Multiplicity is indicated near the end of an association line using numbers, such as 1, 0..1, 0..\*, 1..\*, etc., specifying the minimum and maximum number of instances.

#### Benefits of Class Diagrams:

- **System Design:** Class Diagrams serve as a blueprint for system design, helping designers and developers visualize and organize the system's structure, relationships, and interactions.
- **Code Generation:** They assist in generating code from the model, ensuring consistency and reducing manual coding errors.
- **Documentation:** Class Diagrams act as documentation tools, capturing the system's architecture, relationships, and design decisions, facilitating future maintenance and enhancements.
- **Communication:** They serve as a communication tool between stakeholders, including developers, designers, testers, and business analysts, ensuring a shared understanding of the system's design and requirements.

#### Usage in Software Development:

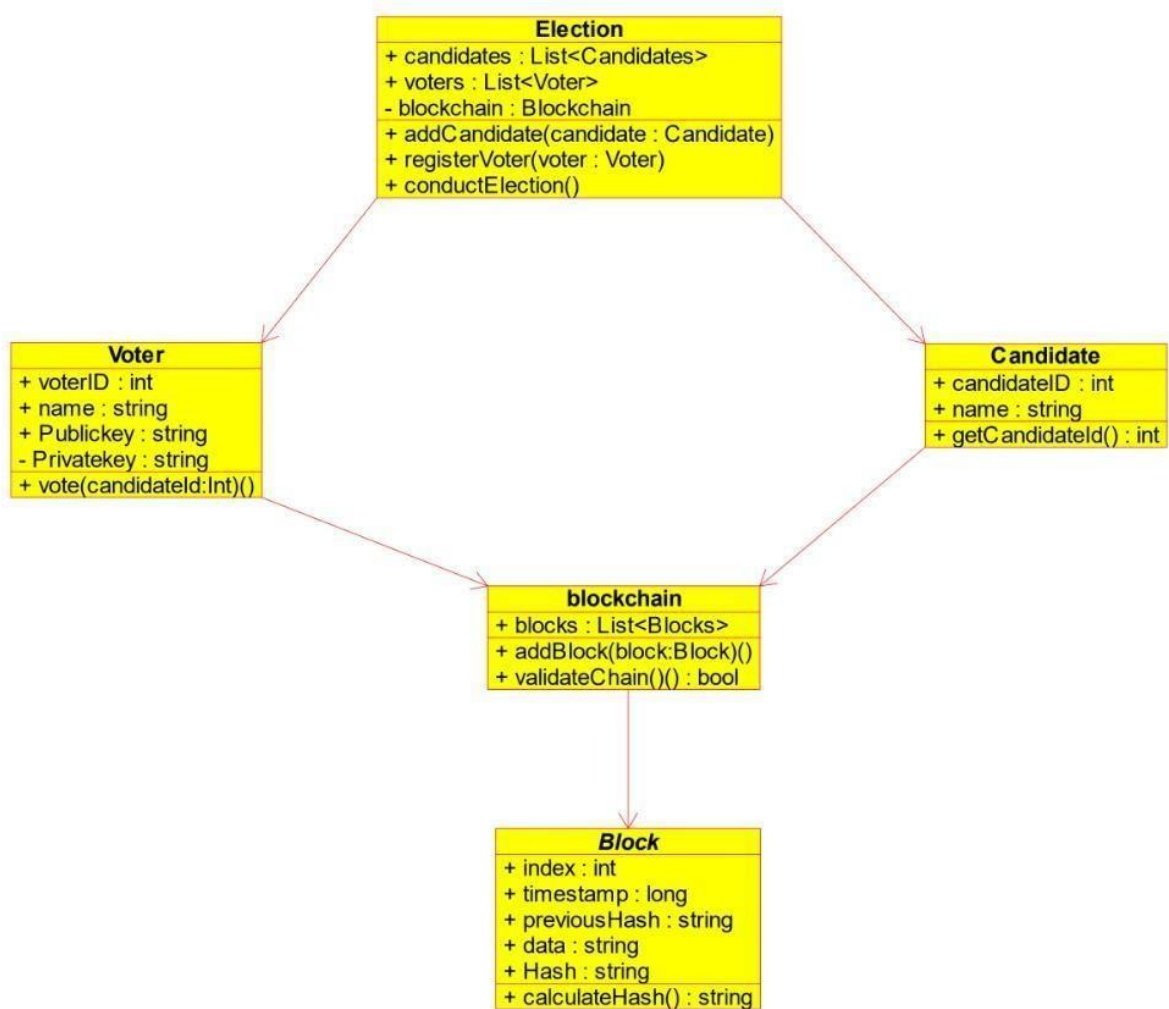
- **Requirement Analysis:** Class Diagrams aid in capturing and analyzing system requirements, identifying classes, attributes, and relationships based on the user's needs and functionalities.
- **Implementation:** They guide the implementation phase by providing a clear structure and design for developers to follow, ensuring the system's consistency and maintainability.
- **Testing:** Class Diagrams help testers understand the system's structure, relationships, and interactions, enabling effective test case design and validation.

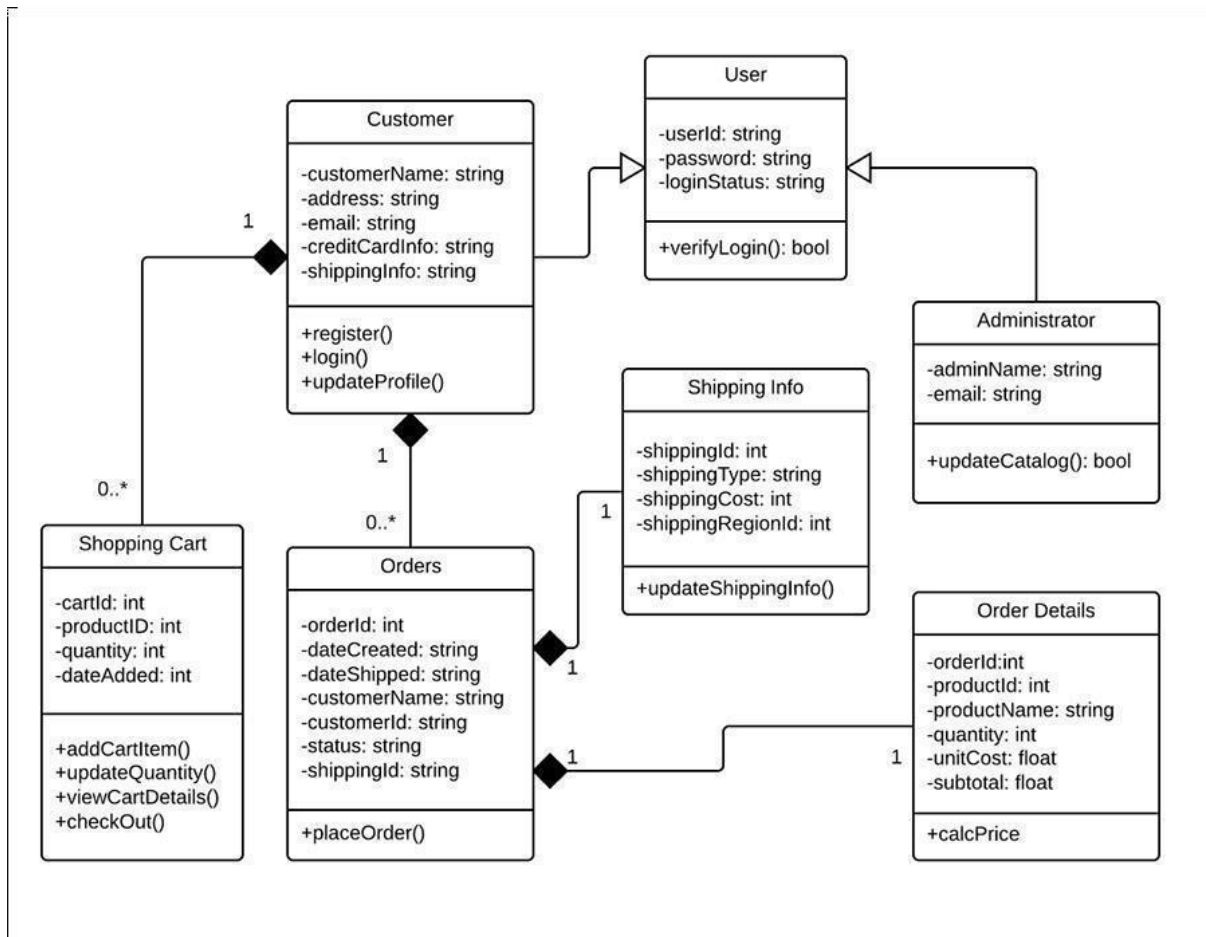
In summary, Class Diagrams are essential tools in object-oriented analysis and design, providing a visual representation of a system's structure and relationships. They facilitate

understanding, communication, design, implementation, and documentation of complex software systems, ensuring clarity, consistency, and scalability throughout the software development lifecycle.

### DIAGRAMS:

#### Online Voting System using Blockchain:



**Online shopping system:**

స్వయం సేవస్వయం భవ

1979

### **EXPERIMENT-5**

**AIM:** To demonstrate Sequence Diagram in Unified Modelling Language (UML).

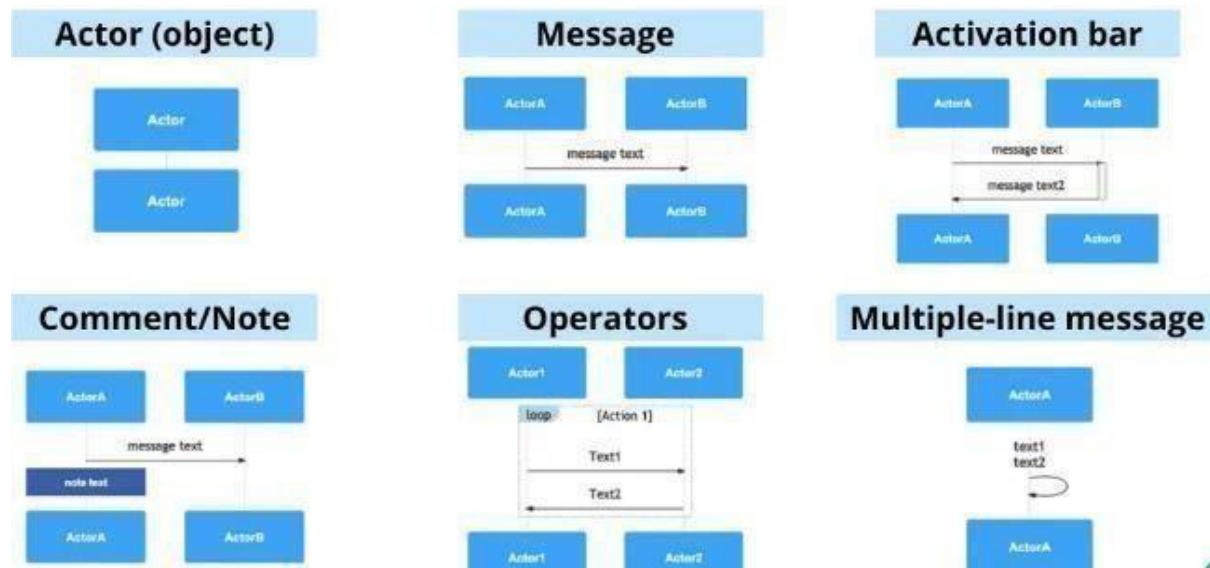
**DESCRIPTION:** Sequence diagrams are a type of interaction diagram that describe how objects interact in a particular scenario of a business or a software system. They are a key component of the Unified Modeling Language (UML), a standardized modeling language used in software engineering and systems design. Here are the key aspects of sequence diagrams:

#### **Purpose**

- Sequence diagrams are used to visualize the sequence of messages exchanged between objects in order to carry out a specific functionality or a process. They are particularly useful for:
- Understanding the flow of a process or operation within the system.
- Designing the interactions between components.
- Documenting the existing systems or software architecture.
- Clarifying complex sequences of events and ensuring all possible scenarios are accounted for.

#### **Components**

- **Actors:** Represent external entities that interact with the system (e.g., users, other systems).
- **Objects:** Instances of classes or other entities that participate in the interaction.
- **Lifelines:** Represent the existence of an object over time. Lifelines are depicted as vertical dashed lines.
- **Activation Bars:** Show the time period during which an object is performing an action.
- **Messages:** Horizontal arrows that indicate communication between objects. They can represent method calls, returns, or signals.
- **Synchronous Messages:** Indicate that the sender waits for the recipient to process the message and return control.
- **Asynchronous Messages:** Indicate that the sender does not wait for the recipient to process the message.
- **Frames:** Enclose a section of the sequence diagram to represent conditional or iterative behavior.



### Benefits

- Clarity: Provides a clear visual representation of interactions over time.
- Communication: Helps in communication between stakeholders, such as developers, analysts, and business users.
- Error Detection: Aids in identifying potential issues and inefficiencies in the process.
- Documentation: Serves as a part of comprehensive system documentation.

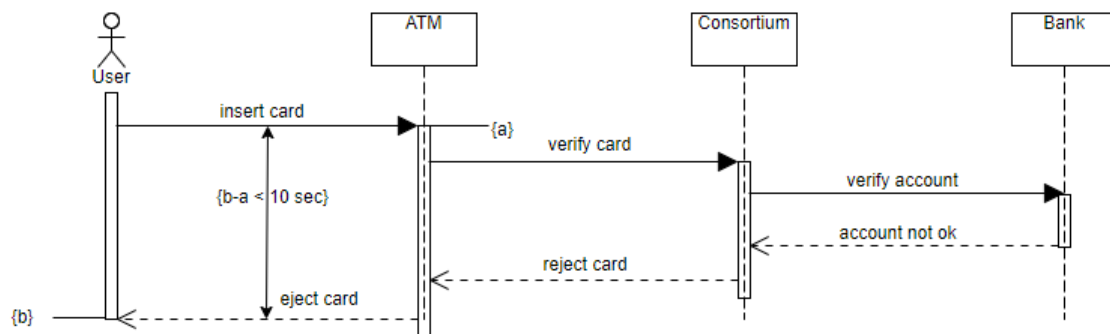
### Limitations

- Complexity: Can become complex and difficult to manage for large systems with many interactions.
- Maintenance: Requires updating along with the system to remain accurate and useful.

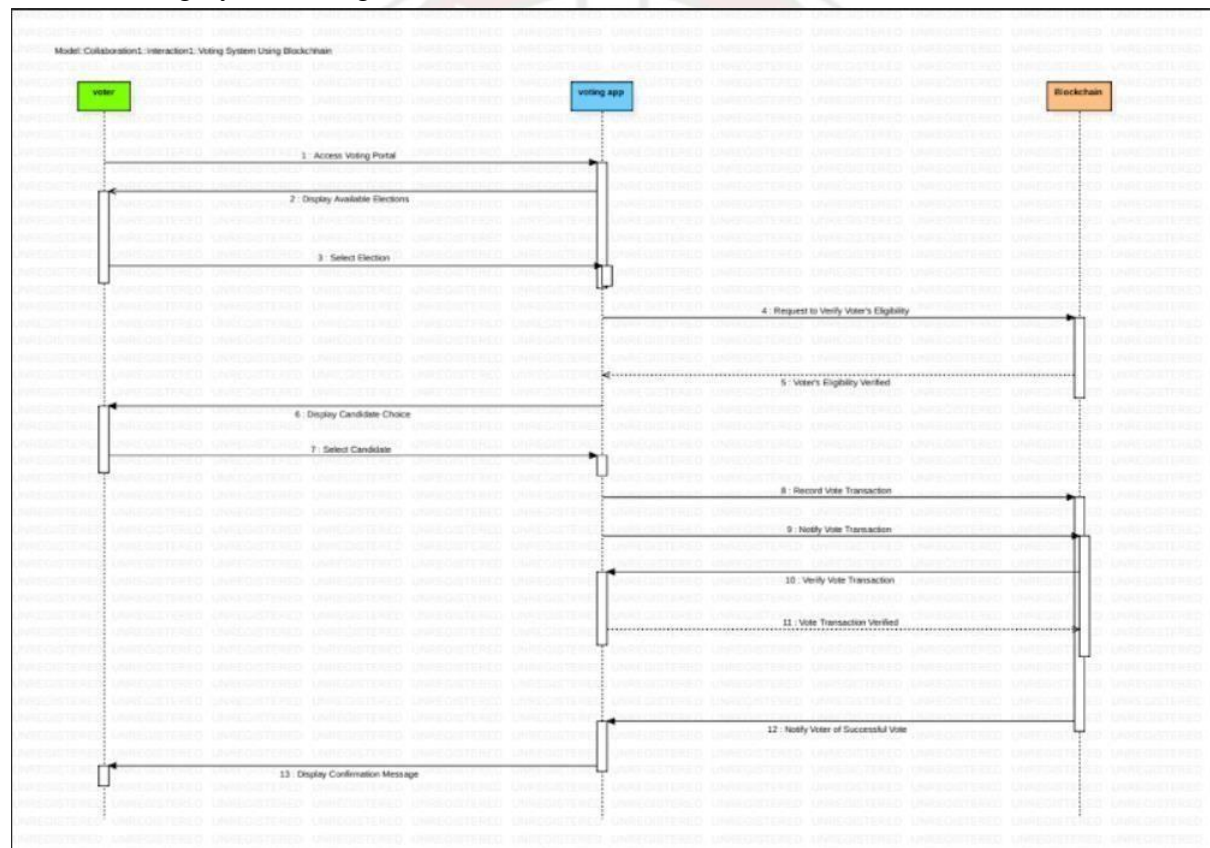
In summary, Sequence diagrams are an essential tool in the toolkit of system designers and software engineers, enabling them to design, analyze, and communicate the dynamic aspects of systems effectively.

**DIAGRAMS:**

## ATM Transfer



## Online Voting System using Blockchain





### **EXPERIMENT-6**

**AIM:** To demonstrate State Diagrams in Unified Modelling Diagram (UML).

**DESCRIPTION:** State diagrams, also known as state machine diagrams, are powerful tools in UML for visualizing the dynamic behavior of an object or system. They depict the various states an entity can exist in and the events that trigger transitions between those states. Here's a detailed breakdown:

Components:




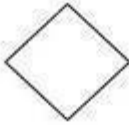
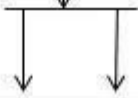
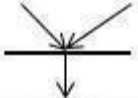

- **States:** Represented by rounded rectangles, states signify distinct conditions or situations the system can be in. Each state has a unique name that reflects its purpose.
- **Transitions:** Transitions are directed arrows connecting states and indicate the events that cause a state change. They are labeled with the triggering event and optionally, actions performed during the transition. These actions can modify the system's attributes or perform specific operations.
- **Initial State:** A solid black circle denotes the starting point of the system's execution. There can only be one initial state per diagram.
- **Final State (Optional):** A state marked with a bull's-eye symbol represents the system's termination point. Final states are optional, and a diagram can have zero or more final states.
- **Guards (Optional):** Guards are Boolean expressions attached to transitions that determine whether a transition can be triggered based on a specific condition. They are written within square brackets [guard\_condition] alongside the event name on the transition label.
- **Activities (Optional):** Activities are actions performed within a state, typically for a specific duration. They are denoted by text enclosed in curly braces {activity\_name} within the state rectangle.

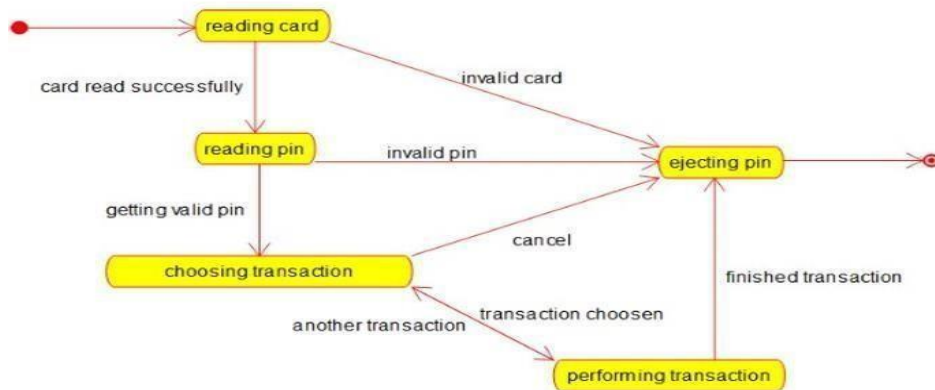
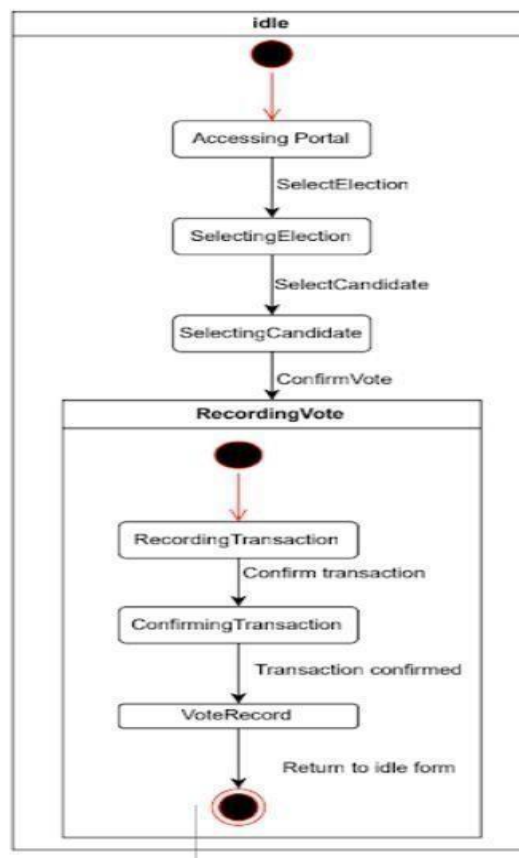
Benefits of using State Diagrams:

- **Clarity and Communication:** State diagrams offer a visual representation of the system's behavior, simplifying communication and understanding between developers, designers, and stakeholders.

- Early Error Detection: By modeling different states and transitions, you can identify potential issues like unreachable states, unintended state changes, or missing transitions, leading to earlier error detection and improved system design.
- Improved Maintainability: State diagrams serve as a form of documentation, making it easier to understand existing code and maintain the system over time.

State Diagrams can get complex with many states and nesting, have limited error handling focus, and might not capture all system aspects. Techniques like decomposition and using other UML diagrams can help address these limitations.

Name	Symbol
Start Node	
Action State	
Control Flow	
Decision Node	
Fork	
Join	
End State	

**DIAGRAMS:****ATM Transfer****Online voting system using blockchain:**

## EXPERIMENT-7

**AIM:** To demonstrate Activity Diagrams in Unified Modelling Diagram (UML).

**DESCRIPTION:** Activity diagram in UML is a type of behavioral diagram that visually portrays the flow of actions within a system. It's essentially a flowchart on steroids, offering a more powerful way to depict the system's dynamic behavior.

**Purpose:**

- To model the sequential or concurrent activities performed by a system in response to a stimulus.

**Elements:**

- **Activities:** Represented by rounded rectangles, they signify actions or processes the system performs.
- **Transitions:** Arrows indicating the flow of control between activities, often triggered by events or completion of the previous activity.

**Additional elements:**

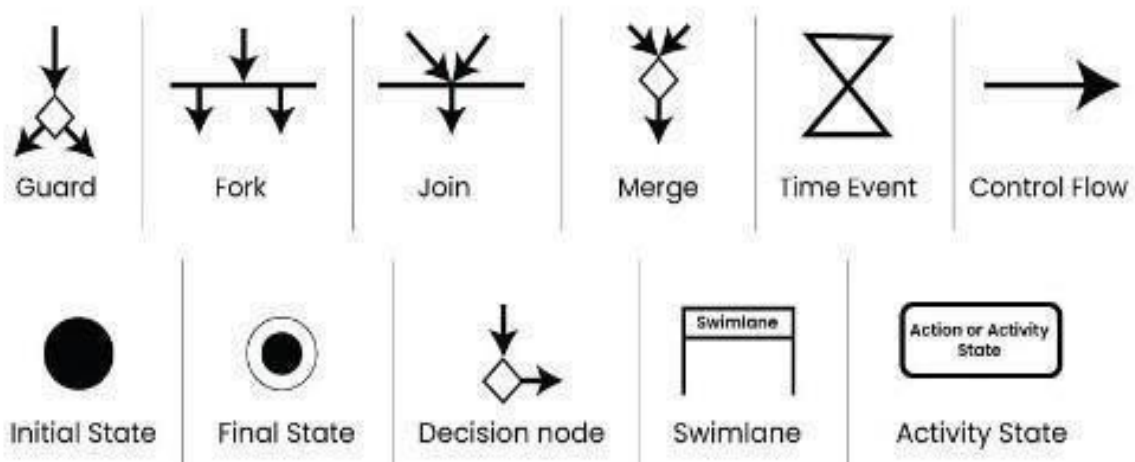
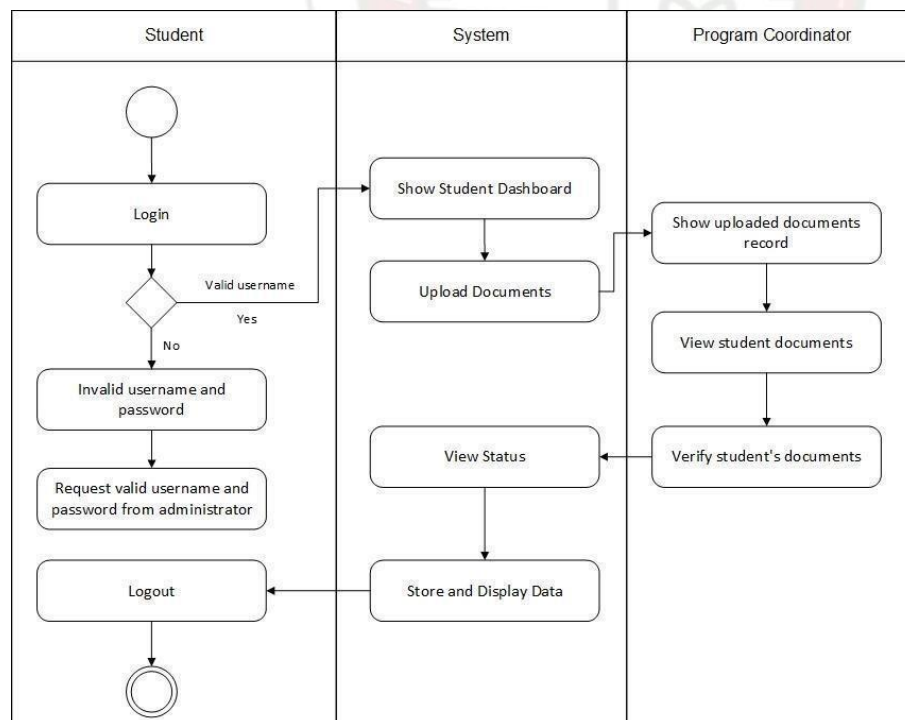
- **Swimlanes:** Divide the diagram horizontally to group activities based on the performers (e.g., actors, objects).
- **Decision/Merge Nodes:** Diamond shapes representing decision points with conditional branching and rejoining of execution paths.
- **Forks/Joins:** Depict splitting or merging of concurrent activities.
- **Object Flows (Optional):** Represent the flow of objects between activities, providing an object-oriented perspective.

**Benefits:**

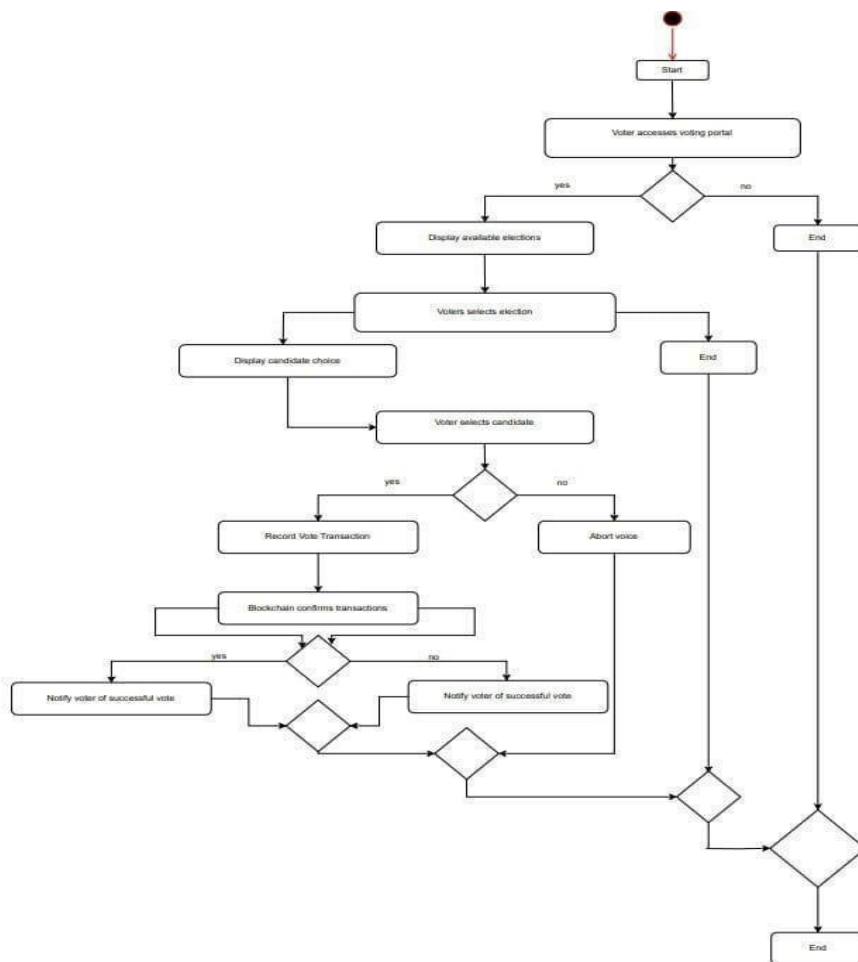
- **Clear Workflow Visualization:** Provides a step-by-step view of the system's actions, making it easier to understand complex processes.
- **Improved Communication:** Facilitates communication between stakeholders regarding the system's functionality.
- **Modeling Parallelism:** Can represent concurrent activities happening simultaneously.

Use Cases include modelling workflows, business processes, use cases, and algorithms.

In summary, activity diagrams are a valuable tool for understanding and documenting the flow of actions within a system, promoting clear communication and effective system design.

**DIAGRAMS:****Student Learning Management System**

Online voting system using blockchain:





## **EXPERIMENT-8**

**AIM:** To demonstrate Component Diagrams in Unified Modelling Diagram (UML).

**DESCRIPTION:** Component diagrams in UML (Unified Modeling Language) are like blueprints for a complex system. They offer a static view, focusing on the individual components, their interactions, and dependencies. Unlike deployment diagrams that showcase the physical layout, component diagrams delve into the modular building blocks and how they work together.

Core Components:

- **Components:** These are the independent, reusable modules that make up the system. They can be physical entities like libraries or executables, or logical groupings of functionalities. Each component is depicted as a rectangle in the diagram.
- **Interfaces:** These act as contracts between components, defining the services offered (provided interfaces) and the services required (required interfaces). Imagine them as agreements that components use to communicate. Interfaces are typically shown as circles or lollipop shapes connected to components by lines.
- **Dependencies:** These represent the relationships between components, signifying how they rely on each other's functionalities. An arrow pointing from component A to component B indicates that A depends on B to function.

Benefits of Component Diagrams:

- **Enhanced Understanding:** They provide a clear view of the system's modular structure, making it easier to grasp how components collaborate to achieve the overall goals.
- **Improved Communication:** These diagrams act as a common language for developers and stakeholders to discuss system architecture, fostering clear communication about component interactions.
- **Promoted Reusability:** By focusing on independent components, they encourage code reusability across different projects. Each component can potentially be used in various systems with minimal modification.
- **Simplified Maintenance:** Component diagrams help visualize dependencies. This allows developers to predict the impact of changes in one component on other parts of the system, streamlining maintenance efforts.

Advanced Concepts:

- **Internal Compartments (Optional):** You can add compartments within a component rectangle to show internal details or stereotypes (like «file» or «database»).

- Ports: These refine the interaction points within a component. Imagine them as specific doorways through which functionalities are provided or required.
- Provided vs. Required Interfaces: A component can have both types of interfaces. Provided interfaces expose its functionalities, while required interfaces specify the services it expects from other components.
- Assemblies: These represent groups of collaborating components that function as a single unit. They are useful for depicting complex subsystems within a larger system.

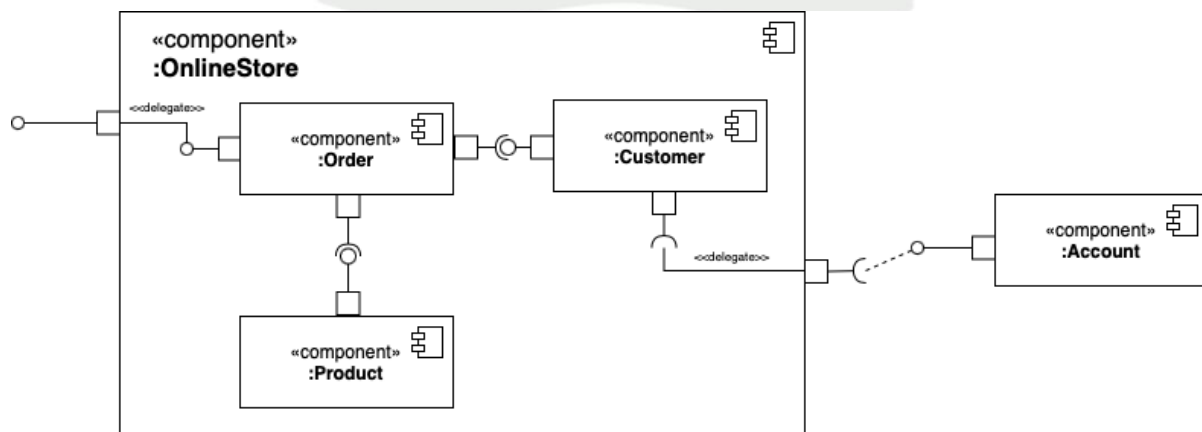
#### Things to Consider:

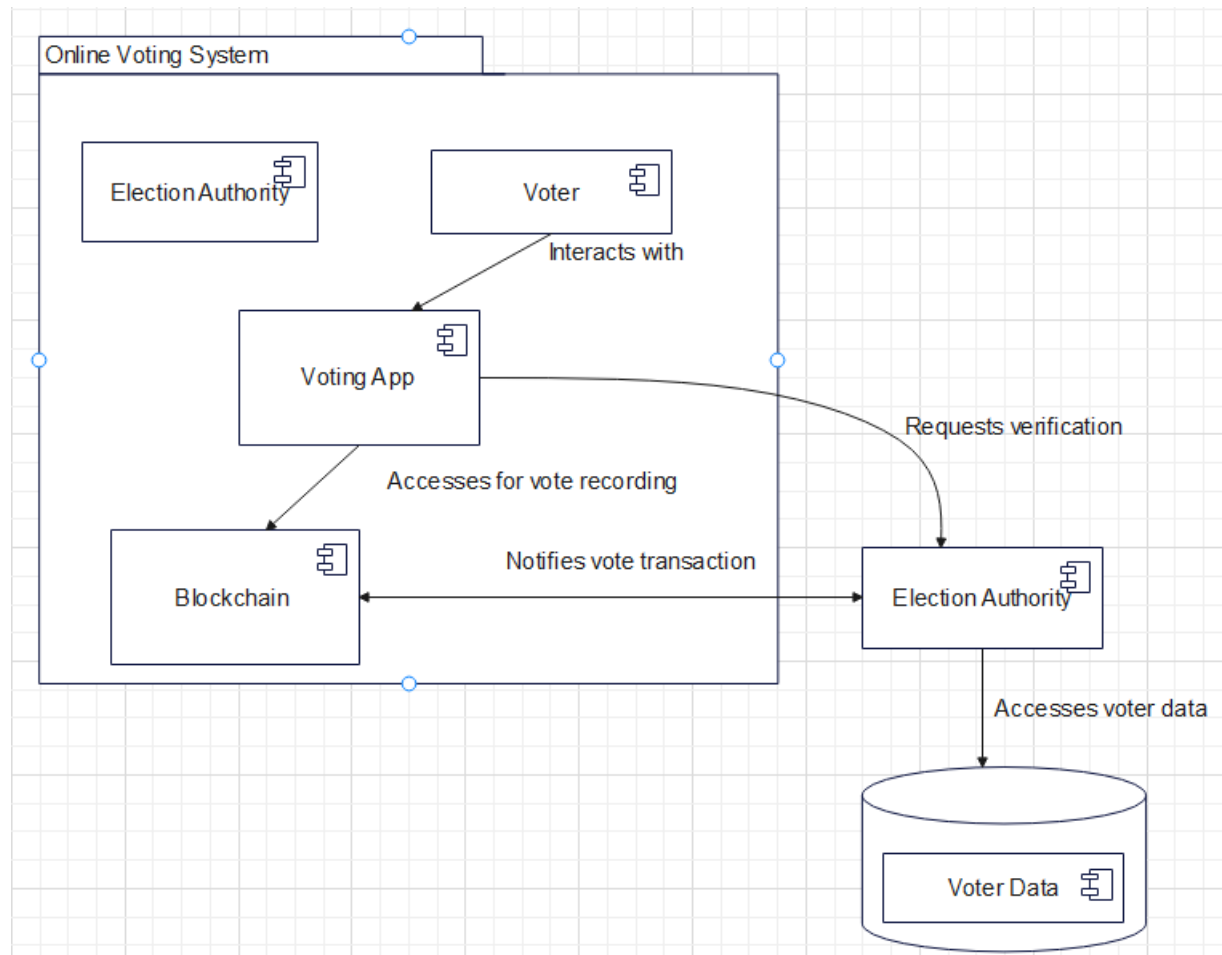
- Focus on Static Structure: Component diagrams capture a system's static view, not the dynamic behavior (how components interact over time). For that, consider using sequence diagrams or communication diagrams.
- Complexity with Large Systems: Diagrams can become cluttered for very large systems with numerous components and dependencies. Consider using packages or subsystems to manage complexity.

In essence, component diagrams are a powerful tool for visualizing and understanding how a system is built from independent, interacting components. They promote modularity, reusability, and clear communication during system design and development.

#### DIAGRAMS:

##### Online Shopping store



**Online voting system using Blockchain**

స్వయం తెలుగు భవ

1979

### **EXPERIMENT-9**

**AIM:** To demonstrate Deployment Diagrams in Unified Modelling Diagram (UML).

**DESCRIPTION:** Deployment diagrams play a crucial role in visualizing the physical architecture of a system. They essentially depict how software components are deployed onto hardware nodes, providing a clear understanding of the system's real-world implementation.

Core Elements:

- **Nodes:** These represent the physical building blocks of the system, such as servers, workstations, databases, network devices, and any other hardware components involved. Nodes are depicted as rectangular boxes in the diagram.
- **Artifacts:** These embody the deployable software entities like executable files, libraries, configuration files, and databases. Artifacts are shown as small rectangles within the nodes, signifying their deployment on those specific hardware components.
- **Deployment Specifications:** These act as guidelines or configurations that dictate how artifacts are deployed onto nodes. They are typically not visualized in the diagram but can be documented alongside it.
- **Dependencies:** Deployment diagrams also depict relationships between nodes, established through communication paths. These dependencies can be physical connections (network cables) or logical interactions (data exchange).
- **Components:** These are the independent, reusable modules that make up the system. They can be physical entities like libraries or executables, or logical groupings of functionalities. Each component is depicted as a rectangle in the diagram.
- **Interfaces:** These act as contracts between components, defining the services offered (provided interfaces) and the services required (required interfaces). Imagine them as agreements that components use to communicate. Interfaces are typically shown as circles or lollipop shapes connected to components by lines.

Benefits of Deployment Diagrams:

- **Clear Visualization:** They provide a comprehensive picture of the system's physical layout, making it easier to understand how software components interact with hardware resources.
- **Improved Communication:** These diagrams act as a bridge between development teams and system administrators, fostering clear communication about deployment strategies.
- **Enhanced Planning:** By visualizing the physical architecture, deployment diagrams aid in capacity planning, resource allocation, and identifying potential bottlenecks.

- Documentation Value: They serve as valuable documentation for future reference, system maintenance, and understanding how the system functions in the real world.

Creating a Deployment Diagram:

1. Identify Nodes: Start by listing all the hardware components involved in the system.
2. Define Artifacts: Determine the software entities that need to be deployed onto the nodes.
3. Map Artifacts to Nodes: Specify which artifacts will be deployed on each node.
4. Establish Dependencies: Show the communication paths or interactions between nodes.
5. Document Deployment Specifications (Optional): If necessary, document any specific configurations or instructions for deployment.

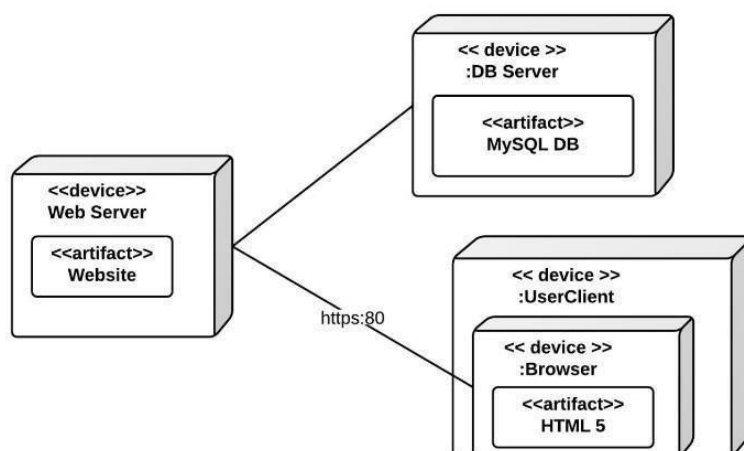
Beyond the Basics:

- Deployment Targets: These represent physical or logical locations where artifacts are deployed (e.g., development server, production environment).
- Deployment Environments: Diagrams can encompass different deployment environments like development, testing, and production.

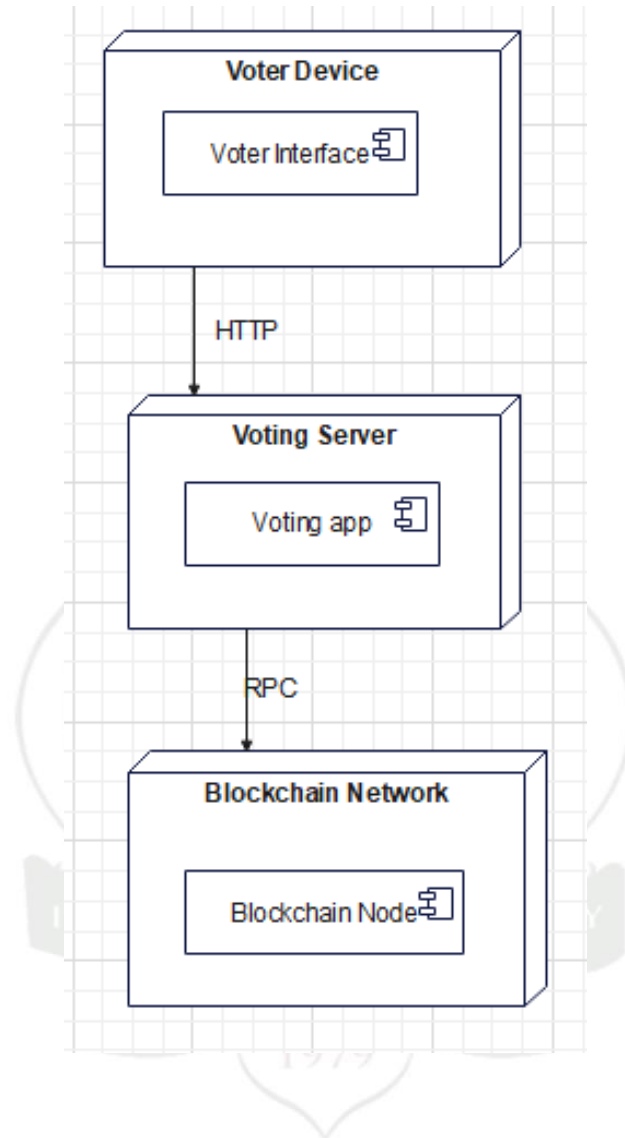
In Conclusion, deployment diagrams are a valuable asset in the UML toolbox. By providing a clear picture of the system's physical architecture, they promote better communication, planning, and understanding of how software interacts with the real world. So, the next time you're designing or deploying a system, leverage deployment diagrams to bridge the gap between the software and the hardware landscape.

## DIAGRAMS:

### Basic Webpage



Online voting system using Blockchain





## **EXPERIMENT-10**

**AIM:** To demonstrate Collaboration Diagrams in Unified Modelling Diagram (UML).

**DESCRIPTION:** Collaboration diagrams, also known as communication diagrams in UML, provide a visual representation of how objects collaborate to fulfill a certain task or accomplish a specific behavior within a system.

Here's a deeper dive into their features:

- **Objects and Actors:** Collaboration diagrams showcase the various objects or actors involved in the interaction. Objects, which are instances of classes, are depicted as rectangles with the object's name and its class type. Actors, representing entities external to the system, are also included if they play a role in the collaboration.
- **Links and Associations:** Links, depicted as lines connecting objects or actors, illustrate the communication pathways between them. These links often represent associations or relationships between the objects, indicating how they interact and exchange information.
- **Messages and Interactions:** Messages are the core elements of collaboration diagrams, representing the communication between objects or actors. Messages can take different forms, such as method calls, signals, or data exchanges, and they are typically labeled with the name of the message and any relevant parameters. The flow of messages demonstrates the sequence of interactions within the collaboration.
- **Roles and Responsibilities:** Collaboration diagrams may highlight the roles and responsibilities of each object or actor involved in the interaction. By assigning roles, the diagram clarifies the specific functions or behaviors that each participant contributes to the collaboration, facilitating a better understanding of the system's dynamics.
- **Contextualization:** Collaboration diagrams provide a contextual view of how objects collaborate within a particular scenario or use case. They focus on illustrating the interactions relevant to a specific behavior or task, allowing stakeholders to grasp the essential communication patterns without overwhelming details.
- **Complementarity with Sequence Diagrams:** While collaboration diagrams emphasize the structural aspects of interactions, sequence diagrams delve into the temporal sequence of messages exchanged between objects over time. Together, these diagram types offer complementary perspectives on system behavior, with collaboration diagrams providing a broader overview and sequence diagrams offering detailed insight into message sequencing.

- **Simplicity and Abstraction:** Collaboration diagrams aim to simplify complex interactions by abstracting away unnecessary details and focusing on the essential elements of the collaboration. This abstraction enables stakeholders to grasp the system's communication patterns more intuitively, fostering clearer communication and collaboration among team members.

In essence, collaboration diagrams serve as powerful tools for modeling and analyzing the interactions between objects and actors within a system, enabling stakeholders to understand how components collaborate to achieve desired functionalities or behaviors.

## DIAGRAMS:

### Basic Job finding platform

