Module-3

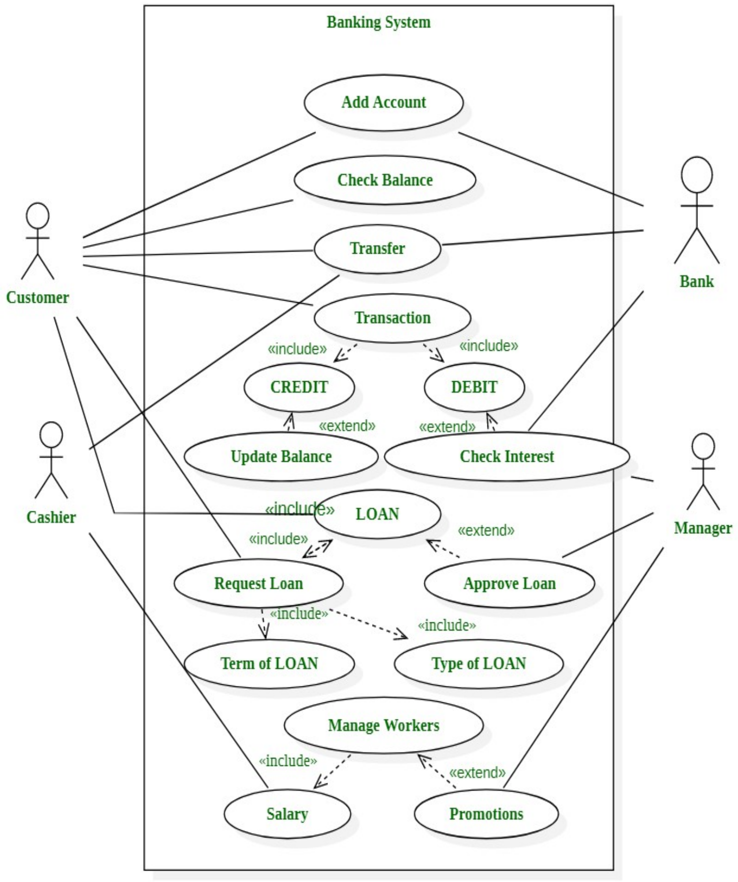
Requirements Modelling-
Scenario-Based Modeling:
Use-Case Model:

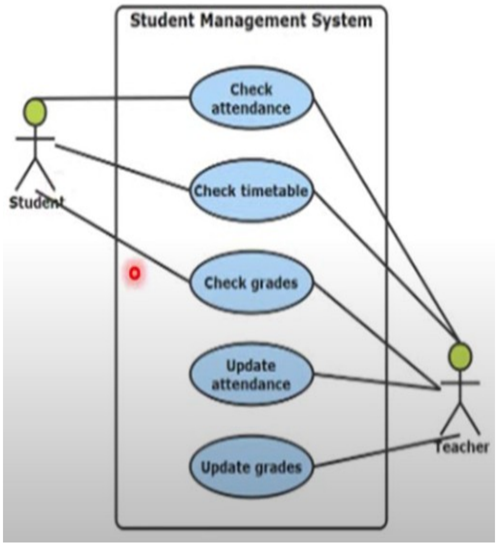
A use case model is a type of UML (Unified Modeling Language) diagram that describes the interactions between users and a system. It is used to capture the functional requirements of a system from the user's perspective.

A use case model consists of three main elements:
1. **Actors:** An actor is a person, other system, or device that interacts with the system. For example, in an online banking system, the actors might be a customer, a teller, or another bank.

2. **Use cases:** A use case is a description of a sequence of actions that the system performs in response to a request from an actor. For example, in an online banking system, a use case might be "Transfer funds between accounts."

3. **Associations:** Associations show the relationships between actors and use cases. For example, in an online banking system, the "Customer" actor might be associated with the "Transfer funds between accounts" use case.





The diagram shows the following actors:
- Customer: A person who uses the online banking system.
- Teller: A bank employee who helps customers with their online banking transactions.
- Other bank: Another bank that the customer's bank can communicate with.

The diagram also shows the following use cases:
- Log in: The customer logs into the online banking system.

- View account balance: The customer views the balance of their account.
- Transfer funds: The customer transfers funds between their accounts.
- Pay bill: The customer pays a bill online.

The associations in the diagram show how the actors and use cases interact with each other. For example, the "Customer" actor is associated with the "Log in" and "View account balance" use cases, which means that the customer can perform these actions.

Advantages of the Use-Case Model:

1. Clarity and Communication: Use-case models provide a clear and visual way to represent system functionality from a user's perspective, making it easier for stakeholders to understand and communicate requirements.

2. User-Centered: They focus on user interactions and goals, ensuring that the system's functionality aligns with user needs and expectations.

3. Requirements Elicitation: Use cases help in identifying, capturing, and documenting functional requirements, ensuring that key user scenarios are considered.

4. Scope Definition: They assist in defining the scope of the system by clearly delineating what is within and outside the system boundaries.

5. Change Management: Use-case models are flexible and can accommodate changes and updates to the system's functionality as requirements evolve.

6. Test Case Generation: Use cases can serve as a basis for generating test cases, helping in the validation and verification process.

7. Project Planning: Use-case models aid in project planning by providing a foundation for task allocation, scheduling, and resource allocation.

8. Documentation: They create a comprehensive documentation of user interactions, which can be valuable for future reference and maintenance.

9. Alignment with Business Goals: Use cases tie system functionality directly to the business goals and processes, ensuring that the software solution aligns with organizational objectives.

10. Visualization: Use-case diagrams provide a visual representation of system functionality and help stakeholders quickly grasp the system's architecture and interactions.

Disadvantages of the Use-Case Model:

1. Over-Emphasis on User Perspective: Use-case models primarily focus on user interactions, which can lead to an underrepresentation of non-functional requirements, system architecture, and technical details.

2. Complexity in Large Systems: In large and complex systems, the number of use cases can become overwhelming, making it challenging to manage and maintain the model.

3. Lack of Formality: Use cases are often written in natural language, which can be imprecise and subject to interpretation, leading to potential misunderstandings.

4. Incomplete Requirements: Use-case modeling may not capture all system requirements, leading to gaps in functionality if not complemented by other modeling techniques.

5. Difficulty in Non-Interactive Systems: Use cases are most suited for systems with user interactions; they may not be as effective in modeling purely technical or batch processing systems.

6. Limited Support for Data Modeling: Use cases focus on behavior but may not provide sufficient guidance for data modeling and database design.

7. Evolution Challenges: Making changes to use cases can be time-consuming, and managing version control becomes crucial as requirements evolve.

8. Misuse or Overuse: If not used appropriately, use-case modeling can lead to overly complex models or neglect of other essential modeling techniques.

9. Skill and Training Requirements: Creating effective use-case models requires skill and training, and misapplication of the technique can result in inaccuracies or inefficiencies.

10. Maintenance Overhead: Keeping use-case models up-to-date can be resource-intensive, especially in long-term projects where requirements change frequently.

Use Case Specification:

A Use Case Specification is a detailed document that provides comprehensive information about a specific use case within a software system. It is a crucial part of the Unified Modeling Language (UML) and serves as a reference guide for software developers, designers, testers, and other stakeholders involved in the software development process.

The Use Case Specification typically includes the following elements:

1. Use Case Identifier: A unique name or identifier for the use case, which is used to reference it in other documents and discussions.

2. Use Case Name: A descriptive name that clearly defines the purpose or goal of the use case, typically in a few words or a short phrase.

3. Scope: A brief description of the system or context in which the use case operates, often including a high-level overview of the system's purpose.

4. Primary Actor: The primary actor is the user or entity that initiates and interacts with the use case. It represents the role or persona responsible for triggering the use case.

5. Stakeholders and Interests: A list of stakeholders (individuals, groups, or systems) who have an interest in the outcome of the use case, along with a description of their interests or expectations.

6. Preconditions: Conditions or requirements that must be met before the use case can be executed. These are the necessary prerequisites for the use case to begin.

7. Postconditions: Conditions or states that the system will be in after the successful completion of the use case. They describe the expected outcomes or changes resulting from executing the use case.

8. Main Flow of Events: A step-by-step description of the primary, or most common, scenario in which the use case is executed. This includes the actions taken by the actor and the system's responses.

9. Alternate and Exceptional Flows: Descriptions of alternative scenarios or exceptional conditions that may occur during the execution of the use case. These describe deviations from the main flow and how the system should respond.

10. Special Requirements: Any specific requirements or constraints related to the use case, such as performance considerations, security requirements, or compliance with regulations.

11. Assumptions: Assumptions made by the use case, such as assumptions about the availability of certain resources or external services.

12. Notes and Comments: Additional information, clarifications, or comments that provide further context or explanations related to the use case.

13. Extensions: Descriptions of extensions or additional use cases that may be triggered during the execution of this use case. These are typically cross-referenced with other use cases.

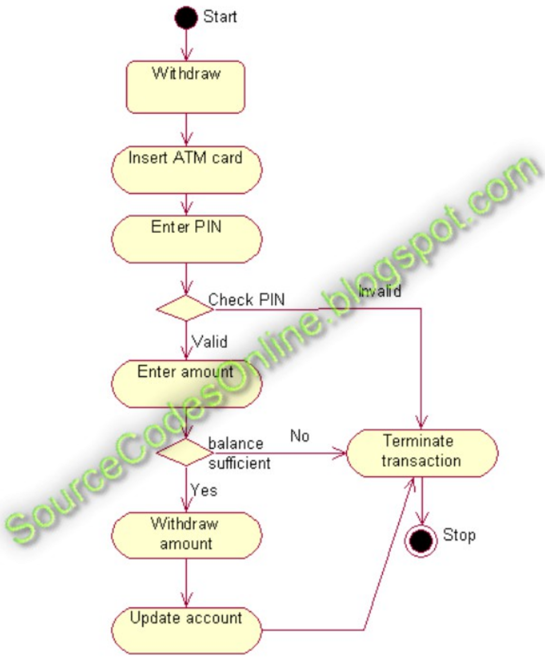
14. References: Any external references or documents related to the use case, such as links to related requirements or design documents.

Use Case Specifications are essential for documenting the functional behavior of a software system in a structured and organized manner. They facilitate effective communication among project stakeholders, guide software development and testing efforts, and serve as a basis for verifying that the system meets user requirements. Use Case Specifications are often used in conjunction with other documentation, such as use-case diagrams, to provide a comprehensive view of the system's functionality.

Activity Diagram:

An activity diagram is a type of UML (Unified Modeling Language) diagram that is used to model the flow of control in a system. It is a graphical representation of the steps involved in a process, and it can be used to depict both sequential and concurrent activities.

Activity diagrams are used in software engineering to model the workflow of a system, to visualize the dynamic aspects of a system, and to understand the conditions and constraints that affect the flow of control. They can also be used to document the design of a system and to communicate the design to stakeholders.



Advantages of Activity Diagrams:

1. Visual Representation: Activity diagrams provide a visual representation of the flow of activities and actions within a system or a specific process. This visual clarity helps in better understanding and communication among stakeholders.

2. Process Modeling: They are particularly useful for modeling business processes, workflows, and system behaviors, making them an essential tool for process analysis and design.

3. Sequential and Parallel Activities: Activity diagrams allow you to represent both sequential and parallel activities, making it easy to visualize complex processes with multiple paths and dependencies.

4. Clear Decision Points: Decision nodes and branches in activity diagrams provide a clear way to represent decision-making processes and conditions that affect the flow of activities.

5. Support for Exception Handling: Activity diagrams can include exception flows, making it possible to model error-handling and recovery processes within a system.

6. Integration with UML: Activity diagrams are part of the Unified Modeling Language (UML), making them compatible with other UML diagrams like use-case diagrams, class diagrams, and sequence diagrams. This enables a comprehensive view of the system.

7. Executable Models: In some modeling tools, activity diagrams can be used to create executable models, allowing for simulation and testing of the system's behavior before implementation.

Disadvantages of Activity Diagrams:

1. Complexity: In highly complex systems or processes, activity diagrams can become intricate and challenging to create and understand, potentially leading to confusion.

2. Lack of Detail: Activity diagrams may not provide a detailed view of certain aspects of a system, such as data handling, which might require supplementary diagrams like data flow diagrams.

3. Limited to Process Modeling: While they excel at process modeling, activity diagrams may not be the best choice for modeling other aspects of a system, such as structural relationships or object interactions.

4. Maintenance Overhead: Keeping activity diagrams up-to-date as requirements evolve can be time-consuming, and failure to do so can lead to discrepancies between the model and the actual system.

5. Complex Decision Logic: Representing complex decision logic with multiple branches and conditions can result in cluttered diagrams that are difficult to follow.

6. Ambiguity: If not properly documented and explained, activity diagrams may be open to interpretation, leading to misunderstandings among team members.

7. Overemphasis on Sequence: Activity diagrams often emphasize the sequence of activities, which may not accurately represent concurrent or asynchronous processes in certain systems.

8. Not Suitable for All Systems: Activity diagrams may not be the most suitable choice for modeling real-time systems or systems with highly concurrent and dynamic behaviors.

In summary, activity diagrams are valuable tools for modeling and visualizing processes and system behaviors, but they have limitations when it comes to representing certain aspects of complex systems or when dealing with highly dynamic or real-time scenarios. Careful consideration of the system's characteristics and requirements is essential when deciding whether to use activity diagrams in a modeling project.

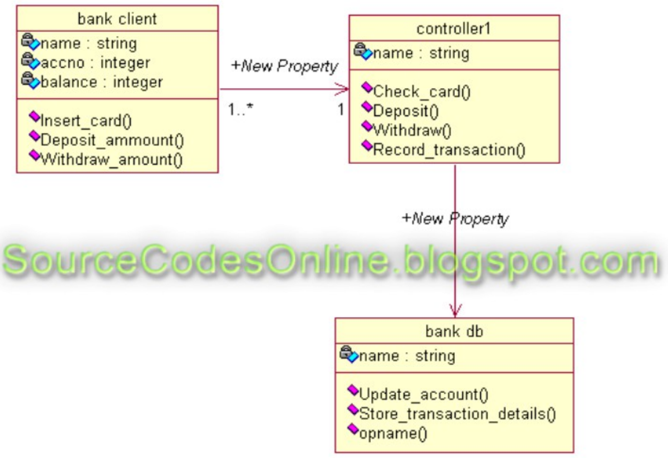
Class-Based Modeling:

Class-based modeling (CBM) is a software engineering technique that uses classes to represent the real-world objects and concepts that a software system will interact with. CBM is used to identify the classes, attributes, and relationships that the system will use, and to design the behavior of the system's objects.

CBM is typically used during the requirements gathering and analysis phases of software development. It is often used in conjunction with use case analysis, which helps to identify the different ways in which the system will be used.

Once the classes have been identified, a class diagram can be created to show the relationships between them. Class diagrams are a type of Unified Modeling Language (UML) diagram, which is a standard notation for modeling software systems.

CBM is a powerful technique for designing complex software systems. It helps to ensure that the system is well-organized and easy to understand. It also helps to identify potential problems early in the development process.



Identifying Analysis Classes:

Identifying Analysis Classes in software engineering is the process of recognizing and defining the essential objects or entities within a software system's problem domain during the initial phases of system design. These classes represent key components or concepts in the system and serve as the building blocks for designing and implementing the software. The process typically involves:

1. Gathering Requirements: Understanding the problem domain and collecting comprehensive requirements for the software system.

2. Analyzing Use Cases: Examining use cases or user stories to identify actors and their interactions with the system. Actors often correspond to potential classes.

3. Identifying Nouns and Noun Phrases: Identifying nouns and noun phrases in the requirements, as they often indicate candidate classes. For example, in an inventory management system, "product," "supplier," and "customer" could be potential classes.

4. Analyzing Relationships: Understanding how the identified objects or entities are related to each other. Relationships between objects can suggest the need for additional classes.

5. Considering Attributes: Thinking about the attributes or properties associated with each identified class. Attributes define the characteristics or data members that a class should have.

6. Exploring Commonalities: Looking for commonalities or shared attributes and behaviors among different objects. If multiple classes have similar attributes or methods, it may be appropriate to create a more abstract, generalized class to represent the commonalities.

7. Determining Inheritance and Generalization: Assessing whether relationships suggest the use of inheritance or generalization. Inheritance can lead to the creation of base or abstract classes that capture

shared attributes and methods.

8. Identifying Opportunities for Reuse: Recognizing opportunities for code reuse, especially when similar classes or functionality appear in different parts of the system. Creating reusable classes can help minimize redundancy.

9. Collaboration and Validation: Collaborating with stakeholders, domain experts, and team members to validate the identified classes, ensuring they accurately represent the problem domain and meet the system's requirements.

10. Iterating and Refining: Recognizing that class identification is an iterative process. As the understanding of the system and its requirements deepens, refinement or the addition of new classes may be necessary.

11. Documentation: Documenting the identified analysis classes, including their attributes, relationships, and relevant behavior. This documentation serves as a reference for design and development.

12. Creating Class Diagrams: Using class diagrams, which are part of the Unified Modeling Language (UML), to visually represent the identified classes and their relationships. Class diagrams provide a clear and structured way to depict the system's architecture.

Identifying analysis classes is a foundational step in object-oriented software design, helping ensure that the software system aligns with the problem domain and effectively meets its intended goals and requirements.

CRC Model:

CRC Model, which stands for "Class-Responsibility-Collaboration" Model, is a lightweight technique used in object-oriented software design to facilitate the identification and organization of classes, their responsibilities, and their collaborations within a software system. It is a simple yet effective way to capture the essential aspects of class design and interaction.

A CRC card is a threecolumn card that is used to represent a class. The first column of the card lists the class name, the second column lists the responsibilities of the class, and the third column lists the collaborators of the class.

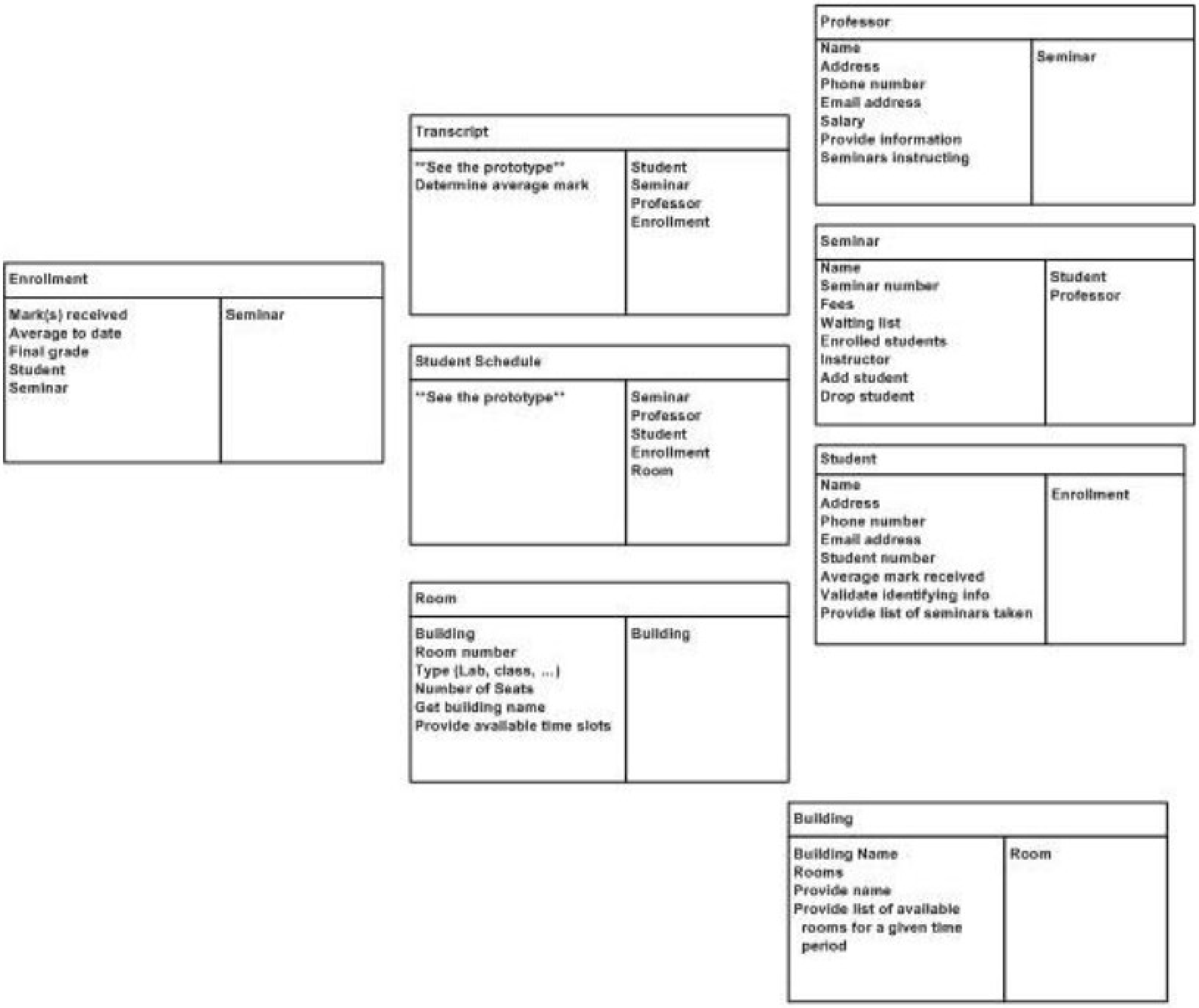
Here's a breakdown of each component of the CRC Model:

1. Class: In the CRC Model, a class represents a blueprint or template for creating objects within a software system. A class encapsulates both data (attributes or properties) and behavior (methods or functions) related to a specific concept or entity. Classes are fundamental building blocks in object-oriented programming.

2. Responsibility: Responsibility refers to the tasks, functions, or behaviors that a class is responsible for performing. Each class in the CRC Model is associated with a list of responsibilities that define what it does within the system. Responsibilities help clarify the purpose and role of each class.

3. Collaboration: Collaboration in the CRC Model defines how classes interact and cooperate with each other to achieve specific functionality or goals. It identifies the relationships and interactions between classes. Collaboration helps determine which classes need to work together to fulfill their responsibilities.

A Class-Responsibility-Collaboration (CRC) model is a simple but effective technique for designing object-oriented software. It uses index cards to represent classes, their responsibilities, and their collaborators.



Professor	
Name Address Phone number Email address Salary Provide information Seminars instructing	Seminar

Transcript	
"See the prototype" Determine average mark	Student Seminar Professor Enrollment

Seminar	
Name Seminar number Fees Waiting list Enrolled students Instructor Add student Drop student	Student Professor

Enrollment	
Mark(s) received Average to date Final grade Student Seminar	Seminar

Student Schedule	
"See the prototype"	Seminar Professor Student Enrollment Room

Student	
Name Address Phone number Email address Student number Average mark received Validate identifying info Provide list of seminars taken	Enrollment

Room	
Building Room number Type (Lab, class, ...) Number of Seats Get building name Provide available time slots	Building

Building	
Building Name Rooms Provide name Provide list of available rooms for a given time period	Room

Each CRC card is divided into three sections:

- **Class:** The name of the class.
- **Responsibilities:** The tasks that the class is responsible for performing.
- **Collaborators:** The other classes that the class needs to interact with in order to perform its responsibilities.
-

To create a CRC model, you start by identifying the main classes in your system. You can use scenarios and use cases to help you identify the classes. Once you have identified the classes, you write the name of each class on a CRC card.

Next, you identify the responsibilities of each class. The responsibilities are the tasks that the class needs to perform in order to fulfill its purpose. You write the responsibilities of each class on the CRC card.

Finally, you identify the collaborators of each class. The collaborators are the other classes that the class needs to interact with in order to perform its responsibilities. You write the names of the collaborators on the CRC card.

Once you have created CRC cards for all of the classes in your system, you can start to arrange the cards into groups. The groups should represent the different subsystems in your system.

CRC models are a valuable tool for designing object-oriented software. They help you to:

- Identify the main classes in your system.
- Understand the responsibilities of each class.

- See how the classes interact with each other.
- Design a more modular and reusable system.

CRC models are also a good way to communicate your design to other team members.

Association and Dependency:

Associations and dependencies are important concepts in software engineering because they help us to design more modular and reusable systems. By understanding the relationships between different elements of a system, we can create systems that are easier to understand, maintain, and extend.

Association is a relationship between two or more elements that indicates that they are related to each other in some way. Associations can be bidirectional or unidirectional. A bidirectional association means that both elements are aware of each other and can interact with each other. A unidirectional association means that only one element is aware of the other element.For example, in a banking system, there might be an association between the "Customer" class and the "Account" class, indicating that customers can be associated with one or more accounts.

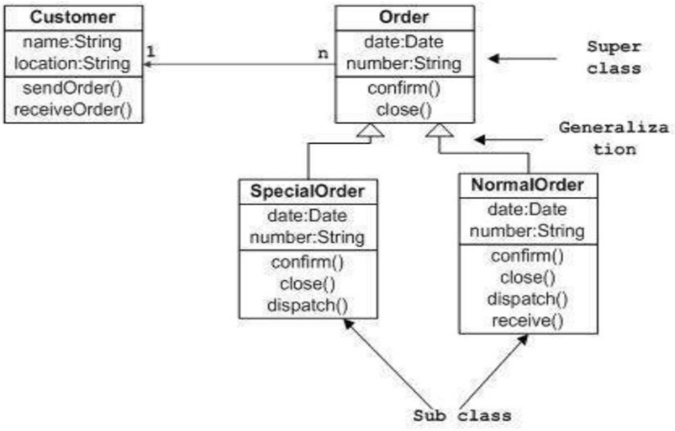
Dependency is a relationship between two elements that indicates that one element depends on the other element in some way. Dependencies are always unidirectional. The element that depends on the other element is called the **dependent** element. The element that is depended on is called the **dependee** element.For example, if a Java class relies on a third-party library to perform certain functions, there is a dependency between the class and the library.

Class Diagram:

Class Diagram is a type of diagram from the Unified Modeling Language (UML) used to visually represent the static structure of a software system or a portion of it. Class diagrams are particularly useful for modeling the classes, objects, attributes, methods, relationships, and constraints that make up the system's design.

Here's a breakdown of the key elements and concepts found in a Class Diagram:

1. **Class:** A class is a blueprint or template for creating objects in an object-oriented programming language. In a Class Diagram, a class is represented as a rectangle divided into three compartments:
    - The top compartment contains the class name.
    - The middle compartment lists the class's attributes (data members or properties) along with their data types.
    - The bottom compartment lists the class's methods (functions or operations) along with their parameters and return types.
2. **Attributes:** Attributes represent the characteristics or properties of a class. They define the data members associated with the class. Each attribute is typically accompanied by its data type.
3. **Methods:** Methods represent the behaviors or functions that the class can perform. These functions encapsulate the operations that can be performed on the class's attributes or properties.
4. **Association:** An association is a bidirectional relationship between two classes. It indicates that objects of the two classes are related to each other in some way.
5. **Dependency:** A dependency is a unidirectional relationship between two classes. It indicates that objects of one class depend on objects of the other class in some way.
6. **Aggregation:** An aggregation is a special type of association that indicates that objects of one class are part of objects of the other class.
7. **Composition:** A composition is a special type of aggregation that indicates that objects of one class are owned by objects of the other class.



Behavior Modeling:

Behavior modeling in software engineering is the process of creating models that describe the behavior of a software system. Behavior models can be used to understand how the system will respond to different inputs and events, and to identify and eliminate potential problems.

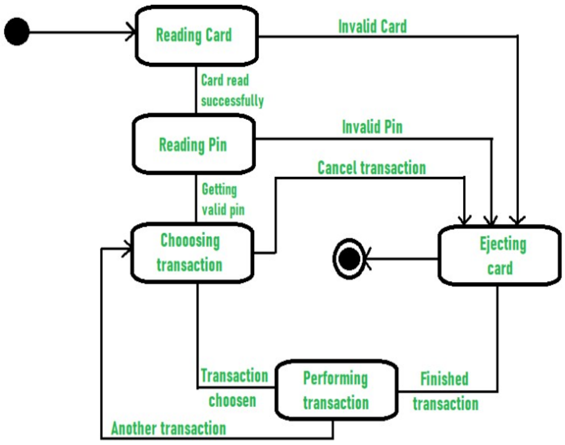
State Diagram:

A state diagram is a type of behavioral diagram used in software engineering to describe the behavior of a system by showing its different states and the transitions between those states. State diagrams are one of the 14 Unified Modeling Language (UML) diagrams.

State diagrams are used to model the dynamic behavior of a system, which means that they show how the system changes its state over time in response to different events. State diagrams are often used in conjunction with other UML diagrams, such as class diagrams and sequence diagrams, to provide a complete picture of how a system works.

Initial state: It defines the initial state (beginning) of a system, and it is represented by a black filled circle.
Final state: It represents the final state (end) of a system. It is denoted by a filled circle present within a circle.
Decision box: It is of diamond shape that represents the decisions to be made on the basis of an evaluated guard.
Transition: A change of control from one state to another due to the occurrence of some event is termed as a transition. It is represented by an arrow labeled with an event due to which the change has ensued.
State box: It depicts the conditions or circumstances of a particular object of a class at a specific point of time. A rectangle with round corners is used to represent the state box.

Example of a State Machine Diagram

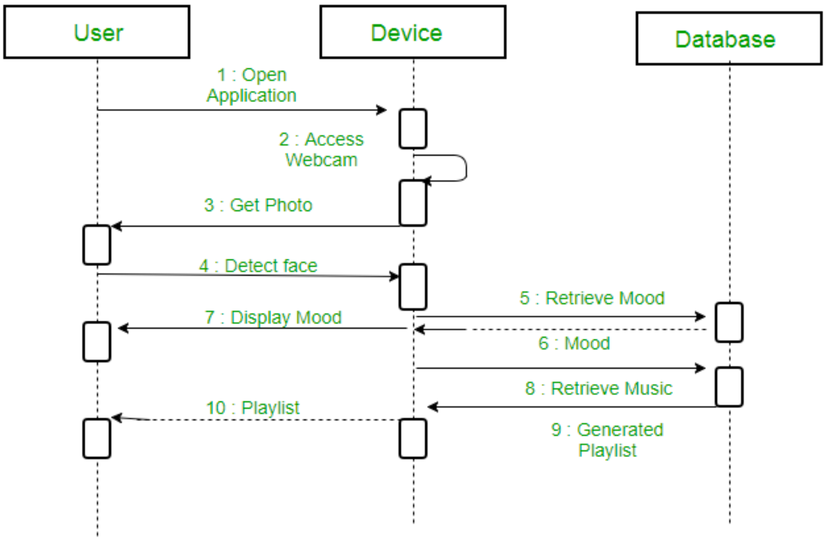


State Transition Diagram for ATM System

Sequence Diagram:

A sequence diagram is a type of behavioral diagram used in software engineering to describe the behavior of a system by showing the interactions between different objects in the system over time. Sequence diagrams are one of the 14 Unified Modeling Language (UML) diagrams.

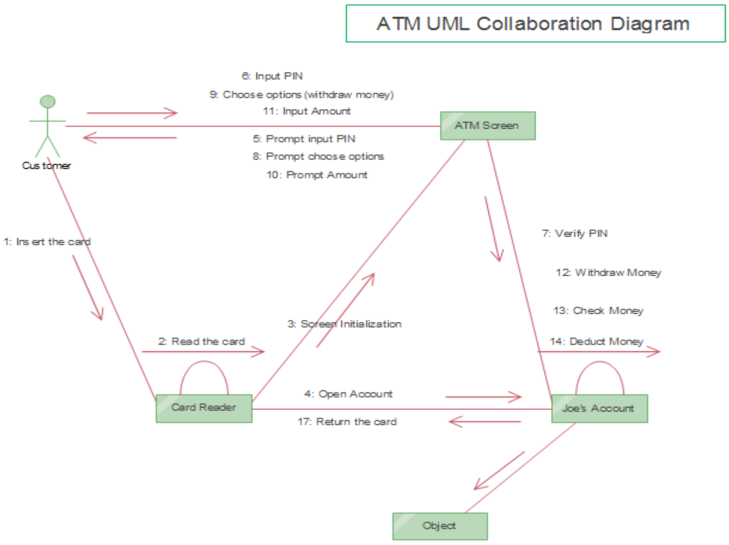
Sequence diagrams are used to model the dynamic behavior of a system, which means that they show how the system changes its state over time in response to different events. Sequence diagrams are often used in conjunction with other UML diagrams, such as class diagrams and state diagrams, to provide a complete picture of how a system works.

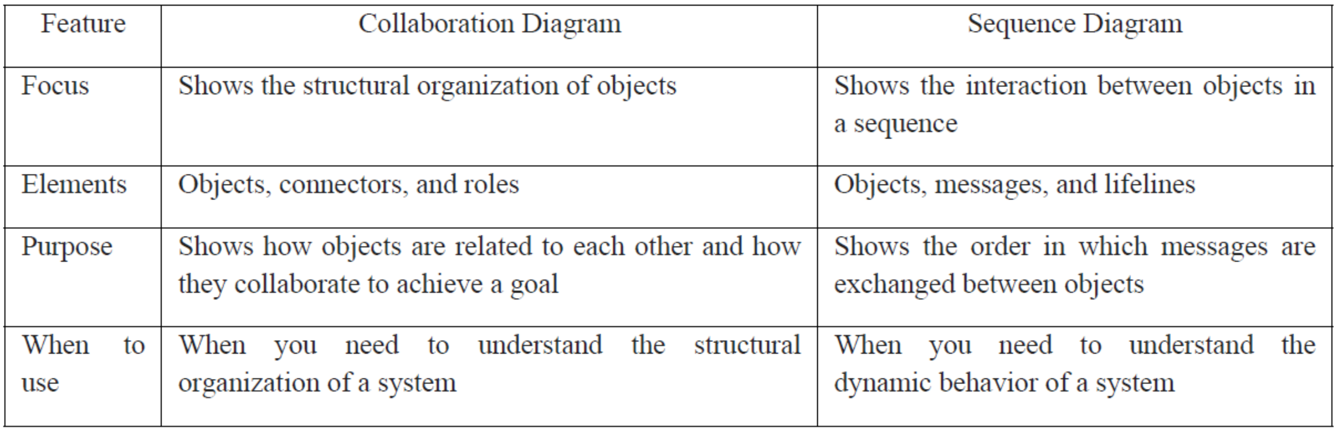


Collaboration Diagram:

A collaboration diagram, also known as a communication diagram, is a type of behavioral diagram used in software engineering to describe the interactions between different objects in a system by showing how they collaborate to perform a specific task or use case. Collaboration diagrams are one of the 14 Unified Modeling Language (UML) diagrams.

Collaboration diagrams are similar to sequence diagrams, but they focus on the relationships between objects rather than the sequence of interactions. Collaboration diagrams are often used in conjunction with sequence diagrams to provide a more complete picture of how a system works.





Feature	Collaboration Diagram	Sequence Diagram
Focus	Shows the structural organization of objects	Shows the interaction between objects in a sequence
Elements	Objects, connectors, and roles	Objects, messages, and lifelines
Purpose	Shows how objects are related to each other and how they collaborate to achieve a goal	Shows the order in which messages are exchanged between objects
When to use	When you need to understand the structural organization of a system	When you need to understand the dynamic behavior of a system

Data Modeling:

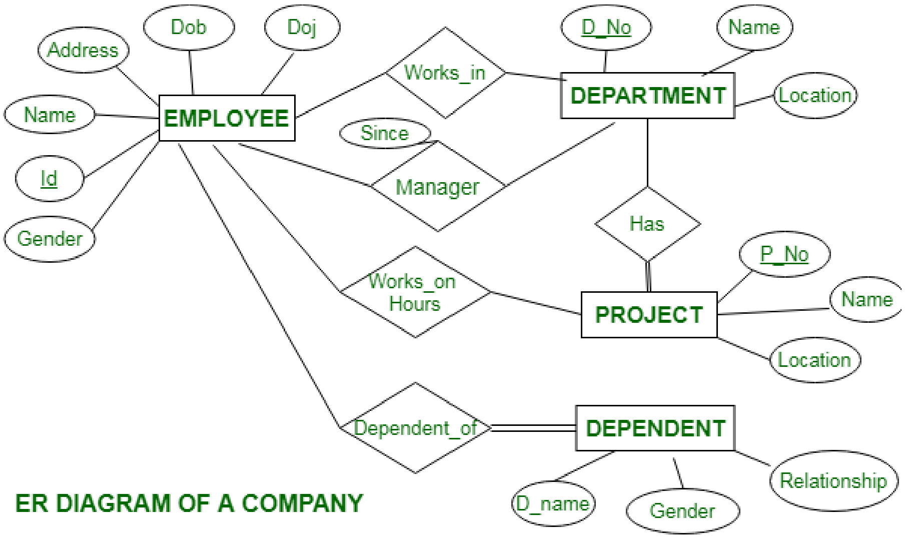
Data modeling is the process of creating a visual representation of the data in a system. It is used to understand, design, and implement databases and other data-driven systems. Data models are used to communicate the structure of the data to stakeholders and to ensure that the data is stored and managed in a way that is efficient and effective.

There are a number of different data modeling techniques, but the most common is entity-relationship modeling (ERM). ERM uses entities, relationships, and attributes to represent the data. Entities are the main things that the system stores data about, such as customers, products, and orders. Relationships are the connections between entities, such as the relationship between a customer and an order. Attributes are the properties of entities, such as a customer's name or address.

E-R Diagram:

An entity relationship (ER) diagram is a type of data model that is used to represent the entities and relationships between entities in a system. It is a graphical way of representing the data that will be stored in a database. ER diagrams are used in software engineering to model the data requirements of a system. They can be used to document the data that will be stored in the system, to visualize the relationships between the different data elements, and to communicate the data requirements to stakeholders.

Here are some of the components of an ER diagram:
Entities: Entities represent the realworld objects that are stored in the database. For example, in an employee database, an entity might be "Employee."
Attributes: Attributes represent the properties of entities. For example, the "Employee" entity might have the attributes "Name," "Address," and "Salary."
Relationships: Relationships represent the connections between entities. For example, the "Employee" entity might be related to the "Department" entity by a "works in" relationship.
Cardinality: Cardinality specifies the number of entities that can be related to each other. For example, the "works in" relationship might be onetoone, onetomany, or manytomany.
An example of an ER diagram for a simple employee database:



ER DIAGRAM OF A COMPANY

The diagram shows the following entities:
- Employee: An employee of the company.
- Department: A department in the company.

The diagram also shows the following relationships:
- Works in: An employee works in a department.
- Manages: An employee manages other employees.

The cardinality of the "Works in" relationship is onetomany. This means that each employee can work in only one department, but a department can have many employees.

The cardinality of the "Manages" relationship is manytomany. This means that an employee can manage many other employees, and an employee can be managed by many other employees.

ER diagrams can be used to model a wide variety of systems, from simple to complex. They are a valuable tool for software engineers, as they can help to ensure that the data requirements of a system are met.

Mapping E-R diagram to Relational Model.

Mapping an Entity-Relationship (E-R) diagram to a relational model involves translating the entities, attributes, and relationships represented in the E-R diagram into tables and columns in a relational database. Here are the key steps involved in this process:

1. Identify Entities: Begin by identifying the entities in your E-R diagram. These are the main objects or concepts you are modeling in your database. Each entity will become a table in the relational database.

2. Identify Attributes: For each entity, identify its attributes. Attributes are the properties or characteristics of the entity. Each attribute will become a column in the corresponding table. Ensure that attributes are atomic and have well-defined data types.

3. Determine Primary Keys: For each entity, identify a primary key attribute. The primary key uniquely identifies each row in the table and is used for indexing and enforcing data integrity constraints.

4. Identify Relationships: Determine the relationships between entities. Relationships can be one-to-one, one-to-many, or many-to-many. Represent these relationships in the E-R diagram using appropriate symbols.

5. Create Tables: For each entity identified in step 1, create a table in the relational model. The table should have columns for each attribute identified in step 2. Include the primary key attribute as well.

6. Establish Foreign Keys: If there are relationships between entities, you'll need to establish foreign keys in the tables. A foreign key is a column in one table that references the primary key in another table, creating a link between the two tables. This enforces referential integrity.

7. Normalize Tables: Ensure that your tables are in at least the first normal form (1NF), which means that each column contains atomic values, and there are no repeating groups. You can further normalize the tables to higher normal forms if necessary to eliminate data redundancy.

8. Define Constraints: Specify any constraints or rules for the tables, such as unique constraints, check constraints, and default values, to maintain data integrity.

9. Map Many-to-Many Relationships: For many-to-many relationships, create an associative table (also known as a junction or linking table) that includes foreign keys from both related entities. This table resolves the many-to-many relationship into two one-to-many relationships.

10. Complete the Relational Model: Continue this process for all entities and relationships in your E-R diagram until you have a complete relational model with tables, columns, primary keys, foreign keys, and constraints.

11. Document the Model: It's essential to document the relational model, including the table structures, data types, constraints, and relationships. This documentation serves as a reference for database developers and users.

12. Implement the Database: Using a database management system (DBMS), create the tables and enforce the constraints defined in your relational model.

Mapping an E-R diagram to a relational model is a crucial step in database design, as it defines the structure and organization of the database to store and manage data effectively. The accuracy and completeness of this mapping are essential for the proper functioning of the database system.