# Chapter Two

## 2.1 High-Level Sequence Diagram

* A high-level sequence diagram shows **the order of interactions** between users (actors) and system components.
* It focuses on the **main functional flow** of each use case such as login, logout, account creation, and employee management.
* Clarifies how **Admin, Employee, and Manager** interact with the system.
* Shows the **flow of messages** between UI, controller, and database.
* Helps with system **analysis and design before coding**.
* Supports better **team communication and documentation**.

## 2.2 Components of High-Level Sequence Diagram

**1. Actors**

* Represent external users who interact with the system.
* In our project, we used three actors:
  + **Admin** – manages employee records and manage attendance.
  + **Employee** – uses system features like login and attendance.
  + **Manager** – manage reports and approvals.

**2. Lifelines**

Lifelines show the roles or components involved in the interactions. We used:

* **Actor Lifeline** – Represents Admin, Employee, and Manager.
* **Boundary Lifeline** – UI components (e.g., LoginPage, Dashboard).
* **Controller Lifeline** – Controls the business logic (e.g., AuthController, EmployeeController).
* **Entity Lifeline** – Business objects like Employee, LeaveRequest.
* **Database Lifeline** – Represents data storage (e.g., UserDB, EmployeeDB).

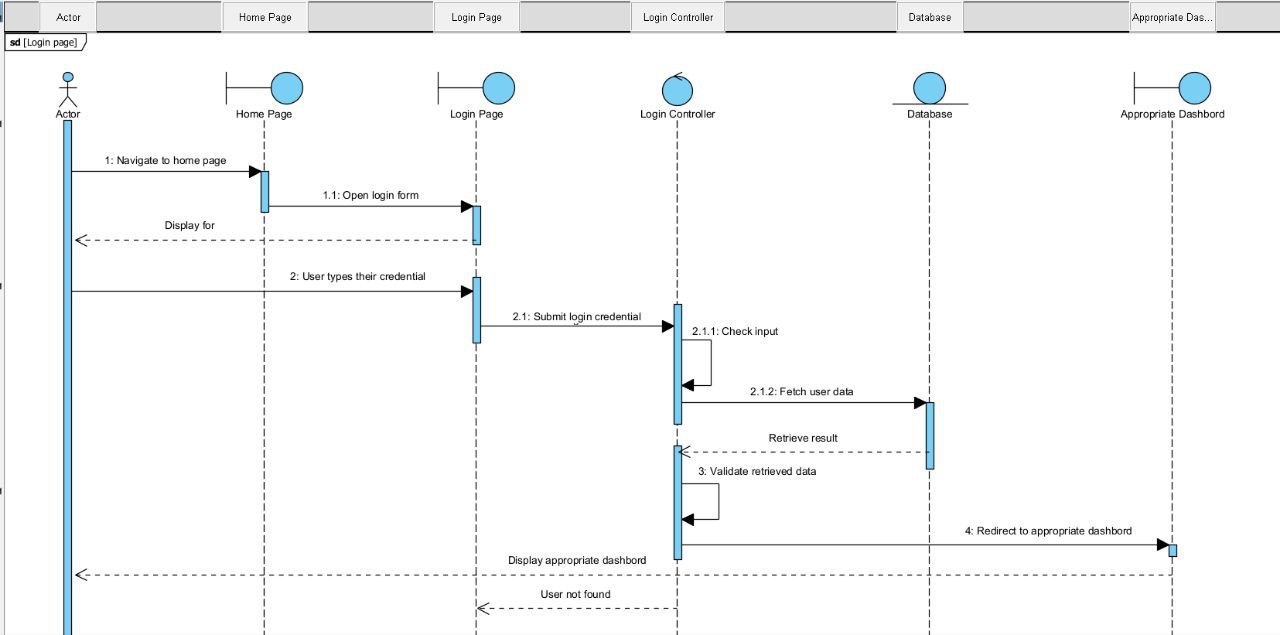
**3. Messages**

* **Solid Line (Synchronous Message):**  
  Used to show direct method calls (e.g., submitLogin()).
* **Dashed Line (Return Message):**  
  Represents a response or result (e.g., loginSuccess).

## 2.3 Example of High-Level Sequence Diagram

We created high-level sequence diagrams for the following use cases:

1. **Login**

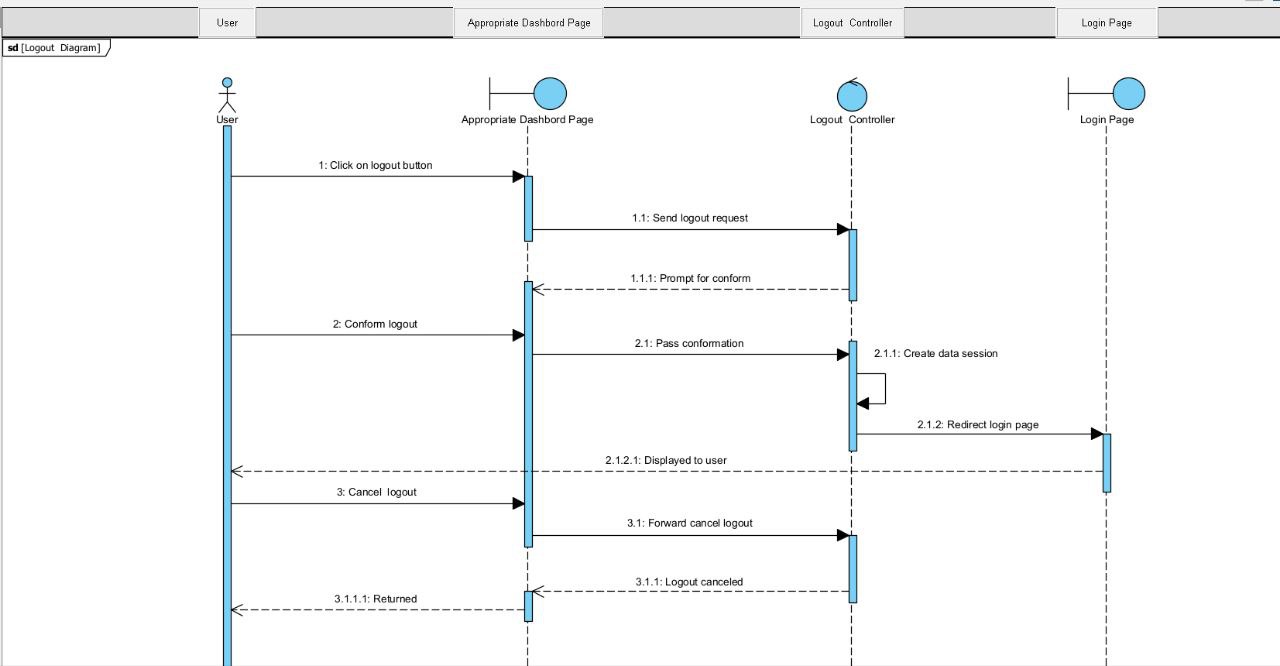


Actor: Employee/Admin/Manager

Flow:

* User navigates to the **homepage**.
* Homepage displays the **login form**.
* User enters credentials on the **login page**.
* Credentials are submitted to the **login controller**.
* Login controller checks input and fetches user data from the **database**.
* Database returns the user record.
* Login controller validates the data.
* If valid, user is redirected to the appropriate **dashboard** based on their role.

1. **Logout**

****

**Actor**: logged-in user

**Flow:**

**User** clicks the logout option on the **dashboard**.

A **confirmation prompt** appears asking the user to confirm or cancel the logout.

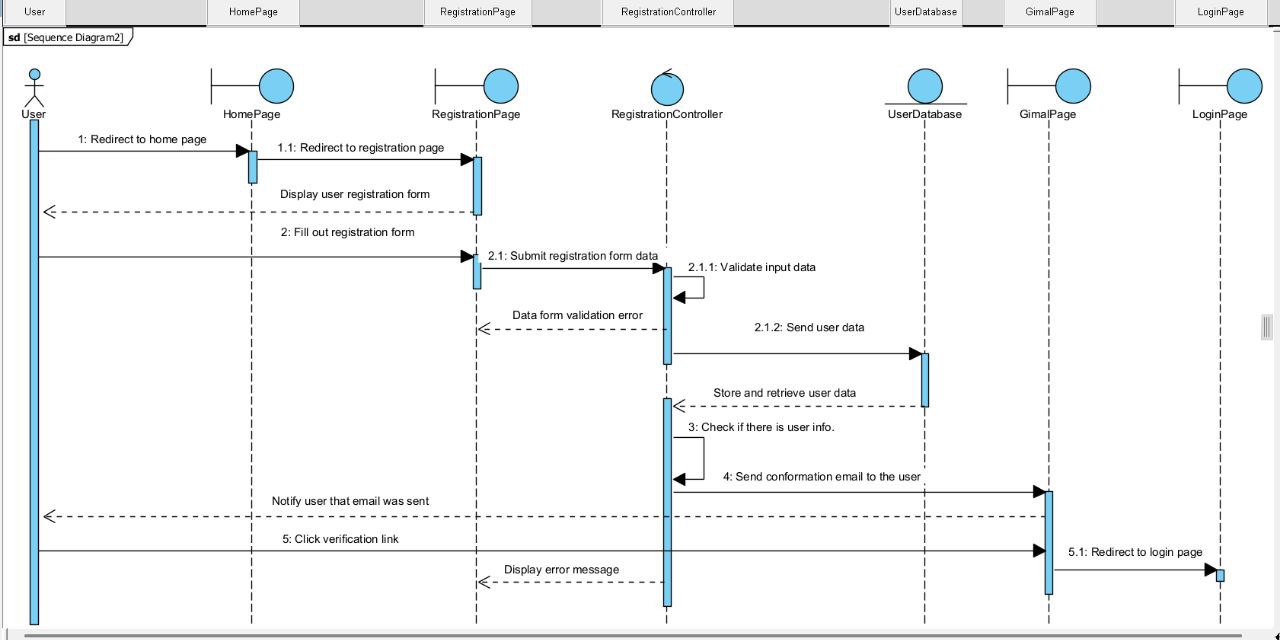
**If the user cancels:**

* The system **closes the prompt** and keeps the user on the **dashboard**.

**If the user confirms:**

* Dashboard sends a logout request to the logout controller.
* Logout controller processes the request (e.g., ends the session).
* User is redirected to the login page.
* Login page is displayed, completing the logout process.

1. **Create Account**

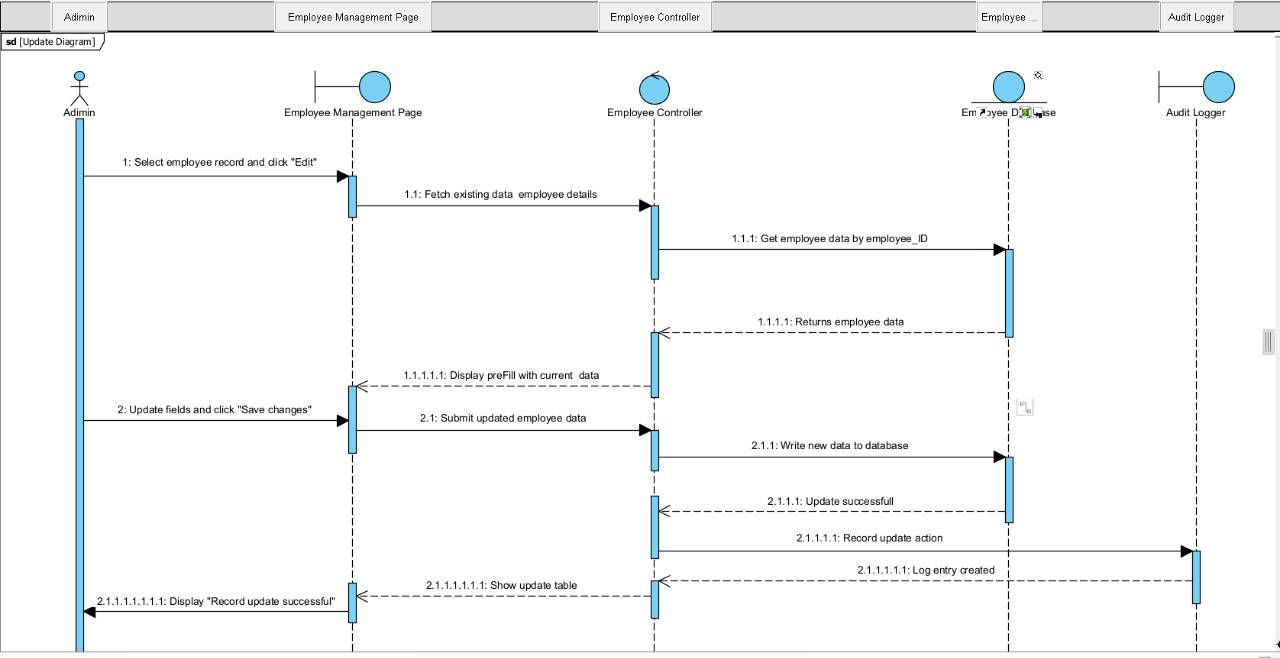


**Actor**: Users

**Flow:**

* **User** navigates to the **homepage**.
* From the homepage, the user selects the **"Create Account"** or **"Register"** option.
* The system opens the **registration page**.
* **User** fills in the required details (e.g., name, email, password) and submits the form.
* The **registration page** sends the data to the **registration controller**.
* **Registration controller** validates the input and sends a request to store the data in the **user database**.
* **User database** saves the new user record and returns a success response.
* **Registration controller** sends a **verification email** via the **Gmail page** or email service.
* User checks their email (Gmail) and clicks the **verification link**.
* After verification, the system redirects the user to the **login page**.
* **Login page** is displayed, allowing the user to log in with the newly created account.

1. **Update Employee Record**

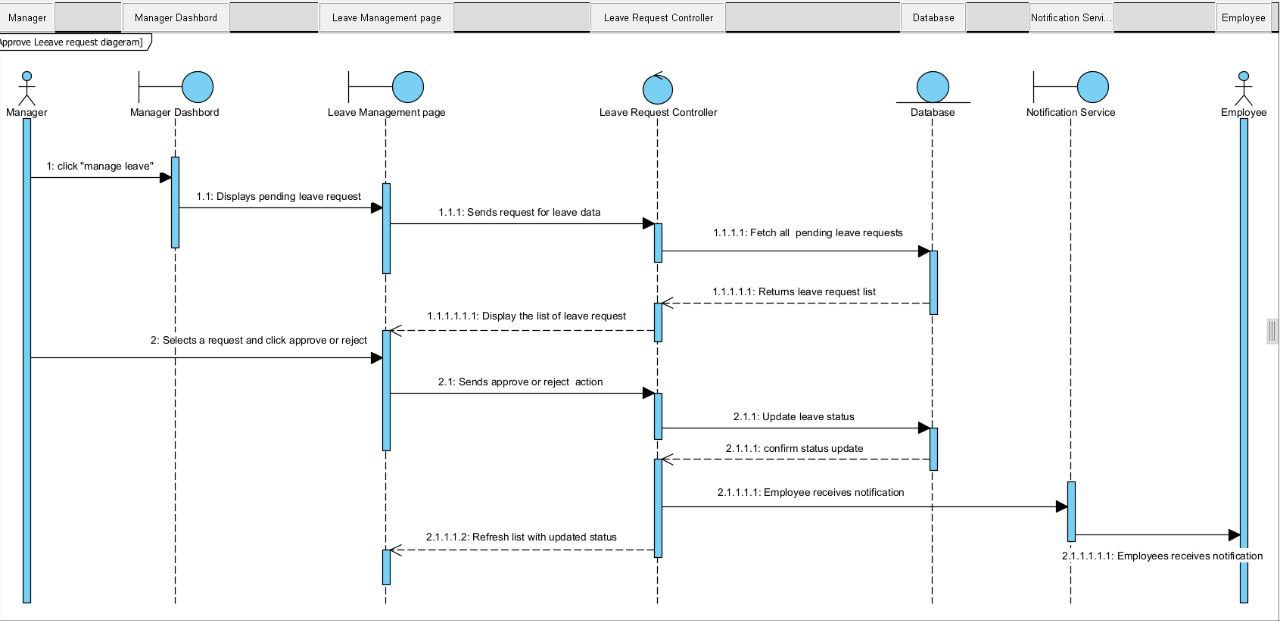
****

**Actor:** Admin

**Flow:**

* **Admin** opens the **employee management page**.
* Selects an employee and clicks **edit**.
* Page displays editable employee form.
* Admin updates detail and submits the form.
* **Employee controller** receives and validates data.
* Sends update request to **employee database**.
* **Database** updates the record and confirms success.
* **Audit logger** records the update action.
* Page shows success message to the admin.

1. **Approve Leave Request**

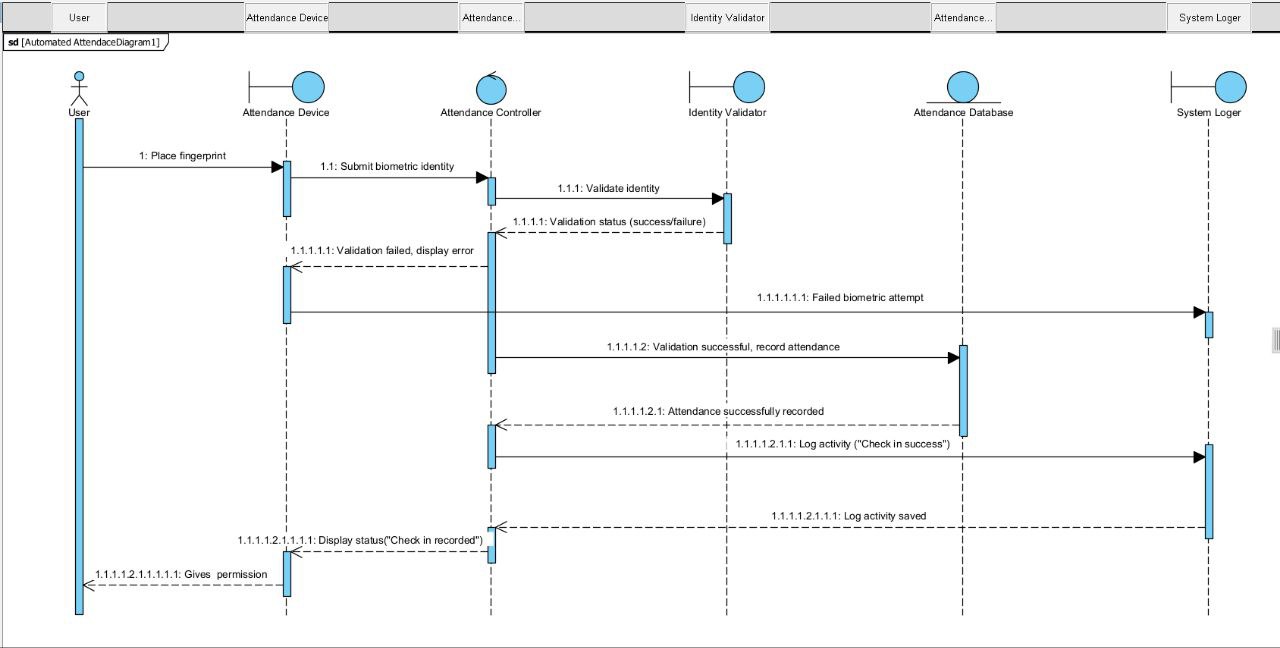


**Actor:** Manager

**Flow:**

* **Manager** logs in and accesses the **manager dashboard**.
* Navigates to the **leave management dashboard**.
* Selects a pending leave request and clicks **approve**.
* **Leave request controller** receives the approval action.
* Controller updates the request status in the **database**.
* **Notification service** sends approval notification to the respective **employee**.
* Page shows confirmation message to the **manager**.

1. **Biometric Attendance**

****

**Actor:** User

**Flow:**

* **User** places fingerprint on the **attendance device**.
* **Attendance device** submits biometric data to the **attendance controller**.
* Controller forwards data to the **identity validator**.
* **Identity validator** checks the fingerprint and returns **validation status** (success or failure).

**If validation fails:**

* Controller instructs the **attendance device** to display an error.
* Failure is logged by the **system logger**.

**If validation succeeds:**

* **Attendance controller** records the attendance in the **attendance database**.
* Database returns confirmation of successful record.
* Controller logs the activity via the **system logger**.
* Logger confirms log entry saved.
* **Attendance device** displays success or grants access accordingly.

## 2.4 Tools and Steps to Draw High-Level Sequence Diagram

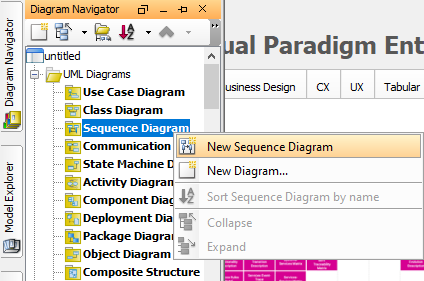
**Tool Used**

* **Visual Paradigm** – A UML modeling tool used to draw our sequence diagrams.

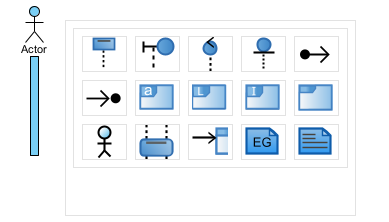
**Steps Followed**

1. Open Visual Paradigm and create a new UML project.
2. Select **Sequence Diagram** from the diagram types.

Right click the **Sequence Diagram -> New Sequence Diagram.**

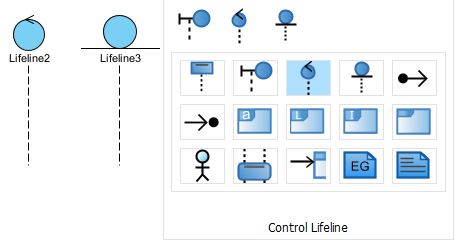


1. Add **Actors** (Admin, Employee, Manager). **Double clicks** the blank space then **Choose** the **Actor**.

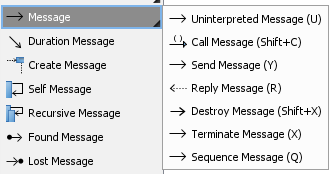


1. Add **lifelines**:

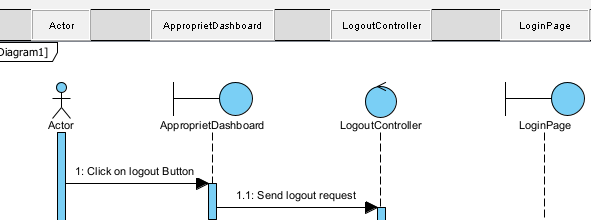
* UI (Boundary)
* Controller
* Entity
* Database



1. Draw **solid arrows** for method calls and **dashed arrows** for responses.



1. Label messages to show clear interaction flow.



1. Save and export diagrams for documentation.

# Chapter 4

## 3.1 Low-Level (Detail) Design (Class Design)

The low-level design focuses on how the internal parts of the system work through class design, using object-oriented programming (OOP) principles such as inheritance, encapsulation, and association. In this project, each class represents a real-world entity and is responsible for a specific part of the system's functionality.

Main Classes and Their Roles

* **User**:  
  Acts as a base class containing shared attributes such as:
  + User id, full Name, password, email, role, address, phone…  
    These attributes are common to all users in the system.
* **Admin**, **Manager**, and **Employee**:  
  These classes **inherit** from the User class. Each has additional functionalities:
  + **Admin** can update employee records, manage policies, manage attendance, manage leaves, manage departments, calculate salaries and view reports.
  + **Manager** can approve leave requests, view reports, create branch, generate reports, assign branch manager and manage policy.
  + **Employee** can request leave, mark attendance, view personal information, view reports and update profile.
* **Policy**:  
  Contains company rules or policies added or updated by the Admin and Manager.
* **Report**:  
  Stores generated reports such as attendance or leave summaries for managerial review.
* **Branch**:  
  Represents different physical or virtual office locations of the company.
* **Department**:  
  Refers to various departments like HR, IT, Sales, etc., to which employees belong.
* **Leave Request**:  
  Handles the process of applying, approving, or rejecting employee leave.
* **Attendance Request**:  
  Records attendance data submitted by employees through biometric or manual check-in.
* **Biometric Device**:  
  Accepts fingerprint data, sends it to the system, and records it after verification.

**Design Benefits**

* Code reusability through inheritance.
* Clear separation of responsibilities across classes.
* Easy to maintain and scale in the future.

## 3.2 Components of Class Diagram

A class diagram is a UML (Unified Modeling Language) tool that visually represents the structure of the system. It shows each class, its attributes and methods, and the relationships between classes.

Key Components

* **Class Box**  
  A rectangle with three parts
  1. **Class Name** (e.g., Employee)
  2. **Attributes** – e.g., +full\_name: String, +phone: String
  3. **Methods** – e.g., +requestLeave (): void, +markAttendance (): void
* **Attributes**  
  Define the data held by a class (e.g., department\_id, hire\_date, status).
* **Methods (Operations)**  
  Functions that can be performed by the class (e.g., approveLeave (), logAttendance ()).
* **Relationships Between Classes**
  + **Inheritance (Generalization)**  
    Represented with a hollow arrow pointing from child to parent class. Which shows the inheritance of classes from the parent class.   
    Example: Admin → User

In this project the Admin, Employee and Manager classes inherit from the User class.

* + **Association**:  
    A line between two classes that communicate or use each other.  
    There are a lot of Association relationship in our project (Manager – Branch, Manager – Report, Admin – Report, Admin – Policy, Employee – Report, Employee – LeaveRequest, and so on).
  + **Aggregation / Composition**
    - **Aggregation** (hollow diamond): A whole-part relationship where parts can exist independently.

**Example:** Employee class with Admin and Department classes, Admin class with Department class.

* + - **Composition** (filled diamond): A strong relationship where parts cannot exist without the whole.

**Example:** Branch class with Department class.

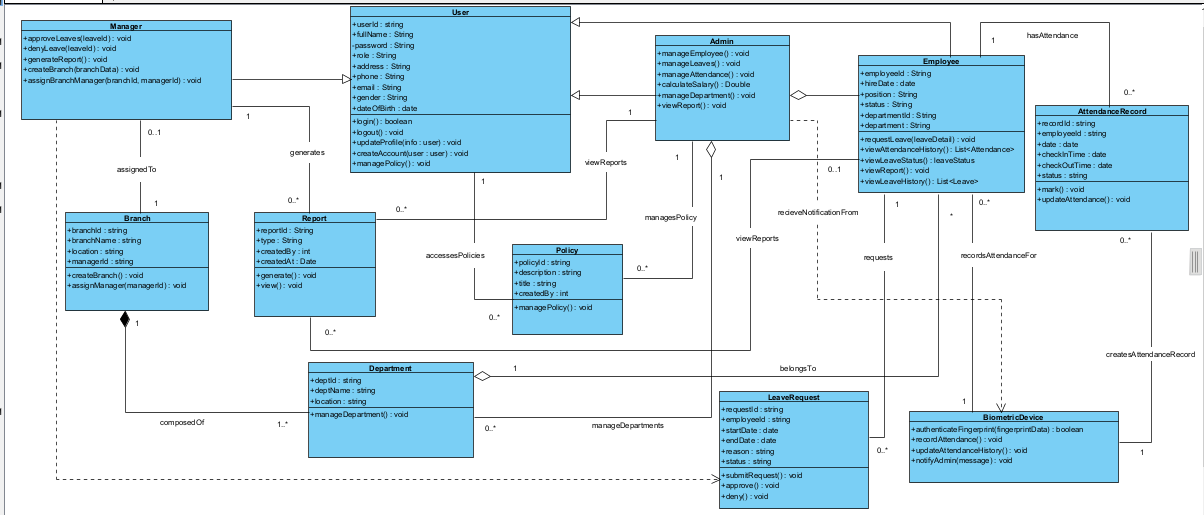
* **Dependency**

Shows that one class relies on another class to work, but it is not tightly connected like association or inheritance.

**Example:** Manager class with LeaveRequest class, Admin class with BiometricDevice class.

* **Multiplicity**:
  1. Shown on association lines to indicate quantity.  
     Example:
     + Admin to Policy (1 – 0.. \*)
     + Admin to Department (1 – 0..\*)
     + Employee to Leave Request (1 – 0..\*)
     + Employee to Biometric Device (0.. \* - 1)
     + Attendance Record to Biometric Device (0..\* - 1)
     + Manager to Branch (0..1 - 1)
     + Manager to Report (1 - 0.. \*) and so on.
* **Access Modifiers**:
  + + Public: Accessible from outside the class.
  + - Private: Accessible only inside the class.

## 3.3 Example of Class Diagram



## 3.4 Tools and Steps to Draw Low-Level Diagram

**Tool Used**

* Visual Paradigm: A UML modeling tool used to draw accurate and clear class diagrams.

**Steps to Draw the Low-Level (Class) Diagram**

* Open Visual Paradigm and start a new project.
* Select Class Diagram from UML diagrams.
* Drag and drop Class elements onto the canvas.
* Define:
* Class names
* Attributes with their data types
* Methods with parameters and return types
* Use generalization arrows to show inheritance (e.g., Admin → User).
* Add associations and multiplicities between classes.
* Apply composition or aggregation where needed (e.g., Department ↔ Employee).
* Organize layout for clarity and label relationships properly.
* Save or export the diagram for documentation.