# UNIT II

## Abstraction: -

When you consider a modular solution to any problem, many levels of abstraction can be posed.

At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment (e.g., a user story).

At lower levels of abstraction, a more detailed description of the solution is provided. Problem-oriented terminology is coupled with implementation-oriented terminology to state a solution (e.g., use case).

Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented (e.g., pseudocode).

Different levels of abstraction -

1. Procedural abstraction

2. Data abstraction

A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

A data abstraction is a named collection of data that describes a data object.

## Modularity: -

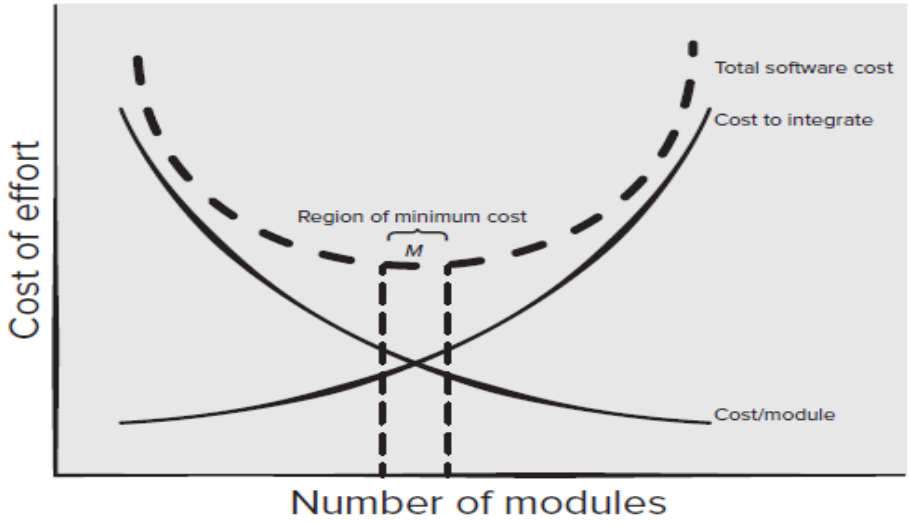
Modularity is the action of separation of concerns. Software is divided into components, sometimes called modules, that are integrated to satisfy problem requirements. Modularity allows a program to be manageable.

Monolithic software (i.e., a large program composed of a single module) contains number of control paths, reference, number of variables, and overall complexity would make understanding impossible.

Software is divided into modules to make understanding easier and reduce the cost required to build the software.

Using too few modules or too many modules should be avoided to reduce the cost required to build the software.

Modularity and software cost




## Cohesion: -

Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to each other and to the class or component itself.

Several types of cohesion –

1. Functional: This level of cohesion occurs when a module performs only one computation and then returns a result.

2. Layer: This type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.

3. Communicational: All operations that access the same data are defined within one class.

Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.

## Coupling: -

Coupling is a qualitative measure of the degree to which classes are connected to one another. As classes (and components) become more interdependent, coupling increases.

An important objective in component-level design is to keep coupling as low as possible.

Several types of coupling –

1. Content coupling : It occurs when one component modifies data that is internal to another component. This violates information hiding.

2. Control coupling occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B. The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result.

3. External coupling occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

# Software Architecture: -

Software architecture suggest to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system"

Architecture is the structure or organization of program components (modules), the ways in which these components interact, and the structure of data that are used by the components.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enables a software engineer to reuse design level concepts.

A set of properties that should be specified as part of an architectural design. Structural properties define "the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another." Extra-functional properties address "how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics (e.g., nonfunctional system requirements)." Families of related systems "draw upon repeatable patterns that are commonly encountered in the design of families of similar systems."

# Architectural Styles: -

## (1) Layered Architecture: -

**Description:** The system is organized into layers, each with specific responsibilities. Higher layers depend on the services provided by lower layers.

**Example:** OSI model in networking, where each layer has a distinct role (e.g., application, transport, network).

**Use Case**: Systems where separation of concerns is essential, like enterprise applications.

## (2) Client-Server Architecture: -

**Description:** The system is divided into two main components: clients (requesting services) and servers (providing services).

**Example:** Web applications where a browser (client) requests pages from a web server.

**Use Case:** Distributed systems requiring centralized data management.

Client-Server Architecture is a fundamental model used in distributed systems where multiple clients request and receive services from a centralized server. This architecture is widely used in various applications, from simple web services to complex enterprise systems.

### Key Concepts of Client-Server Architecture:

### 1. Client –

Role: The client is the service requester in the system. It initiates communication with the server to request resources, data, or services.

### Characteristics:

1. User Interface: Clients handle the user interface (UI), displaying data and receiving input from users.
2. Limited Resources: Clients are usually less resource-intensive, relying on servers for processing power and data storage.
3. Multiple Clients: Many clients may connect to a single server simultaneously.

### 2. Server –

Role: The server is the service provider, processing client requests and returning the necessary data or services.

### Characteristics:

1. Data Management: Servers often manage databases, files, or computational tasks requested by clients.
2. High Performance: Servers are typically powerful machines capable of handling multiple requests from different clients simultaneously.
3. Centralized Control: The server controls access to resources, ensuring that clients only receive the data or services they are authorized to use.

### How Client-Server Architecture Works:

Request-Response Model –

Step 1: The client sends a request to the server, typically over a network.

Step 2: The server processes the request, performing any necessary operations, such as querying a database or performing calculations. Step 3: The server sends a response back to the client with the requested data or an acknowledgment of the completed operation. Step 4: The client receives the response and displays it to the user or performs further actions based on the data received.

## Microservices Architecture: -

Description: The system is composed of small, independent services that communicate over a network. Each service is self-contained and focused on a specific business capability.

Example: E-commerce platforms where services like payment, inventory, and user management are separate.

Use Case: Large-scale applications needing agility, scalability, and independent deployment.

Microservices architecture is a design approach in software development where an application is composed of small, independent services that work together to fulfill the overall functionality of the system. Each service is self-contained and implements a specific business capability. Here's a detailed explanation of microservices architecture:

### 1. Core Concepts of Microservices

**Independence:** Each microservice is a standalone application that can be developed, deployed, and scaled independently. This means that teams can work on different services simultaneously without affecting other parts of the system.

Single Responsibility Principle: Each microservice is responsible for a specific business function or domain, such as user authentication, order processing, or payment management. This aligns with the principle of "Do one thing and do it well."

Inter-Service Communication: Microservices communicate with each other through well-defined APIs, typically using HTTP/REST, gRPC, or messaging protocols like AMQP (Advanced Message Queuing Protocol). This communication is often over a network, which allows services to be distributed across different servers or even geographical locations.

Representational State Transfer (REST) is an architectural style for designing networked applications that uses HTTP as a standard protocol to define communications. REST is often used to develop web services and build APIs because it's simple and uses HTTP methods to match the operations they perform.

gRPC is an open-source, high-performance, language-agnostic Remote Procedure Call (RPC) framework that uses HTTP/2 as its transport layer protocol

The Advanced Message Queuing Protocol (AMQP) is an open-standard application layer protocol that enables reliable communication between devices. It's used to pass business messages between applications or organizations

## 2. Key Characteristics of Microservices

### Decentralized Data Management:

Unlike monolithic architectures, where a single database is shared among all components, each microservice typically has its own database. This helps in maintaining data integrity and allows each service to choose the database technology best suited to its needs (e.g., SQL).

### Autonomy:

Microservices can be developed using different programming languages, frameworks, and tools, as long as they adhere to the agreed-upon communication protocols. This flexibility allows development teams to choose the best technology stack for their specific service.

Scalability: Individual microservices can be scaled independently based on demand. For example, a microservice handling user logins can be scaled up during peak hours without needing to scale the entire system.

Fault Isolation: In a microservices architecture, the failure of one service does not necessarily bring down the entire system. This is because services are decoupled, allowing the system to maintain partial functionality even if one service fails.

**Continuous Delivery and Deployment:**

Microservices enable frequent and independent deployment of services. This is facilitated by CI/CD pipelines, which automate the testing, integration, and deployment processes, allowing rapid iteration and updates.

In microservices architecture, a CI/CD (continuous integration and continuous delivery) pipeline is a series of automated steps that speed up the software delivery process:

CI

Regular commits trigger builds, which can include code compilation and artifact creation. CI aims to reduce merge conflicts, speed up code integration, and automatically trigger builds and tests.

CD

Automatically moves builds through the deployment pipeline, making changes available to users quickly. CD can include deploying to a staging environment for testing, and then to production.

# Design principles for AI systems: -

## 1. Modifiability

Modifiability focuses on the ease with which an AI system can be changed or extended.

Modular Architecture:

1. Separation of Concerns: Divide the system into independent modules with clear responsibilities. This allows changes in one area without affecting others.
2. Microservices: Adopt a microservices architecture where each service is self-contained, allowing individual components to be modified without impacting the whole system.

Loose Coupling:

1. Minimized Dependencies: Reduce interdependencies between modules, making it easier to update or replace components without extensive refactoring.

2. Event-Driven Design: Utilize event-driven or message-passing architectures to decouple components, which supports independent modification and evolution.

Abstract Interfaces:

1. API Design: Implement well-defined APIs that abstract underlying complexities, enabling the replacement or enhancement of specific functionalities without affecting other parts of the system.

Extensibility:

1. Plug-In Systems: Design with plug-in or add-on mechanisms that allow new features or capabilities to be integrated without altering the core system.

2. Configurable Settings: Use external configuration files or databases for system parameters, making it easier to modify behavior without changing the code.

Version Control:

Model and Code Versioning: Implement version control for models, data schemas, and code to manage changes systematically and ensure backward compatibility.

## 2. Scalability

Scalability ensures that an AI system can handle increased workloads efficiently.

Horizontal and Vertical Scaling –

Elastic Scalability: Design systems to scale both horizontally (adding more machines) and vertically (upgrading resources on existing machines) to meet demand.

Load Balancing: Implement load balancers to distribute workloads evenly across resources, ensuring optimal performance as the system scales.

Data Partitioning –

Sharding: Use data partitioning techniques such as sharding to distribute data across multiple databases or storage systems, improving performance and scalability.

Distributed Processing: Leverage distributed computing frameworks (e.g., Apache Spark, Hadoop) to process large datasets in parallel, enhancing scalability.

Stateless Design –

Stateless Components: Design services to be stateless, meaning they don't rely on stored state between requests. This allows easy replication and scaling across multiple instances.

**Resource Decoupling –**

**Independent Scaling:** Ensure that compute, storage, and networking resources can be scaled independently to optimize for specific bottlenecks as demand grows.

**Cloud-Native Infrastructure:** Use cloud-native technologies like Kubernetes or serverless computing to automatically scale resources based on demand.

Asynchronous Processing –

Queue-Based Systems: Use message queues or task queues for handling tasks asynchronously, which allows the system to process tasks at scale without being overwhelmed.

## 3. Adaptability

Adaptability allows the AI system to evolve and respond to new requirements, environments, or changes in data.

1. Continuous Learning –

Online Learning: Implement models that can continuously learn and adapt to new data in real-time, keeping the AI system up-to-date with evolving trends.

Automated Model Retraining: Set up pipelines for automatic model retraining as new data becomes available, ensuring the system adapts to changing conditions.

2. Feedback Loops –

Real-Time Monitoring: Use monitoring tools to gather performance metrics and user feedback in real-time, allowing the system to adapt based on observed behavior.

A/B Testing: Regularly conduct A/B testing to compare different versions or models, enabling the selection of the best performing solution dynamically.

3. Configurable Logic –

Rule-Based Systems: Use rule engines that allow business rules or decision logic to be updated dynamically without code changes.

Dynamic Workflows: Design workflows that can be reconfigured based on changing requirements or conditions, allowing the system to adapt without major redevelopment.

4. Generalization –

Transfer Learning: Employ transfer learning techniques that allow models to be adapted to new tasks with minimal retraining.

Domain Adaptation: Design models that can generalize well across different domains or environments, making them more adaptable to varied data inputs.

5. Resilience and Fault Tolerance –

Graceful Degradation: Ensure that the system can degrade gracefully under stress or failure, adapting its functionality to continue operating at reduced capacity.

Redundancy: Implement redundancy in critical components to adapt to failures without interrupting service.

# Software modeling techniques –

## 1. Entity-Relationship Diagram: -

ER Diagram can express the overall logical structure of a database graphically.

ER Diagram consists of following major components:

1. Rectangles: Rectangles represent Entities in the ER Model.

2. Ellipses: Ellipses represent Attributes in the ER Model.

3. Diamond: Diamonds represent Relationships among Entities.

4. Lines: Lines represent attributes to entities and entity sets with other relationship types.

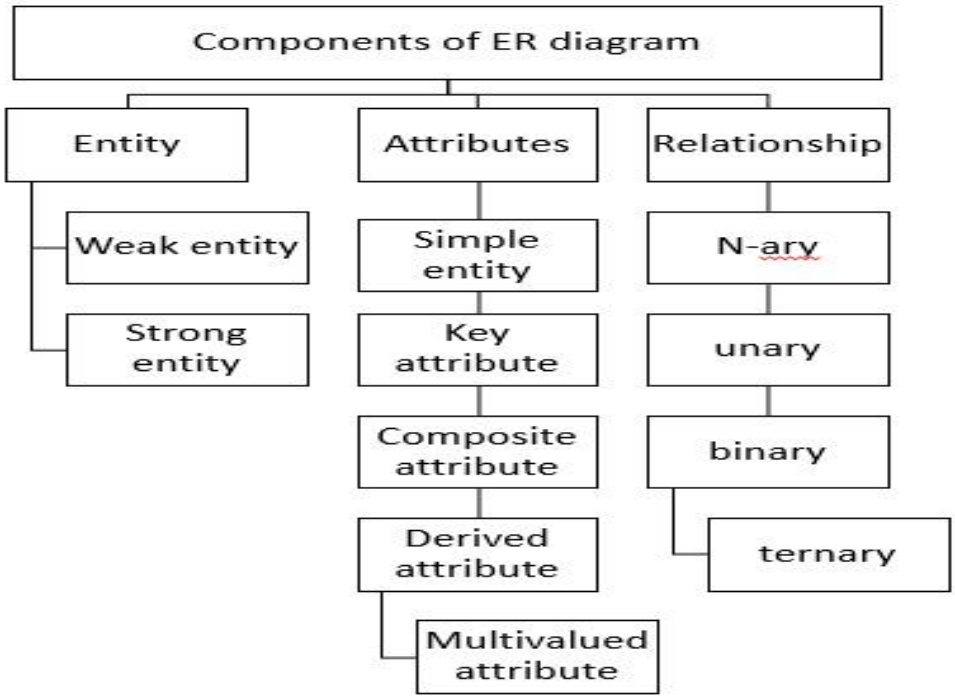
5. Double Lines: It represent total participation.

6. Double Ellipse: Double Ellipses represent Multi-Valued Attributes.

7. Dashed Ellipse: Dashed Ellipse represent Derived attribute.

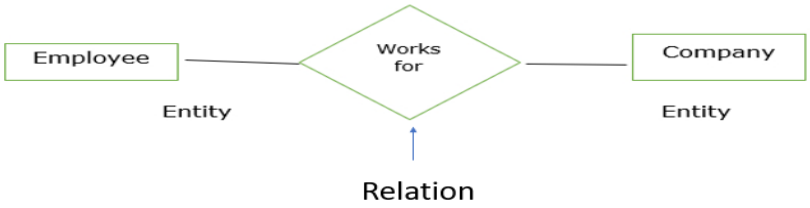
8. Double Rectangle: Double Rectangle represents a Weak Entity.

### Components of ER Diagram



**Entity:**

It may be an object, person, place or event that stores data in a database. In a relationship diagram an entity is represented in rectangle form. For example, entities are Employee, Company.




Strong entity set –

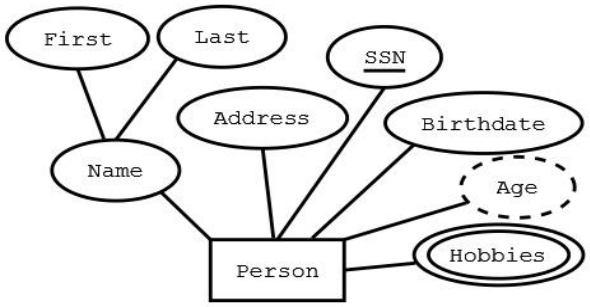
The entity types which consist of key attributes or if there are enough attributes for forming a primary key attribute are called a strong entity set. It is represented by a single rectangle.

Weak entity set –

An entity does not have a primary key attribute and depends on another strong entity via foreign key attribute. It is represented by a double rectangle.

**Attributes:**

It is the name, thing etc. These are the data characteristics of entities or data elements and data fields.




**Degree of Relationship:**

A relationship where a number of different entities set participate is called a degree of a relationship.

It is categorized into the following −

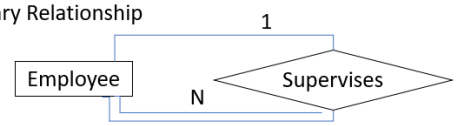
Unary Relationship

Binary Relationship

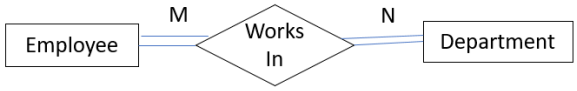
Ternary Relationship

n-ary Relationship

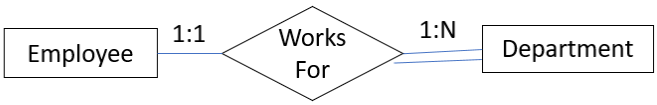
Unary Relationship



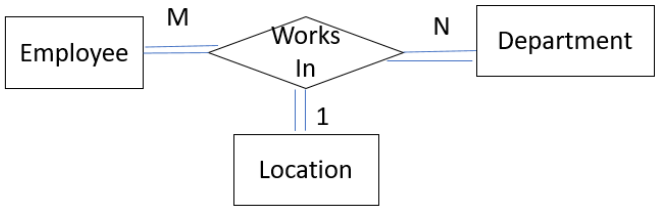

Binary Relationship



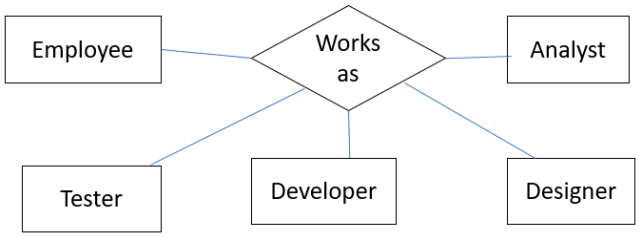

Binary Relationship



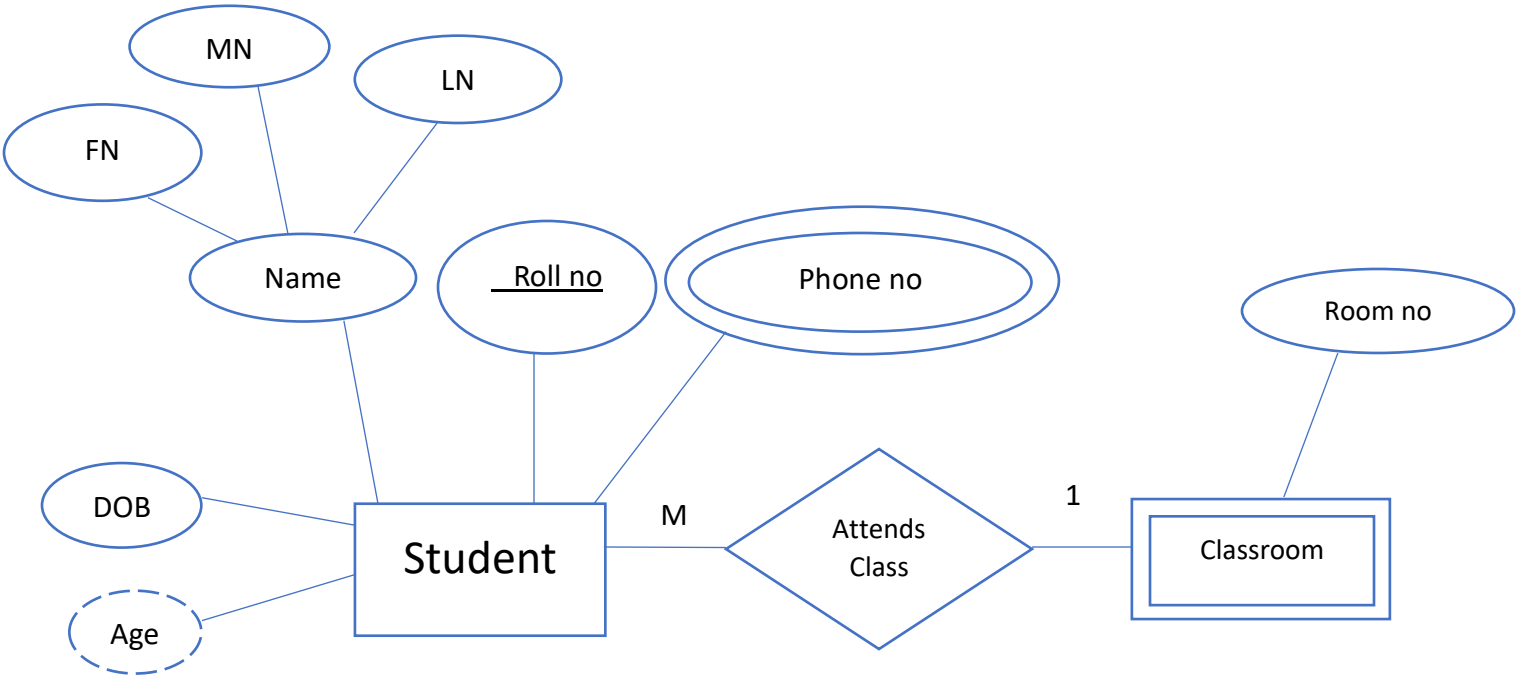

Ternary Relationship




n-ary Relationship



## Converting ER Diagram into structure of a Relation (i.e., Schema of a Relation)




**Schema of Student Entity:**

**Student**

| **Roll no** | **FN** | **MN** | **LN** | **DOB** | **Age** |
|---|---|---|---|---|---|

| **Roll no** | **Phone no** |
|---|---|

Foreign key

**Schema of Classroom Weak Entity:**

**Classroom**

| Roll no | Room no |
|---|---|

Foreign key

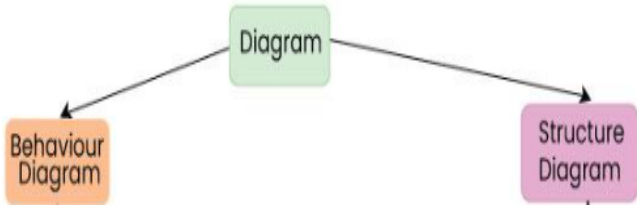
In the above Classroom table 'Room no' is a discriminator. To identify each entry in the weak entity, weak entity is dependent on the primary key of a table (strong entity) on which it is depending along with its own discriminator (i.e., Room no).

## 2. Unified Modeling Language (UML): -

The Unified Modeling Language (UML) is "a standard language for writing software blueprints. UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system".

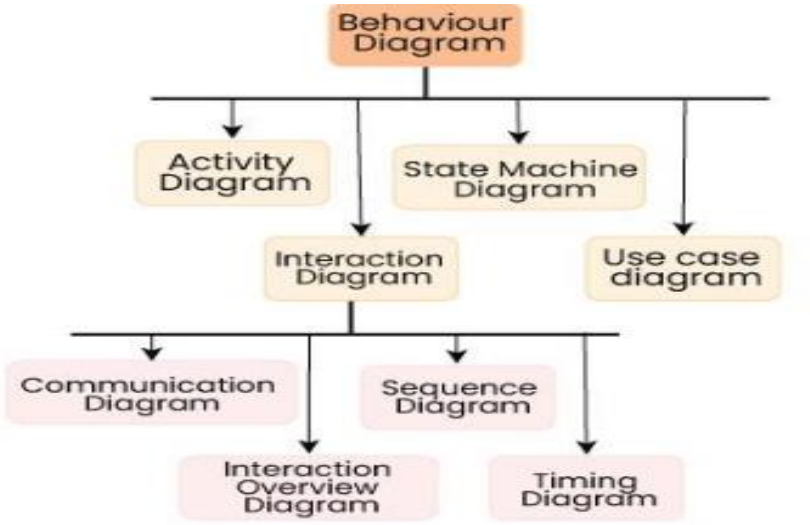
Software architects create UML diagrams to help software developers to build the software.

UML provides different diagrams for use in software modeling.




Behavioral Diagrams:

Behavioral diagrams portray a dynamic view of a system or the behavior of a system, which describes the functioning of the system. It includes use case diagrams, state diagrams, and activity diagrams. It defines the interaction within the system.



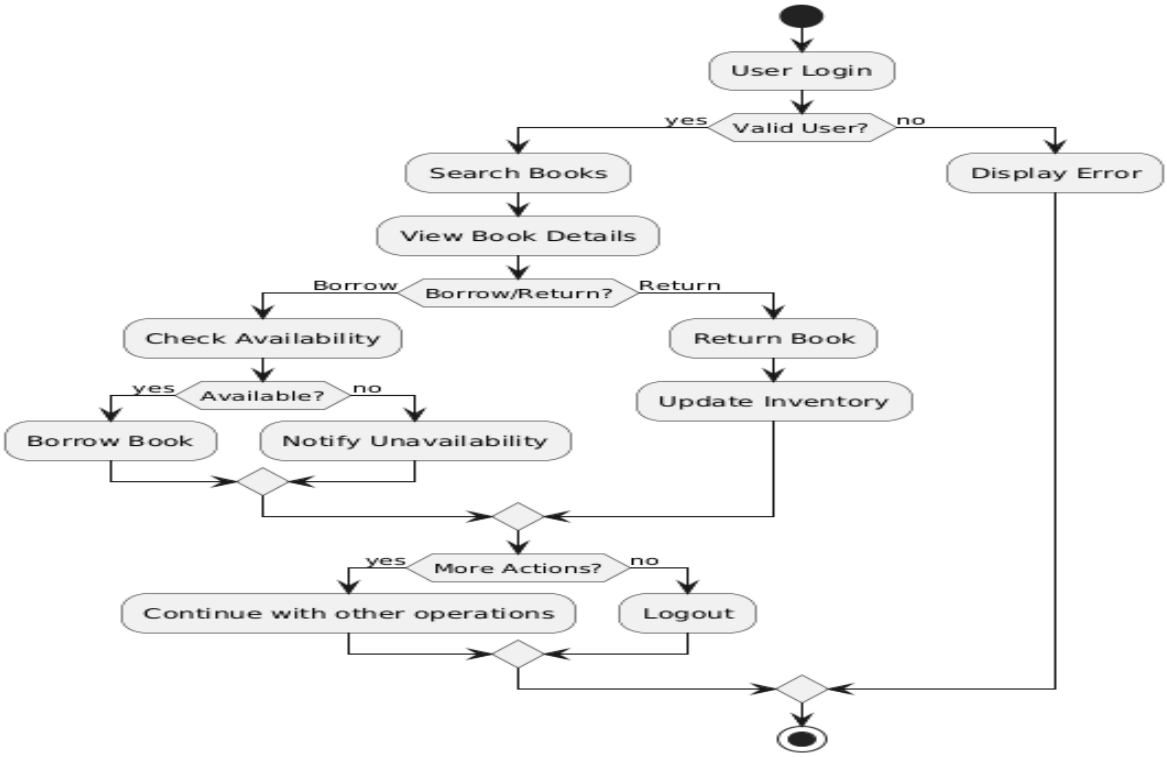

State Machine Diagram:

It is a behavioral diagram. It portrays the system's behavior utilizing finite state transitions. It is also known as the State-charts diagram. It models the dynamic behavior of a class in response to external stimuli that can trigger a behavioral change.

Activity Diagram:

It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur.

Activity Diagram




**Description:**

**Start**: Represents the start of the activity.

**User Login**: The user logs into the system.

**Valid User?**: A decision point that checks if the user credentials are valid.

If "yes": The user can proceed with the operations (e.g., search books, view details).

If "no": An error message is displayed.

**Search Books**: The user searches for books in the library.

**View Book Details**: The user views details of the selected book.

**Borrow/Return?**: A decision point where the user chooses to borrow or return a book.

**Borrow Book**: The system checks if the book is available.

If available, the book is borrowed.

If not, the system notifies the user of unavailability.

**Return Book**: The user returns a borrowed book and the system updates the inventory.

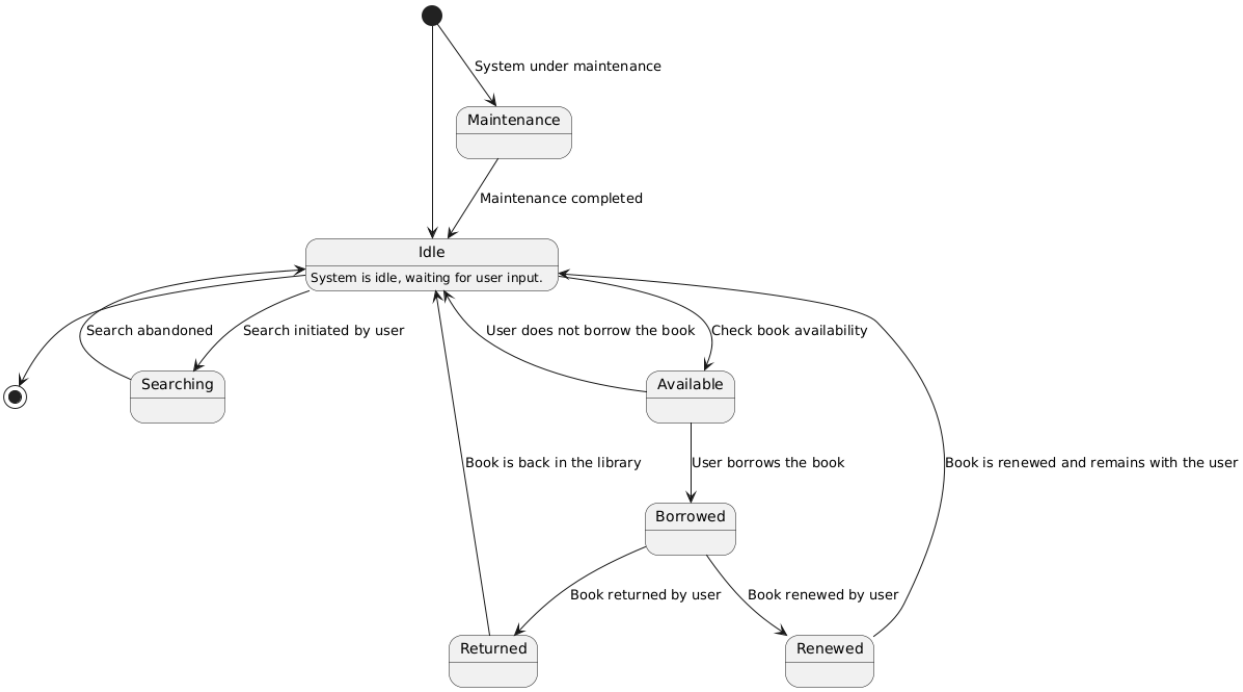
**More Actions?**: A decision point where the user decides whether to perform more operations or logout.

If "yes": The user can continue with other operations.

If "no": The user logs out.

**Stop**: Represents the end of the activity.

## State Diagram




**Description:**

**States:**

**Idle**: The system is waiting for user action.

**Searching**: The user is searching for a book in the system.

**Book Selected**: The user selects a book after searching.

**Checking Availability**: The system checks whether the selected book is available.

**Available**: The book is available for borrowing.

**Unavailable**: The book is not available for borrowing.

**Borrowed**: The book is borrowed by the user.

**Returned**: The book is returned to the library.

**Renewed**: The book is renewed by the user.

**Maintenance**: The system is under maintenance.

**Transitions:**

**[*] → Maintenance**: The system starts in maintenance mode.

**Maintenance → Idle**: After maintenance, the system returns to idle.

**Idle → Searching**: The user initiates a book search.

**Searching → Book Selected**: The user selects a book from the search results.

**Book Selected → Checking Availability**: The system checks if the selected book is available.

**Checking Availability → Available**: The book is available.

**Checking Availability → Unavailable**: The book is not available.

**Available → Borrowed**: The user borrows the available book.

**Borrowed → Idle**: The book is checked out, and the system returns to idle.

**Unavailable → Idle**: The system returns to idle if the book is unavailable.

**Borrowed → Returned**: The user returns the borrowed book.

**Returned → Idle**: The book is back in the library, and the system is idle.

**Borrowed → Renewed**: The user renews the borrowed book.

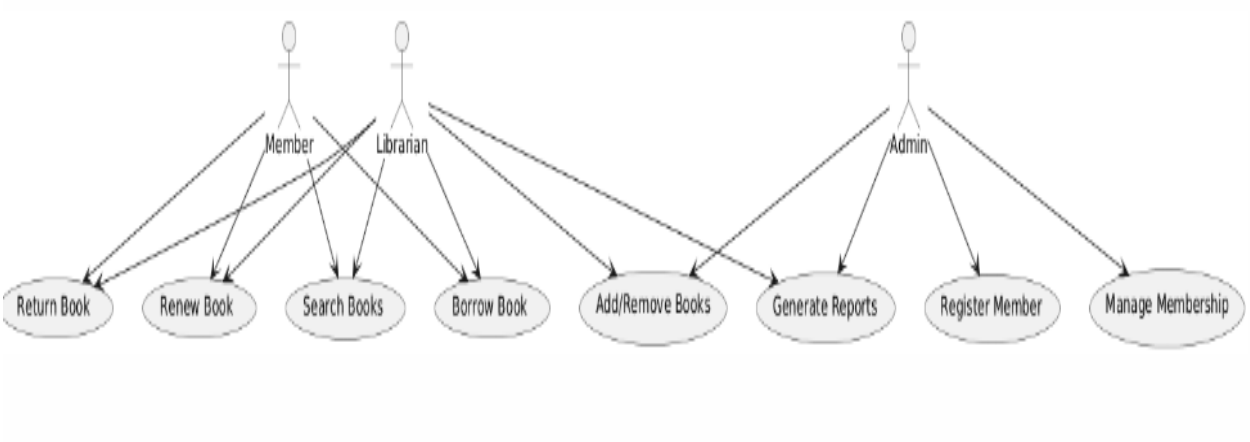
**Renewed → Idle**: After renewal, the system returns to idle.

**Idle → Maintenance**: The system goes into maintenance mode.

## Use Case Diagram:

It encapsulates the functional requirement of a system and its association with actors. It represents the functionality of a system by utilizing actors and use cases.

## Use Case Diagram




Description:

Actors:

Librarian: Manages book operations (searching, borrowing, returning, renewing) and handles book inventory.

Member: Searches for books, borrows, returns, and renews them.

Admin: Manages the overall system, including adding/removing books, generating reports, and managing member registrations.

Use Cases:

Search Books: Allows users to search for available books.

Borrow Book: Members can borrow books from the library.

Return Book: Members can return borrowed books.

Renew Book: Members can renew their borrowed books.

Add/Remove Books: Admin or librarian can add or remove books from the library system.

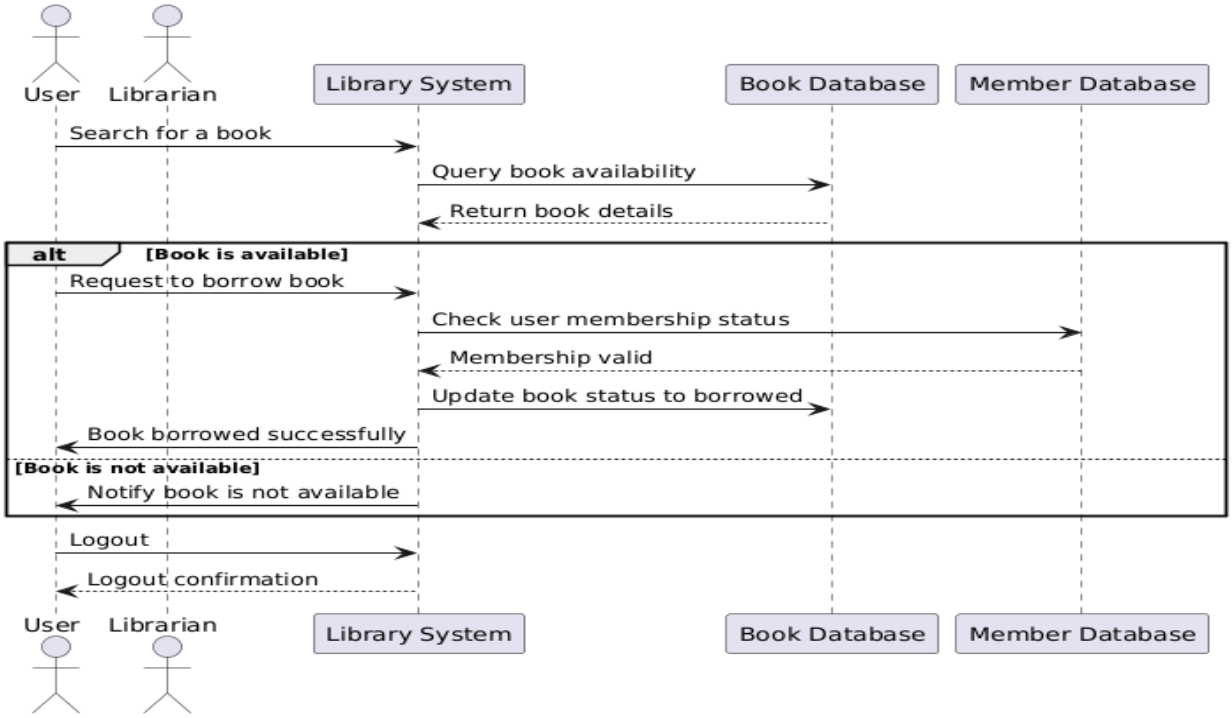
Generate Reports: Admin or librarian can generate reports on library usage and inventory.

Register Member: Admin can register new members.Manage Membership: Admin can manage (update/delete) membership information.

## Interaction Diagrams:

It gives emphasis to object interactions and also depicts the flow between various use case elements of a system. It shows how objects interact with each other and how the data flows within them.

## Interaction Diagrams




**Description:**

**Actors and Participants**:

**User**: The person interacting with the library system.

**Librarian**: The library staff who might assist in the process.

**Library System**: The main system handling user requests.

**Book Database**: The database containing book information.

**Member Database**: The database containing member information.

**Interactions**:

The user initiates a search for a book.

The system queries the book database for availability.

If the book is available, the user requests to borrow the book.

The system checks the user's membership status in the member database.

If the membership is valid, the book status is updated to borrowed, and the user is notified.

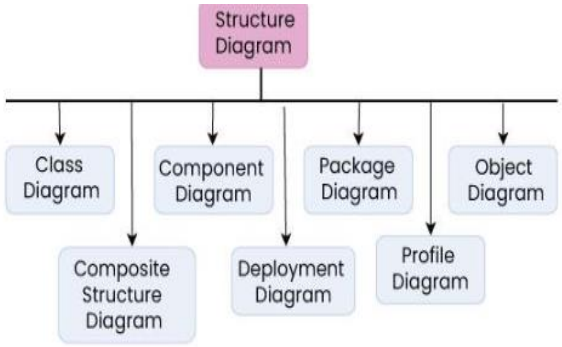
If the book is not available, the user is informed.

The user logs out, and the system confirms the logout.

Conditions: The ‘alt’ block is used to show the alternative flows: one where the book is available and another where it is not.

## Structural Diagrams:

Structural diagrams depict a static view or structure of a system. It is widely used in the documentation of software architecture. It holds class diagrams, component diagrams, object diagrams, composite structure diagrams etc. It presents an outline for the system. It states the elements to be present that are to be modeled.




### Class Diagram:

It depicts the static structure of the system. It displays the system's class, attributes, and methods. It is helpful in recognizing the relation between different objects as well as classes.

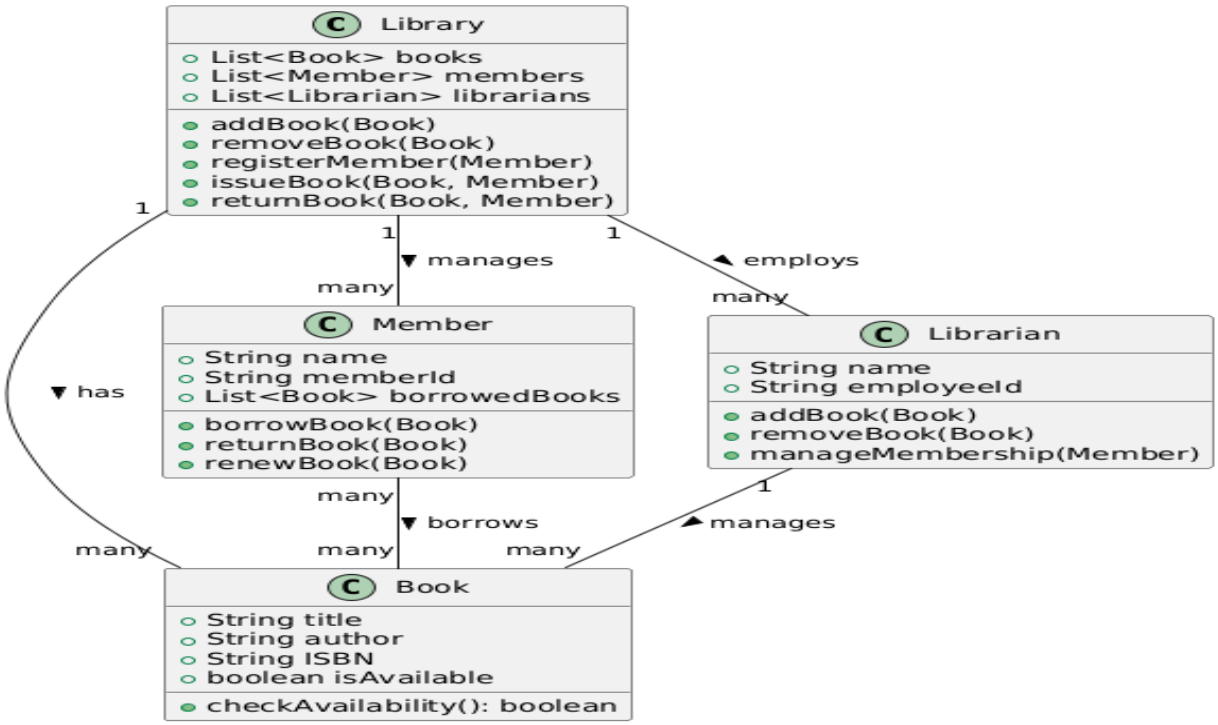
### Component Diagram:

It portrays the organization of the physical components within the system. It is used for modeling execution details.

### Object Diagram:

It describes the static structure of a system at a particular point in time. It can be used to test the accuracy of class diagrams. It represents distinct instances of classes and the relationship between them at a time.

## Class Diagram




Description:

Classes:

Library: Manages collections of books, members, and librarians. It has methods to add/remove books, register members, issue books, and return books.

Book: Represents a book with attributes like title, author, ISBN, and isAvailable. It has a method to check availability.

Member: Represents a library member with attributes name, memberId, and a list of borrowed books. Methods include borrowing, returning, and renewing books.

Librarian: Represents a librarian with attributes name and employeeId. It has methods for managing books and memberships.

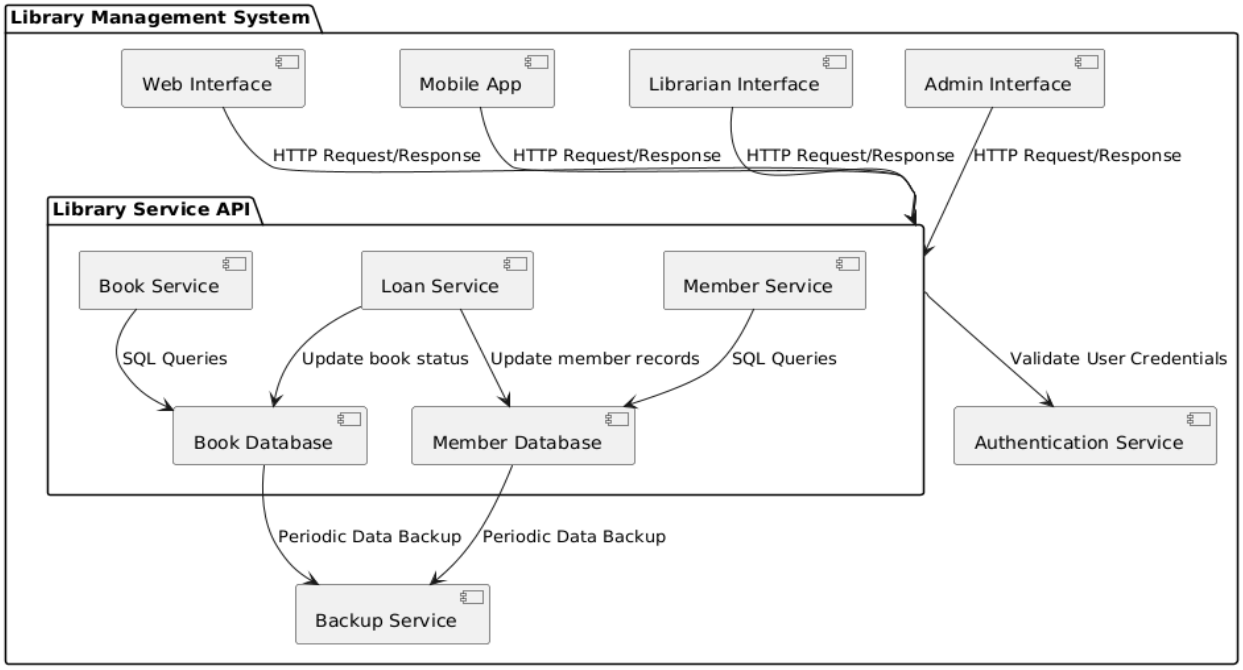
**Relationships**:

**Library** has a one-to-many relationship with **Book**, **Member**, and **Librarian**.

**Member** has a many-to-many relationship with **Book** indicating that members can borrow multiple books.

**Librarian** has a one-to-many relationship with **Book** indicating that a librarian manages multiple books.

## Component Diagram




**Description:**

**Components**:

**Web Interface**: The interface used by users accessing the system via a web browser.

**Mobile App**: The interface used by users accessing the system via a mobile application.

**Librarian Interface**: The interface used by librarians to manage library operations.

**Admin Interface**: The interface used by administrators to manage the system.

**Library Service API**: The core API that handles requests from different interfaces and provides services like book management, member management, and loan processing.

**Book Service**: A service responsible for managing books in the library.

**Member Service**: A service responsible for managing library members.

Loan Service: A service responsible for processing book loans and returns.

Authentication Service: A service used to validate user credentials during login.

Book Database: A database containing information about the books in the library.

Member Database: A database containing information about library members.

Backup Service: A service responsible for periodically backing up the data in the databases.

**Interactions**:

The **Web Interface**, **Mobile App**, **Librarian Interface**, and **Admin Interface** interact with the **Library Service API** through HTTP requests and responses.

The **Library Service API** interacts with the **Book Service**, **Member Service**, and **Loan Service**, which in turn interact with the respective databases.

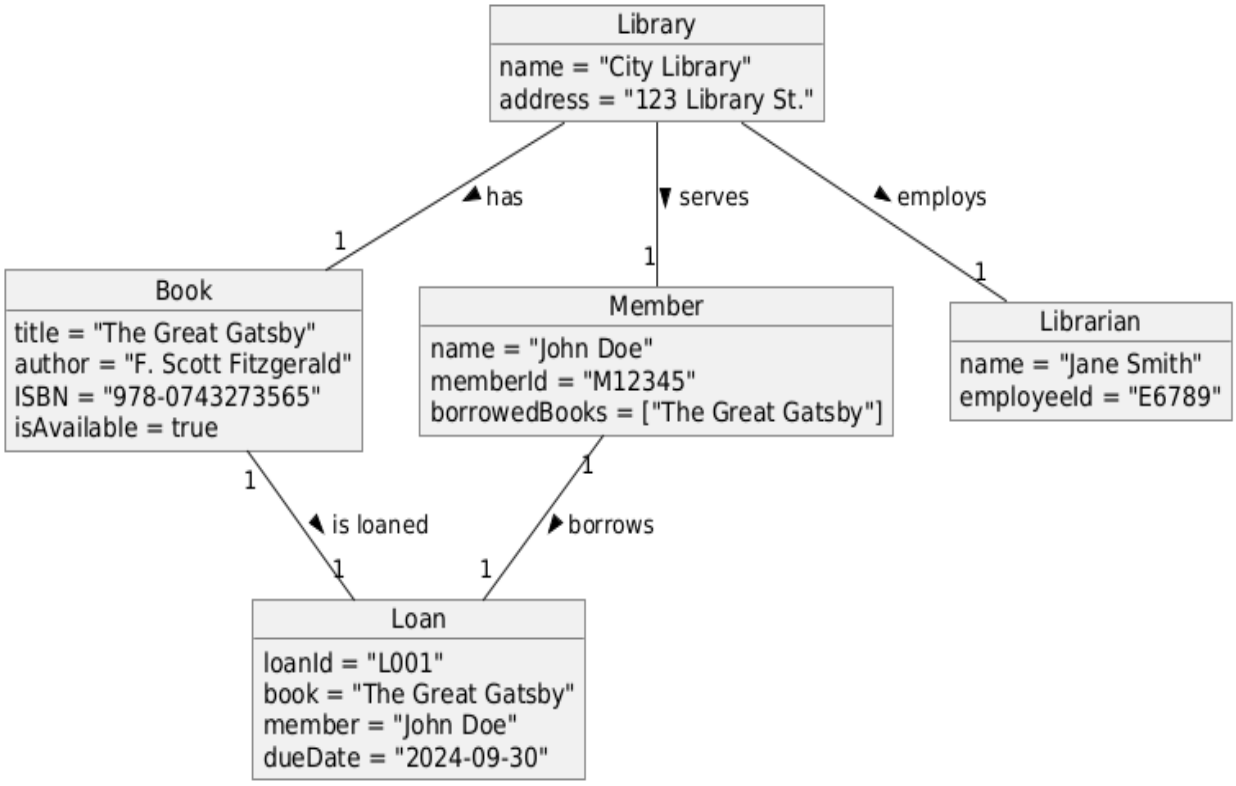
The **Library Service API** uses the **Authentication Service** to validate user credentials.

The **Book Database** and **Member Database** are periodically backed up by the **Backup Service**.

## Composite Structure Diagram:

The composite structure diagrams show parts within the class. It is similar to class diagrams, it represents individual parts in a detailed manner when compared with class diagrams.

## Object Diagram




Description:

Objects:

Library: Represents the library entity with attributes like name and address.

Book: Represents a specific book with attributes like title, author, ISBN, and isAvailable.

Member: Represents a library member with attributes like name, memberId, and a list of borrowedBooks.

Librarian: Represents a librarian with attributes like name and employeeId.

Loan: Represents a loan transaction with attributes like loanId, book, member, and dueDate.

**Relationships**:

The **library** object has associations with **Book**, **Member**, and **Librarian** objects.

The **Member** object is associated with the **Loan** object, indicating that the member has borrowed the book.

The **Book** object is also associated with the **Loan** object, representing the book being loaned out.

# 3. Data Flow Diagram (DFD): -

A Data Flow Diagram (DFD) is a graphical representation of the flow of data through a system, illustrating how data is processed by the system in terms of inputs and outputs.

DFDs are divided into different levels to represent varying degrees of detail:

## 1. Level 0 DFD (Context Diagram)

Purpose: The Level 0 DFD, also known as a Context Diagram, provides a high-level view of the entire system. It shows the system as a single process, with its interactions (data flows) with external entities such as users, other systems, or organizations.

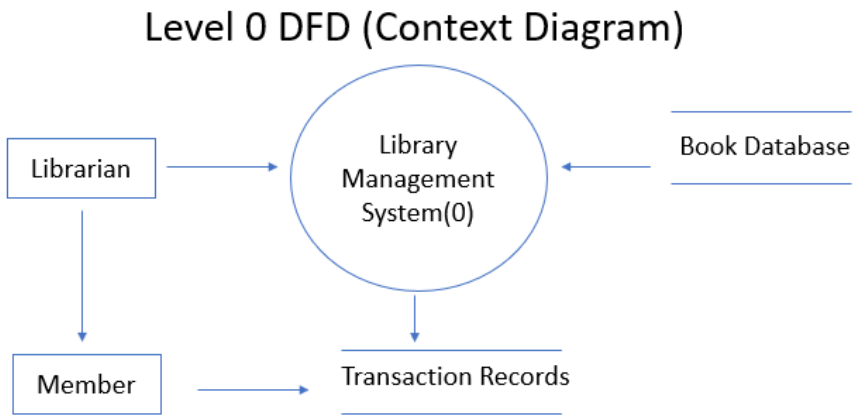
Components:

Single Process: The system is represented as a single process or bubble.

External Entities: Entities that interact with the system are represented as squares or rectangles.

Data Flows: Arrows represent the flow of data between the system (the single process) and external entities.

Detail Level: Minimal, showing only the broad inputs and outputs to/from the system.



Purpose: This diagram provides a high-level overview of the entire Library Management System. It shows how the system interacts with external entities.

Components:

System: Library Management System (as a single process).

External Entities: Librarian, Member, Book Database.

Data Flows: Data flows between the external entities and the system.

Description:

The Librarian interacts with the system to manage books.

The Member interacts with the system to borrow or return books.

The Book Database and Transaction Records are used to store and retrieve data.

## 2. Level 1 DFD

Purpose: The Level 1 DFD decomposes the single process of the Level 0 diagram into major sub-processes and shows how data flows between them. It provides a more detailed view of the system, focusing on the major functions or processes that are part of the system.

Components:

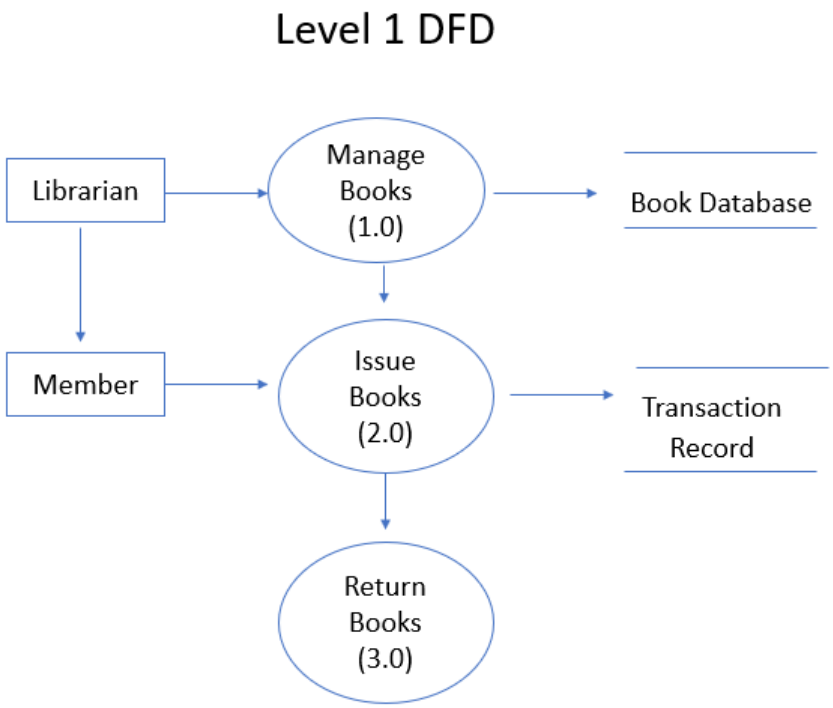
Processes: These represent the main functions of the system. Each process is labeled and numbered (e.g., Process 1.0, Process 2.0).

Data Stores: Represented as open-ended rectangles, data stores are places where data is held within the system.

External Entities: As in the Level 0 DFD, external entities interact with the processes.

Data Flows: Arrows show the flow of data between processes, data stores, and external entities.

Detail Level: Moderate, showing major processes, their interactions, and the data flows.




Purpose:

The Level 1 DFD breaks down the Library Management System into major processes. Each process handles a specific part of the system's functionality.

Components:

Processes: Manage Books (1.0), Issue Books (2.0), Return Books (3.0).

Data Stores: Book Database, Member Database, Transaction Records.

External Entities: Librarian, Member.

Description:

Manage Books (1.0): Involves adding, updating, and removing books.

Issue Books (2.0): Handles the borrowing of books by members.

Return Books (3.0): Manages the return of borrowed books.

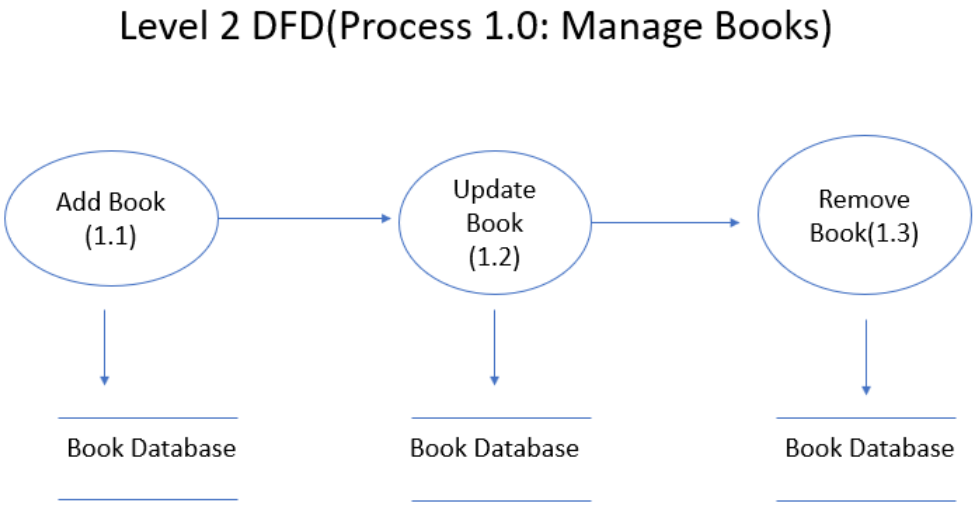
## 3. Level 2 (and Lower) DFDs

Purpose: These levels further decompose each process in the Level 1 DFD into more detailed sub-processes, providing even more granularity about the system’s inner workings. Each Level 1 process can be broken down into multiple Level 2 processes, and so on, depending on the system's complexity.

Components:

Processes: More detailed processes that represent specific tasks within a major function.

Data Stores: These may be the same as in Level 1 or new ones specific to the sub-processes.




Purpose:

The Level 2 DFD further decomposes the "Manage Books" process into more detailed sub-processes.

Components:

Sub-Processes: Add Book (1.1), Update Book (1.2), Remove Book (1.3).

Data Stores: Book Database.

External Entities: Librarian.

Description:

Add Book (1.1): Adding a new book to the database.

Update Book (1.2): Updating details of an existing book.

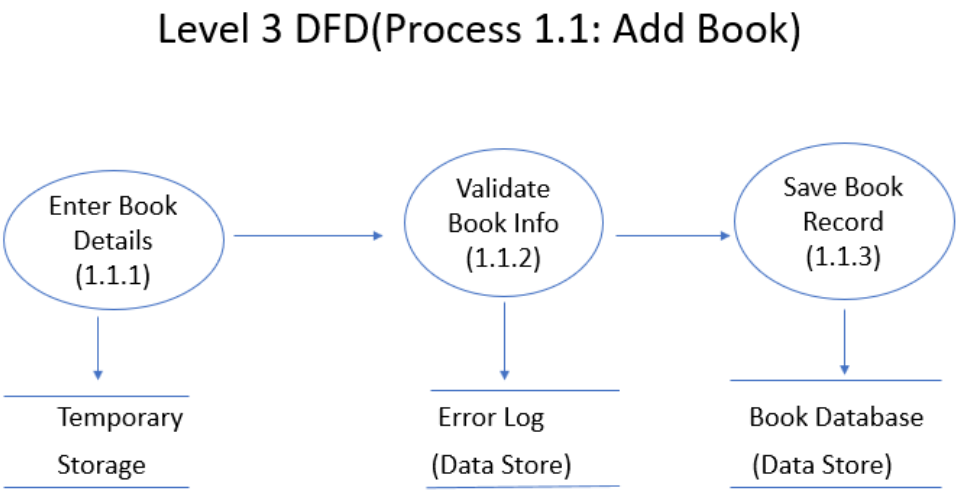
Remove Book (1.3): Removing a book from the database.

## 4. Level 3 (and Beyond) DFDs

Purpose: For very complex systems, some processes within Level 2 may require even further decomposition. Level 3 and beyond break down these processes into even smaller tasks, offering the highest level of detail.

Components: Similar to Level 2 but more granular.

Detail Level: Very high, usually reserved for highly complex systems where even minor details need to be captured.




Purpose:

The Level 3 DFD breaks down the "Add Book" process into even more granular tasks.

Components:

Sub-Processes: Enter Book Details (1.1.1), Validate Book Info (1.1.2), Save Book Record (1.1.3).

Data Stores: Temporary Storage, Error Log, Book Database.

Description:

Enter Book Details (1.1.1): Inputting book information into the system.

Validate Book Info (1.1.2): Checking the entered data for errors or inconsistencies.

Save Book Record (1.1.3): Storing the validated book information in the Book Database.

**Key Points to Remember –**

Balancing: Each level of DFD must balance with the next higher level. This means that the inputs and outputs in a Level 1 DFD must match those in the Level 0 DFD, and so on.

Decomposition: The purpose of moving to lower levels is to break down complex processes into simpler, more manageable tasks.

Iteration: The creation of DFDs is often an iterative process, involving refining and revising the diagrams to ensure clarity and completeness.

These levels of DFDs help in understanding, designing, and analyzing a system by visualizing how data flows through it at varying levels of detail.