SYSTEM DESIGN – COLLEGE LIBRARY MANAGEMENT SYSTEM

TEAM –
Akash A, Aadhithya Balaji, Chezhian V

System Design

System Design defines the structure and behavior of a system.
It converts user requirements into a complete architecture and acts as a blueprint before actual development.
System design focuses on efficiency, scalability, resource management, security, and reliability.

Types of System Design:

- **High Level Design (HLD)**
- **Low Level Design (LLD)**

Goal

- **To meet both functional and non-functional requirements**
- **To efficiently manage books, users, and transactions**
- **To handle a large number of students and records**
- **To reduce system failures and data inconsistency**
- **To enable easy maintenance and future upgrades**

Requirement Handling

I. Functional requirements are solved by clearly defining system features such as book issue, return, and search, and implementing them using appropriate logic, APIs, and workflows that satisfy library operations.

II. Non-functional requirements are addressed using indexing for fast book search, role-based authentication for security, backup mechanisms for reliability, and scalable server architecture to handle many users during peak times.

III. High-Level Design (HLD) describes the overall structure of the system, major modules, their interactions, and data flow.
Example: In a library management system, HLD includes modules such as User, Book, Transaction, and Admin with the flow:
Client → Server → Database

IV. Low-Level Design (LLD) explains internal module logic, database tables, APIs, validation rules, and error handling used during implementation.
Example: It defines tables for students, books, and issue records, along with APIs such as login, issueBook, returnBook, and fineCalculation.

Types of Architecture

1. Monolithic Architecture

All components are built as a single unified application.
Easy to develop and deploy, but difficult to scale for large libraries.

2. Microservices Architecture

Each service such as Book Service, User Service, and Transaction Service operates independently.
Highly scalable and fault-tolerant, but complex to manage.

3. Client–Server Architecture

Clients (students/librarians) request services, and the server processes and responds.
Provides centralized control and clear separation of responsibilities.

4. Layered Architecture

System is divided into layers such as presentation, business logic, and data layers.
Improves maintainability and separation of concerns.

5. MVC (Model–View–Controller) Architecture

Separates application into Model, View, and Controller components.
Improves code organization and easier debugging.

CAP Theorem

CAP Theorem states that a distributed system cannot simultaneously guarantee:

- **Consistency**
- **Availability**
- **Partition Tolerance**

Only two of these properties can be achieved at a time.

Example:
Library transaction systems often prefer Consistency + Partition Tolerance, ensuring correct issue/return records even during network failures.

Visualization

Structural Visualization

- **Class Diagrams (UML)**
- **Component Diagrams**
- **Object Diagrams**
- **Package Diagrams**

Behavioral Visualization

- **Sequence Diagrams**
- **Data Flow Diagrams (DFD)**

- **Activity Diagrams**
- **Use Case Diagrams**

Logical & Conceptual Visualization

- **Entity Relationship (ER) Diagrams**
- **C4 Model (Context, Container, Component, Code)**
- **User Flow Diagrams**

Design Patterns

1. Singleton Pattern

Ensures only one instance of a class exists throughout the system.
Used when a single shared resource is required.

```
class LibraryDatabase {

    private static LibraryDatabase instance;


    private LibraryDatabase() {

        System.out.println("Library Database Connected");

    }


    public static LibraryDatabase getInstance() {

        if (instance == null) {

            instance = new LibraryDatabase();

        }

        return instance;

    }

}


public class SingletonDemo {

    public static void main(String[] args) {

        LibraryDatabase db1 = LibraryDatabase.getInstance();

        LibraryDatabase db2 = LibraryDatabase.getInstance();

    }
```

```
}
```

Explanation:
Only one database connection is created and reused across the system to avoid conflicts in library records.

---

2. Factory Pattern

Separates object creation from usage by deciding the object type at runtime.

```
interface Book {

    void getBookType();

}


class ReferenceBook implements Book {

    public void getBookType() {

        System.out.println("Reference Book");

    }

}


class IssuableBook implements Book {

    public void getBookType() {

        System.out.println("Issuable Book");

    }

}


class BookFactory {

    public static Book createBook(String type) {

        if (type.equalsIgnoreCase("Reference"))

            return new ReferenceBook();

        else

            return new IssuableBook();
```

```
    }
}


public class FactoryDemo {
    public static void main(String[] args) {
        Book book = BookFactory.createBook("Issuable");
        book.getBookType();
    }
}
```

Explanation:
Different book types are created without exposing creation logic to the user.

3. Observer Pattern

Allows users to receive updates automatically when system state changes.

```
import java.util.*;
interface Observer {
    void update(String message);
}
class Student implements Observer {
    private String name;
    Student(String name) {
        this.name = name;
    }
    public void update(String message) {
        System.out.println(name + " received: " + message);
    }
}
class Library {
    private List<Observer> students = new ArrayList<>();
    void addStudent(Observer student) {
        students.add(student);
```

```
    }
    void notifyStudents() {
        for (Observer student : students) {
            student.update("Book is now available!");
        }
    }
}
public class ObserverDemo {
    public static void main(String[] args) {
        Library library = new Library();
        Student s1 = new Student("Student1");
        Student s2 = new Student("Student2");
        library.addStudent(s1);
        library.addStudent(s2);
        library.notifyStudents();
    }
}
```

Explanation:
Students are notified when previously unavailable books become available.

4. Strategy Pattern

Allows selection of different algorithms dynamically at runtime.

```
interface FineStrategy {
    void calculateFine(int days);
}
class StudentFine implements FineStrategy {
    public void calculateFine(int days) {
        System.out.println("Fine: " + days * 2);
    }
}
```

```
class StaffFine implements FineStrategy {

    public void calculateFine(int days) {

        System.out.println("Fine: " + days * 1);

    }

}


class FineCalculator {

    FineStrategy strategy;


    void setStrategy(FineStrategy strategy) {

        this.strategy = strategy;

    }


    void calculate(int days) {

        strategy.calculateFine(days);

    }

}
```

Explanation:
Different fine calculation strategies are applied for students and staff.

Application – College Library Management System

To design a Library Management System, we begin by defining its functional and non-functional requirements.

Functional Requirements (FR)

- **User authentication for students and librarians**
- **Search books by title, author, or category**

- **Issue and return books**
- **Track due dates and calculate fines**
- **View issued books and history**
- **Librarian can add, update, or remove books**
- **Notification for due dates and availability**

Non-Functional Requirements (NFR)

- **Consistency: Accurate issue and return records**
- **Security: Authentication and authorization for users**
- **Availability: System accessible during library working hours**
- **Scalability: Handle many students during exam periods**
- **Performance: Book search results should load within seconds**
- **Reliability: Data backup and recovery mechanisms**

Low Level Design

At this stage, UML diagrams are used to represent:

- **Class relationships**
- **Data flow between modules**
- **User interaction sequences**
- **Database schema**

This ensures clear implementation during development.