- **SUBMITTED BY:** MUHAMMAMD ABUBAKAR
- **REG NO:** FA22-BSE-155
- **SUBMITTED TO:** SIR MUKHTIAR ZAMIN
- **SUBJECT:** SOFTWARE DESIGN AND ARCHITECTURE

# ***LAB ASSIGNMENT 2***

## ***ARCHITECTURAL PROBLEMS:***

### 1. CIRCULAR DEPENDENCIES

- **Description:**
  Circular dependencies occur when two or more modules depend on each other, either directly or indirectly. This creates a loop in the dependency graph, making the system harder to maintain and test. For example, if Module A relies on Module B, and Module B also relies on Module A, any change in one module can propagate issues to the other. This situation complicates system builds and deployments and increases the chances of runtime errors.
- **Solution**:
  To solve this problem, you can refactor the architecture to remove the dependency loop by introducing an intermediary module or abstraction layer. Dependency injection can also be used to break the cycle. This ensures that the modules remain independent and reduces tight coupling.
- **Real-World Example:**
  In a library management system, the Book module might depend on the library module, and the Library module might depend on the Book module, creating a circular dependency. This can be resolved by introducing a Library Service module that acts as a mediator between the two, breaking the dependency cycle.

### 2. POOR LOGGING AND MONITORING

- **Description:**
  Poor logging and monitoring mean the system lacks adequate logs to track events and errors or tools to monitor its performance. This makes it extremely difficult to debug issues or identify the root cause of failures. For example, if a payment fails in an e-commerce application and there is no log entry for the failure, it becomes hard to pinpoint the problem. Furthermore, the absence of monitoring tools can lead to undetected performance bottlenecks or failures until it’s too late.
- **Solution:**
  To address this issue, implement centralized logging systems like Log4j or ELK (Elasticsearch, Logstash, Kibana). Use monitoring tools such as Prometheus or Grafana to track system metrics and generate real-time alerts for any issues. Structured and detailed logs can help trace errors and monitor system performance effectively.
- **Real-World Example:**
  In a food delivery application, a user's payment fails, but no meaningful logs are recorded. This delays the resolution of the issue. By implementing structured logging and monitoring for the payment gateway, developers can quickly identify the failure point and resolve the issue.

### 3. TIGHT COUPLING

- **Description:**
  Tight coupling occurs when modules or services are heavily dependent on one another. This interdependency makes the system rigid, as changes in one module require changes in other dependent modules. For example, if an Order module is directly dependent on an Inventory module, any update to the inventory logic might break the order processing functionality. This approach hinders scalability, reusability, and maintainability.
- **Solution:**
  To reduce tight coupling, adopt loose coupling by designing interfaces or APIs for communication between modules. Use design principles like Dependency Injection (DI) to abstract dependencies and reduce direct interaction between modules. Service-Oriented Architecture (SOA) or microservices can also help decouple components, making the system more modular and scalable.
- **Real-World Example:**
  An online store has tightly coupled Inventory and Order modules. A change in inventory management breaks the order processing system. By introducing an API layer that abstracts inventory logic from the order module, the store can ensure that updates in one module do not affect the other.

### 4. REDUNDANT DATA STORAGE

- **Description:**
  Redundant data storage occurs when the same data is duplicated across multiple systems or modules unnecessarily. This leads to increased storage costs and data synchronization issues. For example, if both Driver and User services in a ride-sharing application store trip details, inconsistencies can arise when updates in one service are not reflected in the other. Redundant storage also complicates database management and increases the risk of stale data.
- **Solution:**
  The solution is to centralize data storage in a shared database or data lake accessible by all systems that require it. Implementing data duplication techniques and ensuring proper database normalization can also help eliminate redundancy and maintain data consistency.
- **Real-World Example:**
  In a ride-sharing app, trip details are redundantly stored in both the Driver and User services. This can lead to inconsistencies if updates occur in only one service. By centralizing trip details in a shared database that both services accesses, the system eliminates redundancy and ensures consistent and synchronized data.

### 5. OUTDATED TECHNOLOGY STACK

- **Description:**
  Using an outdated technology stack can lead to several problems, including poor performance, lack of support, and increased security vulnerabilities. Systems built on obsolete frameworks or programming languages are harder to maintain, as fewer developers are skilled in those technologies. For example, legacy banking systems built with COBOL face challenges because the language is no longer widely used or supported. Such systemalso miss out on new features and performance improvements available in modern technologies.

- **Solution:**
  To address this problem, adopt a phased approach to migrate the system to a modern technology stack. Begin by auditing the current stack to identify critical components for upgrade. Use backward-compatible tools and frameworks to ensure a smooth transition without system downtime. Regularly update libraries, frameworks, and tools to the latest stable versions to stay current with technology trends.
- **Real-World Example:**
  A banking system running on COBOL struggles to find developers for maintenance, increasing operational costs and risks. The bank decided to gradually migrate its core systems to a modern framework like Java Spring Boot, enabling it to leverage a wider talent pool, enhance security, and integrate new features more easily.

## ***SELECTING PROBLEM AND GIVING SOLUTION***:

### 1. CIRCULAR DEPENDENCIES

***INITIAL ISSUE:***

The initial problem, without the mediator (Library Services), might look like this:

- Book needs information about Library.
- Library needs information about Book. This mutual dependency would lead to tight coupling between the two modules, causing issues when changes need to be made, leading to difficult maintenance, testing, and possibly circular references in the code.

CODE SOLUTION:

```
package com.mycompany.assignment;

public class ASSIGNMENT {

    // Book class

    static class Book {

      private String title;

public Book(String title) {

        this.title = title;

     } public String getTitle() {

        return t   }

    }

// Library class

    static class Library {
```

```
    private String name;
  public Library(String name) {
      this.name = name;
    }public String getName() {
      return name;
    }
  } // LibraryService class (Mediator)
  static class LibraryService {
    private Book book;
    private Library library;
  public LibraryService(Book book, Library library) {
      this.book = book;
      this.library = library;
    } public String getBookTitle() {
      return book.getTitle();
    }public String getLibraryName() {
      return library.getName();
    }
  }
// Main method
  public static void main(String[] args) {
    // Create instances of Book and Library
    Book book = new Book("Java Programming");
    Library library = new Library("Central Library")// Create LibraryService as a mediator
    LibraryService libraryService = new LibraryService(book, library);

    // Display information using the mediator
    System.out.println("Book: " + libraryService.getBookTitle() + " is available in Library: " +
  libraryService.getLibraryName());
```

```
        System.out.println("Library: " + libraryService.getLibraryName() + " has Book: " +
libraryService.getBookTitle());
    }
}
```

## ***BENEFITS:***

- **Decoupling**: The Book and Library classes are no longer dependent on each other directly.
- **Maintainability**: The code is more modular and easier to maintain because changes to Book or Library don't affect each other.
- **Testability**: It's easier to test individual components (Book, Library, and LibraryService) in isolation.