CSE 464: Modular Design Case Study Example

Application: A **Student Management System (SMS)** is designed to handle various administrative tasks in a university setting. The current design follows a **monolithic** approach, where all functionalities are handled within a **single large module**. That is the student management system has one large module handling **student registration, fee payments, grade management, and notifications**

Current Design Structure (Monolithic Architecture)

The **main module** handles multiple responsibilities, including:

1. **Student Registration** – Registers new students, updates their details, assigns student IDs.
2. **Fee Payments** – Manages tuition fees, records transactions, and generates payment receipts.
3. **Grade Management** – Stores students' academic records, calculates GPAs, and generates transcripts.
4. **Notifications** – Sends emails and SMS alerts for registration updates, fee reminders, and grades.

These functionalities are implemented as **a single class or module** (e.g., StudentManagementSystem):

```
public class StudentManagementSystem {

public void registerStudent(String name, String dob, String
address) {
        // Adds student to the database
        // Generates student ID
        System.out.println("Student " + name + " registered.");
    }

public void processPayment(String studentID, double amount) {
        // Handles fee payment for student
        System.out.println("Payment of $" + amount + " received
for student " + studentID);
    }
```

```
    public void calculateGPA(String studentID) {
    // Retrieves grades and calculates GPA
  System.out.println("GPA calculated for student " + studentID);
    }

public void sendNotification(String studentID, String message) {
        // Sends email notification
        System.out.println("Notification sent to student " +
studentID + ": " + message);
    }
}
```

This has led to **poor modularity**, resulting in **high coupling and low cohesion**.

High Coupling: All functionalities are in **one big class**, making changes difficult.

Low Cohesion: Unrelated functionalities (registration, payment, GPA, notifications) exist in one class.

Hard to Maintain & Test: If one part fails, the whole system may be affected.

Key Coupling and Cohesion Issues

1. High Coupling (Modules Dependent on Each Other)

- **Tightly Connected Functionalities:**
    - The **fee payment system** directly accesses **student records**, meaning any change in the student structure impacts payments.
    - **Grade management** fetches data directly from **student registration**, making them interdependent.
    - The **notification system** is coupled with both **grades and payments**, as it directly retrieves fee dues and academic results.

- **Global Data Sharing:**
    - All functions interact with a **single student database table**, where different functionalities modify the same dataset.
    - Example: The updateStudentInfo() method is used by both **registration** and **fee payment**, leading to conflicts.
    -
- **Change Ripple Effect:**
    - Modifying the **fee payment logic** could unintentionally impact **student registration** or **notifications**.
    - If a new payment method (e.g., cryptocurrency) is added, the entire system requires modification.

2. Low Cohesion (Unrelated Functionalities in One Module)

- **Student registration, fees, grading, and notifications are all in the same module**, despite being **logically distinct responsibilities**.
- **Unrelated functions grouped together:**
    - registerStudent(), calculateGPA(), processPayment(), and sendEmailNotification() exist in the same module.
    - There's **no clear separation of concerns**, making debugging and maintaining the system difficult.
- **Redundant Code:**
    - Each function requires access to student data, leading to repeated queries and duplicate logic.
- **Difficult Testing & Scalability:**
    - Since **everything is interconnected**, testing one part of the system often affects others.
    - Adding new features (e.g., mobile push notifications) means modifying the monolithic system, increasing risks of breaking existing functionalities.

Refactored Modular Approach for Better Cohesion & Low Coupling

Instead of **one large module**, we can **divide the system into independent, modular components** with **clear separation of concerns**:

1. **Student Registration Module** → Handles student enrollment & profile updates.
2. **Fee Payment Module** → Manages financial transactions independently.
3. **Grade Management Module** → Handles academic records.
4. **Notification Module** → Sends alerts based on messages from other modules.

Each module **communicates through APIs or message queues** instead of direct function calls.

Key Improvements

Lower Coupling

- **No direct dependencies** between modules.
- Changes in **one module** do not break others.

Higher Cohesion

- Each module has **a clear, single responsibility**.
- No **unrelated functions grouped together**.

Better Maintainability & Testing

- Easier to **debug, update, and test** individual components.
- Allows for **future scalability** (e.g., adding new features without affecting existing ones).

Improved Modular Design (Low Coupling & High Cohesion)

Solution: Split the system into **separate classes** for each functionality, using **dependency injection** for better flexibility.

```
public class StudentRegistration {
  public String registerStudent(String name, String dob, String address) {
    // Simulate student registration
```

```
        String studentID = "S1234";  // Normally, this would be generated dynamically
        System.out.println("Student " + name + " registered with ID: " + studentID);
        return studentID;
    }
}

public class FeePayment {
    public void processPayment(String studentID, double amount) {
        // Simulate fee payment processing
        System.out.println("Payment of $" + amount + " received for student " + studentID);
    }
}

public class GradeManagement {
    public void calculateGPA(String studentID) {
        // Simulate GPA calculation
        System.out.println("GPA calculated for student " + studentID);
    }
}



public class NotificationService {
    public void sendNotification(String studentID, String message) {
        // Simulate sending notifications
        System.out.println("Notification sent to student " + studentID + ": " + message);
    }
```

```
}

public class StudentManagementApp {
    public static void main(String[] args) {
        // Creating separate modules
        StudentRegistration registration = new StudentRegistration();
        FeePayment payment = new FeePayment();
        GradeManagement grades = new GradeManagement();
        NotificationService notifications = new NotificationService();

        // Using modules independently
        String studentID = registration.registerStudent("Alice", "2002-05-10", "123 Main St");
        payment.processPayment(studentID, 500.00);
        grades.calculateGPA(studentID);
        notifications.sendNotification(studentID, "Your grades are now available.");
    }
}
```