SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 3rd semester
OBJECT ORIENTED ANALYSIS AND DESIGN (PCA20C07J)

Lab Manual

Lab 1: Case study - the Next Gen POS system

Title: Case Study - The Next Gen POS System

Aim: To understand and analyze the requirements and design principles of a Next Generation Point-of-Sale (POS) system through a detailed case study.

Procedure:

1. **Understand the Domain:** Research the typical functionalities and business rules of a retail POS system.
2. **Identify Actors:** Determine all external entities (users or other systems) that interact with the POS system (e.g., Cashier, Manager, Customer, Payment Gateway, Inventory System).
3. **Identify Use Cases:** Based on the identified actors and system functionalities, list the primary use cases (e.g., Process Sale, Handle Returns, Manage Inventory, Generate Reports).
4. **Develop Use Case Descriptions:** For each major use case, write a detailed description including pre-conditions, post-conditions, main flow, and alternative flows.
5. **Identify Conceptual Classes:** From the use case descriptions, identify key conceptual classes in the problem domain (e.g., Sale, Product, Customer, Payment, Register).
6. **Create a Domain Model:** Develop a conceptual class diagram representing the identified classes, their attributes, and associations.
7. **Analyze Interactions:** Consider specific scenarios (e.g., "Processing a cash sale") and trace the interactions between objects.

Source Code: N/A (This lab is primarily an analysis and design exercise, not a coding one.)

Input: N/A

Expected Output:

- A list of identified actors and use cases.
- Detailed use case descriptions for key functionalities.
- A conceptual domain model (class diagram) for the Next Gen POS system.
- Sample interaction diagrams (e.g., sequence diagram fragments) for a chosen scenario.

Lab 2: Identify a software system that needs to be developed

Title: Software System Identification

Aim: To identify and define a suitable software system for development, considering its scope, purpose, and potential users.

Procedure:

1. **Brainstorming:** Discuss and brainstorm various real-world problems or needs that can be addressed by a software system. Consider the suggested domains (e.g., Passport automation system, Book bank, Exam registration, Stock maintenance system, Online course reservation system).
2. **Problem Definition:** Select one problem and clearly define the core issue it aims to solve.
3. **Stakeholder Identification:** Identify the primary users and other stakeholders who will interact with or be affected by the system.
4. **Preliminary Scope Definition:** Outline the initial boundaries of the system, including what it *will* and *will not* do.
5. **Feasibility Study (Brief):** Conduct a quick assessment of the technical, operational, and economic feasibility of developing the identified system.

Source Code: N/A

Input: N/A

Expected Output:

- A clear statement of the identified software system (e.g., "Online Course Reservation System").
- A brief description of the problem it solves.
- A list of key stakeholders.
- A preliminary scope document.

Lab 3: Document the Software Requirements Specification (SRS) for the identified system.

Title: Software Requirements Specification (SRS) for Online Course Reservation System

Aim: To prepare a comprehensive Software Requirements Specification (SRS) document for the identified software system, detailing its functional and non-functional requirements.

Procedure:

1. **Review Identified System:** Revisit the system identified in Lab 2 (e.g., Online Course Reservation System) and its preliminary scope.
2. **Gather Requirements:** Conduct interviews, surveys, or workshops with potential users/stakeholders to elicit detailed requirements.
3. **Categorize Requirements:** Classify requirements into functional (what the system does) and non-functional (how well the system performs, security, usability, etc.).
4. **Document SRS:** Write the SRS document following a standard template (e.g., IEEE 830). Include sections like:
    - Introduction (Purpose, Scope, Definitions, References)
    - Overall Description (Product Perspective, Product Functions, User Characteristics, Constraints, Assumptions, Dependencies)
    - Specific Requirements (Functional Requirements, Non-functional Requirements, Interface Requirements)
5. **Review and Refine:** Review the SRS with stakeholders to ensure accuracy, completeness, and clarity.

Source Code: N/A

Input: N/A

Expected Output:

- A complete Software Requirements Specification (SRS) document for the Online Course Reservation System.

Lab 4: Identify use cases

Title: Use Case Identification for Online Course Reservation System

Aim: To identify and list all significant use cases for the Online Course Reservation System based on its Software Requirements Specification (SRS).

Procedure:

1. **Analyze SRS:** Carefully read through the functional requirements documented in the SRS.
2. **Identify Actors:** Determine all external entities that interact with the Online Course Reservation System (e.g., Student, Instructor, Administrator, Payment Gateway).
3. **Identify Goals/Tasks:** For each actor, identify the primary goals or tasks they wish to achieve using the system. Each goal typically corresponds to a use case.
4. **List Use Cases:** Create a comprehensive list of all identified use cases (e.g., Register for Course, View Course Catalog, Manage Enrollment, Process Payment, Create Course).
5. **Brief Description:** Provide a short, concise description for each use case.

Source Code: N/A

Input: N/A

Expected Output:

- A list of identified actors for the Online Course Reservation System.
- A list of all major use cases with brief descriptions.

Lab 5: Develop the Use Case model

Title: Use Case Model Development for Online Course Reservation System

Aim: To develop a comprehensive Use Case Model for the Online Course Reservation System, including a Use Case Diagram and detailed use case descriptions.

Procedure:

1. **Refine Use Cases:** Review and refine the use cases identified in Lab 4, ensuring they are at an appropriate level of detail.
2. **Draw Use Case Diagram:** Create a UML Use Case Diagram showing:
    - The system boundary.
    - All identified actors.
    - All primary use cases.
    - Relationships between actors and use cases (association).
    - Relationships between use cases (include, extend, generalization) if applicable.
3. **Write Detailed Use Case Descriptions:** For each critical use case, write a detailed description following a standard template. Include:
    - Use Case Name
    - Actor(s)
    - Pre-conditions
    - Post-conditions
    - Main Flow (step-by-step interaction)
    - Alternative Flows
    - Exception Flows

Source Code: N/A

Input: N/A

Expected Output:

- A UML Use Case Diagram for the Online Course Reservation System.
- Detailed use case descriptions for at least 3-5 key use cases.

Lab 6: Identify the conceptual classes and develop a Domain Model and also derive a Class Diagram from that.

Title: Domain Model and Class Diagram for Online Course Reservation System

Aim: To identify the conceptual classes for the Online Course Reservation System, develop a Domain Model, and subsequently derive a Class Diagram from it.

Procedure:

1. **Review Use Cases and SRS:** Re-examine the use cases and SRS to identify nouns and noun phrases that represent significant concepts in the problem domain.
2. **Identify Conceptual Classes:** List all potential conceptual classes (e.g., Course, Student, Instructor, Enrollment, Payment, Schedule, Department).
3. **Define Attributes:** For each conceptual class, identify relevant attributes that describe its properties.
4. **Identify Associations:** Determine the relationships (associations) between conceptual classes (e.g., a Student *enrolls in* a Course, an Instructor *teaches* a Course). Specify multiplicity for each association.
5. **Develop Domain Model (Conceptual Class Diagram):** Draw a UML Class Diagram representing the conceptual classes, their attributes, and associations. This is a conceptual model, focusing on real-world concepts, not software classes.
6. **Derive Class Diagram (Design Class Diagram):** From the Domain Model, refine it into a Design Class Diagram. This involves adding:
    - Visibility (public, private, protected) for attributes and methods.
    - Data types for attributes.
    - Methods (operations) that classes will perform, derived from use case interactions.
    - Any necessary navigation arrows.

Source Code: N/A

Input: N/A

Expected Output:

- A Domain Model (Conceptual Class Diagram) for the Online Course Reservation System.
- A Design Class Diagram for the Online Course Reservation System, showing attributes, methods, and visibility.

Lab 7: Using the identified scenarios, find the interaction between objects and represent them using UML

Title: Object Interaction Modeling using UML for Online Course Reservation System

Aim: To model the dynamic interactions between objects for specific scenarios within the Online Course Reservation System using appropriate UML diagrams.

Procedure:

1. **Select Key Scenarios:** Choose 2-3 significant scenarios from the use case descriptions (e.g., "Student Registers for a Course," "Administrator Creates a New Course," "Student Views Enrollment History").
2. **Identify Participating Objects:** For each chosen scenario, identify the objects that will participate in the interaction (instances of the classes from your Class Diagram).
3. **Trace Message Flow:** Determine the sequence of messages exchanged between these objects to accomplish the scenario's goal.
4. **Represent using UML:**
    - **Option 1: Sequence Diagram:** Draw a UML Sequence Diagram for each scenario, showing objects as lifelines and messages as arrows ordered by time.
    - **Option 2: Communication (Collaboration) Diagram:** Draw a UML Communication Diagram for each scenario, showing objects as nodes and messages as numbered arrows on associations. (Lab 8 specifically asks for both, so this lab can focus on one or introduce both).

Source Code: N/A

Input: N/A

Expected Output:

- UML Sequence Diagrams (or Communication Diagrams) for 2-3 key scenarios of the Online Course Reservation System.

Lab 8: Sequence and Collaboration Diagrams.

Title: Sequence and Collaboration Diagrams for Online Course Reservation System

Aim: To create detailed Sequence and Collaboration (Communication) Diagrams for critical functionalities of the Online Course Reservation System, illustrating object interactions over time and across collaborations.

Procedure:

1. **Revisit Scenarios:** Use the same key scenarios identified in Lab 7 (e.g., "Student Registers for a Course," "Administrator Creates a New Course").
2. **Draw Sequence Diagrams:** For each scenario, draw a comprehensive UML Sequence Diagram. Ensure it clearly shows:
    - All participating objects (lifelines).
    - The order of messages exchanged.
    - Activations/Execution occurrences.
    - Conditional logic (alt fragments) and loops (loop fragments) if necessary.
3. **Draw Collaboration (Communication) Diagrams:** For each scenario, transform the Sequence Diagram into a UML Communication Diagram. Ensure it shows:
    - All participating objects (nodes).
    - Links between objects.
    - Numbered messages indicating the sequence of interaction.

Source Code: N/A

Input: N/A

Expected Output:

- UML Sequence Diagrams for 2-3 key scenarios of the Online Course Reservation System.
- Corresponding UML Communication (Collaboration) Diagrams for the same scenarios.

Lab 9: Draw relevant State Chart and Activity Diagrams for the same system

Title: State Chart and Activity Diagrams for Online Course Reservation System

Aim: To model the behavior of specific objects and system processes within the Online Course Reservation System using UML State Chart (State Machine) and Activity Diagrams.

Procedure:

1. **Identify State-Dependent Objects:** Choose an object whose behavior changes significantly over its lifetime (e.g., "Enrollment" object, "Course" object, "Student" status).
2. **Draw State Chart Diagram:** For the chosen object, create a UML State Chart Diagram showing:
    - All possible states of the object.
    - Transitions between states.
    - Events that trigger transitions.
    - Actions performed during transitions or within states.
    - Initial and final states.
3. **Identify Business Processes:** Choose a significant business process or workflow within the system (e.g., "Course Registration Process," "Payment Processing Workflow").
4. **Draw Activity Diagram:** For the chosen process, create a UML Activity Diagram showing:
    - Actions/Activities.
    - Control flows between activities.
    - Decision points and merges.
    - Forks and joins for parallel activities.
    - Swimlanes to show responsibility if applicable.

Source Code: N/A

Input: N/A

Expected Output:

- A UML State Chart Diagram for a chosen object (e.g., Enrollment) in the Online Course Reservation System.
- A UML Activity Diagram for a significant business process (e.g., Course Registration) in the Online Course Reservation System.

Lab 10: Implement the system as per the detailed design.

Title: Implementation of Online Course Reservation System

Aim: To implement the core functionalities of the Online Course Reservation System based on the detailed design models (Class Diagrams, Sequence Diagrams, etc.) developed in previous labs.

Procedure:

1. **Choose a Programming Language:** Select an appropriate object-oriented programming language (e.g., Java, Python, C#).
2. **Set up Development Environment:** Configure the necessary tools (IDE, compiler/interpreter).
3. **Translate Design to Code:**
    - Create classes and define their attributes and methods as per the Design Class Diagram.
    - Implement the logic for interactions between objects as depicted in Sequence/Collaboration Diagrams.
    - Focus on implementing at least 2-3 core use cases (e.g., Student Registration, Course Enrollment, Course Listing).
4. **Database Integration (Optional/Basic):** If applicable, implement basic data persistence (e.g., using simple file I/O, in-memory lists, or a lightweight database).
5. **Modular Development:** Develop the system in a modular fashion, ensuring good coding practices (e.g., encapsulation, clear method names, comments).

Source Code:

```
// Placeholder for the actual source code of the Online Course Reservation
System.
// This section should contain the complete, runnable code for the implemented
system,
// adhering to the design principles derived in previous labs.
// Example:
// public class Student {
//     private String studentId;
//     private String name;
//     private List<Course> enrolledCourses;
//
//     public Student(String studentId, String name) {
//         this.studentId = studentId;
//         this.name = name;
//         this.enrolledCourses = new ArrayList<>();
//     }
//
//     public void enrollCourse(Course course) {
//         // Logic to enroll student in a course
//     }
//     // Other methods
// }
//
// public class Course {
//     private String courseId;
//     private String title;
//     private int capacity;
//     private int enrolledCount;
//     // Other attributes and methods
// }
```

```
//
// public class CourseReservationSystem {
//     public static void main(String[] args) {
//         // Main application logic
//     }
// }
```

Input:

```
// Placeholder for sample input data or commands used to run the system.
// Example:
// 1. Create a new student: "createStudent S001 John Doe"
// 2. Create a new course: "createCourse C101 OOP 30"
// 3. Enroll student in course: "enroll S001 C101"
// 4. View course catalog: "viewCourses"
// 5. View student enrollments: "viewEnrollments S001"
```

Expected Output:

```
// Placeholder for the expected output when the system is run with the sample
input.
// Example:
// Student John Doe (S001) created.
// Course OOP (C101) with capacity 30 created.
// S001 enrolled in C101 successfully.
// Course Catalog:
//   - C101: OOP (Enrolled: 1/30)
// Student S001 enrollments:
//   - C101: OOP
```

Lab 11: package diagrams - Component and Deployment Diagrams.

Title: Package, Component, and Deployment Diagrams for Online Course Reservation System

Aim: To create UML Package, Component, and Deployment Diagrams to illustrate the architectural structure and physical deployment of the Online Course Reservation System.

Procedure:

1. **Develop Package Diagram:**
    - Group related classes from your Design Class Diagram into logical packages (e.g., `com.example.courses.model`, `com.example.courses.service`, `com.example.courses.ui`).
    - Show dependencies between packages.
2. **Develop Component Diagram:**
    - Identify the major software components of the system (e.g., User Management Component, Course Management Component, Payment Gateway Integration Component, Database Component).
    - Show the interfaces provided and required by each component.
    - Illustrate the relationships between components.
3. **Develop Deployment Diagram:**
    - Identify the physical hardware nodes where the system will run (e.g., Web Server, Application Server, Database Server, Client Workstation).
    - Show the communication paths between nodes.
    - Place the identified components and artifacts (e.g., executable files, libraries) onto the nodes where they are deployed.

Source Code: N/A

Input: N/A

Expected Output:

- A UML Package Diagram for the Online Course Reservation System.
- A UML Component Diagram for the Online Course Reservation System.
- A UML Deployment Diagram for the Online Course Reservation System.

Lab 12: Test the software system for all the scenarios identified as per the use case diagram

Title: System Testing for Online Course Reservation System

Aim: To thoroughly test the implemented Online Course Reservation System against all scenarios identified in the Use Case Diagram to ensure it meets the specified requirements.

Procedure:

1. **Review Use Cases:** Revisit the Use Case Diagram and detailed use case descriptions.
2. **Develop Test Cases:** For each use case and its alternative/exception flows, design specific test cases. Each test case should include:
    - Test Case ID
    - Use Case/Scenario Covered
    - Pre-conditions
    - Test Steps (detailed actions to perform)
    - Expected Results
    - Actual Results (to be filled during execution)
    - Status (Pass/Fail)
3. **Prepare Test Data:** Create necessary input data for executing the test cases.
4. **Execute Tests:** Run the implemented system with the prepared test data, following the test steps.
5. **Record Results:** Document the actual results and compare them against the expected results. Mark each test case as Pass or Fail.
6. **Report Defects:** For any failed test cases, log defects with detailed information for debugging.

Source Code:

```
// Placeholder for any automated test scripts or frameworks used.
// If manual testing, this section would be N/A.
// Example (JUnit/Pytest/etc. if automated):
// public class CourseReservationSystemTest {
//     @Test
//     public void testStudentRegistration() {
//         // Test steps for student registration
//         // Assertions to verify expected outcome
//     }
//
//     @Test
//     public void testCourseEnrollment() {
//         // Test steps for course enrollment
//         // Assertions to verify expected outcome
//     }
//     // Other test methods
// }
```

Input:

```
// Placeholder for test data used during testing.
// Example:
// Test Data for "Student Registers for Course":
// - Student ID: "S002", Name: "Jane Doe"
```

```
// - Course ID: "C102", Title: "Database Systems", Capacity: 25
// - Enrollment attempt: S002 enrolling in C102
//
// Test Data for "Student Attempts to Enroll in Full Course":
// - Course ID: "C103", Title: "Web Dev", Capacity: 1 (already full)
// - Enrollment attempt: S003 enrolling in C103
```

Expected Output:

```
// Placeholder for the expected output or system behavior for each test case.
// Example:
// Test Case 1: Student Registration
// Expected: Student S002 registered successfully.
// Actual: Student S002 registered successfully. (PASS)
//
// Test Case 2: Course Enrollment
// Expected: Student S002 enrolled in C102 successfully.
// Actual: Student S002 enrolled in C102 successfully. (PASS)
//
// Test Case 3: Student Attempts to Enroll in Full Course
// Expected: Error: Course C103 is full. Enrollment failed.
// Actual: Error: Course C103 is full. Enrollment failed. (PASS)
```

Lab 13: Improve the reusability and maintainability of the software system

Title: Improving Reusability and Maintainability of Online Course Reservation System

Aim: To refactor and enhance the existing Online Course Reservation System to improve its reusability (ability to use components in other systems) and maintainability (ease of modification and bug fixing).

Procedure:

1. **Code Review:** Conduct a thorough review of the existing codebase to identify areas for improvement. Look for:
    - Code duplication.
    - Long methods or classes.
    - Tight coupling between components.
    - Lack of clear separation of concerns.
    - Poor naming conventions.
    - Insufficient comments.
2. **Apply Refactoring Techniques:**
    - **Extract Method/Class:** Break down large methods into smaller, more focused ones, or create new classes for distinct responsibilities.
    - **Introduce Abstraction:** Use interfaces or abstract classes to define common behaviors and allow for different implementations.
    - **Reduce Coupling:** Decouple components by using dependency injection or event-driven architectures.
    - **Improve Naming:** Use clear, descriptive names for variables, methods, and classes.
    - **Add Comments/Documentation:** Ensure critical parts of the code are well-commented and documented.
3. **Modularization:** Further modularize the system into distinct, independent modules or packages.
4. **Version Control:** Utilize a version control system (e.g., Git) to manage changes and track improvements.

Source Code:

```
// Placeholder for the refactored source code of the Online Course Reservation
System.
// This section should show the "before" and "after" if possible, or just the
improved code.
// Highlight the changes made to enhance reusability and maintainability.
// Example:
// // Before (tightly coupled logic)
// public class CourseReservationSystem {
//     public void enrollStudent(Student s, Course c) {
//         // Direct database calls and validation logic here
//     }
// }
//
// // After (improved with service layer and validation)
// public interface EnrollmentService {
//     void enrollStudent(Student student, Course course);
// }
//
// public class EnrollmentServiceImpl implements EnrollmentService {
```

```
//     private CourseRepository courseRepo; // Injected dependency
//     private StudentRepository studentRepo; // Injected dependency
//
//     public EnrollmentServiceImpl(CourseRepository cr, StudentRepository sr) {
//         this.courseRepo = cr;
//         this.studentRepo = sr;
//     }
//
//     public void enrollStudent(Student student, Course course) {
//         // Validation logic
//         // Call repository methods
//     }
// }
```

Input:

```
// Placeholder for input used to demonstrate the functionality of the improved
system.
// This should be similar to previous inputs, but the internal implementation is
improved.
```

Expected Output:

```
// Placeholder for the expected output from the improved system.
// The functional output should be the same as before, but the internal
structure is better.
```

Lab 14: By applying appropriate design patterns.

Title: Applying Design Patterns to Online Course Reservation System

Aim: To identify and apply appropriate design patterns to the Online Course Reservation System to solve recurring design problems and further enhance its flexibility, reusability, and maintainability.

Procedure:

1. **Review System Design:** Analyze the current design and implementation of the Online Course Reservation System to identify areas where design patterns can provide solutions.
2. **Identify Problem Areas:** Look for common problems that design patterns address (e.g., object creation, object composition, algorithm variations, communication between objects).
3. **Select Appropriate Patterns:** Choose one or more suitable design patterns (e.g., Singleton, Factory Method, Strategy, Observer, Decorator, Facade) based on the identified problem areas.
    - **Example:**
        - **Factory Method:** For creating different types of users or courses.
        - **Strategy:** For different payment methods (credit card, PayPal, etc.).
        - **Observer:** For notifying interested parties (e.g., students, instructors) about course updates or enrollment changes.
        - **Singleton:** For a single instance of a configuration manager or logger.
4. **Refactor Code with Patterns:** Modify the existing codebase to incorporate the chosen design patterns. This often involves creating new classes or interfaces and reorganizing existing logic.
5. **Document Pattern Application:** Clearly document which patterns were applied, where they were applied, and the rationale behind their selection.

Source Code:

```
// Placeholder for the source code demonstrating the application of design
patterns.
// Highlight the parts of the code where patterns have been implemented.
// Example (using Strategy Pattern for Payment):
// public interface PaymentStrategy {
//     void pay(double amount);
// }
//
// public class CreditCardPayment implements PaymentStrategy {
//     public void pay(double amount) { /* ... */ }
// }
//
// public class PayPalPayment implements PaymentStrategy {
//     public void pay(double amount) { /* ... */ }
// }
//
// public class Enrollment {
//     private PaymentStrategy paymentMethod;
//
//     public Enrollment(PaymentStrategy method) {
//         this.paymentMethod = method;
//     }
//
//     public void processPayment(double amount) {
//         paymentMethod.pay(amount);
//     }
// }
```

Input:

```
// Placeholder for input used to demonstrate the functionality with applied
design patterns.
// Example:
// Enroll student S001 in C101 using Credit Card payment.
// Enroll student S002 in C102 using PayPal payment.
```

Expected Output:

```
// Placeholder for the expected output from the system with applied design
patterns.
// The functional output should be consistent, but the underlying design is
improved.
// Example:
// Processing credit card payment for 150.00.
// Processing PayPal payment for 200.00.
```

Lab 15: Implement the modified system and test it for various scenarios.

Title: Implementation and Comprehensive Testing of Modified Online Course Reservation System

Aim: To implement any final modifications to the Online Course Reservation System (including design pattern applications) and conduct thorough testing across various scenarios to ensure stability, correctness, and adherence to all requirements.

Procedure:

1. **Final Implementation:** Integrate all changes made in previous labs (improvements, design patterns) into the complete system. Ensure all components work together seamlessly.
2. **Regression Testing:** Re-run all previously passed test cases to ensure that the modifications have not introduced new bugs or regressions in existing functionalities.
3. **New Scenario Testing:** Develop and execute new test cases for any scenarios that might have been introduced or significantly altered by the recent modifications.
4. **Performance Testing (Optional):** Conduct basic performance tests to check system responsiveness under typical loads.
5. **Usability Testing (Optional):** If a user interface is developed, perform basic usability tests.
6. **Defect Management:** Log, track, and resolve any new defects found during this final testing phase.
7. **Documentation Update:** Update any relevant design documents (e.g., Class Diagrams, Sequence Diagrams) to reflect the final implemented system.

Source Code:

```
// Placeholder for the final, complete source code of the Online Course
Reservation System.
// This should be the most refined version of the system.
```

Input:

```
// Placeholder for a comprehensive set of input data and commands covering all
major scenarios,
// including edge cases and error conditions, to thoroughly test the final
system.
```

Expected Output:

```
// Placeholder for the detailed expected output for all test scenarios,
// demonstrating the system's correct behavior and error handling.
```