Westminster Health Centre Management System

OOP Implementation – Case Study

Overview

This coursework specification provides a comprehensive guide for developing a Java-based management system for the Westminster Health Centre.

Students will work incrementally from Week 5 through Week 10, building a fully functional console and GUI-based application that demonstrates mastery of abstraction, encapsulation, inheritance, polymorphism, collections, event handling, exception management, and file persistence.

---

1. Background and Context

The Westminster Health Centre is a small but expanding community clinic providing general medical and administrative services. Currently employing a modest team of doctors, receptionists, and part-time medical staff, the centre faces significant operational challenges with its outdated record-keeping system. Staff details are scattered across spreadsheets and handwritten files, making it cumbersome to track personnel, schedules, and role assignments.

With new government regulations mandating digital record-keeping, the centre's director has commissioned a lightweight, Java-based management application to modernize their operations. The system must record and display staff information, automate administrative reports, and provide a foundation for future appointment scheduling integration.

Core Requirements

The clinic requires a system that can:

- Add new staff members (doctors or receptionists) with complete personal and professional information
- Display all staff details through both console output and graphical table interface
- Search, edit, and remove staff records by their unique ID
- Calculate operational summaries including total staff count, role distribution, and consultation metrics
- Store and retrieve data from files to ensure continuity between sessions

As the clinic's IT consultant, you will build this software entirely from scratch using Object-Oriented Programming principles. You will infer, design, implement, and document every system layer by carefully applying fundamental OOP concepts and design patterns.

---

2. System Architecture and Design

The system architecture follows a layered approach derived from professional software engineering practices. Understanding this conceptual structure is essential before beginning implementation.

System Layers

```
1. Domain Model Layer
   ├── StaffMember (abstract superclass)
   │      ├── Doctor (concrete subclass)
   │      └── Receptionist (concrete subclass)

2. Manager/Controller Layer
   ├── HealthCentreManager (interface)
   └── WestminsterHealthCentreManager (implementation)

3. Presentation Layer
   ├── Console Menu Interface
   └── Swing GUI (JFrame + JTable)

4. Persistence Layer
   └── File I/O (text/CSV or serialization)
```

Domain Model Design

The **StaffMember** abstract class serves as the foundation, containing common attributes and behaviors:

- Core attributes: unique ID, name, surname, date of birth, contact number
- Abstract behavior: role description method that subclasses must implement
- Concrete behavior: string representation for display purposes

The **Doctor** subclass extends StaffMember with specialized medical attributes:

- Medical licence number for regulatory compliance
- Specialization field (e.g., Cardiology, Pediatrics)
- Number of weekly consultations for workload tracking

The **Receptionist** subclass extends StaffMember with administrative attributes:

- Assigned desk number for physical location tracking
- Hours per week for scheduling and payroll purposes

Manager Layer Responsibilities

The manager layer acts as the crucial bridge between user interactions and data storage. Implementing the HealthCentreManager interface, this layer maintains an ArrayList of StaffMember objects for dynamic data management and provides:

- Menu-driven operations for adding, removing, viewing, and saving staff records
- Comprehensive input validation with exception handling
- File handling capabilities for data persistence
- Search and sort functionality for efficient data retrieval

Presentation Layer Components

The console interface provides immediate text-based interaction during early development stages, offering a menu-driven system for testing core functionality before GUI implementation.

The graphical user interface represents a later-stage enhancement, displaying data through Swing components. A custom table model binds the ArrayList data structure to visual table rows and columns, while interactive buttons enable user actions.

---

3. Functional Objectives

Through this coursework, students will progressively build and integrate the following capabilities:

1. **Object-Oriented Design**: Design and implement core classes using encapsulation, inheritance, and polymorphism to create a maintainable and extensible system architecture
2. **Data Management**: Develop a comprehensive console-driven interface supporting full CRUD operations (Create, Read, Update, Delete) for staff records
3. **Collections Framework**: Utilize ArrayList for dynamic data storage, implementing searching, sorting, and filtering operations
4. **Graphical User Interface**: Build an interactive Swing-based GUI with tables, buttons, and custom table models for professional data presentation
5. **Event-Driven Programming**: Implement event handling mechanisms to create responsive and interactive user experiences
6. **Exception Management**: Apply robust exception handling to gracefully manage invalid inputs, duplicate records, and system errors
7. **Data Persistence**: Implement file I/O capabilities to save and load staff data, ensuring continuity between program executions
8. **Quality Assurance**: Develop and execute JUnit test cases to validate system functionality and ensure code reliability

---

4. Technical Requirements

Development Environment

Category	Specification
Language	Java 17 or higher
IDE	IntelliJ IDEA
OOP Concepts	Encapsulation, abstraction, inheritance, polymorphism
Data Structures	ArrayList<StaffMember> for dynamic storage
GUI Toolkit	Java Swing (JFrame, JTable, JButton, JPanel)
Testing Framework	JUnit 5
File I/O	Text/CSV file storage or object serialization
Code Quality	Proper naming conventions, consistent indentation, meaningful comments
Documentation	UML class diagrams and implementation summary report

Quality Standards

- **Input Validation**: System must handle invalid data types, out-of-range values, and duplicate IDs
- **Error Handling**: Graceful error recovery with user-friendly messages
- **Code Readability**: Self-documenting code with clear variable names and logical organization
- **Modularity**: Separate concerns with distinct classes for different responsibilities
- **Maintainability**: Design for future enhancements and easy modification

---

5. Weekly Implementation Plan

This structured timeline guides you from foundational concepts through complete system implementation. Each week builds incrementally on previous work, ensuring deep understanding and mastery of core concepts.

Week 5 – Object-Oriented Foundations and Core Model Design

Theme: Classes, Objects, Inheritance, and Abstraction

Learning Objectives:

- Master fundamental OOP concepts: classes, objects, constructors, methods, and access modifiers
- Understand and apply inheritance, abstract classes, and interfaces

- Translate UML diagrams into working Java code
- Apply reverse engineering principles to infer system design

Lecture Content:

- Introduction to Object-Oriented Programming paradigm
- UML notation and class diagram interpretation
- Abstract classes vs. interfaces: when and why to use each
- The role of inheritance in code reuse and polymorphism

Implementation Tasks:

1. **Project Setup**: Create a new Java project named `HealthCentreSystem` with appropriate package structure
2. **Abstract Base Class**: Implement the `StaffMember` abstract class
    - Define private attributes: `id`, `name`, `surname`, `dob`, `contactNo`
    - Create constructor with all parameters
    - Implement getter and setter methods with appropriate access modifiers
    - Override `toString()` method for readable output
    - Declare abstract method `getRole()` to be implemented by subclasses
3. **Concrete Subclasses**:
    - Implement `Doctor` class extending `StaffMember`
        - Add attributes: `licenceNumber`, `specialisation`, `consultationsPerWeek`
        - Implement constructor calling super constructor
        - Override `getRole()` to return "Doctor"
        - Add specialized getters and setters
    - Implement `Receptionist` class extending `StaffMember`
        - Add attributes: `deskNumber`, `hoursPerWeek`
        - Implement constructor calling super constructor
        - Override `getRole()` to return "Receptionist"
        - Add specialized getters and setters
4. **Interface Definition**: Create `HealthCentreManager` interface

```
5. void addStaff();
6. void viewStaff();
7. void removeStaff();
```

8. **Implementation Class**: Create `WestminsterHealthCentreManager` implementing the interface
    - Initialize `ArrayList<StaffMember>` for data storage
9. **Testing**: Write a simple test driver in the main method creating several Doctor and Receptionist objects, demonstrating polymorphism by storing them in a StaffMember list
10. **Documentation**: Draw a UML class diagram showing the complete inheritance hierarchy with attributes, methods, and relationships

In-Class Demonstration: Show how polymorphism enables an `ArrayList<StaffMember>` to store both Doctor and Receptionist objects, and how method overriding allows different implementations of `getRole()`.

Expected Outcome: Students understand abstraction, inheritance, and polymorphism through hands-on implementation. They can create objects, demonstrate inheritance relationships, and explain why abstract classes are used.

Week 6 – Collections, Data Structures, and Console Interface

Theme: Dynamic Data Management and User Interaction

Learning Objectives:

- Master Java Collections Framework (ArrayList, Set, Map)
- Implement searching, sorting, and filtering algorithms
- Build interactive console-based user interfaces
- Apply input validation and error handling principles

Lecture Content:

- Java Collections Framework overview and when to use each collection type
- Generic programming and type safety
- Comparator and Comparable interfaces for custom sorting
- Scanner class for user input handling

Implementation Tasks:

1. **Data Structure Enhancement**:
   Ensure `WestminsterHealthCentreManager` uses `ArrayList<StaffMember>` for all staff storage
2. **Search Functionality**: Implement `searchById(String id)` method
   - Iterate through staff list
   - Return matching StaffMember or null if not found
   - Consider using Java Streams for more elegant implementation
3. **Sorting Capability**: Implement `sortByName()` method
   - Use `Collections.sort()` with custom Comparator
   - Sort alphabetically by surname, then by name

```
4. Collections.sort(staffList,
   Comparator.comparing(StaffMember::getSurname)
5.
   .thenComparing(StaffMember::getName));
```

6. **Console Menu System**: Build comprehensive menu-driven interface

```
7. === Westminster Health Centre Management System ===
8. 1. Add New Staff Member
9. 2. Remove Staff Member
10.  3. View All Staff
11.  4. Search Staff by ID
12.  5. Sort Staff by Name
13.  6. Display Statistics
14.  7. Save and Exit
15.  0. Exit without Saving
16.
17.  Enter your choice:
```

18. **Input Handling**: Implement robust input validation
    - Check for empty strings
    - Validate numeric inputs (consultations, hours, desk numbers)
    - Ensure date format compliance
    - Prevent duplicate IDs
19. **Staff Addition Flow**:
    - Prompt user to choose staff type (Doctor/Receptionist)
    - Collect all required information with validation
    - Create appropriate object and add to ArrayList
    - Display confirmation message
20. **Staff Removal**:
    - Prompt for staff ID
    - Search for and display staff details
    - Confirm deletion with user
    - Remove from ArrayList and show success message

Expected Outcome: A fully functional console-based system supporting complete CRUD operations with validated input, searchable and sortable staff records.

Week 7 – Graphical User Interface Development

Theme: Building Professional Visual Interfaces with Swing

Learning Objectives:

- Understand Swing component hierarchy and architecture
- Implement custom table models for data display
- Apply layout managers for responsive UI design
- Separate presentation from business logic (MVC pattern)

Lecture Content:

- Introduction to Java Swing framework
- JFrame, JPanel, JLabel, JButton, and JTable components
- AbstractTableModel pattern for dynamic data binding
- Layout managers: BorderLayout, FlowLayout, GridLayout
- Design principles for intuitive user interfaces

Implementation Tasks:

1. **GUI Framework**: Create `StaffManagementGUI` class extending `JFrame`
    - Set appropriate window title, size, and close operation
    - Initialize all components in constructor
2. **Custom Table Model**:
   Implement `StaffTableModel` extending `AbstractTableModel`

```
3. private ArrayList<StaffMember> staffList;
4. private String[] columnNames = {"ID", "Name", "Surname", "Role",
   "Date of Birth", "Contact"};
```

```
5.
6. @Override
7. public int getRowCount() { return staffList.size(); }
8.
9. @Override
10.  public int getColumnCount() { return columnNames.length; }
11.
12.  @Override
13.  public Object getValueAt(int row, int col) {
14.      StaffMember staff = staffList.get(row);
15.      switch(col) {
16.          case 0: return staff.getId();
17.          case 1: return staff.getName();
18.          // ... implement remaining columns
19.      }
20.  }
```

21. **Table Display**: Create and configure JTable
    - Bind StaffTableModel to JTable
    - Set appropriate column widths
    - Enable row selection
    - Add to JScrollPane for scrolling capability
22. **Button Panel**: Design button panel with professional layout
    - "Add Staff" button for creating new records
    - "Remove Staff" button for deleting selected record
    - "Show Statistics" button for displaying role counts
    - "Refresh" button to reload table data
    - "Exit" button to close application
23. **Layout Design**: Organize components using BorderLayout
    - Table in CENTER region
    - Button panel in SOUTH region
    - Optional title label in NORTH region
24. **Initial Data Loading**: Load existing staff data from manager's ArrayList when GUI initializes

In-Class Demonstration: Show how AbstractTableModel automatically updates the JTable when underlying data changes, demonstrating the power of the observer pattern.

Expected Outcome: A functional GUI displaying all staff members in an organized table format with interactive buttons ready for event handling implementation.

Week 8 – Event Handling and Unit Testing

Theme: Interactive Programming and Quality Assurance

Learning Objectives:

- Implement event-driven programming with ActionListener
- Connect GUI actions to business logic methods
- Write effective JUnit test cases
- Apply Test-Driven Development principles

Lecture Content:

- Event-driven programming concepts and the listener pattern
- ActionListener interface and lambda expressions
- JUnit 5 framework and test annotations
- Test case design: boundary values, equivalence partitioning, edge cases

Implementation Tasks:

1. **Button Event Handlers**: Implement ActionListener for each button

   Add Staff Button:

```
addButton.addActionListener(e -> {
    String[] options = {"Doctor", "Receptionist"};
    int choice = JOptionPane.showOptionDialog(...);
    if (choice == 0) {
        // Show dialog to collect doctor information
        // Validate inputs
        // Create Doctor object
        // Add to manager and refresh table
    } else {
        // Handle receptionist creation similarly
    }
});
```

   Remove Staff Button:

```
removeButton.addActionListener(e -> {
    int selectedRow = staffTable.getSelectedRow();
    if (selectedRow >= 0) {
        String id = (String) tableModel.getValueAt(selectedRow, 0);
        int confirm = JOptionPane.showConfirmDialog(...);
        if (confirm == JOptionPane.YES_OPTION) {
            manager.removeStaff(id);
            tableModel.fireTableDataChanged();
        }
    } else {
        JOptionPane.showMessageDialog(..., "Please select a staff
member to remove");
    }
});
```

   Show Statistics Button:

```
statsButton.addActionListener(e -> {
    long doctorCount = staffList.stream()
                                .filter(s -> s instanceof Doctor)
                                .count();
    long receptionistCount = staffList.stream()
                                      .filter(s -> s instanceof
Receptionist)
                                      .count();
    String message = String.format("Total Staff: %d\nDoctors:
%d\nReceptionists: %d",
                                   staffList.size(), doctorCount,
receptionistCount);
```

```
        JOptionPane.showMessageDialog(..., message);
    });
```

2. **Input Dialog Design**: Create comprehensive input forms
    - Use JTextField for text inputs
    - JComboBox for predefined options (specializations, etc.)
    - JSpinner for numeric inputs
    - Validate before creating objects
3. **Validator Utility Class**: Create static validation methods

```
public class Validator {
    public static boolean isValidPhone(String phone) {
        return phone.matches("\\d{10,15}");
    }

    public static boolean isValidDOB(String dob) {
        try {
            LocalDate.parse(dob);
            return true;
        } catch (DateTimeParseException e) {
            return false;
        }
    }

    public static boolean isPositiveInteger(String input) {
        try {
            return Integer.parseInt(input) > 0;
        } catch (NumberFormatException e) {
            return false;
        }
    }
}
```

26. **JUnit Test Suite**: Create `ValidatorTest` class

```
@Test
public void testValidPhoneNumber() {
    assertTrue(Validator.isValidPhone("1234567890"));
    assertFalse(Validator.isValidPhone("12345"));
    assertFalse(Validator.isValidPhone("abcdefghij"));
}

@Test
public void testValidDateFormat() {
    assertTrue(Validator.isValidDOB("1990-05-15"));
    assertFalse(Validator.isValidDOB("15/05/1990"));
    assertFalse(Validator.isValidDOB("invalid-date"));
}

@Test
public void testDuplicateIdPrevention() {
    // Test that adding duplicate ID throws exception or returns
false
}

@Test
public void testRoleCount() {
    // Test statistics calculation accuracy
}
```

50. **GUI-Controller Integration**: Ensure clean separation
    - GUI calls manager methods
    - Manager performs business logic

- GUI updates display based on results

Expected Outcome: Fully interactive GUI where buttons respond to user actions, data is validated before processing, and comprehensive JUnit tests verify system reliability.

Week 9 – Exception Handling and Robust System Design

Theme: Building Resilient and Error-Tolerant Applications

Learning Objectives:

- Understand exception hierarchy and exception propagation
- Create and use custom exception classes
- Implement defensive programming techniques
- Design user-friendly error recovery mechanisms

Lecture Content:

- Java exception hierarchy: checked vs. unchecked exceptions
- Try-catch-finally blocks and resource management
- Throwing and propagating exceptions
- Best practices for exception handling in layered applications

Implementation Tasks:

1. **Custom Exception Classes**: Create domain-specific exceptions

```
public class InvalidDateException extends Exception {
    public InvalidDateException(String message) {
        super(message);
    }
}

public class DuplicateIdException extends Exception {
    public DuplicateIdException(String id) {
        super("Staff member with ID " + id + " already exists");
    }
}

public class InvalidInputException extends Exception {
    public InvalidInputException(String field, String reason) {
        super("Invalid " + field + ": " + reason);
    }
}

public class StaffNotFoundException extends Exception {
    public StaffNotFoundException(String id) {
        super("No staff member found with ID: " + id);
    }
}
```

2. **Enhanced Validation with Exceptions**: Modify validation methods

```
public void validateStaffId(String id) throws DuplicateIdException,
InvalidInputException {
```

```
27.      if (id == null || id.trim().isEmpty()) {
28.          throw new InvalidInputException("ID", "cannot be empty");
29.      }
30.      if (idExists(id)) {
31.          throw new DuplicateIdException(id);
32.      }
33.  }
34.
35.  public void validateDateOfBirth(String dob) throws
   InvalidDateException {
36.      try {
37.          LocalDate date = LocalDate.parse(dob);
38.          if (date.isAfter(LocalDate.now())) {
39.              throw new InvalidDateException("Date of birth cannot be
   in the future");
40.          }
41.          if (date.isBefore(LocalDate.of(1900, 1, 1))) {
42.              throw new InvalidDateException("Date of birth must be
   after 1900");
43.          }
44.      } catch (DateTimeParseException e) {
45.          throw new InvalidDateException("Invalid date format. Use
   YYYY-MM-DD");
46.      }
47.  }
```

48. **Exception Handling in Manager Methods**: Add try-catch blocks

```
49.  public boolean addStaff(StaffMember staff) {
50.      try {
51.          validateStaffId(staff.getId());
52.          validateDateOfBirth(staff.getDob());
53.          staffList.add(staff);
54.          return true;
55.      } catch (DuplicateIdException e) {
56.          System.err.println("Error: " + e.getMessage());
57.          return false;
58.      } catch (InvalidDateException e) {
59.          System.err.println("Error: " + e.getMessage());
60.          return false;
61.      }
62.  }
```

63. **GUI Error Dialogs**: Display user-friendly error messages

```
64.  try {
65.      String id = idField.getText();
66.      manager.validateStaffId(id);
67.      // Continue with staff creation
68.  } catch (DuplicateIdException e) {
69.      JOptionPane.showMessageDialog(this,
70.          e.getMessage(),
71.          "Duplicate ID Error",
72.          JOptionPane.ERROR_MESSAGE);
73.  } catch (InvalidInputException e) {
74.      JOptionPane.showMessageDialog(this,
75.          e.getMessage(),
76.          "Input Validation Error",
77.          JOptionPane.WARNING_MESSAGE);
78.  }
```

79. **Comprehensive Error Testing**: Expand JUnit test suite

```
80.  @Test
81.  public void testDuplicateIdThrowsException() {
82.      assertThrows(DuplicateIdException.class, () -> {
```

```
        manager.validateStaffId("D001"); // Assuming D001 already exists
    });
}

@Test
public void testInvalidDateThrowsException() {
    assertThrows(InvalidDateException.class, () -> {
        manager.validateDateOfBirth("2030-12-31"); // Future date
    });
}

@Test
public void testEmptyNameHandling() {
    assertThrows(InvalidInputException.class, () -> {
        manager.validateName("");
    });
}
```

100. **Edge Case Testing**: Test boundary conditions
    - Maximum and minimum numeric values
    - Very long strings
    - Special characters in names
    - System limits (e.g., maximum staff capacity)

Expected Outcome: A robust application that gracefully handles all types of invalid input, provides clear feedback to users, and maintains data integrity under all circumstances.

Week 10 – File Persistence and Data Management

Theme: Data Storage, Retrieval, and Application Lifecycle

Learning Objectives:

- Implement file I/O operations for data persistence
- Understand serialization vs. text-based storage
- Manage application state across program executions
- Handle file-related exceptions and edge cases

Lecture Content:

- Java File API: File, FileReader, FileWriter, BufferedReader, BufferedWriter
- CSV format for structured data storage
- Object serialization and ObjectOutputStream
- Error handling with I/O operations
- Best practices for file path management

Implementation Tasks:

1. **File Operations Interface**: Add methods to HealthCentreManager

```
2. void saveToFile() throws IOException;
3. void loadFromFile() throws IOException;
```

4. **CSV File Format Design**: Define structured format

```
TYPE,ID,NAME,SURNAME,DOB,CONTACT,SPECIFIC_FIELD_1,SPECIFIC_FIELD_2,SPECIFIC_FIELD_3
D,D001,John,Smith,1978-09-12,1234567890,ML12345,Cardiology,15
R,R002,Alice,Brown,1985-02-18,0987654321,Desk-03,40,
```

8. **Save Implementation**: Write staff data to file

```
public void saveToFile() throws IOException {
     try (BufferedWriter writer = new BufferedWriter(new FileWriter("staffdata.csv"))) {
         // Write header
         writer.write("TYPE,ID,NAME,SURNAME,DOB,CONTACT,FIELD1,FIELD2,FIELD3");
         writer.newLine();

         for (StaffMember staff : staffList) {
             if (staff instanceof Doctor) {
                 Doctor doc = (Doctor) staff;
                 writer.write(String.format("D,%s,%s,%s,%s,%s,%s,%s,%d",
                         doc.getId(), doc.getName(), doc.getSurname(),
                         doc.getDob(), doc.getContactNo(),
                         doc.getLicenceNumber(), doc.getSpecialisation(),
                         doc.getConsultationsPerWeek()));
             } else if (staff instanceof Receptionist) {
                 Receptionist rec = (Receptionist) staff;
                 writer.write(String.format("R,%s,%s,%s,%s,%s,%s,%d,",
                         rec.getId(), rec.getName(), rec.getSurname(),
                         rec.getDob(), rec.getContactNo(),
                         rec.getDeskNumber(), rec.getHoursPerWeek()));
             }
             writer.newLine();
         }
     }
 }
```

34. **Load Implementation**: Read staff data from file

```
 public void loadFromFile() throws IOException {
     File file = new File("staffdata.csv");
     if (!file.exists()) {
         return; // No saved data yet
     }

     staffList.clear();

     try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
         reader.readLine(); // Skip header
         String line;

         while ((line = reader.readLine()) != null) {
             String[] parts = line.split(",");
             String type = parts[0];

             if (type.equals("D")) {
                 Doctor doctor = new Doctor(
                     parts[1], // id
                     parts[2], // name
                     parts[3], // surname
```

```
                    parts[4], // dob
                    parts[5], // contact
                    parts[6], // licence
                    parts[7], // specialisation
                    Integer.parseInt(parts[8]) // consultations
                );
                staffList.add(doctor);
            } else if (type.equals("R")) {
                Receptionist receptionist = new Receptionist(
                    parts[1], parts[2], parts[3], parts[4],
parts[5],
                    parts[6], // desk number
                    Integer.parseInt(parts[7]) // hours
                );
                staffList.add(receptionist);
            }
        }
    }
```

Automatic Data Loading: Initialize data on startup

```
 public WestminsterHealthCentreManager() {
    staffList = new ArrayList<>();
    try {
        loadFromFile();
        System.out.println("Successfully loaded " +
staffList.size() + " staff records");
    } catch (IOException e) {
        System.err.println("No previous data found or error
loading: " + e.getMessage());
    }
 }
```

Save on Exit: Implement graceful shutdown

```
 exitButton.addActionListener(e -> {
    int confirm = JOptionPane.showConfirmDialog(this,
        "Save data before exiting?",
        "Confirm Exit",
        JOptionPane.YES_NO_CANCEL_OPTION);

    if (confirm == JOptionPane.YES_OPTION) {
        try {
            manager.saveToFile();
            JOptionPane.showMessageDialog(this, "Data saved
successfully");
            System.exit(0);
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this,
                "Error saving data: " + ex.getMessage(),
                "Save Error",
                JOptionPane.ERROR_MESSAGE);
        }
    } else if (confirm == JOptionPane.NO_OPTION) {
        System.exit(0);
    }
    // Cancel - do nothing, stay in application
});
```

File Exception Handling: Comprehensive error management

```
try {
    manager.saveToFile();
} catch (FileNotFoundException e) {
    JOptionPane.showMessageDialog(this,
```

```
            "Cannot create file. Check folder permissions.",
            "File Access Error",
            JOptionPane.ERROR_MESSAGE);
    } catch (IOException e) {
        JOptionPane.showMessageDialog(this,
            "Error writing to file: " + e.getMessage(),
            "I/O Error",
            JOptionPane.ERROR_MESSAGE);
    }
```

Backup Strategy: Implement data backup feature

```
public void createBackup() throws IOException {
    String timestamp =
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyyMMdd_HHmm
ss"));
    String backupFile = "staffdata_backup_" + timestamp + ".csv";
    Files.copy(Paths.get("staffdata.csv"), Paths.get(backupFile));
}
```

Expected Outcome: Complete data persistence allowing the application to maintain all staff records between program executions, with robust error handling for file operations.

---

6. Final Integration and Testing

By the end of Week 10, students should have completed a professional-grade Health Centre Management System incorporating:

Core Features

- Complete inheritance hierarchy with abstract classes and interfaces
- Dynamic data management using ArrayList and Collections framework
- Dual interface system: console menu and graphical user interface
- Event-driven programming with responsive button actions
- Comprehensive exception handling with custom exception classes
- Persistent data storage using CSV file format
- Unit testing suite validating key functionality

Quality Assurance Checklist

Functionality:

- [ ] Can add both doctors and receptionists with all required fields
- [ ] Can remove staff by ID with confirmation
- [ ] Can view all staff in both console and GUI
- [ ] Can search for specific staff members
- [ ] Can sort staff alphabetically
- [ ] Can calculate and display statistics
- [ ] Data persists between program runs
- [ ] File operations handle errors gracefully

Code Quality:

- [ ] Proper use of inheritance and polymorphism
- [ ] Consistent naming conventions followed
- [ ] Methods are appropriately sized and focused
- [ ] No code duplication
- [ ] Comments explain complex logic
- [ ] Exception handling covers all error cases
- [ ] JUnit tests pass successfully

User Experience:

- [ ] GUI is intuitive and responsive
- [ ] Error messages are clear and helpful
- [ ] Input validation prevents invalid data
- [ ] Confirmation dialogs prevent accidental deletions
- [ ] Table display is organized and readable

---

7. Assessment and Deliverables

Final Submission Package

Students must prepare a comprehensive submission folder containing:

Source Code (`src/` directory):

- All domain model classes (StaffMember, Doctor, Receptionist)
- Manager classes (interface and implementation)
- GUI classes (StaffManagementGUI, StaffTableModel)
- Utility classes (Validator)
- Custom exception classes
- Main application entry point (HealthCentreSystem.java)

Data Files:

- Sample `staffdata.csv` with at least 5 staff records demonstrating both roles

Testing:

- `ValidatorTest.java` with minimum 5 JUnit test methods covering:
  - Input validation (phone numbers, dates, IDs)
  - Duplicate detection
  - Statistics calculation
  - Exception throwing conditions

Documentation:

- UML class diagram showing complete system architecture
- Brief implementation report (2-3 pages) describing:

- Design decisions and rationale
- Challenges encountered and solutions
- How OOP principles were applied
- Testing strategy and results

---

8. Learning Outcomes Mapping

This coursework directly addresses the following program learning outcomes:

LO2 - Object-Oriented Programming Mastery: Students demonstrate comprehensive understanding of abstraction, encapsulation, inheritance, and polymorphism by designing and implementing a multi-layered class hierarchy that solves real-world problems through appropriate use of abstract classes, interfaces, and inheritance relationships.

LO3 - Advanced Java Programming: Students develop modular, maintainable programs using unfamiliar APIs (Swing GUI toolkit) and advanced data structures (ArrayList with generics, Collections framework), demonstrating ability to learn and apply new technologies effectively.

LO4 - GUI Development: Students build professional graphical user interfaces using Java Swing components, implementing custom table models, event handling mechanisms, and responsive user interactions that separate presentation from business logic.

LO5 - Software Quality and Reliability: Students apply comprehensive exception handling, implement file I/O operations for data persistence, and develop unit tests using JUnit framework to ensure program reliability, robustness, and professional quality standards.

---

9. Assessment Alignment Matrix

Skill Area	Weekly Practice Focus	Assessment Criteria
OOP Concepts	Week 5: Inheritance, abstraction, interfaces	Proper use of abstract classes, inheritance hierarchy, polymorphism (LO2)
Collections	Week 6: ArrayList, searching, sorting	Effective use of Collections framework with appropriate data structures (LO3)
GUI Development	Week 7: Swing components, table models	Professional GUI with proper layout, custom table model, clear navigation (LO4)
Event Handling	Week 8: ActionListener, button events	Responsive interface with properly connected event handlers (LO4)
Exception Handling	Week 9: Custom exceptions, validation	Comprehensive error handling with custom exceptions and user-friendly messages (LO5)
File Persistence	Week 10: CSV I/O, data loading/saving	Reliable data persistence across sessions with error handling (LO5)

Skill Area	Weekly Practice Focus	Assessment Criteria
Unit Testing	Week 8-10: JUnit test cases	Effective test coverage validating key functionality (LO5)
Code Quality	Weeks 5-10: Documentation, style	Professional code style, meaningful comments, clear naming conventions (LO3)

---

10. Success Tips and Best Practices

Development Approach

- **Incremental Development**: Complete each week's tasks thoroughly before moving forward
- **Regular Testing**: Test each feature immediately after implementation
- **Version Control**: Save working versions before making major changes
- **Code Reviews**: Review your code regularly for improvements

Common Pitfalls to Avoid

- Skipping input validation leading to runtime errors
- Poor separation of concerns mixing GUI and business logic
- Insufficient exception handling causing ungraceful failures
- Inadequate testing resulting in hidden bugs
- Last-minute implementation without proper testing

Professional Practices

- Write self-documenting code with clear variable and method names
- Keep methods focused on single responsibilities
- Use meaningful comments for complex logic only
- Follow consistent coding style throughout the project
- Design for maintainability and future enhancements

Conclusion

This coursework provides a comprehensive, professionally-oriented learning experience in Object-Oriented Programming and Java application development. By following the structured weekly plan and implementing each component thoughtfully, students will develop not just a functioning application, but a deep understanding of software engineering principles that will serve them throughout their careers.

The reverse engineering approach mirrors real-world scenarios where developers must understand existing systems, infer requirements, and build solutions from conceptual models. Through this authentic experience, students gain confidence in their ability to analyze, design, implement, and test professional software systems.

Remember: the goal is not just to complete the coursework, but to internalize the principles of object-oriented design, develop problem-solving skills, and build a portfolio-worthy project that demonstrates your capabilities as a software developer.