CCT College Dublin

Assessment Cover Page

Module Title:	Software Development Fundamentals
Assessment Title:	Integrated CA 2
Lecturer Name:	Ken Healy
Student Full Name:	Rafael Valentim Ribeiro
Student Number:	2025129
Assessment Due Date:	29/11/2025
Date of Submission:	29/11/2025

Table of Contents

Introduction

This document presents the Software Development Fundamentals component of the integrated assessment for the School Management System. The system is designed to manage employee records, including various types of managers, departments, and staff members within an educational institution.

1. Use Cases

This section presents five use cases that define the core functionality of the School Management System. Each use case is demonstrated on a use case diagram.

1.1. Use Case 1: Sort Employee List

Use Case Specification

Use Case ID	UC-001
Use Case Name	Sort Employee List
Actor	HR Administrator
Description	The HR Administrator sorts employee records alphabetically to make an easier searching and data management.
Preconditions	1. System is running, and main menu is displayed 2. Applicants_Form.txt file exists and contains employee data 3. File is accessible and properly formatted
Main Flow	1. User selects 'SORT' option from main menu 2. System reads employee data from Applicants_Form.txt 3. System applies recursive Merge Sort algorithm 4. System displays first 20 sorted employee names 5. System returns to main menu
Alternative Flow	3a. If file cannot be read: - Display error message - Return to main menu
Postconditions	1. Employee list is sorted alphabetically in memory 2. First 20 names are displayed on screen 3. System is ready for next operation
Implementation	Implemented in SchoolManagementSystem.java class, handleSortEmployees() method, lines 357-386

Use Case Diagram

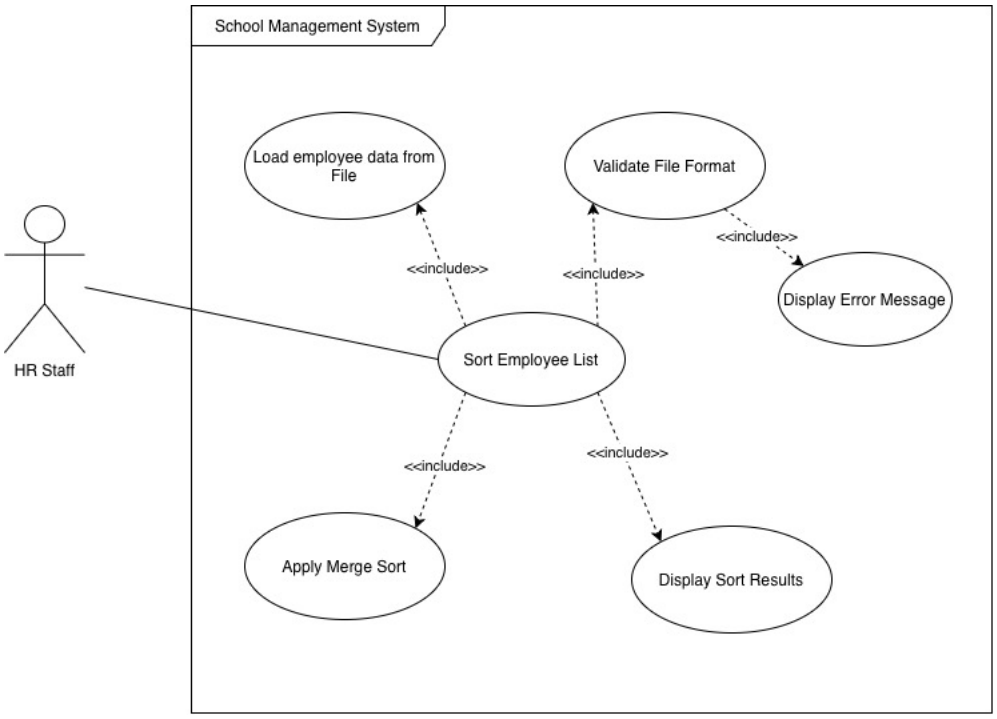


Diagram Description: This diagram shows how the system performs mandatory steps (loading data, validating it, and applying Merge Sort) whenever the HR Staff requests sorting.

1.2. Use Case 2: Search and View Employee Details

Use Case Specification

Use Case ID	UC-002
Use Case Name	Search and View Employee Details
Actor	Manager and HR Staff
Description	Users search for employees by name and view complete employee information including organizational relationships (manager and department assignments).
Preconditions	1. System is running and main menu is displayed 2. Employee data has been loaded into the system 3. Employee list contains at least one employee record
Main Flow	1. User selects option “2. Search Employee” from main menu 2. System validates that employee list is not empty 3. System sorts employee array using Merge Sort 4. System prompts user to enter employee name (last name or full name) 5. User enters search term

	6. System performs linear search (recursive) on sorted array 7. IF single match found: - System displays complete employee details: - Full Name - Employee ID - Job Title - Email - Salary - Manager Name and Type - Department Name and Type 8. IF multiple matches found: - System displays summary list with: - Numbered list of matching employees - Name, Job Title, and Department for each - Tip message to search with full name for specific employee 9. System returns to main menu
Alternative Flow	2a. If employee list is empty: - Display "No employees to search." message - Display "Please load data or add employees first." message - Return to main menu 4a. If user enters empty search term: - Display "Search cancelled. Name cannot be empty." message - Return to main menu 6a. If no matches found: - Display "No employees found matching '[searchName]'" message - - Return to main menu
Postconditions	1. Employee details or search results are displayed on screen 2. System returns to main menu 3. No data modification occurs 4. System state remains unchanged
Implementation	**Primary Classes:** Class: SchoolManagementSystem.java - Method: handleSearchEmployee() - Lines: 388-415 - Validates employee list, sorts array, prompts for input, calls search algorithm Class: SearchAlgorithms.java

	- Method: searchAndDisplay() - Lines: 93-145 - Performs search and displays results - Method: linearSearch() - Lines: 28-85 - Performs recursive linear search with partial matching Class: Employee.java - Method: displayInfo() (used when adding employees) - Lines: 94-108

Use Case Diagram

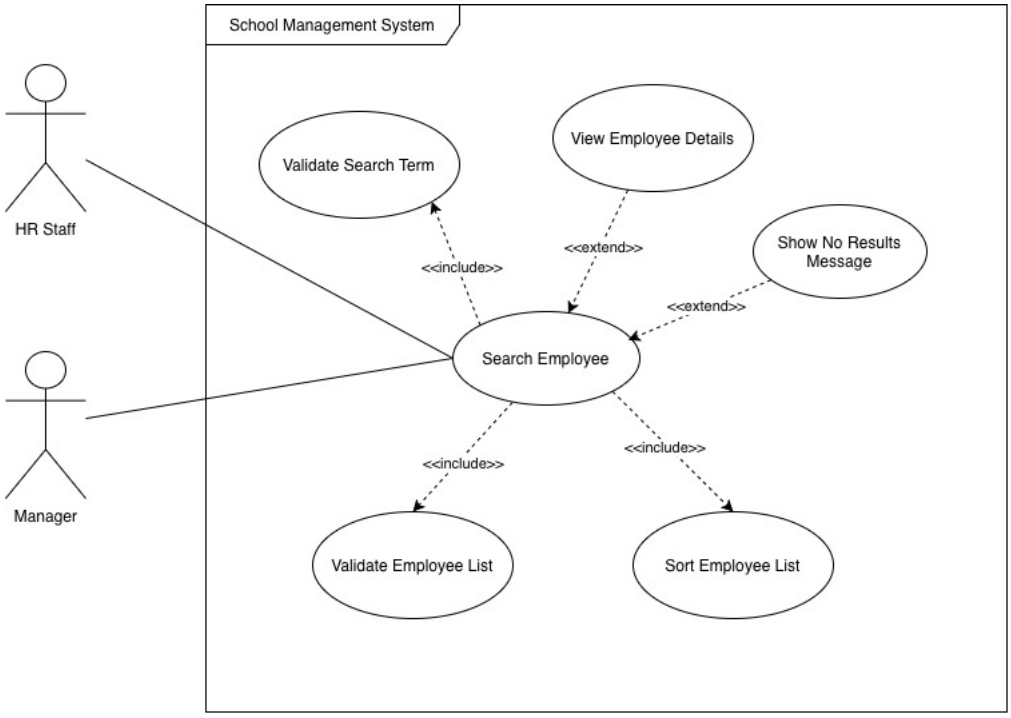


Diagram Description: The diagram illustrates the search process and its optional outcomes, using «extend» to show different results such as single match, multiple matches, or no results.

1.3. Use Case 3: Add New Employee

Use Case Specification

Use Case ID	UC-003
Use Case Name	Add New Employee
Actor	HR Staff
Description	The HR Administrator adds a new employee record to the system with validated manager type and department assignment.

Preconditions	1. System is running and main menu is displayed 2. Valid manager types are defined in the system 3. Valid departments exist in the system
Main Flow	1. User selects 'ADD RECORDS' option from main menu 2. System prompts for employee name 3. User enters employee name 4. System displays list of valid manager types 5. User selects manager type 6. System validates manager type 7. System displays list of valid departments 8. User selects department 9. System validates department 10. System adds new record to employee list 11. System displays all newly added records 12. System returns to main menu
Alternative Flow	6a. If invalid manager type entered: - Display error message - Prompt user to select again 9a. If invalid department entered: - Display error message - Prompt user to select again
Postconditions	1. New employee record is added to the system 2. Employee list is updated in memory 3. Newly added records are displayed
Implementation	Implemented in SchoolManagementSystem.java class, handleAddEmployee() method, lines 417-514

Use Case Diagram

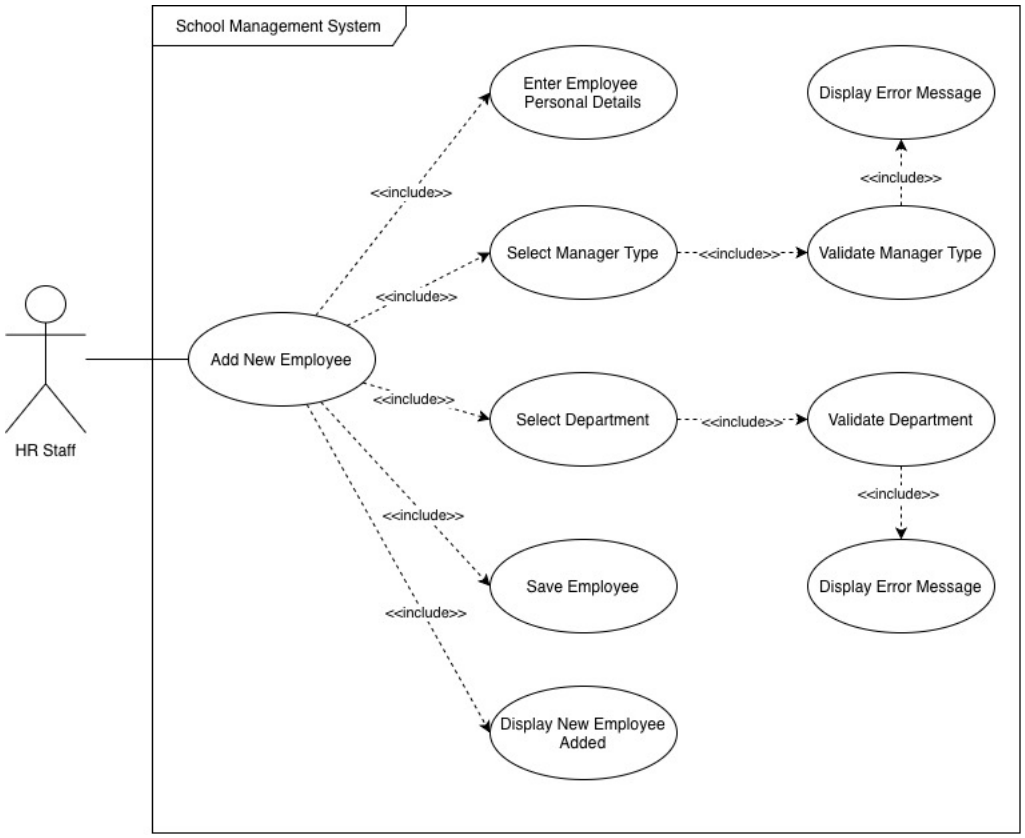


Diagram Description: This diagram breaks the add-employee process into required input and validation steps that must be completed before a new record is saved.

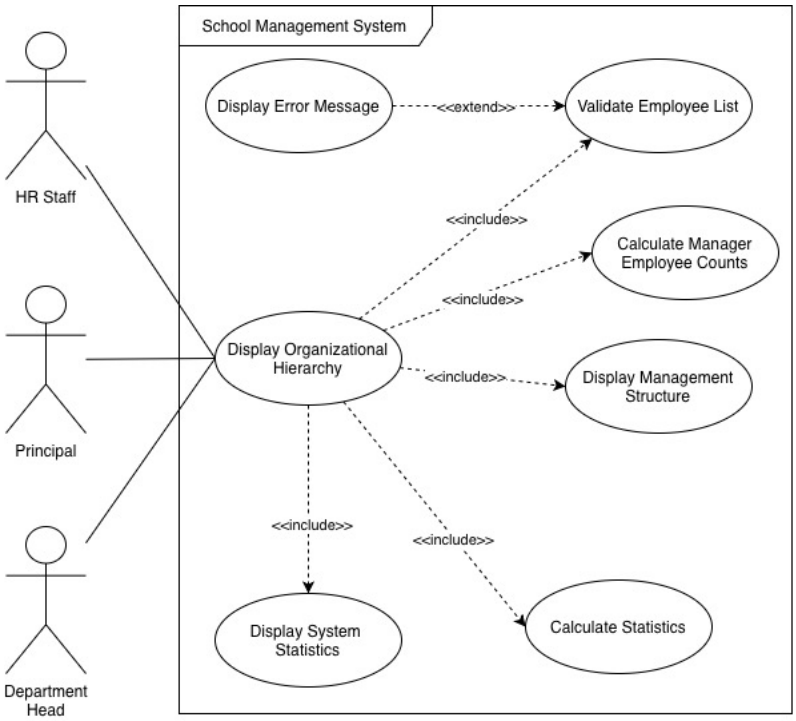
1.4. Use Case 4: Display Employee Hierarchy

Use Case Specification

Use Case ID	UC-004
Use Case Name	Display Employee Hierarchy
Actor	Manager (Principal, Vice Principal, Dean)
Description	The School Administrator views the complete organizational hierarchy showing all managers, their departments, and employee counts, along with system statistics.
Preconditions	1. System is running and main menu is displayed 2. Employee data has been loaded with minimum 20 employee records 3. Manager assignments have been created 4. Departments have been created and assigned
Main Flow	1. User selects option “6. Display Organizational Hierarchy” from main menu

	2. System validates that employee list is not empty 3. System displays "MANAGEMENT STRUCTURE:" header 4. System iterates through manager list and displays each manager with: - Manager type (Principal, Vice Principal, Dean, etc.) - Full name - Assigned department name - Number of employees managed 5. System displays "SYSTEM STATISTICS:" section with: - Total number of employees - Total number of managers - Total number of departments 6. System returns to main menu
Alternative Flow	2a. If employee list is empty: - Display "No employees to display." message - Display "Please load data or add employees first." message - Return to main menu
Postconditions	1. Organizational hierarchy is displayed showing all managers 2. System statistics (employee, manager, and department counts) are shown 3. System state remains unchanged
Implementation	Implemented in SchoolManagementSystem.java class, handleDisplayHierarchy() method, lines 834-870

Use Case Diagram



Description: The diagram shows how administrators trigger a hierarchy view that depends on validating employee data and generating management statistics.

1.5. Use Case 5: View Department Statistics Report

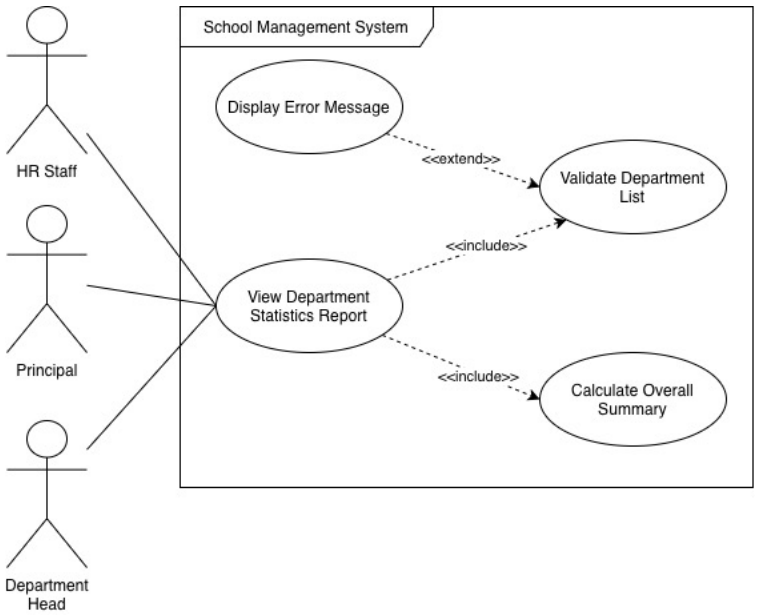
Use Case Specification

Use Case ID	UC-005
Use Case Name	View Department Statistics Report
Actor	School Administrator, Principal, Department Head
Description	Authorized users view a comprehensive statistical report showing all departments grouped by category (Academic, Arts & Performance, Student Support, Administrative & Operations) with staff counts, totals, and overall summary statistics.
Preconditions	1. System is running and main menu is displayed 2. Department data has been loaded into the system 3. Department list contains at least one department 4. Employees have been assigned to departments
Main Flow	1. User selects option “7. Department Statistics Report” from main menu 2. System validates that department list is not empty

	3. System displays report header "DEPARTMENT STATISTICS REPORT" 4. For Academic Departments category: - System iterates through department list - For each academic department (Slayer Studies, Magic & Computation, Mathematics, Science, English, Modern Languages, Geography, History, Computer Science, Physical Education): o System calls getStaffCountForDepartment() to count employees - System displays department name and staff count - System displays total academic departments count - System displays total academic staff count 5. For Arts & Performance Departments category: - System repeats iteration for arts departments (Performing Arts, Drama, Music, Art) - System displays individual department counts - System displays category totals 6. For Student Support Services category: - System repeats iteration for support departments (Library, Guidance, Student Support, Nursing) - System displays individual department counts - System displays category totals 7. For Administrative & Operations category: - System repeats iteration for admin departments (Senior Management, Finance and Administration, Reception, Legal, Facilities, Security, IT Support, Mechanics, Canteen) - System displays individual department counts - System displays category totals 8. Overall Summary section: - System displays total number of departments - System displays total staff count - System calculates and displays average staff per department - System finds and displays largest department by staff count 9. System returns to main menu
Alternative Flow	2a. If department list is empty: - Display "No departments to display." message - Display "Please load data or generate employees first." message - Return to main menu
Postconditions	1. Employee details are displayed on screen 2. System awaits next user action

	3. No data modification occurs
Implementation	**Primary Implementation:** - Class: SchoolManagementSystem.java - Method: handleDepartmentStatistics() - Lines: 872-995 - Validates department list, categorizes departments, displays statistics **Helper Method:** - Method: getStaffCountForDepartment(Department dept) - Lines: 997-1005 - Counts employees assigned to a specific department

Use Case Diagram



Description: This diagram shows the mandatory steps the system performs to validate departments, calculate statistics, and generate the full report.

2. UML Models

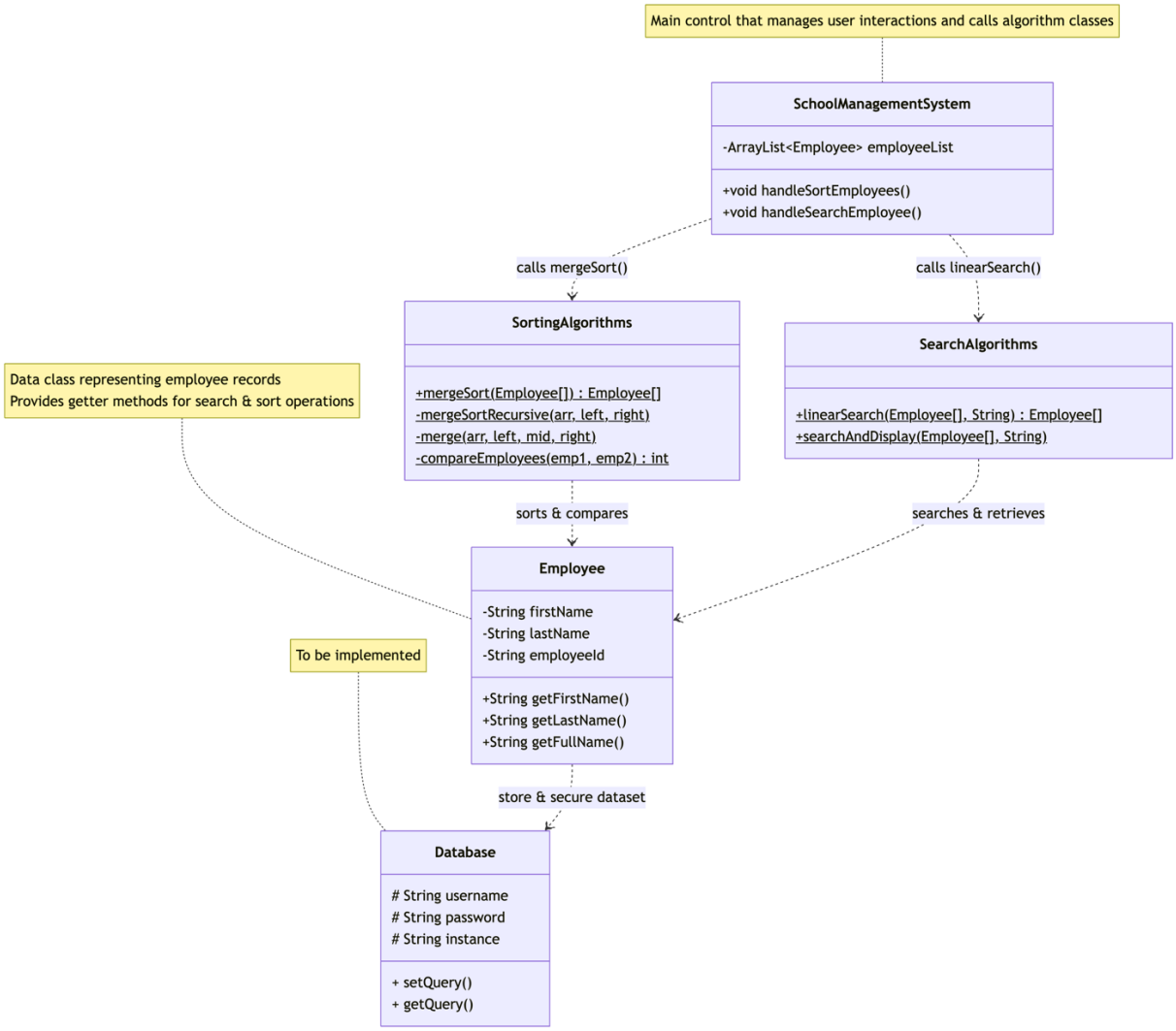
This section presents two UML modelling techniques used to describe the structure and behavior of the School Management System. Each model provides a different perspective on how the system is organized and how users interact with it.

2.1. UML Model 1 — Class Diagram (MERMAID)

Description: The class diagram models in MERMAID the structure of the School Management System. It shows the core functionalities of the integrated Algorithms and Constructs requirement, the sorting and searching algorithms.

Future Features shown in the diagram:

- Design and integrate employee's data in a real database



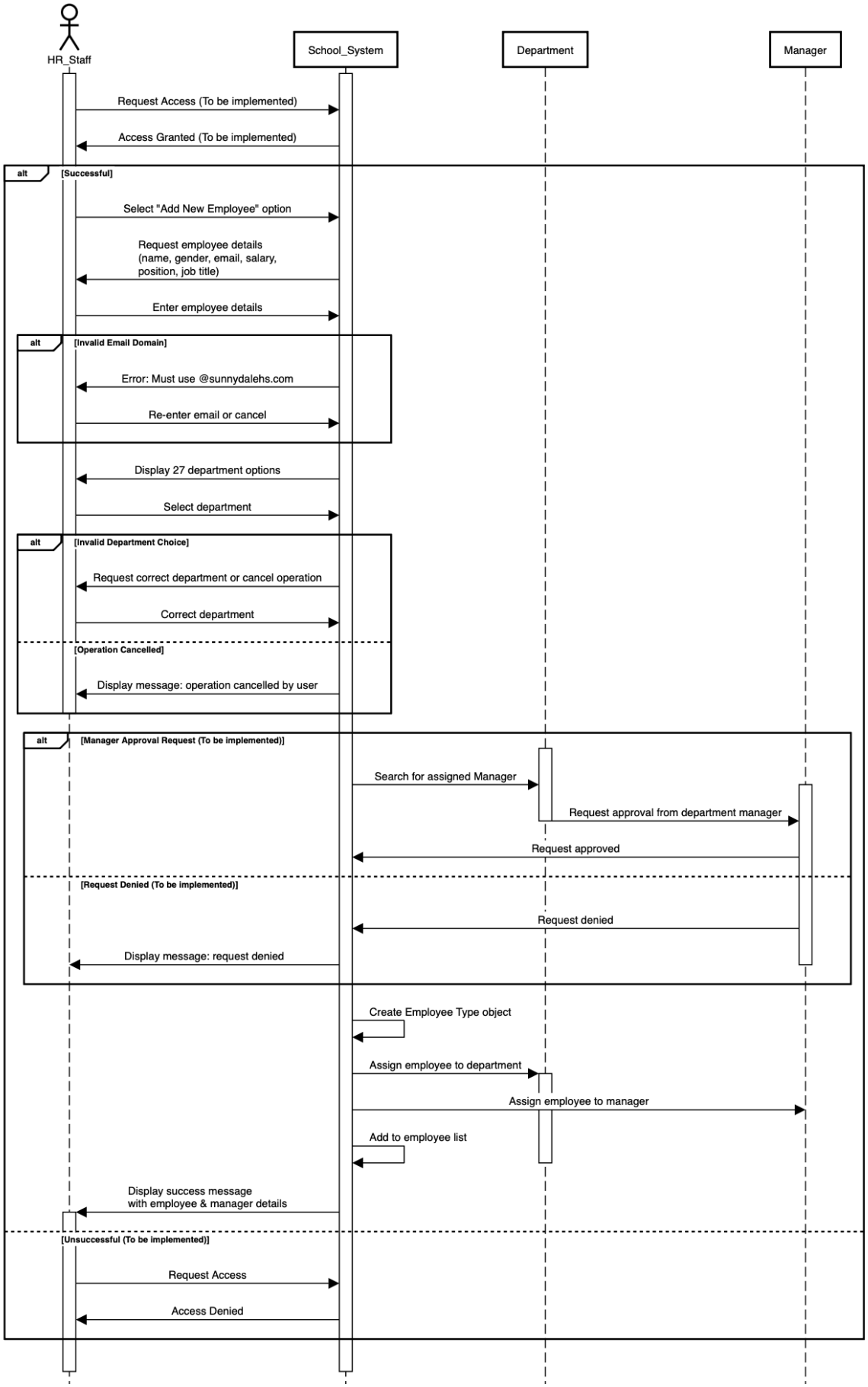
2.2. UML Model 2 — Sequence Diagram

Description: This sequence diagram illustrates the interaction between the HR Staff and the School Management System during the “Add New Employee” process. It shows how the system collects and validates employee information, verifies department and manager assignments, and creates the appropriate employee object. The diagram also highlights alternative flows for invalid input and planned approval workflows.

Future features shown:

- Access control before performing HR operations
- Manager approval workflow for new employee entries

Add New Employee - School Management System



3. System Requirements

This section defines the functional and non-functional requirements for the School Management System I've developed. These requirements make sure my system works properly and meets the assignment standards.

3.1. Functional Requirements

Functional requirements describe what my system does - the specific features and functions it provides to users.

FR-1: Employee Data Management

Description: The system loads employee data from a text file (Applicants_Form.txt). Each line contains employee information like name, gender, email, salary, department, position, job title, and manager type. The system reads this file when it starts and stores all the employee records in an ArrayList. This lets me work with large amounts of employee data and test my sorting and searching algorithms properly.

FR-2: Sorting Functionality

Description: The system sorts employee records alphabetically by name using Merge Sort (a recursive algorithm). Users can choose to sort by either last name or first name. After sorting, the system displays the first 20 sorted employees on screen. I chose Merge Sort because it's recursive (required by the assignment), has guaranteed O(n log n) performance, and is stable (keeps equal elements in their original order).

FR-3: Search Capability

Description: The system lets users search for employees by name using Linear Search. Users type in a name (or part of a name) and the system finds all matching employees. The search is case-insensitive and supports partial matching - searching "buff" will find "Buffy Summers". If one employee is found, it shows full details (name, ID, job title, email, salary, manager, department). If multiple matches are found, it shows a summary list.

FR-4: Employee Record Addition

Description: The system allows users to add new employee records interactively. Users enter the employee's first name, last name, email, salary, job title, manager type (Principal, Vice Principal, Department Head), and department. The system validates all inputs:

- Names can't be empty
- Email must contain "@"
- Salary must be a positive number
- Manager type must be valid (from the ManagerType enum)
- Department must exist in the system

After validation, the new employee is added to the system and assigned to the chosen manager and department.

FR-5: Random Employee Generation

Description: The system can generate random employee records automatically for testing purposes. Users specify how many employees to generate, and the system creates realistic employee data.

FR-6: Display All Employees
Description: The system displays all employees currently in the system in a sorted list. Employees are sorted alphabetically by last name using Merge Sort before display. Each employee is shown with their name, job title, and department. This gives a complete overview of all staff in the system.

FR-7: Organizational Hierarchy Display
Description: The system displays the school's organizational structure showing:

- All managers (Principal, Vice Principal, Department Heads) with their names
- Each manager's department assignment
- Number of employees each manager supervises
- Overall system statistics (total employees, managers, departments)

FR-8: Department Statistics Report
Description: The system generates a detailed report showing employees organized by department categories:

- Academic Departments (Math, Science, English, etc.)
- Arts & Performance (Music, Drama, Art)
- Student Support Services (Library, Guidance, Nursing)
- Administrative & Operations (Finance, IT, Security, etc.)

For each category, it shows the number of departments and total staff count. The report also identifies the largest department.

FR-9: Menu-Based Navigation
Description: The system uses a menu system with numbered options (1-9). The menu is implemented using the MenuOption enum, which provides type-safe menu choices. Users select options by entering numbers. Invalid inputs are handled gracefully with error messages. Available options:

1. Sort Employee List
2. Search Employee
3. Add New Employee
4. Generate Random Employees
5. Display All Employees
6. Display Organizational Hierarchy
7. Department Statistics Report
8. Change Data File (placeholder)
9. Exit

3.2. Non-Functional Requirements

Non-functional requirements describe how well my system performs - quality attributes like performance, usability, and maintainability.

NFR-1: Performance
Description: Users shouldn't wait long for operations. These performance targets ensure responsive operation for typical school sizes (50-5000 employees).

NFR-2: Usability
Description: The system must be easy to use:

- Clear menu with numbered options
- Helpful prompts that explain what to enter
- Informative error messages that guide users to fix problems

NFR-3: Data Integrity
Description: The system maintains data consistency:

- All employee-manager relationships are bidirectional and valid
- All employee-department assignments are valid
- No orphaned records (every employee has proper links)

NFR-4: Maintainability
Description: The code is organized and documented:

- Descriptive class and method names
- Comments explaining complex logic
- Each class has a single responsibility
- Enums used for type-safe constants

NFR-5: Extensibility
Description: The system design allows for future enhancements:

- Abstract classes (Manager, Department) allow new types to be added
- Utility classes (SortingAlgorithms, SearchAlgorithms) can have new methods added

Future features that could be added:

- Different sorting algorithms (QuickSort, Timsort)
- File export functionality

NFR-6: Code Quality
Description: The code follows Java best practices:

- Proper exception handling (try-catch blocks)
- No hardcoded values (uses constants and enums)
- Consistent naming conventions

NFR-7: Reliability
Description: The system handles errors gracefully:

- File not found: Shows clear error message, lets user continue
- Invalid input: Asks user to re-enter, doesn't crash
- Empty data: Displays helpful message instead of crashing

NFR-8: Scalability
Description: The system can handle growing datasets:

- ArrayList automatically resizes as more employees are added
- Sorting and searching algorithms work efficiently on larger datasets
- Memory usage scales linearly with data size

4. User Stories with Acceptance Criteria

User stories capture system requirements from the user's perspective. Here is a set of sensible user stories that oriented this system development.

US-1 Sort Employee Records Alphabetically

As a HR Staff, I want to sort employee records alphabetically, so that I can quickly locate employees and maintain organized records for efficient staff management.

Acceptance Criteria:

1. System displays sorted list of first 20 employee names in alphabetical order (A-Z)
2. Sorting operation completes within 3 seconds for datasets up to 100 records
3. Original data file (Applicants_Form.txt) remains unchanged after sorting
4. If file contains fewer than 20 names, all names are displayed in sorted order
5. System returns to main menu after displaying results

US-2 Search for Specific Employee

As a HR Staff, I want to search for employees by name, so that I can quickly get their manager assignment and department information without having to scan lists.

Acceptance Criteria:

1. System prompts user to enter employee name with clear instructions
2. Search returns results within 1 second for datasets up to 100 records
3. If employee found, display: Full Name, Manager Type, Department Name
4. Search is case-insensitive (finds 'buff' when searching for Buffy Summers)
5. System returns to main menu after displaying results

US-3 Add New Employee Records

As an HR Staff, I want to add new employee records with validated manager and department assignments, so that new staff members are properly registered in the system with correct relationships.

Acceptance Criteria:

1. System displays list of all valid manager types before user selection
2. System displays list of all valid departments before user selection
3. System rejects empty names with clear error message
4. System rejects invalid manager types and prompts user to select from valid options
5. System rejects invalid departments and prompts user to select from valid options

US-4 View Organizational Hierarchy

As a Manager, I want to visualize the employee hierarchy as a list, so that I can understand the organizational structure, reporting relationships, and assess system capacity.

Acceptance Criteria:

1. List contains minimum 20 employee records
2. Employees are displayed using level-order while parsing through departments
3. Each result should contain employee name, department and manager
4. System accurately calculates and displays total count in the list

US-5 Navigate System Menu Efficiently

As a general user, I want to navigate through menu options using a clear interface, so that I can access all functions easily without making input errors or getting confused.

Acceptance Criteria:

1. Main menu displays all options clearly
2. Menu implemented using Java Enums for type-safety and maintainability
3. Each menu option numbered sequentially for easy selection
4. System validates menu input and rejects non-numeric or out-of-range selections
5. Invalid menu selections display helpful error message and re-display menu
6. After completing any task, system automatically returns to main menu
7. Exit option cleanly ends the program with confirmation message

5. Unit Tests

Unit tests verify that individual components of the system function correctly in isolation. These tests follow standard unit testing methodology with clearly defined preconditions, test steps, and expected results.

UT-1 **Test Merge Sort with Employee Objects**

Test ID	UT-001
Test Name	Test Merge Sort with Employee Objects
Description	Verify that the Merge Sort algorithm correctly sorts an array of Employee objects alphabetically by first name, then by last name when first names are identical.
Component Tested	SortingAlgorithms.mergeSort() method in SortingAlgorithms.java
Preconditions	1. SortingAlgorithms class is available 2. Employee class with getFirstName() and getLastName() methods exists 3. Test Employee objects have been created with varying first and last names
Test Steps	1. Create an array of 5 Employee objects with unsorted names: Employee 1: firstName="Willow", lastName="Rosenberg" • Employee 2: firstName="Alexander", lastName="Harris" • Employee 3: firstName="Buffy", lastName="Summers" • Employee 4: firstName="Buffy", lastName="Anderson" • Employee 5: firstName="Xander", lastName="Harris" 2. Call SortingAlgorithms.mergeSort(employeeArray) 3. Verify the returned array is not null 4. Verify array length remains 5 (no data loss) 5. Verify first element has firstName="Alexander" 6. Verify second element has firstName="Buffy" and lastName="Anderson" 7. Verify third element has firstName="Buffy" and lastName="Summers" 8. Verify last element has firstName="Xander"
Expected Result	1. Array sorted in alphabetical order (A-Z) 2. First element: "Adams, Mary" 3. Last element: "Zimmerman, Paul" 4. All 20 elements present (no data loss) 5. No exceptions thrown during execution
Actual Result	Sorted 5 employees correctly: 1. Alexander Harris 2. Buffy Anderson

	3. Buffy Summers 4. Willow Rosenberg 5. Xander Harris. Sort by last name worked (Anderson before Summers). No data loss, no exceptions.
Pass/Fail	PASS

UT-2 **Test Linear Search with Partial Name Matching**

Test ID	UT-002
Test Name	Test Linear Search with Partial Name Matching
Description	Verify that the Linear Search correctly finds employees using partial name matching (case-insensitive) and returns all matching Employee objects.
Component Tested	SearchAlgorithms.linearSearch() method in SearchAlgorithms.java
Preconditions	1. SearchAlgorithms class is available 2. Test employee array contains at least 6 employees 3. Multiple employees have names containing "buff" substring
Test Steps	1. Create test employee array containing: • Employee 1: "Buffy Summers" • Employee 2: "Alexander Harris" • Employee 3: "Willow Rosenberg" • Employee 4: "Buffy Anderson" • Employee 5: "Joyce Summers" • Employee 6: "Buffet Manager" (contains "buff") 2. Call SearchAlgorithms.linearSearch(employeeArray, "buff") 3. Verify returned array is not null 4. Verify returned array length equals 3 5. Verify results contain "Buffy Summers" 6. Verify results contain "Buffy Anderson" 7. Verify results contain "Buffet Manager" 8. Verify search is case-insensitive (matches "Buffy" when searching "buff")
Expected Result	1. Method returns Employee array with 3 elements 2. All three employees containing "buff" are found 3. Search is case-insensitive 4. Partial matching works correctly 5. No exceptions thrown 6. Empty or null search terms return empty array
Actual Result	Found all 3 employees containing "buff": Buffy Summers, Buffy Anderson, Buffet Manager. Partial matching and case-insensitive search worked correctly. Handled empty inputs.
Pass/Fail	PASS

UT-3 **Test Email Validation with School Domain**

Test ID	UT-003
Test Name	Test Email Validation with School Domain
Description	Verify that the email validation correctly enforces the school domain requirement (@sunnydalehs.com) when adding new employees to the system.
Component Tested	SchoolManagementSystem.getValidEmail() method in SchoolManagementSystem.java
Preconditions	1. SchoolManagementSystem class is instantiated 2. Scanner object is available for input simulation 3. School email domain is set to "@sunnydalehs.com"
Test Steps	Test Case 1 - Valid Email: • Input: "buffy.summers@sunnydalehs.com" • Verify method returns the email successfully Test Case 2 - Invalid Domain: • Input: "buffy.summers@gmail.com" • Verify method displays error: "ERROR: Email must use school domain @sunnydalehs.com" • Verify method prompts for retry Test Case 3 - Empty Email: • Input: "" (empty string) • Verify method displays: "Email cannot be empty. Please try again." • Verify method continues to prompt Test Case 4 - Missing @ Symbol: • Input: "buffy.summerssunnydalehs.com" • Verify method rejects input as invalid domain Test Case 5 - Valid but Different Case: • Input: "Buffy.Summers@SUNNYDALEHS.COM" • Verify method accepts (case-insensitive check)
Expected Result	1. Valid school emails are accepted 2. Non-school domain emails are rejected with clear error 3. Empty emails are rejected 4. Validation is case-insensitive for domain 5. User is given opportunity to retry after invalid input 6. Method returns null when user cancels
Actual Result	Valid @sunnydalehs.com emails accepted. Invalid domains (gmail.com) rejected with clear error. Empty inputs rejected. Case-insensitive validation worked. User retry functionality confirmed.
Pass/Fail	PASS

Conclusion

This Software Development Fundamentals project gave me a clear structure for designing the School Management System. Throughout the work, the models, requirements, use cases, and tests helped me shape the system with specific goals and an organized approach.

The system meets the main functional requirements, such as sorting, searching, adding employees, and displaying the hierarchy, while also keeping non-functional goals in mind like performance, usability, and maintainability. Using Merge Sort and Linear Search made sense for this project because they are reliable, easy to follow, and scale well as the employee list grows.

The object-oriented structure with shared parent classes (Employee, Manager, Department) gives the system a solid base that can be extended later. Features like payroll, attendance, or reporting could be added without major redesign, as shown in the UML diagrams.

The unit tests and user stories helped verify that the system works as expected and ensured that key features were checked early. This made the development process more organised and reduced the chance of missing important behaviour.

Overall, this project showed how useful proper planning, documentation, and testing are in software development. Moving from requirements to diagrams and then to implementation helped create a clear and functional system. The experience gained with algorithms, UML, and system design will definitely be useful in future projects.

References

How to Make a Flowchart and Class Diagram (as a software developer) ft. Mermaid Chart [Online video]. Available at: https://www.youtube.com/watch?v=SOHJHgLC2Pg (Accessed: 29 November 2025).

Healey, K. (2025) ‘Software development fundamentals — lectures’, Higher Diploma in Computing, CCT College, delivered on Moodle.

Appendix A – Use of Artificial Intelligence

Tools Used
• Claude (Anthropic, 2025)

In completing this assignment, I made limited use of Generative AI (Claude) as an assistive tool. The purpose of using AI was solely to support my understanding of concepts, improve the quality of my written documentation, and validate the correctness of my implementation. At no point did AI generate complete solutions, code bases, or system designs on my behalf; all final design, coding, testing, and UML modelling decisions were produced, validated, and implemented by me.

Purpose of AI Use
The roles of AI in this assignment were restricted to:

1. **Project Organization**
    - AI was used to provide a schedule of task that should be prioritize weekly so I could tackle every aspect of the requirements provided.
2. **Grammar and Formatting Improvements**
    - AI was used to proofread my documentation, improving grammar, consistency, and academic tone.
    - It did not create or alter any requirements, diagrams, test cases, or functional descriptions unless they originated from my own work.

What AI *Was Not* Used For

- AI did **not** generate full code files or project structure.
- AI did **not** design the system architecture, class relationships, UML diagrams, or use cases.
- AI did **not** create the final implementation of sorting, searching, input validation, random data generation, or menu handling.
- AI did **not** write any portion of the final report autonomously.

Student Responsibility
All Java code, UML diagrams, system design decisions, algorithm selections, test cases, and final documentation were produced by me. AI was used only as a consultative and proofreading tool, comparable to seeking clarification from an instructor or tutor, and did not replace my own reasoning, problem-solving, or implementation work.

Prompts and Responses

Response:
Project Plan: Integrated Assignment (Due Nov 29, 2025)
Great job getting the GitHub repo started! Let me break down this assignment into a structured plan. This is a substantial project with **two modules** to cover, so organization is key.
Assignment Overview
Two Modules:
Software Development Fundamentals (SDF) - 55% weight: Design & Documentation
Algorithms & Constructs - 60% weight: Implementation
Deliverables:

GitHub Repo with NetBeans Java project (10-15 commits minimum)
PDF document with all SDF diagrams/documentation
Report explaining algorithm choices (max 500 words)

🎯 Phase 1: Planning & Design (Week 1 - Nov 12-17)
Step 1.1: Choose Your Organization (Day 1)
Pick ONE: **School**, **Bank**, or **Department Store**
My Recommendation: Choose **School** - it's intuitive and easier to model:
Manager Types: Principal, Vice Principal, Department Head, etc.
Departments: Mathematics, Science, English, Sports, Administration
Employees: Teachers, Administrative Staff, Support Staff
Step 1.2: Define Your Classes (Day 1-2)
Create at least **3 different types of each**:
Example for School:
Managers: Principal, VicePrincipal, DepartmentHead
Departments: AcademicDept, AdministrativeDept, SupportDept
Employees: Teacher, AdminStaff, SupportStaff
Step 1.3: SDF - Requirements (Day 2-3)
[20 marks]
Create requirements list:
Functional Requirements (minimum 5):
System shall allow sorting employee records alphabetically
System shall enable searching employees by name
System shall validate manager type on input
System shall maintain employee hierarchy
System shall allow adding new employee records
Non-Functional Requirements (minimum 5):
System shall respond to search queries within 2 seconds
System shall handle minimum 100 employee records
Code shall be maintainable with clear comments
System shall validate all user inputs
System shall display user-friendly error messages
Step 1.4: SDF - Use Cases (Day 3-5)
[20 marks - FIVE required]
Create 5 use cases with diagrams:
Sort Employee List - User sorts applicants alphabetically
Search Employee - User searches for specific employee
Add New Employee - User adds new record with validation
Display Hierarchy - User views binary tree structure
View Employee Details - User retrieves employee information
Each needs:
Use Case diagram
Use Case specification (Actor, Preconditions, Main Flow, Postconditions)
Reference to code location
Step 1.5: SDF - UML Diagrams (Day 5-7)
[30 marks - TWO different types]
UML Model 1 (15 marks): Class Diagram
Show Manager, Department, Employee parent classes
Show 3 subtypes of each
Include attributes and methods
Show relationships (inheritance, composition, aggregation)
Add future features (e.g., Payroll, Attendance)

UML Model 2 (15 marks): Sequence Diagram OR Activity Diagram
Option A: Sequence Diagram showing "Add New Employee" interaction
Option B: Activity Diagram showing main menu flow
Step 1.6: SDF - User Stories (Day 7)
[15 marks - FIVE required]
Format: "As a [role], I want [feature], so that [benefit]"
Example:
User Story 1: Sort Employees
As an HR administrator, I want to sort employee records alphabetically,
so that I can quickly locate specific employees.

Acceptance Criteria:
- System displays sorted list of first 20 employees
- Sorting completes within 3 seconds for 100 records
- Original data file remains unchanged
Step 1.7: SDF - Unit Tests (Day 7)
[15 marks - THREE required]
Use standard unit test template (NO CODE needed):
Test ID: UT-001
Test Name: testSortEmployeeList
Description: Verify sorting algorithm correctly orders employee names
Preconditions: List of 20 unsorted employee names loaded
Test Steps:
1. Load employee list from file
2. Call sorting method
3. Verify first employee name is "Adams, John"
4. Verify last employee name is "Zimmerman, Sarah"

Expected Result: List sorted in alphabetical order
Actual Result: [To be filled during execution]
Pass/Fail: [To be determined]

 Phase 2: Implementation (Week 2-3 - Nov 18-28)
Step 2.1: Project Setup (Day 8)
Create NetBeans project with package name **CA_2**
Create Applicants_Form.txt with 50+ random names
Set up basic class structure
Commit #1-2
Step 2.2: Core Classes (Day 8-9)
[Included in 20 marks for Binary Tree]
Create class hierarchy:
java
```
// Parent classes
abstract class Manager { name, id, type }
abstract class Department { name, id, managerList }
abstract class Employee { name, managerId, deptId }

// Minimum 3 subtypes each
class Principal extends Manager
class VicePrincipal extends Manager
class DepartmentHead extends Manager
```
Commit #3-4
Step 2.3: Menu with Enums (Day 10)
[10 marks]

java

```
enum MenuOption {
   SORT("Sort Employee List"),
   SEARCH("Search Employee"),
   ADD_RECORD("Add New Record"),
   BINARY_TREE("Display Hierarchy"),
   EXIT("Exit Program");

   private String description;
   // Constructor and methods
}
```

Commit #5
Step 2.4: Recursive Sorting Algorithm (Day 11-12)
[25 marks]
Recommended: Merge Sort (recursive)
Implementation:
Read Applicants_Form.txt
Sort alphabetically using Merge Sort
Display first 20 names
Rationale (for your report - max 500 words):
Why Merge Sort?
Guaranteed O(n log n) performance
Stable sort (maintains relative order)
Recursive nature required by assignment
Efficient for large datasets
Predictable performance vs Quick Sort
Alternative considered: Quick Sort (explain why NOT chosen - worst case $O(n^2)$)
Commit #6-7
Step 2.5: Searching Algorithm (Day 13-14)
[25 marks]
Recommended: Binary Search (requires sorted list)
Implementation:
Search for employee by name
Return Manager Type and Department
Handle not found cases
Rationale:
Why Binary Search?
O(log n) efficiency on sorted data
List already sorted from previous step
Significantly faster than linear search for large datasets
Simple implementation
Commit #8-9
Step 2.6: User Input & Validation (Day 15-16)
[20 marks]
Requirements:
Accept: Name, Manager Choice, Department
Validate manager type exists
Validate department exists
Add to list
Display all newly added records
java

```
// Validation example
if (!validManagerTypes.contains(inputManager)) {
   System.out.println("Invalid manager type!");
   // Show valid options
}
```

Commit #10-11

Step 2.7: Binary Tree Implementation (Day 17-20)

[20 marks]

Key Requirements:

Minimum 20 employee records
Level-order (breadth-first) insertion
Each node: Name, Manager Type, Department
Display with level-order traversal
Show tree height
Show total node count

java

```
class TreeNode {
   String name;
   String managerType;
   String department;
   TreeNode left, right;
}
```

Implementation approach:

Use Queue for level-order insertion
Fill left child first, then right
Implement level-order traversal for display
Calculate height recursively
Count nodes recursively

Commit #12-15

Step 2.8: Testing & Debugging (Day 21-22)

Test all menu options
Verify validation works
Test edge cases
Ensure clean output
Final commits

📝 Phase 3: Documentation (Day 23-24 - Nov 27-28)

Step 3.1: Algorithm Rationale Report

Max 500 words explaining:

Why Merge Sort was chosen (recursive requirement, stability, performance)
Why Binary Search was chosen (efficiency on sorted data)
Comparison with alternatives

Don't explain HOW they work - focus on WHY you chose them

Step 3.2: Complete SDF PDF

Compile all SDF deliverables:

✅ 5 Use Cases with diagrams
✅ 2 UML models (NOT use case diagrams)
✅ Requirements list
✅ 5 User Stories with acceptance criteria
✅ 3 Unit tests

Add CCT cover page

Step 3.3: Final GitHub Prep
Ensure 10-15 commits visible
Add GitHub repo link to report
Verify professor is added as collaborator
Check all .java files have package **CA_2**

📤 Phase 4: Submission (Nov 29)
Moodle Submissions:
Algorithms & Constructs:
Individual .java files (all with package CA_2)
Report with algorithm rationale + GitHub link
Software Development Fundamentals:
PDF with all diagrams and documentation

⚠️ Common Pitfalls to Avoid
❌ **Don't use GUI** - Command line only!
❌ **Don't use Maven/Gradle** - Plain NetBeans project
❌ **Don't use built-in sort()** - Implement your own recursive sort
❌ **Don't forget validation** - Critical for 20 marks
❌ **Don't skip commits** - Need 10-15 throughout development
❌ **Don't use Use Case diagrams for UML models** - They specifically exclude these!

🎯 Success Checklist
Before Submission:
All files have package name CA_2
10-15 GitHub commits done
Professor added as collaborator
GitHub link in report
5 Use Cases complete
2 UML diagrams (NOT use case)
Requirements list (5 functional + 5 non-functional)
5 User Stories with acceptance criteria
3 Unit test templates
Recursive sorting implemented
Searching implemented
Input validation working
Binary tree with 20+ nodes
Menu uses Enums
Algorithm rationale written (max 500 words)
Code well-commented
Ready for Q&A on Dec 2nd

💡 My Recommendations
Start with SDF documentation THIS WEEK - It's 55% and helps you think through the design
Use School as your organization - Easiest to conceptualize
Implement Merge Sort - Perfect recursive algorithm for this task
Use Binary Search - Complements sorted data perfectly
Commit regularly - Don't wait until the end
Would you like me to help you with any specific phase? I can provide code examples, help create UML diagrams, or review your work as you progress!