Workforce CLI — System Modelling & Build Report

Object-Oriented Design, UML Modelling, and Testing Documentation

Project Overview

This document presents the **system architecture, UML models, user stories, and testing framework** for the **Workforce CLI Application**, a Java-based console program developed to demonstrate **object-oriented programming, recursion, and algorithmic techniques** for organizational data management.

It includes detailed **use case diagrams, class and sequence models**, and **unit test scenarios** aligned with real-world system analysis and software design principles.

Contents

Section 1: System and Use Case Planning

1. System Overview

The Dream Tech Company System is designed to support core internal operations like managing employee records, sorting, searching, and department-role alignment. While the current system runs on a command-line interface, it's been built with scalability in mind—future versions could include user accounts, web interfaces, and smart reporting tools. The system's structure reflects both current functionality and planned upgrades based on real-world organizational workflows. For the system boundary and its' elements (IBM, 2023) clear explanations helped.

2. Modelling Tools Used

All UML diagrams in this report were created using https://draw.io a free, browser-based modelling tool that supports all standard UML shapes and notations. The Class Diagram was designed using this tool to illustrate the system's structure, including inheritance, associations, enums, and planned future components.

The Use Case Diagrams were also created using draw.io and follow the standard UML notation format for actors, system boundaries, and use case ovals. Relationships such as <<use >> and <<extend>> are used where applicable.

The Sequence Diagrams were developed using https://sequencediagram.org , an online tool specifically designed for creating clear and structured UML sequence diagrams. It allowed for easy modelling of control flow, object creation/destruction, and alternate paths (alt) using a text-based syntax that rendered into visual diagrams.

3. System Boundary

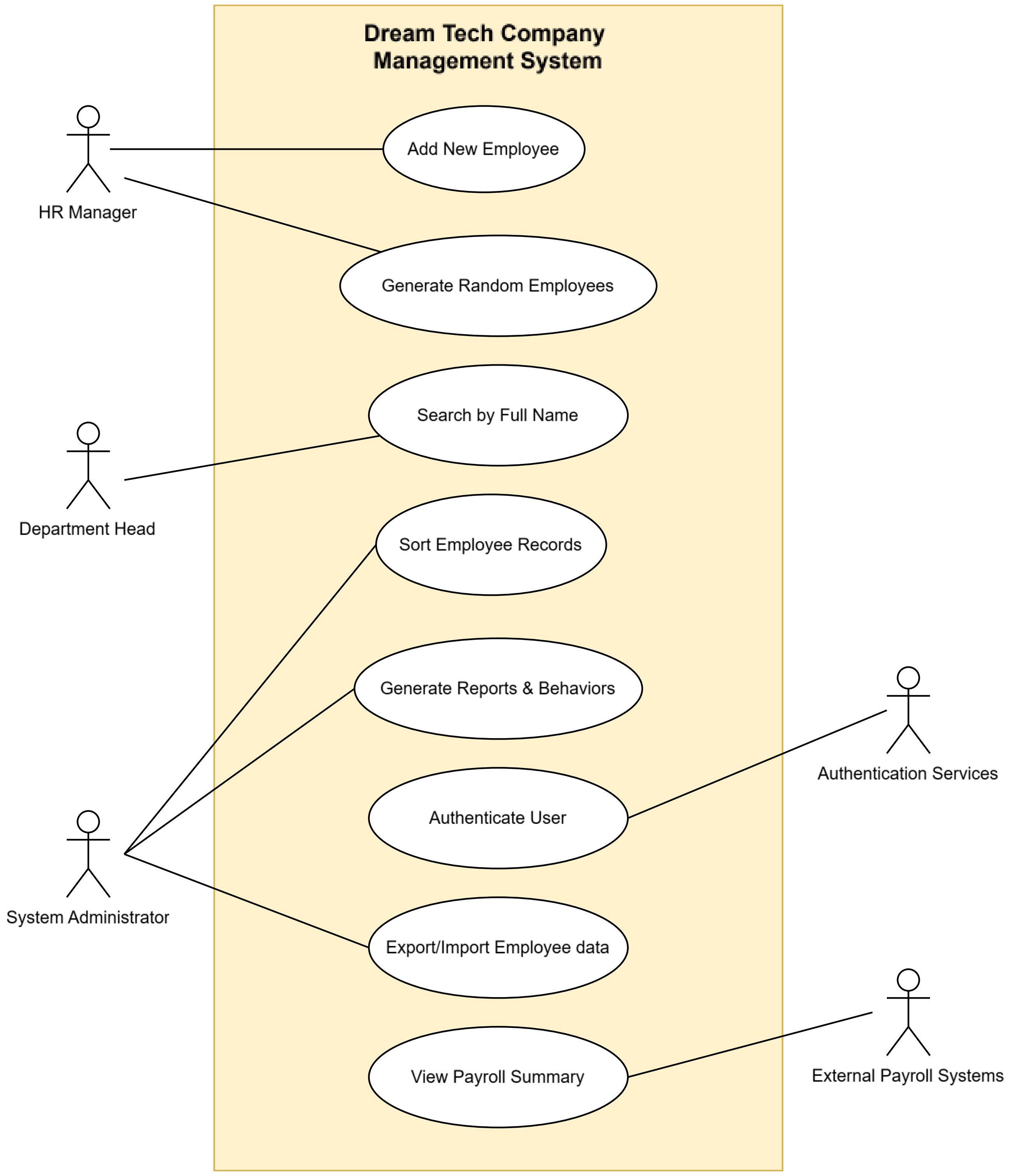
1. Description

The Dream system includes employee creation (manual or random), sorting and searching employee data, assigning roles and departments, and generating behavior-based summaries. It's designed to grow into a full enterprise tool with onboarding, role-based access, external HR integrations, and reporting features.

2. External Elements

- ➢ HR Manager
- ➢ Department Head
- ➢ System Administrator
- ➢ External Payroll System
- ➢ Authentication Service

3. Boundary Diagram



4. Actors in the System

Actor	Description
HR Manager	Responsible for manually adding employees or generating random ones for testing and onboarding. Represents a real-world HR role in both CLI and future GUI versions.
Department Head	Searches for specific employees by name or role. Will eventually manage project assignment and team composition in future upgrades.
System Administrator	Handles tasks like sorting records and checking data structure integrity. Could later manage access permissions and system-level operations.
External Systems	Represents services like payroll APIs, authentication tools, and third-party integrations that will be incorporated in future iterations of Dream.

5. Use Case Planning

This section presents the five main use cases designed for the Dream Company Management System. These reflect the real interactions that staff (like HR or system admins) would carry out in the organization. Even though the current version is console-based, the use cases are written to reflect full system functionality in future builds—like dashboards, access control, and team analytics.

Use Case ID	Role-Based Actor	Notes
CCT-TC-UC01	HR Manager	Manual record entry
CCT-TC-UC02	HR Manager / System Tester	Dummy data generation
CCT-TC-UC03	System Administrator	Data visibility and integrity
CCT-TC-UC04	Department Head	Employee lookup
CCT-TC-UC05	All Users (mainly Admin & HR)	Reporting and behavior testing

Each use case will be documented using the standard Use Case Narrative Template (as provided by CCT) and will be accompanied by a Use Case Diagram. Where applicable, a reference will be provided to the Java code (in the CA_2 package) that implements the use case. This use case format is adapted from CCT's official use case template (Healy, 2025), and extended with common practice elements recommended by (Cockburn, 1998).

Section 2: Use Case Narratives & Diagrams

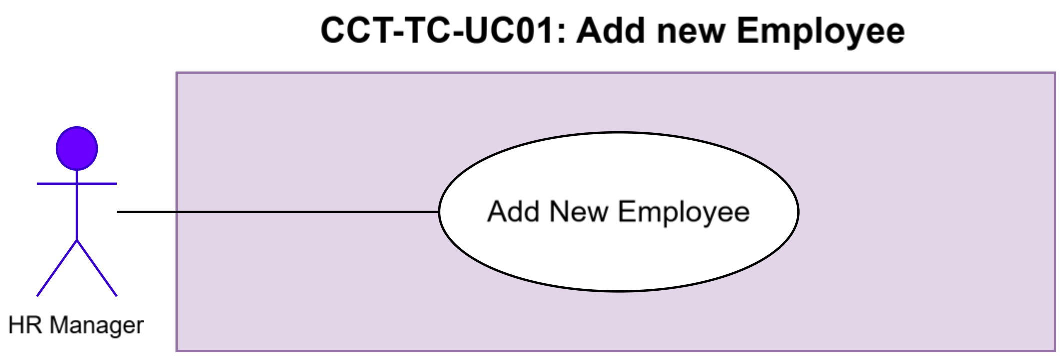
1. Use Case 1: Add New Employee

1. Use Case Specification (CCT-TC-UC01)

Use Case ID	CCT-TC-UC01	
Use Case Title	Add New Employee (Manual Entry)	
Use Case Goal	To manually input and save a new employee's details into the system.	
Description (or Overview)	This use case allows a user to input personal and employment details for a new employee, validate the data, create the corresponding object, and save it to the system and CSV file. It also models how an HR Manager manually adds a new employee. Currently, this action is performed through a command-line interface, but the system is structured to support future enterprise versions with role-based access.	
Dependencies	None	
Trigger	User selects "ADD > Manual Entry" from the main CLI menu	
Precondition(s)	User has launched the CLI program A valid CSV file was loaded at startup Less than 20 employees currently exist in the system	
Primary Actor	HR Manager (for current CLI and future GUI system interactions)	
Secondary Actor(s)	Input Validator – ensures data quality (via EmployeeInputValidator) File System – saves employee to applicant_data.csv	
Main Sequence	**Step**	**Action**
	1	User selects option "3. ADD" from the main menu
	2	User selects "1. Add new employee" from add options
	3	System prompts for first name, validates it
	4	System prompts for last name, validates it
	5	System prompts for gender selection
	6	System prompts for valid email address
	7	System prompts for salary amount and validates range (1000–10000)
	8	User selects a department (IT Development, HR, Finance)
	9	User selects a job role (e.g., Developer, QA, Clerk, Team Lead, etc.)
	10	System creates employee object via EmployeeBuilder.createEmployee()
	11	System adds employee to internal list (DreamCompany.addEmployee())
	12	Employee is saved to CSV via EmployeeFileWriter.appendEmployeeToFile()
	13	System displays current employee list back to user

Postcondition(s)	New employee is added to memory (FIFO queue, max 20) Employee data is written to applicant_data.csv System list is refreshed and displayed	
Alternative Paths (or Exceptions)	**Step**	**Action**
	3a	Invalid name entered → system re-prompts with error
	6a	Invalid email format → system re-prompts
	7a	Invalid salary entered (non-number or out of range) → system re-prompts
	10a	Job title doesn't match valid roles → fallback to Developer
Frequency of Use	Regular (used each time a new employee is added manually)	
Business Rules / Constraints	- Name must start with a capital and only contain letters/hyphens - Salary must be between €1000 and €10000 - Email must be valid format - Total employees max: 20 (FIFO structure)	
Technology / Implementation Notes	- CLI-based menu (Switch case, Scanner input) - Uses Enums (MainMenuChoice, AddingChoice) - Validation: EmployeeInputValidator - Persistence: EmployeeFileWriter	
Code References	- DreamCompanyApp.main() - DreamCompany.addEmployee() - EmployeeBuilder.createEmployee() - EmployeeFileWriter.appendEmployeeToFile() - EmployeeInputValidator.isFirstNameValid(), etc.	
Comments (or Notes)	This is one of the core user-facing actions and ties directly into your object model and validation strategy. Recommended to link this use case to your first User Story. The HR Manager role represents staff responsible for onboarding and employee record entry in both current and planned versions of the system.	

2. Use Case Diagram 1 (Add New Employee)



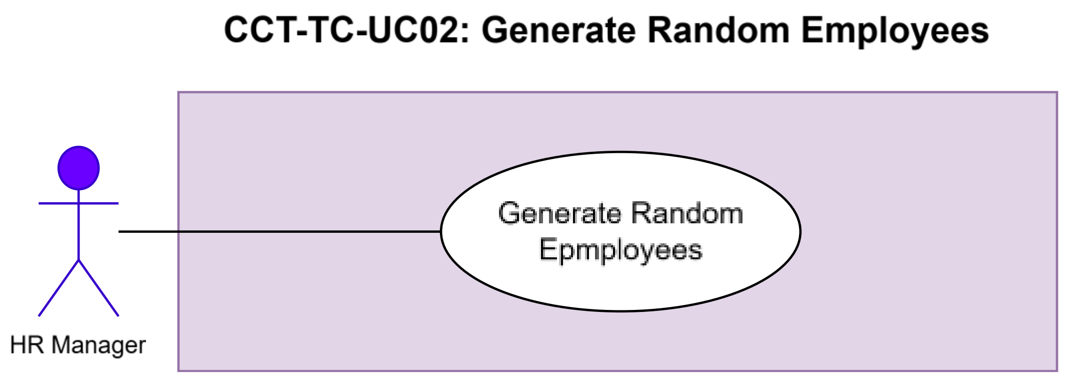
2. Use Case 2: Generate Random Employees

1. Use Case Specification (CCT-TC-UC02)

Use Case ID	CCT-TC-UC02	
Use Case Title	Generate Random Employees	
Use Case Goal	To create and save randomly generated employees to populate the system quickly.	
Description (or Overview)	This use case allows the user to generate between 1 to 10 employee records with randomized but valid data (names, departments, roles, etc.) and store them in the internal list and output file. It also reflects how an HR Manager or system tester generates random employee records. While primarily used in development for testing purposes, this feature can be retained for admin tools or HR simulations in future builds.	
Dependencies	None	
Trigger	User selects "ADD > Random Entry" from the main CLI menu	
Precondition(s)	User has launched the CLI program A valid CSV file was loaded at startup Less than 20 employees currently exist	
Primary Actor	HR Manager (used for testing and data population tasks)	
Secondary Actor(s)	RandomEmployeeDataGenerator (creates random valid employees) File System (appends data to CSV) Department Selector (assigns valid department classes)	
Main Sequence	**Step**	**Action**
	1	User selects option "3. ADD" from the main menu
	2	User selects "2. Add random employee" from add options
	3	System prompts user for number of employees (1–10)
	4	System validates number is within range
	5	System generates employee records using Disney-themed names and valid data
	6	System adds each employee to the internal FIFO list (max 20)
	7	System writes all generated employees to applicant_data.csv
	8	System displays the new and current employee list
Postcondition(s)	New employees are added to internal memory (FIFO queue) Employee data is saved to the CSV file System list is refreshed and shown	
Alternative Paths (or Exceptions)	**Step**	**Action**
	3a	Invalid input (non-number or out-of-range) -> system re-prompts with error
	6a	Employee count exceeds 20 → oldest employees are removed (FIFO enforcement)

	7a	File writing error -> message is shown but app continues
Frequency of Use	Moderate (often used for testing and populating the system)	
Business Rules / Constraints	- Only 1–10 employees can be generated at a time - Queue holds max 20 employees - Data must be valid (names, roles, departments, etc.)	
Technology / Implementation Notes	- CLI menu using enum AddingChoice - Uses RandomEmployeeDataGenerator class - Saves to applicant_data.csv	
Code References	- DreamCompanyApp (ADD > RANDOM logic) - DreamCompany.addEmployee() - RandomEmployeeDataGenerator.generateRandomEmployee() - EmployeeFileWriter.appendEmployeesToFile()	
Comments (or Notes)	Useful for quickly populating the system with test data. Helps simulate real usage scenarios. The role here represents both actual HR staff and technical staff responsible for populating the database with initial or dummy data in large systems.	

2. Use Case Diagram 2 (Generate Random Employees)


CCT-TC-UC02: Generate Random Employees
Generate Random Epmployees
HR Manager

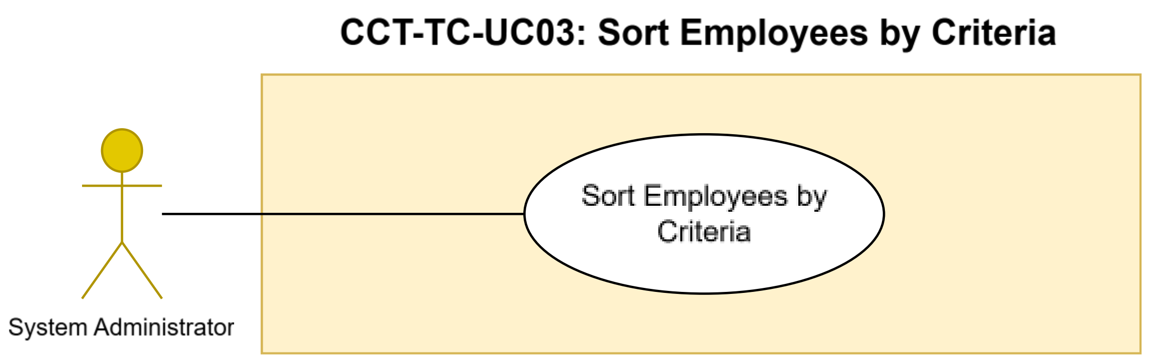
3. Use Case 3: Sort Employees By Criteria

1. Use Case Specification (CCT-TC-UC03)

Use Case ID	CCT-TC-UC03
Use Case Title	Sort Employees by Criteria
Use Case Goal	To organize employee records by selected attributes such as name, salary, or department.
Description (or Overview)	This use case allows users to sort the current employee list based on different criteria to improve readability and prepare for searching or reporting. System Administrator can sort employee records by name, salary, or department. Though CLI-based for now, future implementations may support real-time dashboards or filter controls.
Dependencies	None
Trigger	User selects “SORT” from the main CLI menu
Precondition(s)	User has launched the CLI program A valid CSV file was loaded At least one employee exists in the list
Primary Actor	System Administrator (for maintaining data visibility and order)

Secondary Actor(s)	Internal Employee List (FIFO Queue) – holds and displays sorted data	
Main Sequence	**Step**	**Action**
	1	User selects option “1. SORT” from the main menu
	2	System displays sorting options (Name, Salary, Department)
	3	User selects desired sort criteria
	4	System validates the selection
	5	System performs appropriate sort: - Recursive Insertion Sort (by name) - Bubble Sort (by salary) - Bubble Sort (by department then name)
	6	Sorted employee list is displayed to user
Postcondition(s)	The employee list is reordered in memory and displayed in the selected order.	
Alternative Paths (or Exceptions)	**Step**	**Action**
	3a	User enters an invalid option -> system displays error and re-prompts
	5a	List is empty -> system warns user that sorting has no effect
Frequency of Use	Frequent – used whenever employees need to be organized for display or searching.	
Business Rules / Constraints	- Sort by Name uses recursive insertion sort - Sort by Salary uses descending bubble sort - Sort by Department uses bubble sort with tie-breaker by name	
Technology / Implementation Notes	- CLI input via SortingChoice enum - Methods: sortByNameRecursive(), sortBySalaryDesc(), sortByDepartmentThenName()	
Code References	- DreamCompanyApp (SORT section) - DreamCompany.sortByNameRecursive() - DreamCompany.sortBySalaryDesc() - DreamCompany.sortByDepartmentThenName()	
Comments (or Notes)	This use case supports improved usability and aligns with searching and reporting features. Sorting tasks are typically handled by those managing visibility and integrity of employee records, hence mapped to the System Admin role.	

2. Use Case Diagram 3 (Sort Employees By Criteria)


CCT-TC-UC03: Sort Employees by Criteria
Sort Employees by Criteria
System Administrator

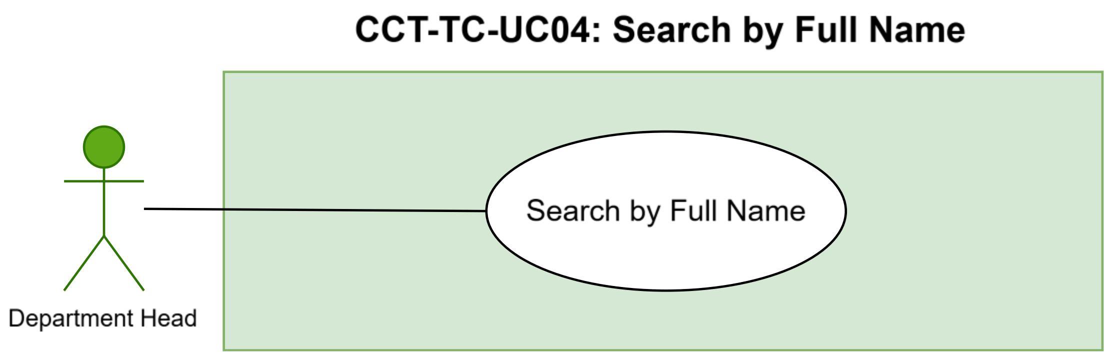
4. Use Case 4: Search for Employee by Full Name

1. Use Case Specification (CCT-TC-UC04)

Use Case ID	CCT-TC-UC04	
Use Case Title	Search for Employee by Full Name	
Use Case Goal	To allow the user to locate a specific employee record using the full name.	
Description (or Overview)	This use case enables users to search for an employee by entering their full name (first and last). The system uses binary search to find and display the matching employee's details. It allows a Department Head to look up an employee by full name. In future versions, this could be integrated into team dashboards or HR portals.	
Dependencies	Sort by Name must have occurred before binary search.	
Trigger	User selects "SEARCH > Full Name" from the main CLI menu	
Precondition(s)	The employee list must not be empty The list must be sorted by name	
Primary Actor	Department Head (searching for employees under their supervision)	
Secondary Actor(s)	Sorted Employee List (data structure used for binary search)	
Main Sequence	**Step**	**Action**
	1	User selects option "2. SEARCH" from the main menu
	2	System displays search options; user chooses 'Full Name'
	3	System prompts user for first name
	4	System validates first name format
	5	System prompts user for last name
	6	System validates last name format
	7	System combines names and prepares for search
	8	System performs binary search on sorted list
	9	If found, system displays full employee details
	10	If not found, system notifies user of failure
Postcondition(s)	Employee details are shown if a match is found, or an error message is displayed if not found.	
Alternative Paths (or Exceptions)	**Step**	**Action**
	3a	Invalid first name input -> system re-prompts
	5a	Invalid last name input -> system re-prompts
	8a	No matching employee found -> system displays error message
	2a	User selects wrong option -> system re-prompts for valid input
Frequency of Use	Frequently used when users need quick access to specific employee data.	
Business Rules / Constraints	- Names must be alphabetic and properly formatted - List must be sorted before performing binary search	
Technology / Implementation Notes	- Uses binarySearchByName() - Requires prior sort via sortByNameRecursive() - Validates input using EmployeeInputValidator	

Code References	- DreamCompanyApp (SEARCH section) - DreamCompany.binarySearchByName() - EmployeeInputValidator
Comments (or Notes)	This is one of the key user-facing features and showcases algorithmic efficiency with binary search. This role models a future user (e.g., line manager or department lead) who requires fast access to employee data without admin privileges.

2. Use Case Diagram 4 (Search for Employee by Full Name)



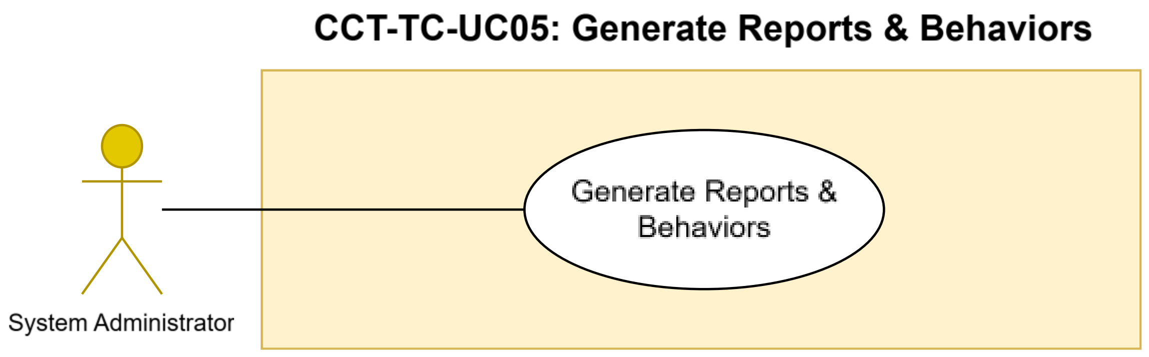
5. Use Case 5: Generate Role-Based Reports & Show Behavior

1. Use Case Specification (CCT-TC-UC05)

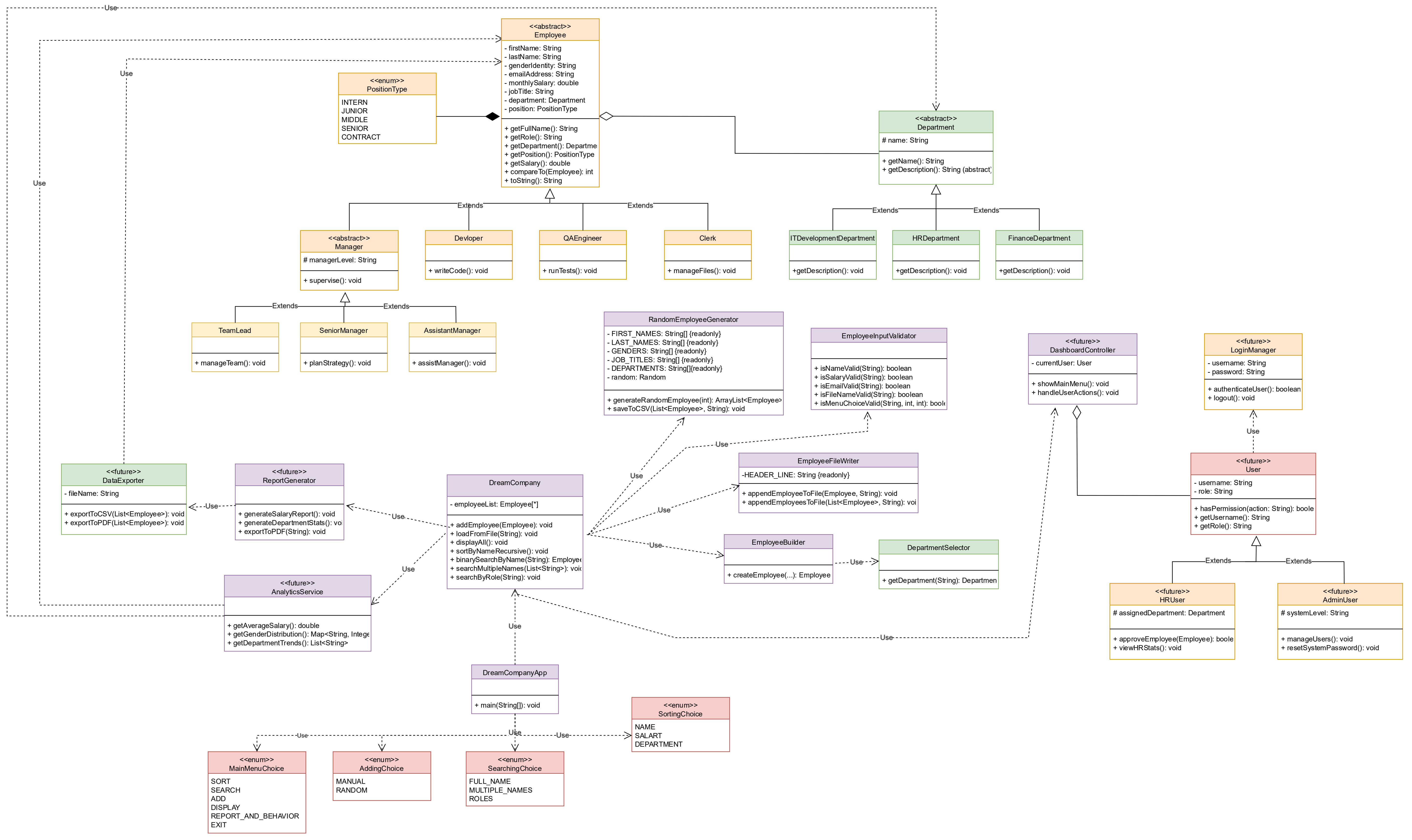
Use Case ID	CCT-TC-UC05	
Use Case Title	Generate Role-Based Report & Show Behavior	
Use Case Goal	To provide a count summary of employee types and demonstrate role-specific behaviors.	
Description (or Overview)	This use case generates a summary report by employee subclass (e.g., Developer, Clerk, Manager) and simulates behaviors like coding or testing based on the role. It reflects both testing functionality and future HR reporting features.	
Dependencies	None	
Trigger	User selects “REPORT & BEHAVIORS” from the main CLI menu	
Precondition(s)	The employee list must not be empty	
Primary Actor	All Users (primarily HR Managers and Admins)	
Secondary Actor(s)	None	
Main Sequence	**Step**	**Action**
	1	User selects option '5. REPORT & BEHAVIORS' from the main menu
	2	System retrieves all current employees
	3	System counts number of employees by subclass (e.g., Developer, QA Engineer)
	4	System displays a formatted report with role-based counts
	5	System checks type of each employee and performs role-specific method (e.g., writeCode, manageFiles)

	6	System prints output of each behavior to the console
Postcondition(s)	User sees a printed report of employee types and demonstration of their job-specific behaviors.	
Alternative Paths (or Exceptions)	**Step**	**Action**
	1a	User selects wrong option -> system re-prompts for valid input
	2a	List is empty -> system shows message 'No employees available for report.'
Frequency of Use	Occasional, mainly used for summaries or debugging system behavior.	
Business Rules / Constraints	- Employee must match subclass for behavior (e.g., Developer can only call writeCode)	
Technology / Implementation Notes	- Uses instanceof checks - Methods: reportBySubclass(), writeCode(), runTests(), manageFiles()	
Code References	- DreamCompanyApp (REPORT_AND_BEHAVIOR case) - DreamCompany.reportBySubclass() - Subclass methods (writeCode, runTests, etc.)	
Comments (or Notes)	Helpful for visualizing role distribution and ensuring role behaviors function correctly. Also useful for demos. This use case is valuable for reporting, training, or debugging. In future builds, it could be linked to analytics dashboards or employee performance views.	

2. Use Case Diagram 5 (Generate Role-Based Reports & Show Behavior)


CCT-TC-UC05: Generate Reports & Behaviors
Generate Reports & Behaviors
System Administrator

Section 3: UML Model 1 – Class Diagram (Based on Dream Tech Entire System)


Use
<<abstract>> Employee
- firstName: String
- lastName: String
- genderIdentity: String
- emailAddress: String
- monthlySalary: double
- jobTitle: String
- department: Department
- position: PositionType
+ getFullName(): String
+ getRole(): String
+ getDepartment(): Departme
+ getPosition(): PositionType
+ getSalary(): double
+ compareTo(Employee): int
+ toString(): String
<<enum>> PositionType
INTERN
JUNIOR
MIDDLE
SENIOR
CONTRACT
<<abstract>> Department
# name: String
+ getName(): String
+ getDescription(): String (abstract)
Extends
<<abstract>> Manager
# managerLevel: String
+ supervise(): void
Devloper
+ writeCode(): void
QAEngineer
+ runTests(): void
Clerk
+ manageFiles(): void
ITDevelopmentDepartment
+getDescription(): void
HRDepartment
+getDescription(): void
FinanceDepartment
+getDescription(): void
TeamLead
+ manageTeam(): void
SeniorManager
+ planStrategy(): void
AssistantManager
+ assistManager(): void
RandomEmployeeGenerator
- FIRST_NAMES: String[] {readonly}
- LAST_NAMES: String[] {readonly}
- GENDERS: String[] {readonly}
- JOB_TITLES: String[] {readonly}
- DEPARTMENTS: String[]{readonly}
- random: Random
+ generateRandomEmployee(int): ArrayList<Employee>
+ saveToCSV(List<Employee>, String): void
EmployeeInputValidator
+ isNameValid(String): boolean
+ isSalaryValid(String): boolean
+ isEmailValid(String): boolean
+ isFileNameValid(String): boolean
+ isMenuChoiceValid(String, int, int): boole
<<future>> DashboardController
- currentUser: User
+ showMainMenu(): void
+ handleUserActions(): void
<<future>> LoginManager
- username: String
- password: String
+ authenticateUser(): boolean
+ logout(): void
<<future>> DataExporter
- fileName: String
+ exportToCSV(List<Employee>): void
+ exportToPDF(List<Employee>): void
<<future>> ReportGenerator
+ generateSalaryReport(): void
+ generateDepartmentStats(): vo
+ exportToPDF(String): void
DreamCompany
- employeeList: Employee[*]
+ addEmployee(Employee): void
+ loadFromFile(String): void
+ displayAll(): void
+ sortByNameRecursive(): void
+ binarySearchByName(String): Employee
+ searchMultipleNames(List<String>): void
+ searchByRole(String): void
EmployeeFileWriter
-HEADER_LINE: String {readonly}
+ appendEmployeeToFile(Employee, String): void
+ appendEmployeesToFile(List<Employee>, String): voi
<<future>> User
- username: String
- role: String
+ hasPermission(action: String): boole
+ getUsername(): String
+ getRole(): String
EmployeeBuilder
+ createEmployee(...): Employee
DepartmentSelector
+ getDepartment(String): Departmen
<<future>> AnalyticsService
+ getAverageSalary(): double
+ getGenderDistribution(): Map<String, Intege
+ getDepartmentTrends(): List<String>
<<future>> HRUser
# assignedDepartment: Department
+ approveEmployee(Employee): boole
+ viewHRStats(): void
<<future>> AdminUser
# systemLevel: String
+ manageUsers(): void
+ resetSystemPassword(): void
DreamCompanyApp
+ main(String[]): void
<<enum>> SortingChoice
NAME
SALART
DEPARTMENT
<<enum>> MainMenuChoice
SORT
SEARCH
ADD
DISPLAY
REPORT_AND_BEHAVIOR
EXIT
<<enum>> AddingChoice
MANUAL
RANDOM
<<enum>> SearchingChoice
FULL_NAME
MULTIPLE_NAMES
ROLES

Section 4: Justification of UML Model 1

The first UML model chosen for the Dream Tech Company system is the class diagram, as it most accurately reflects the system's architecture and object-oriented structure. Dream Tech is designed to model employees across various roles, manage them in a queue, and support both manual and random employee creation, validation, and reporting. These features are implemented using a layered and highly modular class-based approach, which makes the class diagram the most suitable choice.

At the core of this application is the abstract class Employee, which contains shared attributes such as firstName, genderIdentity, and monthlySalary. It also defines shared behaviour such as getFullName() and compareTo(). Concrete classes like Developer, QAEngineer, Clerk, and Manager inherit from this class. Manager then serves as an abstract subclass, extended by TeamLead, SeniorManager, and AssistantManager. These inheritance and generalisation relationships are best shown using a class diagram, where arrows can clearly indicate how subclasses extend from base classes (Lupidchart, 2024).

The system's core logic is managed by the DreamCompany class, which interacts with a queue of employees. It relies on several utility classes such as EmployeeBuilder, RandomEmployeeDataGenerator, EmployeeInputValidator, and EmployeeFileWriter. These classes are not part of the same hierarchy but interact through associations, which are precisely the type of relationships that a class diagram is designed to model (Paradigm, 2024). Other UML models, such as object diagrams or activity diagrams, would not represent these class-level dependencies effectively. Object diagrams only show a static snapshot at runtime, which is not useful in this case, as employees are added dynamically through CLI interaction. Activity diagrams, while helpful for user flow, do not capture inheritance, class dependencies, or future extensibility.

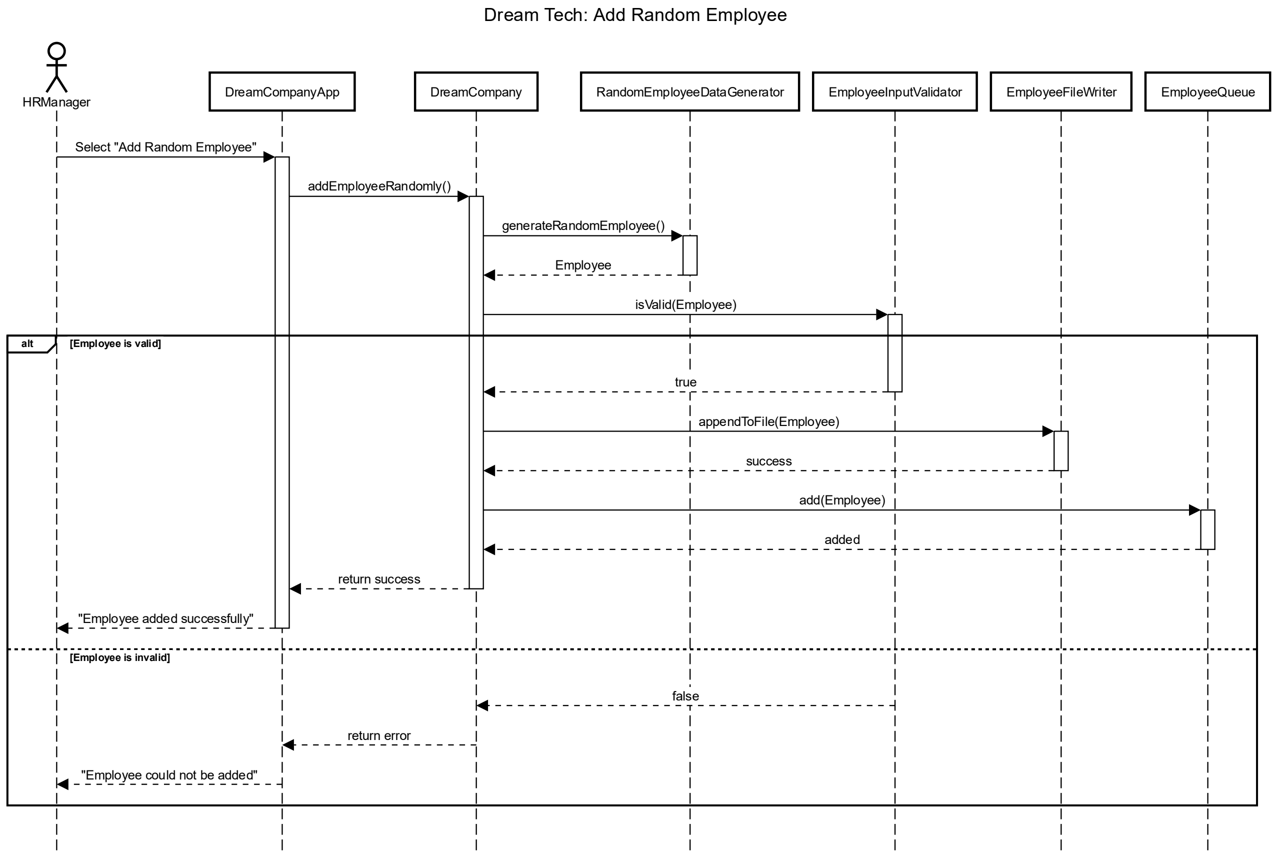
The class diagram also allows for the inclusion of <<enum>> types such as PositionType, SortingChoice, and SearchingChoice, which are used across the application for structured input and logic handling. These enumerations are referenced in various classes and directly affect method decisions and sorting/filtering operations — another reason why a class-level model is essential to convey how the system functions internally.

Additionally, the class diagram shows the system's planned scalability. Future classes like LoginManager, DashboardController, User, HRUser, AdminUser, ReportGenerator, AnalyticsService, and DataExporter are marked with <<future>> stereotypes. These demonstrate how the system will evolve into a role-based, GUI-ready HR platform with reporting and analytics capabilities. Including these in the class diagram shows stakeholders how the design supports future expansion, which is not something other diagram types can do as clearly (Object Management Group, 2017).

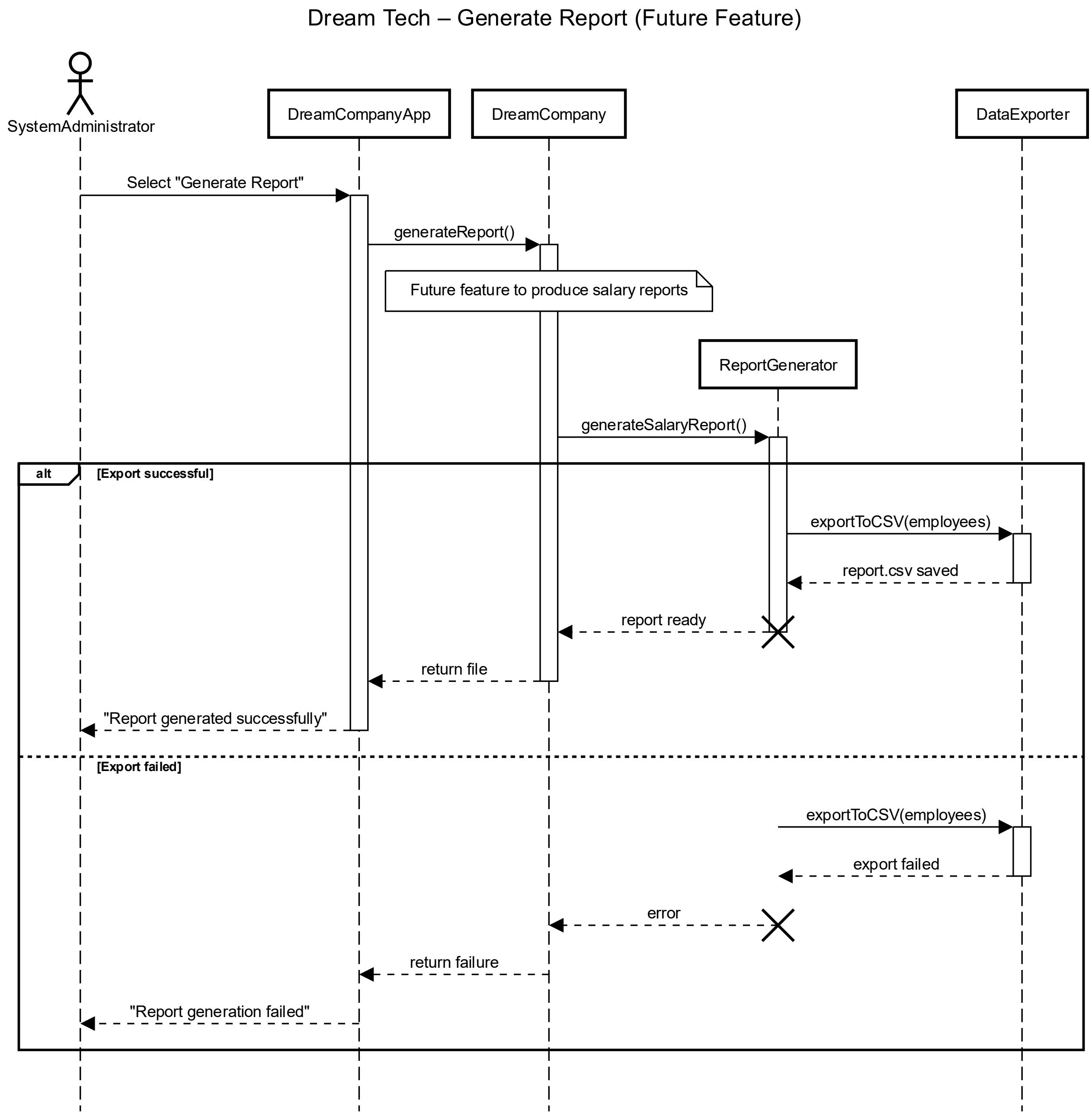
In conclusion, the class diagram is not only the most appropriate model for the current Dream Tech Company system, but also the most effective way to visualise its relationships, logic, modular structure, and long-term vision. Its ability to capture inheritance, utility class usage, enums, and scalability makes it a strong and justifiable choice.

Section 5: UML Model 2 – Sequence Diagrams

1. Sequence Diagram 1: Add Random Employee



2. Sequence Diagram 2: Generate Report (Future Feature)



Section 6: Justification of UML Model 2 - Sequence Diagram (For 2 use cases)

1. Justification of Sequence Diagram 1 (Add Random Employee)

For my second UML model, I selected a sequence diagram to represent the use case *“Add Random Employee”*. This was the most appropriate choice for Dream Tech Company because this specific scenario involves a series of method-level interactions across multiple classes in a strict order. When a user selects this option in the CLI, the system generates an employee using RandomEmployeeDataGenerator, validates the input through EmployeeInputValidator, saves the record using EmployeeFileWriter, and finally adds the employee to the queue. All of these steps involve separate classes that interact via method calls, making a sequence diagram the most accurate way to visualise this flow (Lupidchart, 2024).

I considered using an activity diagram for this use case, but activity diagrams are more focused on decision-making and high-level process flow. They do not clearly show how objects or classes pass control and messages between each other. Since the Dream Tech system is class-heavy and modular, a sequence diagram allowed me to illustrate the internal communication between objects in a way that aligns directly with my code structure (Paradigm, 2024). It also helped communicate which object controls the logic at each point using activation bars, and how success or failure in validation can affect the overall flow using alt blocks. This level of detail would not be possible in a use case or activity diagram (Object Management Group, 2017).

2. Justification of Sequence Diagram 2 (Generate Report)

I used another sequence diagram to represent the future use case *"Generate Report"*. This diagram was chosen to show how the Dream Tech system is designed to scale, by incorporating future classes such as ReportGenerator and DataExporter. The system administrator triggers the report generation, and the flow includes object creation, data export, and conditional logic depending on whether the export succeeds or fails. This future-facing model reflects both the structure and extensibility of the system, which was a key goal in my class diagram as well.

I considered modelling this scenario with a component diagram or state diagram, but those alternatives would not have captured the behaviour and control flow between classes. A component diagram would be better suited for a deployed, multi-module application, which Dream Tech is not at this stage. A state diagram would only show lifecycle states of one object (e.g., "report generated", "report failed"), but wouldn't capture the collaboration between classes. The sequence diagram, by contrast, allowed me to visualise the interaction between planned future components and the existing core system in a way that supports technical planning and stakeholder communication (Paradigm, 2024), (Object Management Group, 2017).

Section 7: Comparison Between Models

1. Comparison

In designing the Dream Tech Company system, I considered several UML modelling techniques before selecting the class diagram and sequence diagrams as the most appropriate. The class diagram was chosen to represent the static architecture of the system, which is highly object-oriented. My design includes abstract classes like Employee, subclass hierarchies (e.g., Developer, Manager, TeamLead), and supporting utility classes such as EmployeeBuilder, Validator, and FileWriter. These structural relationships, including inheritance and associations, could not have been represented effectively in other models like activity diagrams, which focus on workflows rather than class structure (Paradigm, 2024).

I also considered using a component diagram, but this would have been more suitable for a large-scale, modular or web-based system (Lupidchart, 2024). Since Dream Tech is currently a CLI-based, single-application system, the component model would not have added value. A state diagram was also ruled out, as it would only track the state transitions of a single object, such as Employee, and not capture system-wide logic or class collaboration (Paradigm, 2024).

For behavioural modelling, I selected sequence diagrams because they show how the system behaves in response to specific use cases — namely UC02: Add Random Employee and UC05: Generate Report. These diagrams clearly map the flow of method calls and control between multiple objects in order, using constructs like alt blocks and object creation/destruction. This level of detail was essential for showing how the DreamCompany class interacts with helper classes like RandomEmployeeDataGenerator and EmployeeInputValidator, something a use case diagram or activity diagram could not adequately visualise. Use case diagrams are great for user goals but lack class-level interactions, and activity diagrams are more suitable for simple user-driven workflows, not internal object behaviour (Paradigm, 2024).

In summary, the class diagram provided a complete view of the system's static structure and scalability, while sequence diagrams allowed me to represent the dynamic behaviour of both implemented and future features. Together, they were the most appropriate combination to fully represent the architecture and logic of the Dream Tech Company system.

2. Comparison table

Feature / Purpose	**Class Diagram (Used)**	**Sequence Diagram (Used)**	**Component Diagram (Not used)**	**Activity Diagram (Not used)**	**State Diagram (Not used)**
Focus	Static structure of the system	Runtime behaviour and object interaction	High-level module/component architecture	Process flow and user actions	Lifecycle states of a single object
Best Suited For	Showing class relationships, inheritance, utility usage, and future scalability	Showing detailed method calls and interaction flow in use cases	Systems with services, packages, or deployable modules	Describing decisions and task sequences	Tracking object status transitions
Why I Chose It for Dream Tech	My system is class-heavy with clear inheritance and modular class structure	My app logic is driven by object collaboration (e.g., Validator -> FileWriter -> Queue)	My app is not modularised into separate deployable components	Doesn't show method-level object interaction	Not useful for multi-class use cases
Specific Use in My System	Models Employee subclasses, DreamCompany, helper classes, enums, and <<future>> features	UC02: Add Random Employee and UC05: Generate Report (shows control, return values, alt blocks)	Not suitable — all logic is contained within a single monolithic CLI app	Could represent "Add Employee" flow but not internal method logic	Would only work for a single object like Employee, not whole process
Limitations	Doesn't show order/timing of logic flow	Doesn't show overall structural design or data types	Can't show inheritance, class interactions, or internal object behaviour	Lacks technical depth for object communication	Too specific and disconnected from class-driven design

Sources used to create this table: (Lupidchart, 2024), (Lupidchart, 2024), (Lupidchart, 2024), (Paradigm, 2024), (Paradigm, 2024)

Section 8: User Stories and Acceptance Criteria

This section presents five user stories designed to capture the core functional requirements of the Dream Tech Company system. Each story is directly linked to the five use cases identified earlier and is informed by the system's UML modelling techniques, including class diagrams and sequence diagrams.

The format used for the user stories follows the recommended pattern provided in the course material, which is based on agile practices. Each story follows the structure: "As a [role], I want [function], So that [benefit]", ensuring clarity, user focus, and traceability. The acceptance criteria for each story are written to be clear, testable, and outcome-focused, providing a foundation for system validation and test case development later in the report.

The structure and phrasing of the user stories were also supported by external industry guidance from (Mitrofanskiy, 2024), which offered additional practical examples and formatting techniques. By combining class-taught structure with industry-validated practices, the resulting user stories offer both academic alignment and real-world applicability.

1. USER STORY 1: Add Employee (Manual)

User Story 1 – Add Employee (Manual)	
Scenario	**Acceptance Criteria**
User Story: As an HR Manager, I want to manually enter employee details, So that I can ensure the data is accurate and personalised.	• The system must allow entry of first name, last name, gender, salary, department, and job title • All fields must be validated (e.g. names must be alphabetic, salary must be numeric and within range). • If a field is invalid, the system must provide an error message and not proceed. • The employee must be added to the internal queue if valid. • A confirmation message must be displayed once added.

2. USER STORY 2: Generate Random Employee

User Story 2 – Generate Random Employee	
Scenario	**Acceptance Criteria**
User Story: As a System User, I want to generate employees automatically, So that I can quickly test and populate the system.	• The system must allow selecting a number of random employees to generate. • Each generated employee must include valid data from predefined lists. • All generated employees must pass validation. • Valid employees are written to file and added to the queue. • A success message is shown after completion.

3. USER STORY 3: Search Employee by Full Name

User Story 3 – Search Employee by Full Name	
Scenario	**Acceptance Criteria**
User Story: As a System User, I want to search for an employee by their full name, So that I can view their information quickly.	• The system must allow the user to input a full name. • If the employee exists, their full details must be displayed. • If not found, an error message must be shown. • Inputs must be validated (e.g. two parts, alphabetic).

4. USER STORY 4: Sort Employees

User Story 4 – Sort Employees	
Scenario	**Acceptance Criteria**
User Story: As a System User, I want to sort the list of employees, So that I can view them in a specific order like by name or salary.	• The user must be able to choose the sort type: by name, by salary, or by department. • The list must update immediately based on the selected option. • The sorting must be accurate and reflect in the display. • No duplicates or invalid entries should be present.

5. USER STORY 5: Generate Report (Future Feature)

User Story 5 – Generate Report (Future Feature)	
Scenario	**Acceptance Criteria**
User Story: As a System Administrator, I want to generate a report of all employees, So that I can review and export data for management.	• A report must be generated that includes all employee data. • The data must be exportable (e.g. to a CSV file). • The export must succeed only if the file path is valid. • The system must return a confirmation or error message. • The report must reflect the current state of the employee queue.

Section 9: Unit Testing Scenarios

The following test cases were designed based on the Dream Tech system's core features. They follow a structured template based on best practices outlined in the unit testing material (*Unit Test.docx*). The tests are linked directly to the system's use cases and modelling techniques, demonstrating coverage across input validation, object creation, file handling, sorting, searching, and reporting.

1. Test Cases by Use Case

1. CCT-TC-UC01 – Add Employee (Manual)

a) TC_UC01_001 – Validate Name Input

TEST ID NUMBER	**TC_UC01_001**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Validate Name Input**
Module Tested	*EmployeeInputValidator*						
Description of Test	*Check name entry when manually adding an employee*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	System is running and form is open						
Dependencies (if any)	None						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Open Add Employee form		Form loads without issue	Add Employee form displayed	Pass	
2. Enter valid name	"Peter"	Input accepted	Field accepted input; no validation errors shown	Pass	Valid input
3. Enter invalid name	"Pet0r!"	Error message: "Invalid characters"	Error message displayed as expected	Pass	Detected number + symbol
4. Submit with valid name	"Peter"	No validation errors shown	Form submitted; no name errors	Pass	
5. Submit with invalid name	"123"	System blocks submission	Submission blocked; error prompt shown	Pass	Input rejected as expected

b) TC_UC01_002 – Reject Invalid Email

TEST ID NUMBER	**TC_UC01_002**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Reject Invalid Email**
Module Tested	*EmployeeInputValidator*						
Description of Test	*Validate rejection of improperly formatted email.*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	System running and form is open						
Dependencies (if any)	None						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Open the Add Employee form		Form loads correctly	Add Employee form opened successfully	Pass	
2. Enter invalid email in email field	peterdream.com	System detects invalid format	System displayed error: "Invalid email format"	Pass	Missing '@' symbol
3. Submit the form	Click "Add Employee"	System shows error message: "Invalid email format"	Error message blocked submission	Pass	Form not submitted
4. Verify employee is not added	Check queue	Employee not added to queue	Queue unchanged; employee not added	Pass	Validated via queue status

c) TC_UC01_003 – Queue Accepted Employee

TEST ID NUMBER	**TC_UC01_003**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Queue Accepted Employee**
Module Tested	*DreamCompany*						
Description of Test	*Ensure validated employee is added to queue.*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	Valid employee data is available, form inputs pass validation						
Dependencies (if any)	EmployeeBuilder, EmployeeInputValidator						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Build employee object	Name, Email, Salary, Department, etc.	Employee object is created successfully	Employee object created using builder	Pass	Uses EmployeeBuilder class
2. Validate employee data	Same data	Data passes all validation checks	Validator returned true	Pass	Uses EmployeeInputValidator
3. Add validated employee to queue	Call addToQueue(Employee)	Employee is pushed into queue	Employee successfully added to system queue	Pass	Queue size increased by one
4. Verify employee in queue	Read from queue	Employee data present at correct position	Employee object is at end of queue	Pass	FIFO logic validated
5. Confirm success message	System confirmation output	Display: “Employee added successfully”	Success message displayed in CLI	Pass	Confirms completion of workflow

2. CCT-TC-UC02 – Generate Random Employee

a) TC_UC02_001 – Generate One Employee

TEST ID NUMBER	**TC_UC02_001**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Generate One Employee**
Module Tested	*RandomEmployeeDataGenerator*						
Description of Test	*Ensure that when one random employee is generated, all required fields are populated with valid data*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	System is running, and generation function is accessible						
Dependencies (if any)	Enum classes (e.g., PositionType, DepartmentType)						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Open system	Start app	System loads without error	Application started successfully	Pass	Console output confirmed
2. Select random generation	Choose “Generate 1”	System calls random generation method	generateRandomEmployee(1) executed	Pass	Random generator triggered
3. Check employee object fields	Inspect generated object	All fields populated (name, salary, department, etc.)	Object contains valid values for each attribute	Pass	No null/empty values detected
4. Validate enums used	Check DepartmentType, PositionType	All enums valid, no unrecognised values	Department and position values are valid enums	Pass	Valid options from predefined types
5. Confirm CLI output	Console display	“Random employee added successfully” message shown	Confirmation message appeared	Pass	Final output matches expectation

b) TC_UC02_002 – Validate Random Employee

TEST ID NUMBER	**TC_UC02_002**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Validate Random Employee**
Module Tested	*EmployeeInputValidator*						
Description of Test	*Ensure that a randomly generated employee passes all validation checks required for system entry*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	A random employee object must be generated first						
Dependencies (if any)	RandomEmployeeDataGenerator, EmployeeBuilder						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Generate random employee	Call generateRandomEmployee()	Employee object with auto-filled fields	Employee object created with valid-looking data	Pass	Fields: name, email, salary, department
2. Validate random employee	Pass employee to validator	Validation result: true	Validator returned true	Pass	All fields passed checks
3. Test invalid override (optional)	Manually set salary = -1000	Validation result: false	Validator returned false after override	Pass	Negative salary rejected
4. Restore valid salary	Set salary to 4000	Re-run validator: true	Validator returned true	Pass	Employee now valid again
5. Confirm validation success	Capture confirmation message	“Employee validated successfully”	Console output matched expected	Pass	Test complete

c) TC_UC02_003 – Write Random to File

TEST ID NUMBER	**TC_UC02_003**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Write Random to File**
Module Tested	*EmployeeFileWriter*						
Description of Test	*Confirm that a validated random employee is successfully written to the system's employee data file.*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	A random employee has been generated and passed validation						
Dependencies (if any)	RandomEmployeeDataGenerator, EmployeeInputValidator						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Generate random employee	Call generateRandomEmployee()	Employee object created	Random employee created with valid fields	Pass	Prepares test object
2. Validate employee	Pass object to validator	Returns true (passes all checks)	Validator returned true	Pass	Uses EmployeeInputValidator
3. Call appendToFile()	Employee object	Employee data appended to employees.csv	File updated, new line added	Pass	No overwrite occurred
4. Verify file contents	Open employees.csv	Employee data appears on last line	Line present with correct structure and values	Pass	Name, email, salary fields match object
5. Test file access after save	Attempt to reopen file	File can be opened and read again	File opened successfully and data intact	Pass	Confirms persistence

3. CCT- UC03 – Sort Employees

a) TC_UC03_001 – Sort by Name

TEST ID NUMBER	**TC_UC03_001**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Sort by Name**
Module Tested	*DreamCompany*						
Description of Test	*Ensure that the list of employees is correctly sorted alphabetically by first name when the user selects "Sort by Name"*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	Employee list must contain at least three unsorted entries.						
Dependencies (if any)	Employee class, sorting method, CLI input handling						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Load employee list	Add: "Alice", "Snow", "Henny"	List contains unsorted employees	List loaded: Alice, Snow, Henny	Pass	Order before sorting: S, A, H
2. Select sorting option	CLI option: "Sort by Name"	Sorting function is triggered	Sorting triggered successfully	Pass	CLI input accepted
3. Apply recursive insertion sort	Call sorting method	Employees reordered alphabetically	List sorted: Alice, Henny, Snow	Pass	Alphabetical order confirmed
4. Display sorted list	CLI output	Sorted list shown on screen	Console displayed sorted employee names	Pass	Visual confirmation complete
5. Validate first and last entries	Compare first/last names in list	First = "Alice", Last = "Snow"	Matches expected order	Pass	Test logic validated with boundaries

b) TC_UC03_002 – Sort by Salary Descending

TEST ID NUMBER	**TC_UC03_002**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Sort by Salary Descending**
Module Tested	*DreamCompany*						
Description of Test	*Ensure that employees are sorted from highest to lowest salary when the sort option is selected*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	System has access to a populated employee list with varying salary values						
Dependencies (if any)	Employee list, sorting method, console menu interaction						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Add test employees to list	Salaries: 3200, 5800, 4500	List created with 3 salary values	List added: [3200, 5800, 4500]	Pass	Unsorted input prepared
2. Select sorting option	CLI: “Sort by Salary (High to Low)”	System triggers sorting logic	Sort option activated through menu	Pass	CLI logic functioning
3. Execute sort method	Sort function called	Salaries reordered in descending order	Sorted: [5800, 4500, 3200]	Pass	Correct internal order
4. Display result	CLI Output	Sorted salaries displayed to user	Salaries displayed in correct descending order	Pass	Output verified visually
5. Validate order logic	Check first and last salary in list	First = 5800, Last = 3200	Order matches expected logic	Pass	Confirms sorting integrity

c) TC_UC03_003 – Sort by Department and Name

TEST ID NUMBER	**TC_UC03_003**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Sort by Department and Name**
Module Tested	*DreamCompany*						
Description of Test	*Ensure that employees are grouped by department and sorted alphabetically within each group when the correct sort option is selected*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	A list of employees across multiple departments exists.						
Dependencies (if any)	Employee class, sorting method						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Populate list with mixed departments	HR: “Zayn”, Dev: “Liam”, HR: “Alice”, Dev: “Ben”	List with multiple departments	Employees added successfully	Pass	HR & Dev mixed in random order
2. Select sorting option	CLI: “Sort by Department then Name”	System triggers compound sort	Sort option selected	Pass	Menu interaction confirmed
3. Execute compound sort	Apply department + name logic	Employees grouped by department, names ordered within group	Sorted: HR → Alice, Zayn; Dev → Ben, Liam	Pass	Nested sort structure successful
4. Display sorted list	CLI Output	Correct groupings and order shown	Output displays expected sort structure	Pass	Visual check passed
5. Validate logic for both criteria	Check department blocks and name order	Each department group sorted alphabetically	Confirmed correct grouping and name sorting	Pass	Logic verified with multiple fields

4. CCT- UC04 – Search Employee by Name

a) TC_UC04_001 – Search Existing Employee

TEST ID NUMBER	**TC_UC04_001**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Search Existing Employee**
Module Tested	*DreamCompany*						
Description of Test	*Verify that the system can successfully locate an employee by full name using a binary search when the employee exists in the sorted list*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	The employee list is sorted alphabetically by name and contains the target employee						
Dependencies (if any)	Employee, sorting method, binary search method						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Populate list with employees	“Alice Smith”, “Ben Carter”, “Liam Gray”	List contains 3 employees including Alice	List created successfully	Pass	Sorted beforehand
2. Sort list by name	Use "Sort by Name"	List sorted alphabetically	Order: Alice, Ben, Liam	Pass	Required for binary search
3. Call binary search function	Search full name: “Alice Smith”	Employee found at index or object returned	Binary search returned Alice Smith	Pass	Matching record found
4. Display search result	CLI Output	Employee details shown in console	Name, email, and department printed	Pass	Visual match confirmed
5. Validate correct index (optional)	Index in array/list	Employee position corresponds with sorted list	Alice found at index 0	Pass	Logic validated using index check

b) TC_UC04_002 – Search Non-existent Employee

TEST ID NUMBER	**TC_UC04_002**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Search Non-existent Employee**
Module Tested	*DreamCompany*						
Description of Test	*Verify that the system handles binary search correctly when the employee is not found in the sorted list and displays an appropriate error message*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	The employee list is sorted and the target name is **not** present.						
Dependencies (if any)	Employee, sorting method, binary search logic						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Populate list with employees	"Ben Carter", "Liam Gray"	List created excluding the search name	List added successfully	Pass	"Zoe Grey" is not in this list
2. Sort list by name	Sort option: "Sort by Name"	List ordered alphabetically	List: Ben, Liam	Pass	Binary search precondition satisfied
3. Execute binary search	Full name: "Zoe Grey"	Search returns "not found" indicator	Search result: null or -1	Pass	Null result confirms name not present
4. Display system response	CLI Output	Show message: "Employee not found."	Message displayed correctly in console	Pass	User feedback provided
5. Verify no crash or exception	Edge case handling	Application remains stable	No crash; system handled input properly	Pass	Negative path handled gracefully

c) TC_UC04_003 – Validate Search Input

TEST ID NUMBER	**TC_UC04_003**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Validate Search Input**
Module Tested	*EmployeeInputValidator*						
Description of Test	*Verify that invalid search inputs (e.g. numeric or special characters) are correctly rejected, and an appropriate error message is displayed*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	The system is running, and the search menu is accessible						
Dependencies (if any)	CLI interface, search method, validator						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Launch system and open search	Start CLI	Search option available	Search feature displayed in menu	Pass	
2. Enter numeric input	12345	Input rejected as invalid	Error: "Please enter a valid full name."	Pass	Numeric input blocked
3. Enter special characters	!@#$%	Input rejected as invalid	Same error message shown	Pass	Edge case validated
4. Leave input blank	Press Enter	Input rejected; prompt re-displayed	Re-prompted to enter valid full name	Pass	Blank input handled
5. Confirm no search performed	After invalid input	No search logic executed	No employee search attempted	Pass	Confirms validation before execution

5. CCT-UC05 – Generate Report (Future Feature)

a) TC_UC05_001 – Generate Report File

TEST ID NUMBER	**TC_UC05_001**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Generate Report File**
Module Tested	*ReportGenerator*						
Description of Test	*Verify that the system can create a report file containing current employee data from the queue.*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	The employee queue contains at least one validated employee						
Dependencies (if any)	DreamCompany, EmployeeQueue, DataExporter						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Add employee(s) to queue	Valid employee object(s)	Queue updated with entries	3 employees added to queue	Pass	Ready to generate report
2. Select "Generate Report"	CLI option	Triggers generateReport() function	Report method successfully called	Pass	CLI trigger confirmed
3. Collect queue data	Call within method	Employee data fetched from queue	Queue retrieved: 3 records	Pass	Internal logic verified
4. Create CSV or report structure	Export to file format	report.csv or report.txt file created	report.csv generated in output directory	Pass	File created without overwrite
5. Open and verify file contents	Open report.csv	Employee info written in correct format	Report contains expected fields and values	Pass	File readable and accurate

b) TC_UC05_002 – Fail on Empty Queue

TEST ID NUMBER	**TC_UC05_002**	**Developer:**	**Egshiglen**	**Date Test Carried Out:**	**01.05.2025**	**Test Name:**	**Fail on Empty Queue**
Module Tested	*ReportGenerator*						
Description of Test	*Ensure that if the report is triggered with an empty queue, the system responds appropriately without error and does not generate a report*			**Test Carried out by:**		**2024359**	
Test Precondition(s)	Employee queue is empty						
Dependencies (if any)	DreamCompany, EmployeeQueue						

TEST STEP	**DATA (Input)**	**Expected Result(s)**	**Actual Result(s)**	**PASS/FAIL**	**NOTES**
1. Confirm queue is empty	Check EmployeeQueue	No employees present	Queue count = 0	Pass	Precondition verified
2. Attempt to generate report	Select “Generate Report”	Report method triggered	generateReport() called	Pass	No crash occurred
3. Detect empty queue inside method	Conditional check	System detects empty queue	Condition returned false (no data)	Pass	Internal logic worked
4. Display error message	Console output	“No employees in queue. Report not generated.”	Exact message displayed in console	Pass	User notified clearly
5. Ensure no file is created	Check file directory	No report.csv or output file exists	No file created; no export attempted	Pass	File system clean

References

Cockburn, A. (1998). *Basic Use Case Template*. Retrieved from https://cis.bentley.edu/lwaguespack/CS360_Site/Downloads_files/Use%20Case%20Template%20%28Cockburn%29.pdf

IBM. (2023). *Defining the boundaries of a system*. Retrieved from https://www.ibm.com/docs/en/dma?topic=diagrams-defining-boundaries-system

Lupidchart. (2024). *Component Diagram Tutorial*. Retrieved from https://www.lucidchart.com/pages/uml-component-diagram

Lupidchart. (2024). *UML Class Diagram Tutorial*. Retrieved from https://www.lucidchart.com/pages/uml-class-diagram

Lupidchart. (2024). *UML Sequence Diagram Tutorial*. Retrieved from https://www.lucidchart.com/pages/uml-sequence-diagram

Mitrofanskiy, K. (2024). *User Story Acceptance Criteria Explained with Examples*. Retrieved from https://intellisoft.io/user-story-acceptance-criteria-explained-with-examples/

Object Management Group. (2017). *Unified Modelling Language*. Retrieved from https://www.omg.org/spec/UML/2.5.1

Paradigm, V. (2024). *What is Activity Diagram?* Retrieved from https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/

Paradigm, V. (2024). *What is Class Diagram?* Retrieved from https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/

Paradigm, V. (2024). *What is Sequence Diagram?* Retrieved from https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

Paradigm, V. (2024). *What is State Machine Diagram?* Retrieved from https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/