CSc3350 Software Development
Group Team Project
Fall 2024

Team 1 - The ZQL Surgeons

Z-Force Employee Manager

Software Design Document

Names:
Suzal Regmi
Puru Saluja

Date (mm/dd/yy): 12/09/24

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Purpose

This Software Design Document (SDD) details the design and structure of Z-Force Employee Manager, an employee management system developed for Company Z. The system, built by The ZQL Surgeons, aims to handle current and future HR needs as the company scales from its current 25 full-time employees to three times that number within 18 months.

Key functionalities include:

• Searching employees by name, SSN, or empid

• Updating employee details (e.g., name, email, SSN)

• Increasing salaries by a certain percentage within a specified salary range

• Generating monthly pay reports by job title or division

No user authentication is required. The focus is on reliable data handling and easy retrieval/reporting. This SDD is intended for developers, QA testers, and stakeholders who require an in-depth technical understanding of the system's architecture and logic.

1.2 Definitions and Acronyms

• empid: Employee ID, a unique identifier for each employee.

• SSN: Social Security Number

• DAO: Data Access Object, an abstraction layer for database operations

• UX: User Experience, referring to the front-end (console/GUI) interface.

• JDBC: Java Database Connectivity, a Java API for database communication.

2.0 DATABASE SCHEMA DIAGRAM

The following database schema was generated via dBeaver and reflects the logical structure of the **employeeData** schema. Our design ensures that Company Z's employee records and related data are stored in a normalized, scalable manner. Each table and its relationships are designed to allow for straightforward queries, updates, and expansions as the organization grows from 25 full-time employees to three times that number.

Core Tables:

• **employees**(empid PK, Fname, Lname, email, SSN, Salary): Basic employee info.

• **address**(empid PK/FK, gender, pronouns, race, DOB, phone, city_id FK, state_id FK): Demographic and contact data.

• **city**(city_id PK, city_name), **state**(state_id PK, state_name): Normalized location tables.

• **division**(ID PK, division_name, address details): Stores division-level data.

• **employee_division**(empid PK/FK, div_ID FK): Associates employees with divisions.

• **job_titles**(job_title_id PK, job_title): Job title reference.

• **employee_job_titles**(empid PK/FK, job_title_id FK): Maps employees to their job titles.

• **payroll**(payID PK, empid FK, pay_date, earnings, etc.): Pay statements for historical and reporting use.

Key Relationships:

• employees.empid → employee_division.empid

• employees.empid → employee_job_titles.empid

• employees.empid → payroll.empid

• employee_division.div_ID → division.ID

• employee_job_titles.job_title_id → job_titles.job_title_id

• employees.empid → address.empid

• address.city_id → city.city_id

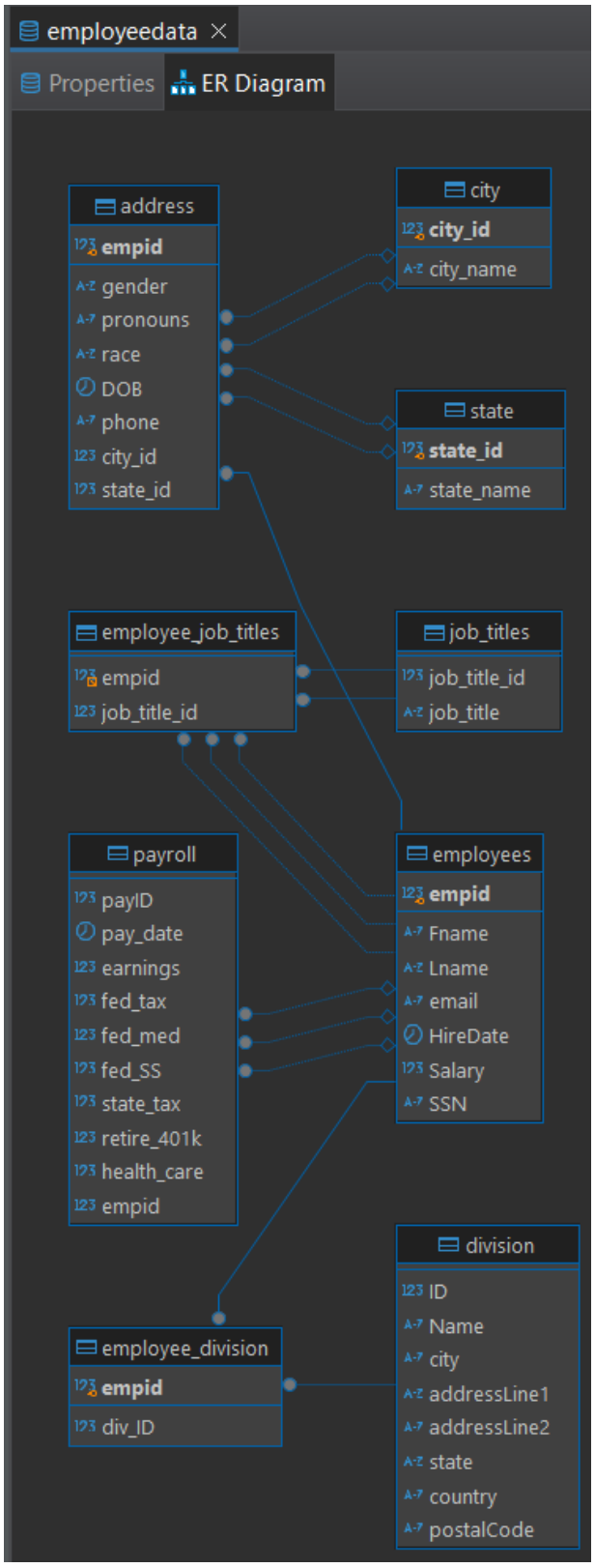
• address.state_id → state.state_id

Scalability Considerations:

• By separating data into multiple tables, such as employee_job_titles, employee_division, and address, the schema can easily scale to accommodate additional attributes, new divisions, or updated job titles as the company grows.

• Normalizing city and state data reduces data redundancy and ensures consistent references, making reporting and data analysis more efficient.

Below is the ER diagram, as generated by dBeaver.



3.0 JAVA CLASS DIAGRAM

The following classes have been implemented as part of the Employee Management System:

• **Employee.java**:
A simple POJO (Plain Old Java Object) representing an employee's core data (empid, firstName, lastName, email, SSN, salary). It encapsulates employee attributes and provides getters and setters. In terms of system architecture, this class aligns with the model layer, holding the data that corresponds to an employees table record.

• **DatabaseConnection.java**:
A singleton class responsible for establishing a connection to the MySQL database. It manages a single Connection instance, ensuring efficient reuse of this resource and encapsulating database connection details away from other parts of the system. This aligns with our design goal of separating infrastructure (DB connections) from business logic.

• **EmployeeService.java**:
A service class that interacts with the database to perform CRUD (Create, Read, Update, Delete) operations and handle specific queries such as updating salaries and generating reports. It uses DatabaseConnection to communicate with the database and contains methods that:

- Insert a new employee (aligns with the "Add new employee" feature)
- Search an employee by name/SSN/empid (aligns with the "Search Employee" use case)
- Update employee information (aligns with the "Update Employee Data" use case)
- Update salaries by a given percentage for employees in a certain salary range (aligns with the "Update Employee Salaries" use case)
- Generate pay reports by job title (part of the "Generate Reports" requirement)

The EmployeeService acts as a DAO layer in practice, though it could be split into separate DAO and Service layers if desired. Currently, it provides a straightforward approach consistent with the initial requirements.

• **EmployeeManagementSystem.java**:
A main class (entry point) demonstrating how to use EmployeeService and Employee classes. It simulates basic user interactions, effectively playing the role of a simple UX (in console form). This class:

- Inserts a new Employee record
- Searches for an employee
- Updates the employee's details
- Updates salaries for certain employees
- Generates and prints a report by job title

While a fully fleshed-out user interface (UI) might rely on JavaFX or a web-based UI, this main class provides a console-based test scenario that can be integrated into the system's UX later.

4.0 TEST CASES

User Story / Scenario

Task 1: Update Employee Information
Alex, an HR specialist, selects "Update Employee Information," enters the employee ID, and modifies details such as the employee's last name and email. After submitting the changes, the system confirms the update.

Test Cases for Task 1:
Test Case 1 (Pass): Valid Employee ID and Updated Details
Input: Alex enters Employee ID = 101 and updates the last name to "Johnson" and email to "johnson.new@example.com".
Expected Result: The employee with empid=101 now has last name "Johnson" and email "johnson.new@example.com". A confirmation message indicates a successful update.

Test Case 2 (Fail): Non-Existent Employee ID
Input: Alex enters Employee ID = 9999, which does not exist, and attempts to update details.
Expected Result: The system displays an error message: "Employee not found. No updates applied."

Test Case 3 (Fail): Database Connection Issue
Input: Alex enters Employee ID = 101 and attempts an update during a database outage.
Expected Result: The system cannot connect to the database and displays an error message. No changes are made.

Task 2: Search Employee by Employee ID
Mark, a manager, needs to find an employee by their ID. He selects "Search Employee," enters the employee ID, and submits the request. The system retrieves and displays the employee's details.

Test Cases for Task 2:
Test Case 1 (Pass): Valid Employee ID
Input: Mark enters Employee ID = 202 and submits a search.
Expected Result: The system displays the employee's name, email, and other details associated with empid=202.

Test Case 2 (Fail): Non-Existent Employee ID
Input: Mark enters Employee ID = 9999.
Expected Result: The system reports "No matching employee found."

Test Case 3 (Fail): Invalid Format for Employee ID
Input: Mark enters “ID202” instead of a numeric value.
Expected Result: The system prompts Mark to enter a valid numeric Employee ID.

Task 3: Update Salary for All Employees Below a Certain Amount
Michael, a payroll manager, wants to update salaries for all employees earning below a certain threshold. He selects “Update Salaries,” enters the salary threshold and the percentage increase, and the system updates all qualifying employees’ salaries.

Test Cases for Task 3:
Test Case 1 (Pass): Valid Threshold and Percentage
Input: Michael sets the salary threshold to $50,000 and chooses a 5% increase.
Expected Result: All employees earning below $50,000 receive a 5% salary increase. A summary confirms the updates.

Test Case 2 (Fail): No Employees Below Threshold
Input: Michael sets the threshold to $10,000, lower than any current salary.
Expected Result: No changes are made, and the system reports “No employees met the criteria.”

Test Case 3 (Fail): Invalid Percentage Input
Input: Michael enters a negative percentage (-5%).
Expected Result: The system rejects the input and asks for a positive increment percentage.

Task 4: Add a New Employee
Emily, a recruiter, adds a new employee using the “Add Employee” option. She enters the employee’s name, email, and hire date. Upon submitting, the system confirms the new employee addition and provides a unique empid.

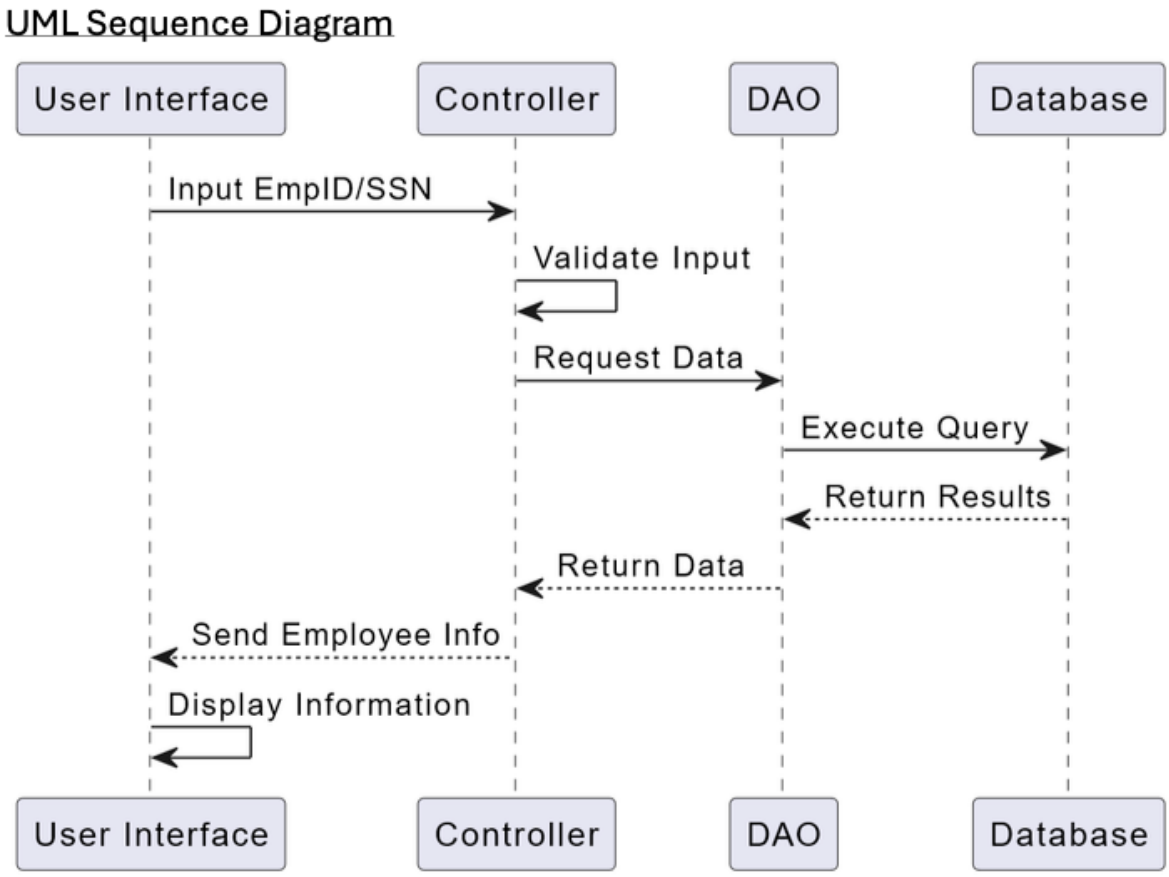
Test Cases for Task 4:
Test Case 1 (Pass): Valid Employee Information
Input: Emily enters “Alice Johnson”, “alice.johnson@example.com”, and hire date “2024-12-01”.
Expected Result: A new employee record is created. The system displays a confirmation message and a unique empid for Alice.

Test Case 2 (Fail): Missing Mandatory Fields
Input: Emily provides only the name “Alice Johnson” but no email or hire date.
Expected Result: The system displays an error indicating that all mandatory fields (email, hire date) must be provided before adding the employee.

Test Case 3 (Fail): Database Connection Issue
Input: Emily provides complete details, but the database is down.
Expected Result: The system shows a connection error and no new employee is added.

5.0 UML SEQUENCE DIAGRAM

The provided EmployeeManagementSystem.java class, when run, follows the sequence logic depicted in the UML Sequence Diagram:



- The User Interface role is simulated by the main method's code (command-line inputs in a real scenario).
- The Controller role corresponds to EmployeeManagementSystem main logic deciding which EmployeeService methods to call.
- The DAO functionality is represented by EmployeeService, which communicates with the Database via JDBC queries.

The sequence of calls in the code aligns with the diagram: Input (simulated in code) → EmployeeService methods → Prepared Statements and SQL queries → Database results → returned data → displayed output.

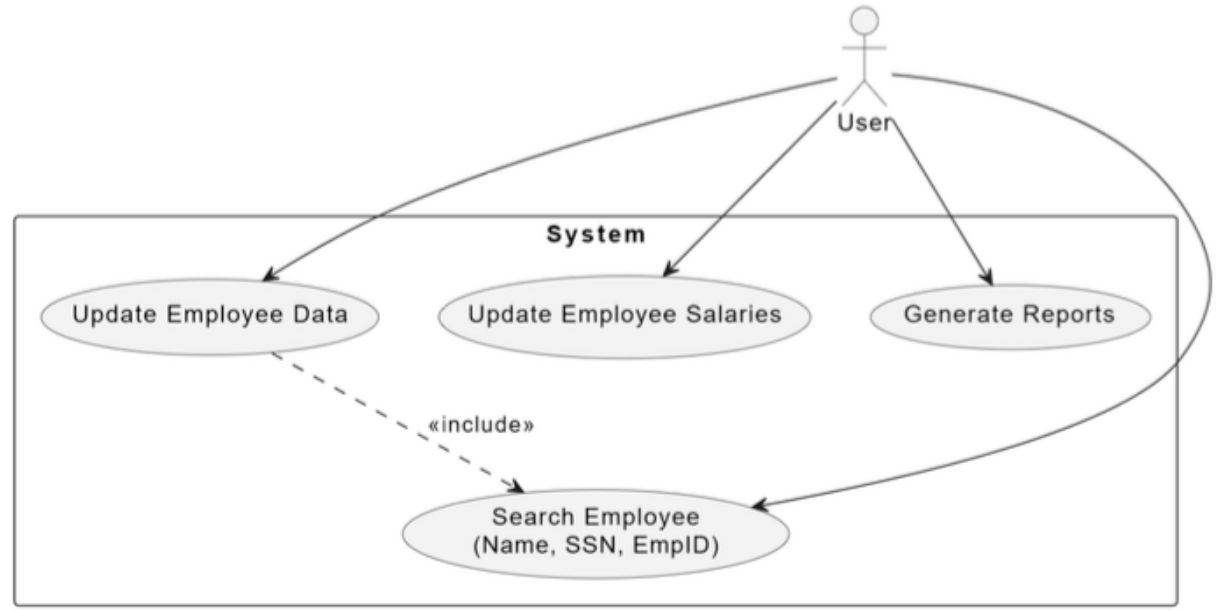
6.0 UML USE CASE DIAGRAM

The UML Use Case Diagram provides a high-level perspective of the system's capabilities and the interactions between the user and the system. Each oval represents a use case—an action or service the system performs that yields value for the user.

Key Use Cases:

- **Search Employee (Name, SSN, EmpID):**
  A foundational action needed before updating data or salaries. Since both "Update Employee Data" and "Update Employee Salaries" often require identifying a specific employee, "Search Employee" is included within them, as indicated by an «include» relationship.
- **Update Employee Data:**
  Allows modification of employee details (e.g., name, email, SSN) after searching for the correct employee record.
- **Update Employee Salaries:**
  Enables applying a percentage-based salary increase to employees within a specified salary range, typically following an initial search to confirm the target group.
- **Generate Reports:**
  Lets the user produce reports such as total pay by job title or by division, offering valuable insights into the company's payroll distribution.

UML Use Case Diagram



Explanation:

• The **User** actor represents an HR administrator or a payroll specialist interacting with the system.

• The **System** boundary encapsulates the functionalities provided.

• The diagram shows the relationships:

  • **Update Employee Data** → «include» → **Search Employee**

  • **Update Employee Salaries** → «include» → **Search Employee**

7.0 CODE INTEGRATION

Key Implementation Files:

• **Employee.java**: Defines the Employee entity.

• **DatabaseConnection.java**: Manages a single, reusable JDBC connection.

• **EmployeeService.java**: Encapsulates business logic for CRUD operations, salary updates, and report generation.

• **EmployeeManagementSystem.java**: Demonstrates the system's features via console-based interactions.

Example Usage:

```
EmployeeService service = new EmployeeService();

// Insert a new employee
Employee emp = new Employee(0, "John", "Doe", "john.doe@example.com", "123-45-6789",
60000);
service.insertEmployee(emp);

// Search for the employee
Employee found = service.searchEmployee("John");
System.out.println("Found: " + found.getFirstName() + " " + found.getLastName());

// Update employee data
service.updateEmployee(found.getEmpid(), "John", "Smith", "john.smith@example.com", "987-
65-4321");
```

```
// Update salary range
service.updateSalary(58000, 105000, 3.2);

// Generate pay reports by job title
List<String> report = service.generatePayByJobTitle();
report.forEach(System.out::println);
```