CSc3350 Software Development

Software Design Document

Group Team Project Fall 2025

TEAM 8

Aayoush Chowdhury, Zaim Shakeel, Humza Fawwaz, Philani Ruwoko

Date: (12/08/2025)

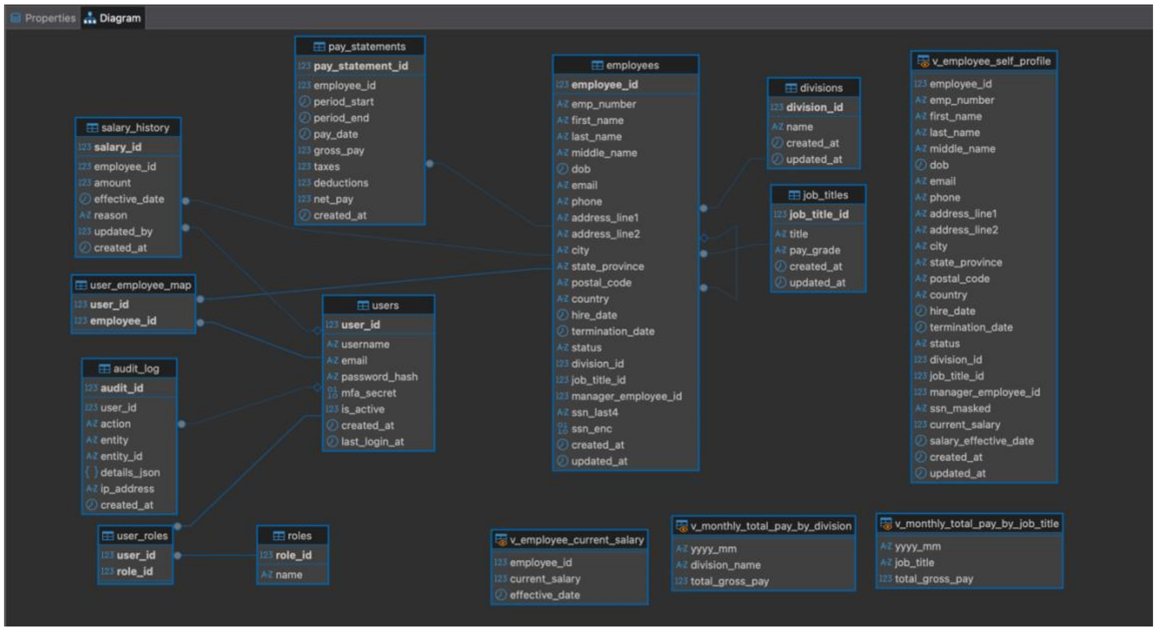
TABLE OF CONTENTS

1. INTRODUCTION

1.1. Software Purpose

The purpose of the Employee Management System (EMS) is to provide a secure, efficient, and scalable software solution for Company Z to manage employee data, payroll, and related administrative tasks. The system will replace the current manual process of using dBeaver and MySQL scripts with a structured Java-based application featuring role-based access control (HR Admin and General Employee) and a MySQL database backend. The intended audience includes HR administrators who need full CRUD capabilities and employees who require read-only access to their personal and pay-related information. The system is designed to support approximately 55 current employees with scalability to triple that amounts within 18 months.

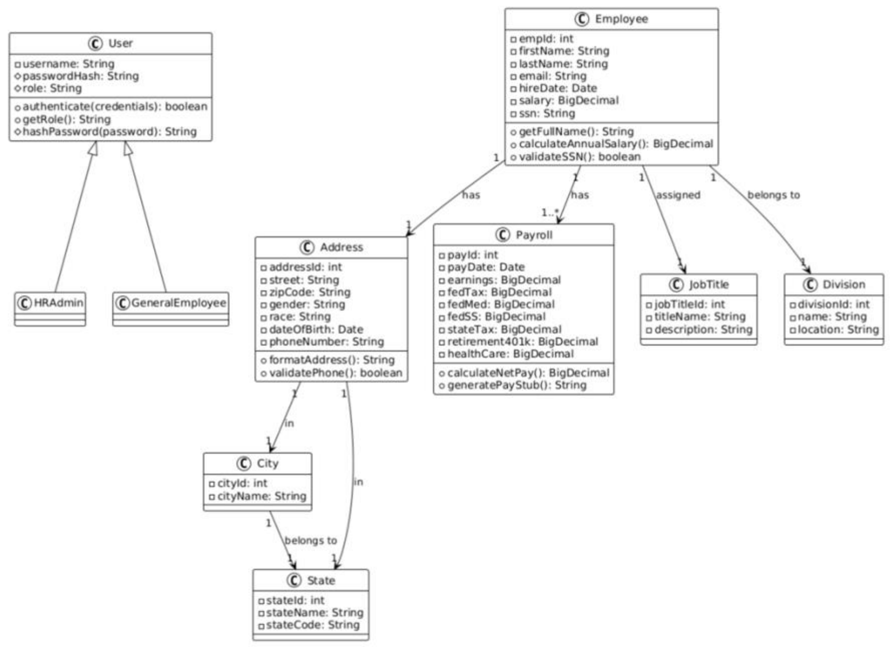
2. DATABASE SCHEMA DIAGRAM



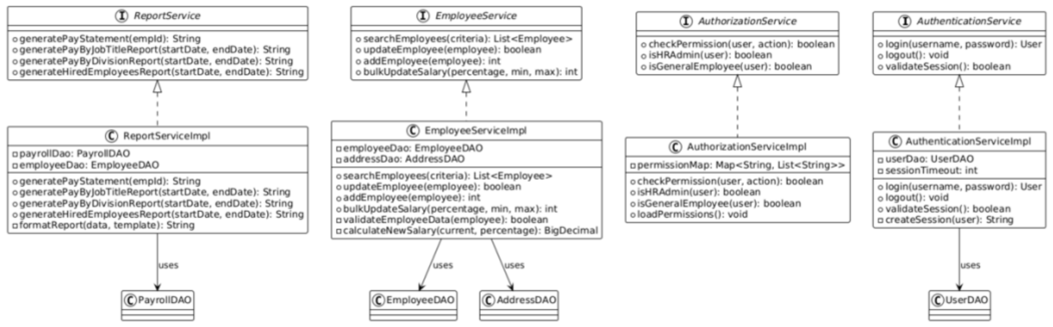
3. JAVA CLASS DIAGRAMS

3.1 Domain Model Classes (Figure 3.1)

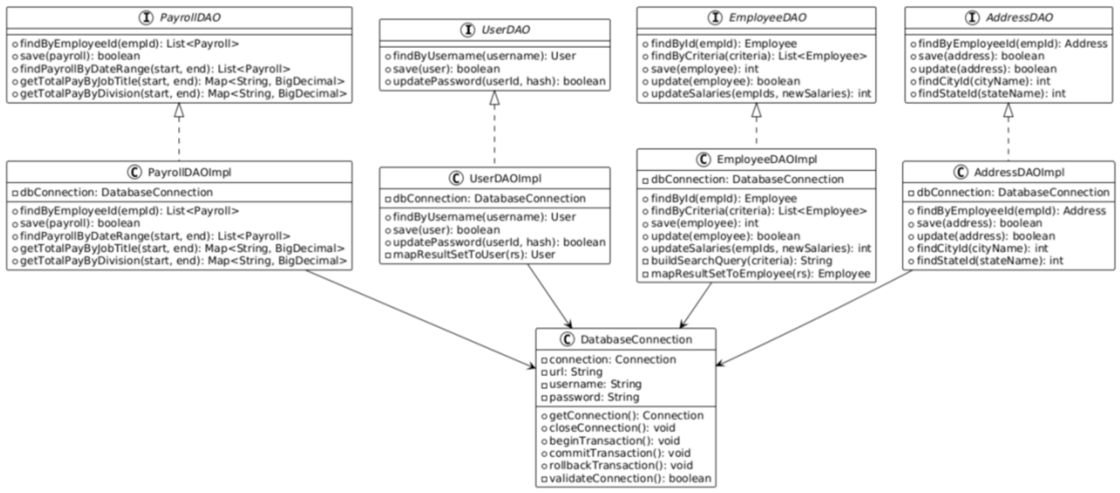
This diagram shows the core domain entities including Employee, User, Address, Payroll, JobTitle, and Division. The User class demonstrates inheritance with HRAdmin and GeneralEmployee subclasses, implementing role-based access control. Relationships between entities are shown through composition and association.



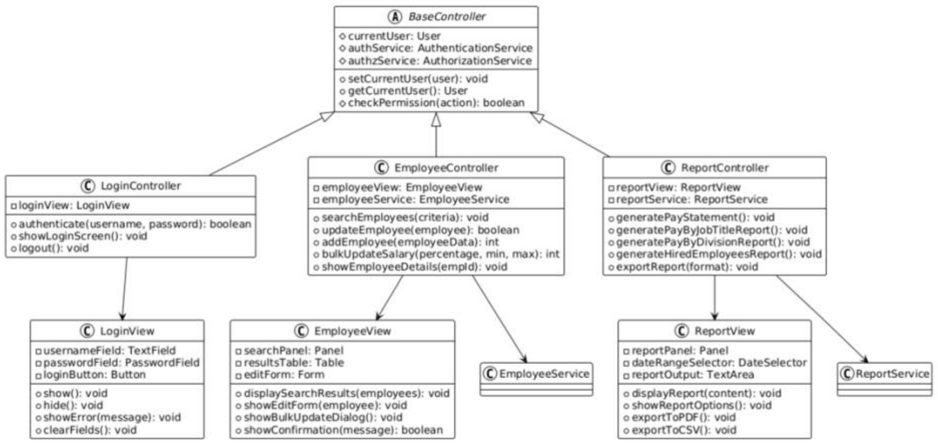
3.2 Service Layer Architecture (Figure 3.2)
The service layer consists of four main interfaces: AuthenticationService, AuthorizationService, EmployeeService, and ReportService. Each interface has a corresponding implementation class that coordinates business logic and interacts with the DAO layer.


ReportService
generatePayStatement(empId): String
generatePayByJobTitleReport(startDate, endDate): String
generatePayByDivisionReport(startDate, endDate): String
generateHiredEmployeesReport(startDate, endDate): String
ReportServiceImpl
payrollDao: PayrollDAO
employeeDao: EmployeeDAO
generatePayStatement(empId): String
generatePayByJobTitleReport(startDate, endDate): String
generatePayByDivisionReport(startDate, endDate): String
generateHiredEmployeesReport(startDate, endDate): String
formatReport(data, template): String
uses
PayrollDAO
EmployeeService
searchEmployees(criteria): List<Employee>
updateEmployee(employee): boolean
addEmployee(employee): int
bulkUpdateSalary(percentage, min, max): int
EmployeeServiceImpl
employeeDao: EmployeeDAO
addressDao: AddressDAO
searchEmployees(criteria): List<Employee>
updateEmployee(employee): boolean
addEmployee(employee): int
bulkUpdateSalary(percentage, min, max): int
validateEmployeeData(employee): boolean
calculateNewSalary(current, percentage): BigDecimal
uses
uses
EmployeeDAO
AddressDAO
AuthorizationService
checkPermission(user, action): boolean
isHRAdmin(user): boolean
isGeneralEmployee(user): boolean
AuthorizationServiceImpl
permissionMap: Map<String, List<String>>
checkPermission(user, action): boolean
isHRAdmin(user): boolean
isGeneralEmployee(user): boolean
loadPermissions(): void
AuthenticationService
login(username, password): User
logout(): void
validateSession(): boolean
AuthenticationServiceImpl
userDao: UserDAO
sessionTimeout: int
login(username, password): User
logout(): void
validateSession(): boolean
createSession(user): String
uses
UserDAO

3.3 Data Access Layer (Figure 3.3)
The DAO layer provides database abstraction through interface-implementation pairs for Employee, User, Payroll, and Address entities. All DAO implementations depend on the DatabaseConnection class which manages JDBC connections and transaction handling.


PayrollDAO
findByEmployeeId(empId): List<Payroll>
save(payroll): boolean
findPayrollByDateRange(start, end): List<Payroll>
getTotalPayByJobTitle(start, end): Map<String, BigDecimal>
getTotalPayByDivision(start, end): Map<String, BigDecimal>
UserDAO
findByUsername(username): User
save(user): boolean
updatePassword(userId, hash): boolean
EmployeeDAO
findById(empId): Employee
findByCriteria(criteria): List<Employee>
save(employee): int
update(employee): boolean
updateSalaries(empIds, newSalaries): int
AddressDAO
findByEmployeeId(empId): Address
save(address): boolean
update(address): boolean
findCityId(cityName): int
findStateId(stateName): int
PayrollDAOImpl
dbConnection: DatabaseConnection
findByEmployeeId(empId): List<Payroll>
save(payroll): boolean
findPayrollByDateRange(start, end): List<Payroll>
getTotalPayByJobTitle(start, end): Map<String, BigDecimal>
getTotalPayByDivision(start, end): Map<String, BigDecimal>
UserDAOImpl
dbConnection: DatabaseConnection
findByUsername(username): User
save(user): boolean
updatePassword(userId, hash): boolean
mapResultSetToUser(rs): User
EmployeeDAOImpl
dbConnection: DatabaseConnection
findById(empId): Employee
findByCriteria(criteria): List<Employee>
save(employee): int
update(employee): boolean
updateSalaries(empIds, newSalaries): int
buildSearchQuery(criteria): String
mapResultSetToEmployee(rs): Employee
AddressDAOImpl
dbConnection: DatabaseConnection
findByEmployeeId(empId): Address
save(address): boolean
update(address): boolean
findCityId(cityName): int
findStateId(stateName): int
DatabaseConnection
connection: Connection
url: String
username: String
password: String
getConnection(): Connection
closeConnection(): void
beginTransaction(): void
commitTransaction(): void
rollbackTransaction(): void
validateConnection(): boolean

3.4 Controller and View Classes (Figure 3.4)
The MVC pattern is implemented through the BaseController abstract class, which is extended by LoginController, EmployeeController, and ReportController. Each controller manages its corresponding view component and coordinates with service layer classes.


BaseController
currentUser: User
authService: AuthenticationService
authzService: AuthorizationService
setCurrentUser(user): void
getCurrentUser(): User
checkPermission(action): boolean
LoginController
loginView: LoginView
authenticate(username, password): boolean
showLoginScreen(): void
logout(): void
EmployeeController
employeeView: EmployeeView
employeeService: EmployeeService
searchEmployees(criteria): void
updateEmployee(employee): boolean
addEmployee(employeeData): int
bulkUpdateSalary(percentage, min, max): int
showEmployeeDetails(empId): void
ReportController
reportView: ReportView
reportService: ReportService
generatePayStatement(): void
generatePayByJobTitleReport(): void
generatePayByDivisionReport(): void
generateHiredEmployeesReport(): void
exportReport(format): void
LoginView
usernameField: TextField
passwordField: PasswordField
loginButton: Button
show(): void
hide(): void
showError(message): void
clearFields(): void
EmployeeView
searchPanel: Panel
resultsTable: Table
editForm: Form
displaySearchResults(employees): void
showEditForm(employee): void
showBulkUpdateDialog(): void
showConfirmation(message): boolean
EmployeeService
ReportView
reportPanel: Panel
dateRangeSelector: DateSelector
reportOutput: TextArea
displayReport(content): void
showReportOptions(): void
exportToPDF(): void
exportToCSV(): void
ReportService

4. PROGRAMMING TASKS

1. User Authentication and Authorization System
   Implement a secure login system that authenticates users and determines their role (HR Admin or General Employee). The system must validate credentials against the MySQL database and establish appropriate permission levels for CRUD operations vs. read-only access.
   Components:
   - User class with username, password (hashed), and role attributes

   - Authentication method to verify credentials
   - Authorization method to check permissions before database operations
   - Session management to maintain logged-in state

2. Employee Search Functionality
   Create a search module that allows users to query employee data using multiple criteria (name, DOB, SSN, empid). The search must respect user permissions: HR Admin can search for editing purposes, while General Employees can only search to view their own data.
   Components:
   - Search interface accepting multiple search parameters
   - SQL query builder that constructs dynamic WHERE clauses
   - Result set handler that returns employee data
   - Permission filter that restricts results based on user role
3. Employee Data Update Module (HR Admin)
   Develop functionality for HR Admin to update employee information after selecting from search results. This includes updating personal information in the employees table and related data in the address table (street, city, state, zip, gender, race, DOB, phone).
   Components:
   - Employee data model class with all attributes
   - Update method that accepts modified employee object
   - SQL UPDATE statement generator
   - Validation for required fields and data types
   - Transaction management to ensure data integrity across multiple tables
4. Bulk Salary Update with Percentage and Range
   Implement a salary increase function that updates multiple employees' salaries by a specified percentage, but only for employees whose current salary falls within a given range. For example, 3.2% increase for salaries ≥$58K and <$105K.
   Components:
   - Input parameters: percentage, minimum salary, maximum salary
   - SQL query to select eligible employees
   - Calculation logic to compute new salaries
   - Batch UPDATE operation
   - Confirmation mechanism showing affected employees before commit
5. Employee Pay Statement History Report
   Create a report generation system for General Employees to view their own pay statement history, sorted by most recent pay date. The report pulls data from the payroll table and displays payment information in descending chronological order.
   Components:
   - Report generator class
   - SQL JOIN query connecting employees and payroll tables
   - Date sorting functionality (DESC order)
   - Permission check to ensure employees only see their own data
   - Formatted output displaying: pay date, amount, period covered, deductions
6. New Employee Insertion Module
   Develop functionality for HR Admin to add a new employee with complete profile including personal details, address, division, and job title.

Components:
- Employee creation form with validation
- Address assignment with city/state lookup
- Division and job title assignment
- Transactional insert across multiple tables

7. Report: Total Pay by Job Title (Monthly)
   Generate a monthly payroll summary report grouped by job title, accessible only by HR Admin.
   Components:
   - Date range selection interface
   - SQL aggregation with GROUP BY on job title
   - Formatted tabular output
   - Permission validation for HR Admin only
8. Report: Total Pay by Division (Monthly)
   Generate a monthly payroll summary report grouped by division, accessible only by HR Admin.
   Components:
   - Date range selection interface
   - SQL JOIN across payroll, employee, and division tables
   - Aggregation by division ID and name
   - Permission validation
9. Report: Employees Hired Within Date Range
   Generate a report listing all employees hired between two dates, including their personal and job information.
   Components:
   - Date picker for start and end dates
   - SQL query with BETWEEN clause on HireDate
   - Joined data from employees, address, division, and job titles
   - Export option (CSV/PDF)
10. Session Management and Logout System
    Implement secure session handling including automatic timeout, session cleanup, and safe logout.
    Components:
    - Session token generation and validation
    - Inactivity timeout (e.g., 30 minutes)
    - Secure logout that invalidates session
    - Audit logging of login/logout events

5. TEST CASES (cleaned up from #6 in Deliverables)

This section describes comprehensive test cases for three critical system functions: employee data updates, employee search, and bulk salary updates. Each test case includes preconditions, inputs, expected results, and verification methods to ensure proper system functionality and data integrity.

5.1 Test Case A: Update Employee Data

User Story/Scenario:

As an HR Administrator, I need to update employee information (personal and address data) after searching for an employee, so that I can maintain accurate employee records in the system. The update must maintain data integrity across multiple tables (employees and address) and validate all inputs before committing changes. The system must prevent unauthorized users from modifying employee data and ensure that all database transactions can be rolled back if any part of the update fails.

Programming Task Description:

Create a comprehensive employee data update function that modifies employee information across multiple tables (employees, address). The function must validate input data, maintain referential integrity, handle transactions, and work with the Employee and Address class structures.

Test Cases:

Test Case A1: Valid Employee Data Update (Personal Info)

- Pre-condition: HR Admin logged in, valid empid exists (e.g., empid=1001)
- Input: empid=1001, firstName="John", lastName="Smith", DOB="1985-03-15", gender="M", race="Caucasian", phone="404-555-1234"
- Expected Result: PASS - Database updated successfully, confirmation message displayed
- Verification: Query database to confirm all fields updated correctly

Test Case A2: Valid Employee Data Update (Address Info)

- Pre-condition: HR Admin logged in, valid empid exists (e.g., empid=1001)
- Input: empid=1001, street="123 Main St", city_id=5 (Atlanta), state_id=10 (GA), zip="30301"
- Expected Result: PASS - Address table updated successfully
- Verification: Query address table to confirm changes

Test Case A3: Update with Invalid Employee ID

- Pre-condition: HR Admin logged in
- Input: empid=9999 (non-existent), firstName="Jane", lastName="Doe"
- Expected Result: FAIL - Error message: "Employee ID not found"
- Verification: No database changes occur

Test Case A4: Update with Missing Required Fields

- Pre-condition: HR Admin logged in, valid empid=1002
- Input: empid=1002, firstName="", lastName="" (empty required fields)
- Expected Result: FAIL - Error message: "Required fields cannot be empty"
- Verification: No database changes occur

Test Case A5: Update with Invalid Data Types

- Pre-condition: HR Admin logged in, valid empid=1003
- Input: empid=1003, DOB="invalid-date", phone="not-a-number"
- Expected Result: FAIL - Error message: "Invalid data format for DOB and phone"
- Verification: No database changes occur

Test Case A6: Unauthorized Update Attempt (General Employee)

- Pre-condition: General Employee logged in
- Input: Attempt to update any employee data
- Expected Result: FAIL - Error message: "Access denied. Insufficient permissions"
- Verification: No database changes occur

Test Case A7: Update with Transaction Rollback

- Pre-condition: HR Admin logged in, valid empid=1004
- Input: empid=1004, valid employee data but invalid foreign key reference (e.g., city_id=999 doesn't exist)
- Expected Result: FAIL - Transaction rolled back, error message displayed
- Verification: No partial updates in database, data remains in original state

5.2 Test Case B: Search for Employee (Admin User)

User Story/Scenario:

As an HR Administrator, I need to search for employees using various criteria (name, DOB, SSN, empid) to locate their records for viewing or editing. The search must support single or multiple criteria and return complete employee information from all related tables (employees, address, division, job_titles). General employees should only be able to search for and view their own information, while HR Admins can search for any employee in the system. The search results must include joined data from multiple tables to provide a complete employee profile.

Programming Task Description:

Implement a flexible search function for HR Admin that accepts multiple search criteria (name, DOB, SSN, empid) individually or in combination. The function must return matching employees with complete information from joined tables (employees, address, division, job_titles) and allow selection for editing.

Test Cases:

Test Case B1: Search by Employee ID (Exact Match)

- Pre-condition: HR Admin logged in, employee with empid=2001 exists
- Input: empid=2001
- Expected Result: PASS - Returns exactly one employee record with all details
- Verification: Displayed data matches database record for empid=2001

Test Case B2: Search by Full Name (Exact Match)

- Pre-condition: HR Admin logged in, employee "Sarah Johnson" exists
- Input: firstName="Sarah", lastName="Johnson"
- Expected Result: PASS - Returns one or more matching records
- Verification: All returned records have matching first and last names

Test Case B3: Search by Partial Name

- Pre-condition: HR Admin logged in, multiple employees with last name starting with "Smith"
- Input: lastName="Smith%" (wildcard search)
- Expected Result: PASS - Returns all employees with last name starting with "Smith"
- Verification: Result set includes Smith, Smithson, Smithers, etc.

Test Case B4: Search by Date of Birth

- Pre-condition: HR Admin logged in, employees with DOB="1990-05-20" exist
- Input: DOB="1990-05-20"
- Expected Result: PASS - Returns all employees born on that date
- Verification: All returned records have matching DOB

Test Case B5: Search by SSN (Exact Match)

- Pre-condition: HR Admin logged in, employee with SSN="123-45-6789" exists
- Input: SSN="123-45-6789"
- Expected Result: PASS - Returns exactly one employee (SSN is unique)
- Verification: Returned record shows correct SSN and employee details

Test Case B6: Search with Multiple Criteria (AND logic)

- Pre-condition: HR Admin logged in
- Input: firstName="John", lastName="Doe", DOB="1985-01-15"
- Expected Result: PASS - Returns employee(s) matching ALL criteria
- Verification: Returned records match all three search parameters

Test Case B7: Search with No Results

- Pre-condition: HR Admin logged in
- Input: empid=99999 (non-existent)
- Expected Result: PASS - Returns empty result set with message "No employees found"
- Verification: No errors, appropriate user feedback displayed

Test Case B8: Search with Invalid Input Format

- Pre-condition: HR Admin logged in
- Input: empid="ABC" (non-numeric for integer field)
- Expected Result: FAIL - Error message: "Invalid employee ID format"
- Verification: No database query executed

Test Case B9: Unauthorized Search (General Employee searching others)

- Pre-condition: General Employee logged in
- Input: Search for empid not matching their own
- Expected Result: FAIL - Error message: "Access denied" or returns no results
- Verification: General employee cannot view other employees' data

Test Case B10: Search Results Display Joined Data

- Pre-condition: HR Admin logged in, empid=3001 exists with address, division, and job title
- Input: empid=3001
- Expected Result: PASS - Returns complete employee profile including:
    - Basic employee info
    - Address (street, city name, state name, zip)
    - Division name
    - Job title
- Verification: All joined table data displayed correctly

5.3 Test Case C: Update Salary for Employees in Range

User Story/Scenario:

As an HR Administrator, I need to apply percentage-based salary increases to groups of employees whose salaries fall within a specified range, so that I can efficiently process annual raises or cost-of-living adjustments without manually updating each employee individually. The system must validate the salary range and percentage, preview the affected employees before committing changes, execute all updates as a single atomic transaction, and provide a summary of the changes made. This ensures accuracy and allows me to verify the updates before they are permanently applied to the database.

Programming Task Description:

Create a bulk salary update function that increases salaries by a specified percentage for all employees whose current salary falls within a defined range. The function must validate inputs, preview affected employees, execute the update as a transaction, and provide summary results.

Test Cases:

Test Case C1: Valid Bulk Salary Update

- Pre-condition: HR Admin logged in, 10 employees have salaries between $58,000 and $104,999
- Input: percentage=3.2, minSalary=58000, maxSalary=105000
- Expected Result: PASS - All 10 employees' salaries increased by 3.2%, confirmation message shows count
- Verification:
  - Query database to confirm new salaries = old salary × 1.032
  - Employees outside range remain unchanged
  - All 10 updates successful

Test Case C2: Update with Zero Percentage

- Pre-condition: HR Admin logged in
- Input: percentage=0, minSalary=50000, maxSalary=100000
- Expected Result: FAIL or PASS with no changes - Error/warning: "Percentage must be greater than zero"
- Verification: No salary changes occur

Test Case C3: Update with Negative Percentage (Salary Decrease)

- Pre-condition: HR Admin logged in, employees exist in range
- Input: percentage=-5.0, minSalary=60000, maxSalary=80000
- Expected Result: PASS - Salaries decreased by 5% (if business rules allow) OR FAIL with validation error
- Verification: Depends on business requirements; verify intended behavior

Test Case C4: Update with Invalid Range (Min > Max)

- Pre-condition: HR Admin logged in
- Input: percentage=5.0, minSalary=100000, maxSalary=50000
- Expected Result: FAIL - Error message: "Minimum salary cannot exceed maximum salary"
- Verification: No database query executed

Test Case C5: Update with No Matching Employees

- Pre-condition: HR Admin logged in, no employees have salaries between $200,000-$300,000
- Input: percentage=10.0, minSalary=200000, maxSalary=300000

- Expected Result: PASS - Message: "No employees found in specified salary range. 0 records updated"
- Verification: No errors, appropriate feedback

Test Case C6: Update Boundary Conditions (Inclusive)

- Pre-condition: HR Admin logged in, employees exist with salaries exactly \$58,000 and \$105,000
- Input: percentage=3.0, minSalary=58000, maxSalary=105000
- Expected Result: PASS - Employees with exactly \$58,000 ARE updated, employees with exactly \$105,000 are NOT updated (per requirement: ≥58K but <105K)
- Verification: Check boundary salary employees specifically

Test Case C7: Preview Before Update

- Pre-condition: HR Admin logged in, 5 employees in range \$70,000-\$90,000
- Input: percentage=4.5, minSalary=70000, maxSalary=90000, previewMode=true
- Expected Result: PASS - Displays list of 5 employees with current and proposed new salaries, no database changes yet
- Verification: Database unchanged, preview data accurate

Test Case C8: Unauthorized Bulk Update (General Employee)

- Pre-condition: General Employee logged in
- Input: percentage=5.0, minSalary=50000, maxSalary=100000
- Expected Result: FAIL - Error message: "Access denied. Insufficient permissions"
- Verification: No salary changes occur

Test Case C9: Update with Very Large Percentage

- Pre-condition: HR Admin logged in, employees exist in range
- Input: percentage=500.0 (500% increase), minSalary=60000, maxSalary=80000
- Expected Result: WARNING or CONFIRMATION PROMPT - "This will increase salaries by 500%. Confirm? (Y/N)"
- Verification: Requires explicit confirmation before executing

Test Case C10: Transaction Integrity (Database Failure Mid-Update)

- Pre-condition: HR Admin logged in, 20 employees in range, simulate database connection failure after 10 updates
- Input: percentage=3.0, minSalary=50000, maxSalary=100000
- Expected Result: FAIL - Transaction rolled back, error message, NO partial updates
- Verification: All 20 employees have original salaries (rollback successful)

Test Case C11: Decimal Precision Handling

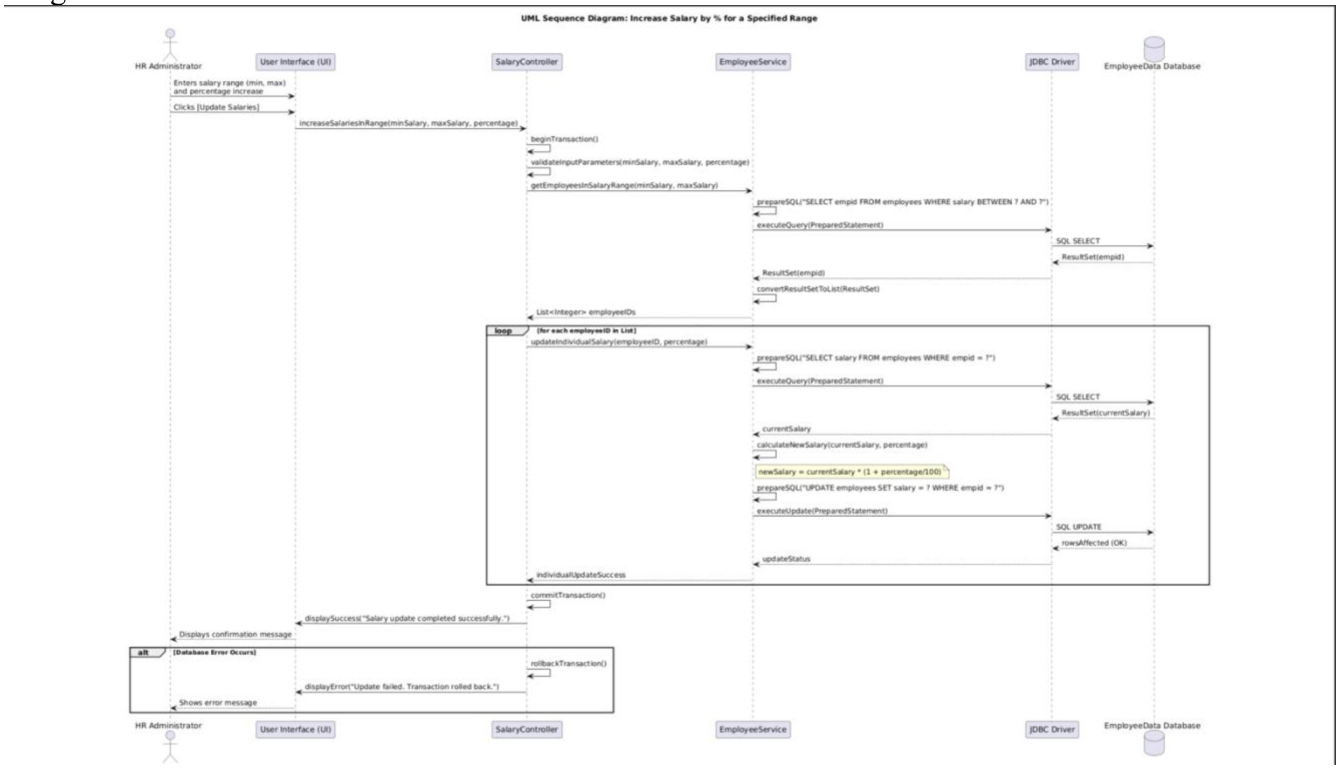
- Pre-condition: HR Admin logged in, employee has salary=$75,123.45
- Input: percentage=3.25, minSalary=70000, maxSalary=80000
- Expected Result: PASS - New salary = $77,569.11 (rounded to 2 decimal places)
- Verification: Database stores correct decimal value, no rounding errors

6. SEQUENCE DIAGRAMS (cleaned up from #7 in Deliverables)

6.1 Sequence Diagram: Increase Salary by % for a Specified Range
Diagram Description:

This diagram shows the bulk salary update workflow. An HR Admin enters a percentage increase and salary range through the UI. The request flows to EmployeeService, which validates inputs, begins a transaction, and queries for eligible employees with salary BETWEEN min AND max. For each eligible employee, the service calculates newSalary = currentSalary × (1 + percentage/100) and executes an UPDATE. If all updates succeed, the transaction commits and a success message show the count of updated employees. If any error occurs, the transaction rolls back completely, and an error message is displayed.

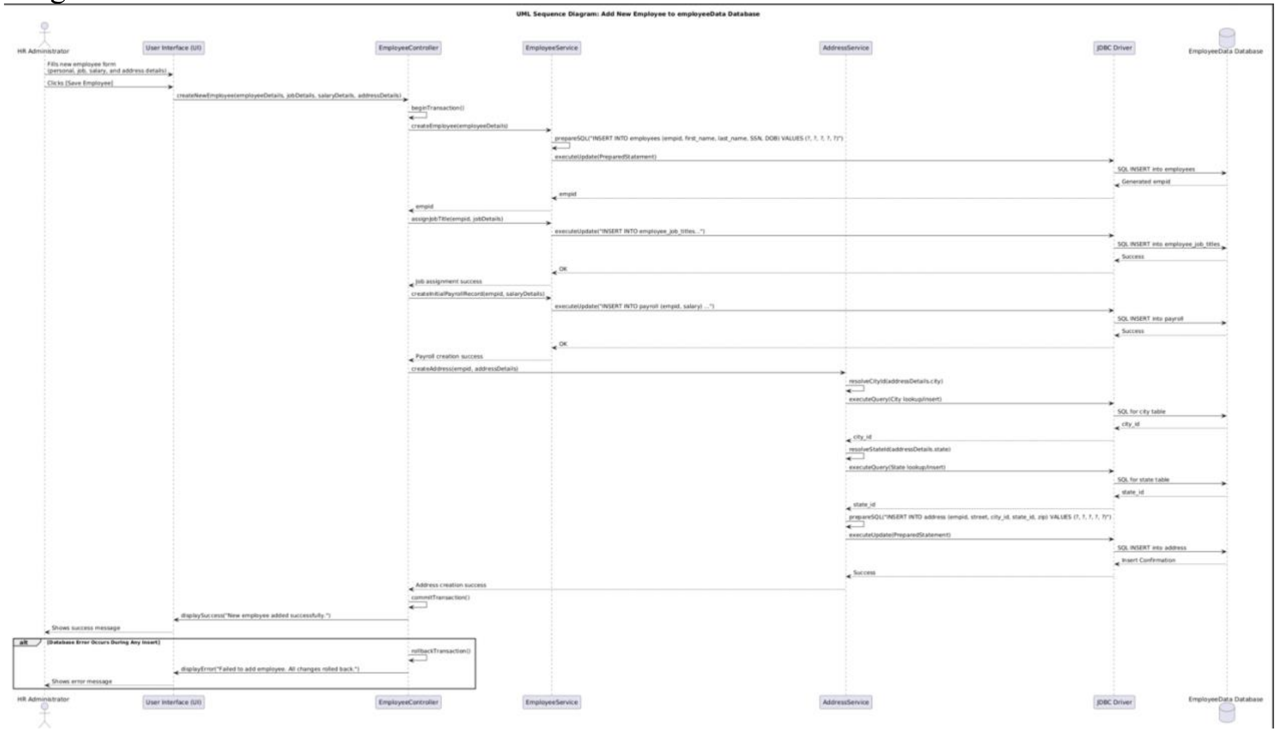
Diagram:


UML Sequence Diagram: Increase Salary by % for a Specified Range
HR Administrator
User Interface (UI)
SalaryController
EmployeeService
JDBC Driver
EmployeeData Database
Enters salary range (min, max) and percentage increase
Clicks [Update Salaries]
increaseSalariesInRange(minSalary, maxSalary, percentage)
beginTransaction()
validateInputParameters(minSalary, maxSalary, percentage)
getEmployeesInSalaryRange(minSalary, maxSalary)
prepareSQL("SELECT empid FROM employees WHERE salary BETWEEN ? AND ?")
executeQuery(PreparedStatement)
SQL SELECT
ResultSet(empid)
ResultSet(empid)
convertResultSetToList(ResultSet)
List<Integer> employeeIDs
loop
[for each employeeID in List]
updateIndividualSalary(employeeID, percentage)
prepareSQL("SELECT salary FROM employees WHERE empid = ?")
executeQuery(PreparedStatement)
SQL SELECT
ResultSet(currentSalary)
currentSalary
calculateNewSalary(currentSalary, percentage)
newSalary = currentSalary * (1 + percentage/100)
prepareSQL("UPDATE employees SET salary = ? WHERE empid = ?")
executeUpdate(PreparedStatement)
SQL UPDATE
rowsAffected (OK)
updateStatus
individualUpdateSuccess
commitTransaction()
displaySuccess("Salary update completed successfully.")
Displays confirmation message
alt
[Database Error Occurs]
rollbackTransaction()
displayError("Update failed. Transaction rolled back.")
Shows error message
HR Administrator
User Interface (UI)
SalaryController
EmployeeService
JDBC Driver
EmployeeData Database

6.2 Sequence Diagram: Add New Employee to Database
Diagram Description:

This diagram shows the multi-table insertion process for adding a new employee. An HR Admin submits a completed employee form via the UI. EmployeeService starts a transaction and first inserts the core employee record into the employees table. The generated empId is then used to insert related records into employee_job_titles, payroll, and address tables (the latter via AddressService, which resolves city/state IDs). If all inserts succeed, the transaction commits and a success confirmation is shown. If any step fails, the entire transaction rolls back, ensuring no partial data is saved, and an error is reported.

Diagram:


UML Sequence Diagram: Add New Employee to employeeData Database
HR Administrator
User Interface (UI)
EmployeeController
EmployeeService
AddressService
JDBC Driver
EmployeeData Database
Fills new employee form (personal, job, salary, and address details)
Clicks [Save Employee]
createNewEmployee(employeeDetails, jobDetails, salaryDetails, addressDetails)
beginTransaction()
createEmployee(employeeDetails)
prepareSQL("INSERT INTO employees (empid, first_name, last_name, SSN, DOB) VALUES (?, ?, ?, ?, ?)")
executeUpdate(PreparedStatement)
SQL INSERT into employees
Generated empid
empid
empid
assignJobTitle(empid, jobDetails)
executeUpdate("INSERT INTO employee_job_titles...")
SQL INSERT into employee_job_titles
Success
OK
Job assignment success
createInitialPayrollRecord(empid, salaryDetails)
executeUpdate("INSERT INTO payroll (empid, salary) ...")
SQL INSERT into payroll
Success
OK
Payroll creation success
createAddress(empid, addressDetails)
resolveCityId(addressDetails.city)
executeQuery(City lookup/insert)
SQL for city table
city_id
city_id
resolveStateId(addressDetails.state)
executeQuery(State lookup/insert)
SQL for state table
state_id
state_id
prepareSQL("INSERT INTO address (empid, street, city_id, state_id, zip) VALUES (?, ?, ?, ?, ?)")
executeUpdate(PreparedStatement)
SQL INSERT into address
Insert Confirmation
Success
Address creation success
commitTransaction()
displaySuccess("New employee added successfully.")
Shows success message
alt
[Database Error Occurs During Any Insert]
rollbackTransaction()
displayError("Failed to add employee. All changes rolled back.")
Shows error message

7. APPENDIX

7.1. Definitions and Acronyms

Acronyms

API – Application Programming Interface: A set of protocols and tools for building software applications that specify how software components should interact.

CRUD – Create, Read, Update, Delete: The four basic operations of persistent storage in database management systems.

DAO – Data Access Object: A design pattern that provides an abstract interface to a database or other persistence mechanism, separating low-level data access operations from high-level business logic.

DOB – Date of Birth: An employee's birth date stored in the system.

EMS – Employee Management System: The software system developed for Company Z to manage employee data and payroll.

FK – Foreign Key: A field in a database table that uniquely identifies a row in another table, establishing a relationship between the two tables.

GUI – Graphical User Interface: A visual interface that allows users to interact with the application through graphical elements rather than text-based commands.

HR – Human Resources: The department responsible for managing employee-related functions and the primary administrator of the EMS.

JDBC – Java Database Connectivity: A Java API that enables Java applications to interact with databases through SQL statements.

MVC – Model-View-Controller: A software architectural pattern that separates an application into three interconnected components to separate internal representations of information from how information is presented to the user.

PK – Primary Key: A unique identifier for a record in a database table that cannot be null and must be unique across all records.

SQL – Structured Query Language: A standardized programming language used for managing and manipulating relational databases.

SSN – Social Security Number: A unique nine-digit identification number issued to U.S. citizens and residents, used as a unique identifier for employees.

UML – Unified Modeling Language: A standardized modeling language consisting of integrated diagrams used to visualize the design of a software system.

UI/UX – User Interface/User Experience: The visual elements and interaction design through which users engage with the application.

Technical Terms

Abstract Class: A class that cannot be instantiated and is designed to be subclassed. It may contain abstract methods that must be implemented by subclasses. BaseController in the system architecture is an abstract class.

Aggregation: A relationship where one class is composed of or contains another class, but the contained class can exist independently. Service classes using DAO objects demonstrate aggregation.

Authentication: The process of verifying the identity of a user attempting to access the system through credentials such as username and password.

Authorization: The process of determining what actions an authenticated user is permitted to perform within the system. HR Admin users have different permissions than General Employee users.

Batch Update: A database operation that executes multiple UPDATE statements as a single unit, improving performance and ensuring consistency.

Composition: A strong form of aggregation where the contained object cannot exist independently of the container. Employee has Address demonstrates composition.

Database Connection Pool: A cache of database connections maintained to improve performance by reusing connections rather than creating new ones for each request.

Hashing: A one-way cryptographic function that transforms a password into a fixed-length string of characters, making it secure for storage.

Implementation Class: A concrete class that provides the actual code for methods declared in an interface. EmployeeServiceImpl implements EmployeeService interface.

Inheritance: An object-oriented programming concept where a class (subclass) derives properties and methods from another class (superclass). HRAdmin extends User demonstrates inheritance.

Interface: A contract in Java that specifies a set of method signatures that implementing classes must provide, without defining the implementation.

Join: A SQL operation that combines rows from two or more tables based on a related column between them.

Normalization: The process of organizing database tables to reduce redundancy and improve data integrity by dividing large tables into smaller, related tables.

Referential Integrity: A database constraint that ensures relationships between tables remain consistent, preventing orphaned records.

Role-Based Access Control (RBAC): A security approach that restricts system access based on the user's role within the organization. The system implements two roles: HR Admin and General Employee.

Service Layer: An architectural layer that contains business logic and coordinates between the presentation layer (controllers) and data access layer (DAOs).

Session: A temporary state that tracks a user's interaction with the application from login to logout, maintaining authentication and authorization status.

Transaction: A sequence of database operations that are treated as a single unit of work. All operations must succeed for the transaction to commit; if any operation fails, all changes are rolled back.

Transaction Rollback: The process of undoing all database changes made within a transaction when an error occurs, returning the database to its state before the transaction began.

Validation: The process of checking user input to ensure it meets required formats, types, and business rules before processing or storing in the database.

Wildcard Search: A search technique using special characters (% in SQL) to match patterns rather than exact values, enabling partial name searches.