Employee Management System

Introduction:

This project implements an Employee Management System (EMS) using the Singleton and Factory design patterns,Observer, Strategy, and Template. The EMS provides a GUI for managing employees and departments, integrates with an SQL Server database, and uses object-oriented design principles to ensure modularity and maintainability.

Github link :https://github.com/AbdulrahmanElshaphei/Employee-Management-System.git

---

1-Singleton Pattern

Purpose

The Singleton pattern ensures that a class has only one instance while providing a global point of access to it. In this project, the Singleton pattern is used for:

1. **Database Connection: Ensures only one instance of the database connection is created throughout the application's lifecycle.**
2. **Payroll System: Ensures the payroll system is centralized and consistent.**

Implementation

1. Database Connection Singleton

Class Name: `DatabaseConnection`

Responsibilities:

- **Create a single database connection instance.**
- **Provide a global access point to the connection.**

Code:

```
package com.ems.core;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnection {
    private static final String URL = "jdbc:sqlserver://localhost:1433;databaseName=ems";
    private static final String USER = "sa";
    private static final String PASSWORD = "your_password";
    private static Connection connection;

    private DatabaseConnection() {}

    public static Connection getInstance() {
        if (connection == null) {
            try {
                connection = DriverManager.getConnection(URL, USER, PASSWORD);
                System.out.println("Connected to SQL Server database.");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return connection;
    }
}
```

How It Works:

- **The `getInstance` method initializes the connection only once.**
- **Subsequent calls return the same connection instance.**

2. Payroll System Singleton

Class Name: `PayrollSystem`

Responsibilities:

- **Centralize payroll processing.**
- **Maintain consistency in salary calculations.**

Code:

```
package com.ems.core;

import com.ems.model.Employee;

public class PayrollSystem {
    private static PayrollSystem instance;

    private PayrollSystem() {}

    public static PayrollSystem getInstance() {
        if (instance == null) {
            instance = new PayrollSystem();
        }
        return instance;
    }

    public void processPayroll(Employee employee) {
        System.out.println("Processing payroll for: " + employee.getName());
        System.out.println("Salary: " + employee.getSalary());
    }
}
```

How It Works:

- **The `getInstance` method ensures only one `PayrollSystem` instance exists.**
- **The `processPayroll` method is accessible globally.**

2- Factory Pattern

Purpose

The Factory pattern provides an interface for creating objects without specifying their concrete classes. This project uses the Factory pattern for:

1. **Employee Creation: To instantiate different types of employees (e.g., Full-Time, Part-Time, Contractor).**
2. **Department Creation: To instantiate department objects dynamically.**

Implementation

1. Employee Factory

Class Name: `EmployeeFactory`

Responsibilities:

- **Create employee objects based on the type parameter.**

Code:

```
package com.ems.factory;

import com.ems.model.Employee;
import com.ems.model.FullTimeEmployee;
import com.ems.model.PartTimeEmployee;
import com.ems.model.Contractor;
public class EmployeeFactory {
   public static Employee createEmployee(String type, String name, double salary, int
departmentId) {
      switch (type.toLowerCase()) {
         case "full-time":
            return new FullTimeEmployee(name, salary, departmentId);
         case "part-time":
            return new PartTimeEmployee(name, salary, departmentId);
         case "contractor":
            return new Contractor(name, salary, departmentId);
         default:
            throw new IllegalArgumentException("Unknown employee type: " + type);
     }}}
```

How It Works:

- **The `createEmployee` method takes a type parameter and returns the corresponding employee object.**
- **This simplifies object creation in the main application logic.**

2. Department Factory

Class Name: `DepartmentFactory`

Responsibilities:

- **Create department objects based on the name parameter.**

Code:

```
package com.ems.factory;

import com.ems.model.Department;

public class DepartmentFactory {
    public static Department createDepartment(String name) {
        return new Department(name);
    }
}
```

How It Works:

- **The `createDepartment` method dynamically creates a department instance.**

Integration with Database

Database Schema

1. **Employees Table:**

```
CREATE TABLE employees (
   id INT PRIMARY KEY IDENTITY(1,1),
   name NVARCHAR(100) NOT NULL,
   type NVARCHAR(50) NOT NULL,
   salary FLOAT NOT NULL,
   department_id INT NOT NULL,
   FOREIGN KEY (department_id) REFERENCES departments(id)
);
```

2. **Departments Table:**

```
CREATE TABLE departments (
   id INT PRIMARY KEY IDENTITY(1,1),
   name NVARCHAR(100) NOT NULL
);
```

Database Connection

- The `DatabaseConnection` class (Singleton) handles the connection.
- Query execution is handled in service methods.

Integration with GUI

The GUI (`EmployeeManagementGUI`) interacts with the `EmployeeManagementSystem`, which:

- **Uses the Factory pattern for employee and department creation.**
- **Relies on the Singleton database connection to perform SQL operations.**

Example GUI Action:

```
JButton addEmployeeButton = new JButton("Add Employee");
addEmployeeButton.addActionListener(e -> {
    Employee employee = EmployeeFactory.createEmployee("Full-Time", "Alice", 60000,
1);
    EmployeeManagementSystem ems = new EmployeeManagementSystem();
    ems.addEmployee(employee);
});
```

3- Observer Design Pattern

Purpose

The Observer pattern is used to notify dependent objects (observers) when the state of a subject changes. In this project, the observer pattern is applied to log changes whenever an employee is added or removed.

Components

- **Subject: `EmployeeManager`**
  - **Manages the list of employees and notifies observers of any changes.**
- **Observer: `Logger`**
  - **Logs changes to the console.**

Implementation

`EmployeeManager` (Subject)

- **Maintains a list of observers.**
- **Notifies all observers when an employee is added or removed.**

```
public class EmployeeManager {
    private final List<Observer> observers = new ArrayList<>();

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void addEmployee(int id, String name, String department, double salary) {
        // Add employee to database
        notifyObservers("Employee added: " + name);
    }

    public void removeEmployee(int id) {
        // Remove employee from database
        notifyObservers("Employee removed: ID " + id);
    }
```

```
}
```

`Logger` (Observer)

Logs notifications from `EmployeeManager`.

```
public class Logger implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Log: " + message);
    }
}
```

4- Strategy Design Pattern

Purpose

The Strategy pattern is used to define a family of algorithms, encapsulate each one, and make them interchangeable. In this project, it calculates employee benefits based on their department.

Components

- Context: `BenefitCalculator`
    - Encapsulates the strategy object and provides a method to calculate benefits.
- Strategy Interface: `BenefitStrategy`
    - Defines the method for benefit calculation.
- Concrete Strategies: `HRBenefitStrategy`, `ITBenefitStrategy`, `FinanceBenefitStrategy`
    - Implement different benefit calculation rules.

Implementation

`BenefitStrategy` (Interface)

Defines the method for calculating benefits.

```
public interface BenefitStrategy {
    double calculateBenefit(double baseSalary);
}
```

Concrete Strategies

Each strategy implements `BenefitStrategy` for specific departments.

```
public class HRBenefitStrategy implements BenefitStrategy {
    @Override
    public double calculateBenefit(double baseSalary) {
        return baseSalary * 0.15;
    }
}

public class ITBenefitStrategy implements BenefitStrategy {
    @Override
    public double calculateBenefit(double baseSalary) {
        return baseSalary * 0.20;
    }
}

public class FinanceBenefitStrategy implements BenefitStrategy {
    @Override
    public double calculateBenefit(double baseSalary) {
        return baseSalary * 0.25;
    }
}
```

`BenefitCalculator` **(Context)**

Holds a reference to the strategy and calculates benefits using the selected strategy.

```
public class BenefitCalculator {
    private BenefitStrategy strategy;

    public void setStrategy(BenefitStrategy strategy) {
        this.strategy = strategy;
    }

    public double calculate(double baseSalary) {
        if (strategy == null) {
            throw new IllegalStateException("Benefit strategy not set.");
        }
        return strategy.calculateBenefit(baseSalary);
    }
}
```

5- Template Method Design Pattern

Purpose

The Template Method pattern defines the skeleton of an algorithm, deferring some steps to subclasses. In this project, it generates reports for employees based on predefined templates.

Components

- **Abstract Class:** `EmployeeReport`
  - **Defines the template for report generation.**
- **Concrete Classes:** `DepartmentReport`, `SalaryReport`
  - **Implement specific steps of the template.**

Implementation

`EmployeeReport` (Abstract Class)

Defines the template for generating reports.

```
public abstract class EmployeeReport {
    public final void generateReport() {
        fetchData();
        processData();
        printReport();
    }

    protected abstract void fetchData();
    protected abstract void processData();
    protected abstract void printReport();
}
```

Concrete Reports

Implement the steps defined in `EmployeeReport`.

```
public class DepartmentReport extends EmployeeReport {
    @Override
    protected void fetchData() {
        System.out.println("Fetching employee data by department...");
    }

    @Override
    protected void processData() {
        System.out.println("Processing department data...");
    }

    @Override
    protected void printReport() {
        System.out.println("Department Report generated.");
    }
}

public class SalaryReport extends EmployeeReport {
    @Override
    protected void fetchData() {
        System.out.println("Fetching employee salary data...");
    }
```

```
    @Override
    protected void processData() {
        System.out.println("Processing salary data...");
    }

    @Override
    protected void printReport() {
        System.out.println("Salary Report generated.");
    }
}
```

GUI Integration

Key Features

- **Observer Pattern:**
    - **Logs employee additions and removals in the console.**
- **Strategy Pattern:**
    - **Allows users to calculate benefits based on the selected department.**
- **Template Method Pattern:**
    - **Generates department and salary reports with a single button click.**

GUI Components

- **Employee Table: Displays employee details from the database.**
- **Add/Remove Employee: Allows users to manage employees.**
- **Benefit Calculation: Enables calculation of benefits using the strategy pattern.**
- **Report Generation: Implements the template method pattern to generate reports.**

Database Schema

The database uses a SQL Server table to store employee data:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name NVARCHAR(100) NOT NULL,
    Department NVARCHAR(50) NOT NULL,
    BaseSalary DECIMAL(10, 2) NOT NULL
);
```

Conclusion

This Employee Management System demonstrates the practical use of Singleton , Factory Observer, Strategy, and Template Method design patterns in a real-world scenario.

The Singleton and Factory patterns provide:

- **Centralized Management (Singleton): For database and payroll processing.**
- **Dynamic Object Creation (Factory): For employees and departments.**

The system provides a robust and modular approach to managing employees, calculating benefits, and generating reports, with seamless integration of a database and a user-friendly GUI.This modular design ensures scalability, maintainability, and separation of concerns, making the Employee Management System robust and extensible.