Minor Project Report

On

Employee Management System Using Python

Bachelor of Technology

In
Computer Science and Engineering (AIML)

2023-2027

By
Name – Karan Palan
Registration Number – 23FE10CAI00523

Under the supervision of
Sanjay Kumar Tehariya

School of Computer Science and Engineering
Department of Artificial Intelligence and Machine Learning
Manipal University Jaipur, Jaipur, Rajasthan, India

SEP-NOV 2024

Introduction

The **Employee Management System (EMS)** is a Python-based project developed to demonstrate the principles of Object-Oriented Programming (OOP). It simplifies the management of employee records, such as adding, updating, removing employees, and calculating their salaries.

This project is built using OOP concepts like **Abstraction**, **Encapsulation**, **Inheritance**, and **Polymorphism**, showcasing Python's flexibility and power for implementing such systems.

How the Project Works

Objective

- Automate the management of employee records.
- Support multiple employee types with specific attributes (e.g., bonuses for managers, stipends for interns).
- Implement all OOP concepts effectively with practical use cases.

Key Features

- **Encapsulation**: Protects sensitive data using private attributes.
- **Inheritance**: Creates specialized employee types by extending a base class.
- **Polymorphism**: Demonstrates method overriding for dynamic behavior.
- **Abstraction**: Hides implementation details using an abstract base class.

Code Overview

The project consists of three files:

1. **`employee.py`** : Defines the `Employee`, `Manager`, and `Intern` classes, implementing the core functionality.
2. **`employee_management_system.py`** : Contains the `EmployeeManagementSystem` class for managing employee records.
3. **`main.py`** : Demonstrates the system in action and handles user interactions.

File: `employee.py`

Key Code:

```
from abc import ABC, abstractmethod

# ----------------------- Abstraction -----------------------
```

```
class AbstractEmployee(ABC):
    """
    Abstract base class to define the interface for Employee.
    This class uses abstraction to hide implementation details.
    """

    @abstractmethod
    def get_details(self):
        pass

    @abstractmethod
    def calculate_salary(self):
        pass

# ------------------------ Encapsulation ------------------------

class Employee(AbstractEmployee):
    """
    Employee class representing a general employee.
    Demonstrates encapsulation by using private attributes.
    """

    def __init__(self, emp_id, name, age, department, base_salary):
        self.__emp_id = emp_id        # Private attribute
        self.__name = name            # Private attribute
        self.__age = age              # Private attribute
        self.__department = department# Private attribute
        self.__base_salary = base_salary # Private attribute

    # Getter methods to access private attributes
    def get_emp_id(self):
        return self.__emp_id

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def get_department(self):
        return self.__department

    def get_base_salary(self):
        return self.__base_salary

    # Setter methods to modify private attributes with validation
    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            raise ValueError("Age must be positive.")

    def set_department(self, department):
        self.__department = department

    def set_base_salary(self, base_salary):
        if base_salary >= 0:
            self.__base_salary = base_salary
        else:
            raise ValueError("Base salary cannot be negative.")

    def get_details(self):
        """
        Implementation of abstract method to return employee details.
        """
```

```
        return f"ID: {self.__emp_id}, Name: {self.__name}, Age: {self.__age}, " \
               f"Department: {self.__department}, Base Salary: {self.__base_salary}"

    def calculate_salary(self):
        """
        Calculates salary. For a general employee, it's just the base salary.
        """
        return self.__base_salary

# ------------------------ Inheritance and Polymorphism ------------------------

class Manager(Employee):
    """
    Manager class inheriting from Employee.
    Demonstrates inheritance and polymorphism by overriding methods.
    """

    def __init__(self, emp_id, name, age, department, base_salary, bonus):
        super().__init__(emp_id, name, age, department, base_salary)
        self.__bonus = bonus  # Private attribute specific to Manager

    # Getter and Setter for bonus
    def get_bonus(self):
        return self.__bonus

    def set_bonus(self, bonus):
        if bonus >= 0:
            self.__bonus = bonus
        else:
            raise ValueError("Bonus cannot be negative.")

    def get_details(self):
        """
        Overridden method to include bonus information.
        """
        base_details = super().get_details()
        return f"{base_details}, Bonus: {self.__bonus}"

    def calculate_salary(self):
        """
        Overridden method to include bonus in salary calculation.
        """
        return super().calculate_salary() + self.__bonus

class Intern(Employee):
    """
    Intern class inheriting from Employee.
    Demonstrates polymorphism by overriding methods differently.
    """

    def __init__(self, emp_id, name, age, department, base_salary, stipend):
        super().__init__(emp_id, name, age, department, base_salary)
        self.__stipend = stipend  # Private attribute specific to Intern

    # Getter and Setter for stipend
    def get_stipend(self):
        return self.__stipend

    def set_stipend(self, stipend):
        if stipend >= 0:
            self.__stipend = stipend
        else:
            raise ValueError("Stipend cannot be negative.")

    def get_details(self):
```

```
        """
        Overridden method to include stipend information.
        """
        base_details = super().get_details()
        return f"{base_details}, Stipend: {self.__stipend}"

    def calculate_salary(self):
        """
        Overridden method to include stipend instead of bonus.
        """
        return super().calculate_salary() + self.__stipend
```

Explanation:

1. **Abstract Base Class**:
    - Implements `AbstractEmployee` using Python's `abc` module.
    - Forces subclasses to implement `get_details` and `calculate_salary`.
2. **Employee Class**:
    - Encapsulates attributes like employee ID, name, age, and salary.
    - Provides getter and setter methods for controlled access.
3. **Manager and Intern Classes**:
    - Extend the `Employee` class to add specific attributes (e.g., bonus for `Manager`, stipend for `Intern`).
    - Demonstrate polymorphism through method overriding.

File: `employee_management_system.py`

Key Code:

```
from employee import Employee, Manager, Intern

class EmployeeManagementSystem:
    """
    Employee Management System to manage employees.
    Demonstrates usage of classes, objects, inheritance, polymorphism, etc.
    """

    def __init__(self):
        # Dictionary to store employees with emp_id as key
        self.__employees = {}

    def add_employee(self, employee):
        """
        Adds an employee to the system.
        Handles edge case of duplicate emp_id.
        """
        if employee.get_emp_id() in self.__employees:
            raise ValueError(f"Employee with ID {employee.get_emp_id()} already exists.")
        self.__employees[employee.get_emp_id()] = employee
        print(f"Employee {employee.get_name()} added successfully.")

    def remove_employee(self, emp_id):
        """
        Removes an employee from the system.
        Handles edge case of non-existent emp_id.
        """
        if emp_id in self.__employees:
            removed_employee = self.__employees.pop(emp_id)
            print(f"Employee {removed_employee.get_name()} removed successfully.")
        else:
            raise KeyError(f"No employee found with ID {emp_id}.")

    def update_employee(self, emp_id, **kwargs):
```

```
        """
        Updates employee details.
        Handles edge cases like non-existent emp_id and invalid attributes.
        """
        if emp_id not in self.__employees:
            raise KeyError(f"No employee found with ID {emp_id}.")

        employee = self.__employees[emp_id]

        for key, value in kwargs.items():
            if key == 'name':
                employee._Employee__name = value
            elif key == 'age':
                employee.set_age(value)
            elif key == 'department':
                employee.set_department(value)
            elif key == 'base_salary':
                employee.set_base_salary(value)
            elif isinstance(employee, Manager) and key == 'bonus':
                employee.set_bonus(value)
            elif isinstance(employee, Intern) and key == 'stipend':
                employee.set_stipend(value)
            else:
                raise AttributeError(f"Invalid attribute '{key}' for employee type.")

        print(f"Employee {employee.get_name()} updated successfully.")

    def get_employee_details(self, emp_id):
        """
        Retrieves details of a specific employee.
        Handles edge case of non-existent emp_id.
        """
        if emp_id in self.__employees:
            return self.__employees[emp_id].get_details()
        else:
            raise KeyError(f"No employee found with ID {emp_id}.")

    def calculate_total_salary(self):
        """
        Calculates the total salary of all employees.
        """
        total = 0
        for employee in self.__employees.values():
            total += employee.calculate_salary()
        return total

    def list_all_employees(self):
        """
        Lists details of all employees.
        """
        if not self.__employees:
            print("No employees in the system.")
        else:
            for emp_id, employee in self.__employees.items():
                print(employee.get_details())
```

Explanation:

1. **EmployeeManagementSystem Class**:
    - Manages a collection of employees using a dictionary for efficient lookups.
    - Includes methods for adding, updating, and removing employees.
    - Handles edge cases like duplicate IDs and invalid updates.
2. **Dynamic Updates**:
    - Uses Python's `setattr` function to update attributes dynamically based on user input.
3. **Error Handling**:

3. **Error Handling**:
    - Raises appropriate exceptions (`ValueError`, `KeyError`) for invalid operations.

File: `main.py`

Key Code:

```
 from employee import Employee, Manager, Intern
from employee_management_system import EmployeeManagementSystem

def main():
    # Create an instance of EmployeeManagementSystem
    ems = EmployeeManagementSystem()

    try:
        emp1 = Employee(emp_id=1, name="Karan", age=18, department="HR", base_salary=50000)
        ems.add_employee(emp1)

        mgr1 = Manager(emp_id=2, name="Raghav", age=40, department="IT", base_salary=80000, bonus=10000)
        ems.add_employee(mgr1)

        intern1 = Intern(emp_id=3, name="Kavyaa", age=22, department="Marketing", base_salary=30000, stipend=5000)
        ems.add_employee(intern1)

        print("\nList of all employees:")
        ems.list_all_employees()

        ems.update_employee(1, department="Finance", base_salary=55000)

        total_salary = ems.calculate_total_salary()
        print(f"\nTotal Salary of all employees: {total_salary}")

        ems.remove_employee(3)

        print("\nList of all employees after removal:")
        ems.list_all_employees()

    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    main()
```

Explanation:

1. **Demonstrates EMS Functionalities**:
    - Adds employees of different types (General, Manager, Intern).
    - Lists all employees and calculates total salaries.
    - Updates and removes employees while handling edge cases.
2. **User Interaction**:
    - The `main` function showcases all key operations with clear outputs.

Output

Sample Execution:

1. **Adding Employees:**

```
Employee Karan added successfully.
Employee Raghav added successfully.
Employee Kavyaa added successfully.
```

2. **Listing Employees:**

```
List of all employees:
ID: 1, Name: Karan, Age: 18, Department: HR, Base Salary: 50000
ID: 2, Name: Raghav, Age: 40, Department: IT, Base Salary: 80000, Bonus: 10000
ID: 3, Name: Kavyaa, Age: 22, Department: Marketing, Base Salary: 30000, Stipend: 5000
```

3. **Updating Employee:**

```
Employee Karan updated successfully.
```

4. **Calculating Total Salary:**

```
Total Salary of all employees: 180000
```

5. **Removing an Employee:**

```
Employee Kavyaa removed successfully.
```

6. **Listing Employees After Removal:**

```
List of all employees after removal:
ID: 1, Name: Karan, Age: 18, Department: Finance, Base Salary: 55000
ID: 2, Name: Raghav, Age: 40, Department: IT, Base Salary: 80000, Bonus: 10000
```

Conclusion

The **Employee Management System** is a comprehensive application that demonstrates the effective use of OOP principles in Python. By utilizing abstraction, encapsulation, inheritance, and polymorphism, the project achieves modularity, scalability, and security.

Future improvements could include:

- Integration with a database for persistent storage.
- A graphical user interface for enhanced usability.
- Advanced analytics for employee performance evaluation.

This project serves as a robust foundation for building real-world employee management solutions.