CASE STUDY

PAYEXPERT

LOGESH. D

Classes:

• Employee:

• Properties: EmployeeID, FirstName, LastName, DateOfBirth, Gender, Email, PhoneNumber, Address, Position, JoiningDate, TerminationDate

• Methods: CalculateAge()

```
from datetime import date

class Employee:
    def __init__(self, employee_id=None, first_name=None,
last_name=None, date_of_birth=None, gender=None,
                 email=None, phone_number=None, address=None,
position=None, joining_date=None, termination_date=None):
        self.employee_id = employee_id
        self.first_name = first_name
        self.last_name = last_name
        self.date_of_birth = date_of_birth
        self.gender = gender
        self.email = email
        self.phone_number = phone_number
        self.address = address
        self.position = position
        self.joining_date = joining_date
        self.termination_date = termination_date

    def calculate_age(self):
        today = date.today()
        age = today.year - self.date_of_birth.year
        if today.month < self.date_of_birth.month or (today.month
== self.date_of_birth.month and today.day <
self.date_of_birth.day):
            age -= 1
        return age
```

• Payroll:

  • Properties: PayrollID, EmployeeID, PayPeriodStartDate, PayPeriodEndDate, BasicSalary, OvertimePay, Deductions, NetSalary

```
class Payroll:
    def __init__(self, payroll_id=None, employee_id=None,
pay_period_start_date=None, pay_period_end_date=None,
                 basic_salary=None, overtime_pay=None,
deductions=None, net_salary=None):
        self.payroll_id = payroll_id
        self.employee_id = employee_id
        self.pay_period_start_date = pay_period_start_date
        self.pay_period_end_date = pay_period_end_date
        self.basic_salary = basic_salary
        self.overtime_pay = overtime_pay
        self.deductions = deductions
        self.net_salary = net_salary
```

• Tax:

  • Properties: TaxID, EmployeeID, TaxYear, TaxableIncome, TaxAmount

```
class Tax:
    def __init__(self, tax_id=None, employee_id=None,
tax_year=None, taxable_income=None, tax_amount=None):
        self.tax_id = tax_id
        self.employee_id = employee_id
        self.tax_year = tax_year
        self.taxable_income = taxable_income
        self.tax_amount = tax_amount
```

• FinancialRecord:

  • Properties: RecordID, EmployeeID, RecordDate, Description, Amount, RecordType

```
class FinancialRecord:
    def __init__(self, record_id=None, employee_id=None,
record_date=None, description=None, amount=None,
record_type=None):
        self.record_id = record_id
        self.employee_id = employee_id
        self.record_date = record_date
        self.description = description
        self.amount = amount
        self.record_type = record_type
```

EmployeeService (implements IEmployeeService):

• Methods:

  • GetEmployeeById

  • GetAllEmployees

  • AddEmployee

  • UpdateEmployee

  • RemoveEmployee

```
def employee_management(employee_service):
    while True:
        print("\nEmployee Management")
        print("1. Add Employee")
        print("2. Update Employee")
        print("3. Remove Employee")
        print("4. View Employee Details")
        print("5. Back to Main Menu")

        choice = input("Enter your choice: ")

        if choice == "1":
            add_employee(employee_service)
        elif choice == "2":
            update_employee(employee_service)
        elif choice == "3":
            remove_employee(employee_service)
        elif choice == "4":
            view_employee_details(employee_service)
        elif choice == "5":
            break
        else:
            print("Invalid choice. Please try again.")


def add_employee(employee_service):
    first_name = input("Enter first name: ")
    last_name = input("Enter last name: ")
    date_of_birth = input("Enter date of birth (YYYY-MM-DD): ")
    gender = input("Enter gender (M/F): ")
    email = input("Enter email: ")
    phone_number = input("Enter phone number: ")
    address = input("Enter address: ")
    position = input("Enter position: ")
    joining_date = input("Enter joining date (YYYY-MM-DD): ")

    employee = Employee(
        first_name=first_name,
        last_name=last_name,
        date_of_birth=date.fromisoformat(date_of_birth),
        gender=gender,
        email=email,
        phone_number=phone_number,
        address=address,
        position=position,
        joining_date=date.fromisoformat(joining_date)
    )

    employee_service.add_employee(employee)
    print("Employee added successfully.")


def update_employee(employee_service):
    employee_id = int(input("Enter employee ID: "))

    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    first_name = input(f"Enter first name
({employee.first_name}): ") or employee.first_name
    last_name = input(f"Enter last name ({employee.last_name}):
") or employee.last_name
    date_of_birth = input(f"Enter date of birth
({employee.date_of_birth.isoformat()}): ") or
employee.date_of_birth
    gender = input(f"Enter gender ({employee.gender}): ") or
employee.gender
    email = input(f"Enter email ({employee.email}): ") or
employee.email
    phone_number = input(f"Enter phone number
({employee.phone_number}): ") or employee.phone_number
    address = input(f"Enter address ({employee.address}): ") or
employee.address
    position = input(f"Enter position ({employee.position}): ")
or employee.position
    joining_date = input(f"Enter joining date
({employee.joining_date.isoformat()}): ") or
employee.joining_date
    termination_date = input(
        f"Enter termination date
({employee.termination_date.isoformat() if
employee.termination_date else None}): ") or
employee.termination_date

    updated_employee = Employee(
        employee_id=employee_id,
```

```python
        first_name=first_name,
        last_name=last_name,
        date_of_birth=date.fromisoformat(date_of_birth) if
isinstance(date_of_birth, str) else date_of_birth,
        gender=gender,
        email=email,
        phone_number=phone_number,
        address=address,
        position=position,
        joining_date=date.fromisoformat(joining_date) if
isinstance(joining_date, str) else joining_date,
        termination_date=date.fromisoformat(termination_date) if
isinstance(termination_date, str) else termination_date
    )

    employee_service.update_employee(updated_employee)
    print("Employee updated successfully.")


def remove_employee(employee_service):
    employee_id = int(input("Enter employee ID: "))
    employee_service.remove_employee(employee_id)
    print("Employee removed successfully.")


def view_employee_details(employee_service):
    employee_id = int(input("Enter employee ID: "))

    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    print(f"\nEmployee Details:")
    print(f"Employee ID: {employee.employee_id}")
    print(f"First Name: {employee.first_name}")
    print(f"Last Name: {employee.last_name}")
    print(f"Date of Birth: {employee.date_of_birth.isoformat()}")
    print(f"Gender: {employee.gender}")
    print(f"Email: {employee.email}")
    print(f"Phone Number: {employee.phone_number}")
    print(f"Address: {employee.address}")
    print(f"Position: {employee.position}")
    print(f"Joining Date: {employee.joining_date.isoformat()}")
    print(f"Termination Date:
{employee.termination_date.isoformat() if
employee.termination_date else 'N/A'}")
```

PayrollService (implements IPayrollService):

- Methods:
  - GeneratePayroll
  - GetPayrollById
  - GetPayrollsForEmployee
  - GetPayrollsForPeriod

```python
def payroll_processing(employee_service, payroll_service):
    while True:
        print("\nPayroll Processing")
        print("1. Generate Payroll")
        print("2. View Payroll Details")
        print("3. Back to Main Menu")

        choice = input("Enter your choice: ")

        if choice == "1":
            generate_payroll(employee_service, payroll_service)
        elif choice == "2":
            view_payroll_details(employee_service,
payroll_service)
        elif choice == "3":
            break
        else:
            print("Invalid choice. Please try again.")


def generate_payroll(employee_service, payroll_service):
    employee_id = int(input("Enter employee ID: "))
    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    start_date = input("Enter pay period start date (YYYY-MM-DD):
")
    end_date = input("Enter pay period end date (YYYY-MM-DD): ")

    try:
        payroll_service.generate_payroll(employee_id,
date.fromisoformat(start_date), date.fromisoformat(end_date))
    except PayrollGenerationException as e:
        print(e)
        return

    print("Payroll generated successfully.")
```

```
def view_payroll_details(employee_service, payroll_service):
    employee_id = int(input("Enter employee ID: "))
    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    payrolls =
payroll_service.get_payrolls_for_employee(employee_id)

    if not payrolls:
        print("No payroll records found for this employee.")
        return

    print(f"\nPayroll Records for Employee {employee.first_name}
{employee.last_name}:")
    for payroll in payrolls:
        print(f"\nPayroll ID: {payroll.payroll_id}")
        print(f"Pay Period:
{payroll.pay_period_start_date.isoformat()} -
{payroll.pay_period_end_date.isoformat()}")
        print(f"Basic Salary: {payroll.basic_salary}")
        print(f"Overtime Pay: {payroll.overtime_pay}")
        print(f"Deductions: {payroll.deductions}")
        print(f"Net Salary: {payroll.net_salary}")
```

TaxService (implements ITaxService):

• Methods:

  • CalculateTax

  • GetTaxById

  • GetTaxesForEmployee

```
def tax_calculation(employee_service, tax_service):
    while True:
        print("\nTax Calculation")
        print("1. Calculate Tax")
        print("2. View Tax Details")
        print("3. Back to Main Menu")
        choice = input("Enter your choice: ")

        if choice == "1":
            calculate_tax(employee_service, tax_service)
        elif choice == "2":
            view_tax_details(employee_service, tax_service)
        elif choice == "3":
            break
        else:
            print("Invalid choice. Please try again.")


def calculate_tax(employee_service, tax_service):
    employee_id = int(input("Enter employee ID: "))
    tax_year = int(input("Enter tax year: "))
    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    try:
        tax_service.calculate_tax(employee_id, tax_year)
    except TaxCalculationException as e:
        print(e)
        return

    print("Tax calculated successfully.")


def view_tax_details(employee_service, tax_service):
    employee_id = int(input("Enter employee ID: "))
    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    taxes = tax_service.get_taxes_for_employee(employee_id)

    if not taxes:
        print("No tax records found for this employee.")
        return

    print(f"\nTax Records for Employee {employee.first_name}
{employee.last_name}:")
    for tax in taxes:
        print(f"\nTax ID: {tax.tax_id}")
        print(f"Tax Year: {tax.tax_year}")
        print(f"Taxable Income: {tax.taxable_income}")
        print(f"Tax Amount: {tax.tax_amount}")
```

FinancialRecordService (implements IFinancialRecordService):

• Methods:

  • AddFinancialRecord

  • GetFinancialRecordById

  • GetFinancialRecordsForEmployee

```
def financial_record_management(employee_service,
financial_record_service):
    while True:
        print("\nFinancial Record Management")
        print("1. Add Financial Record")
        print("2. View Financial Records")
        print("3. Back to Main Menu")
        choice = input("Enter your choice: ")

        if choice == "1":
            add_financial_record(employee_service,
financial_record_service)
        elif choice == "2":
            view_financial_records(employee_service,
financial_record_service)
        elif choice == "3":
            break
        else:
            print("Invalid choice. Please try again.")


def add_financial_record(employee_service,
financial_record_service):
    employee_id = int(input("Enter employee ID: "))
    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    description = input("Enter description: ")
    amount = float(input("Enter amount: "))
    record_type = input("Enter record type (income/expense): ")

    try:

financial_record_service.add_financial_record(employee_id,
description, amount, record_type)
    except FinancialRecordException as e:
        print(e)
        return

    print("Financial record added successfully.")


def view_financial_records(employee_service,
financial_record_service):
    employee_id = int(input("Enter employee ID: "))
    try:
        employee =
employee_service.get_employee_by_id(employee_id)
    except EmployeeNotFoundException as e:
        print(e)
        return

    financial_records =
financial_record_service.get_financial_records_for_employee(emplo
yee_id)

    if not financial_records:
        print("No financial records found for this employee.")
        return

    print(f"\nFinancial Records for Employee
{employee.first_name} {employee.last_name}:")
    for record in financial_records:
        print(f"\nRecord ID: {record.record_id}")
        print(f"Record Date: {record.record_date.isoformat()}")
        print(f"Description: {record.description}")
        print(f"Amount: {record.amount}")
        print(f"Record Type: {record.record_type}")


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

DatabaseContext:

• A class responsible for handling database connections and interactions.

```
import mysql.connector
from exception.database_connection import
DatabaseConnectionException

def get_connection():
    try:
        return mysql.connector.connect(
            host="127.0.0.1",
            user="root",
            password="root",
            database="casestudy"
        )
    except mysql.connector.Error as e:
        raise DatabaseConnectionException(f"Error connecting to
the database: {e}")
```

Interfaces/Abstract class:

- IEmployeeService:
  - GetEmployeeById(employeeId)
  - GetAllEmployees()
  - AddEmployee(employeeData)
  - UpdateEmployee(employeeData)
  - RemoveEmployee(employeeId)

```
from abc import ABC, abstractmethod
from entity.EMPLOYEE import Employee
from exception.employee_not_found_exc import
EmployeeNotFoundException

class IEmployeeService(ABC):
    @abstractmethod
    def get_employee_by_id(self, employee_id):
        pass

    @abstractmethod
    def get_all_employees(self):
        pass

    @abstractmethod
    def add_employee(self, employee_data):
        pass

    @abstractmethod
    def update_employee(self, employee_data):
        pass

    @abstractmethod
    def remove_employee(self, employee_id):
        pass
```

- IPayrollService:
  - GeneratePayroll(employeeId, startDate, endDate)
  - GetPayrollById(payrollId)
  - GetPayrollsForEmployee(employeeId)
  - GetPayrollsForPeriod(startDate, endDate)

```
from abc import ABC, abstractmethod
from entity.PAYROLL import Payroll
from exception.payroll_gen_exc import PayrollGenerationException
from exception.employee_not_found_exc import
EmployeeNotFoundException

class IPayrollService(ABC):
    @abstractmethod
    def generate_payroll(self, employee_id, start_date,
end_date):
        pass

    @abstractmethod
    def get_payroll_by_id(self, payroll_id):
        pass

    @abstractmethod
    def get_payrolls_for_employee(self, employee_id):
        pass

    @abstractmethod
    def get_payrolls_for_period(self, start_date, end_date):
        pass
```

- ITaxService:
  - CalculateTax(employeeId, taxYear)
  - GetTaxById(taxId)
  - GetTaxesForEmployee(employeeId)
  - GetTaxesForYear(taxYear)

```
from abc import ABC, abstractmethod
from entity.TAX import Tax
from exception.tax_calculation_exc import TaxCalculationException
from exception.employee_not_found_exc import
EmployeeNotFoundException
from decimal import Decimal
```

```python
class ITaxService(ABC):
    @abstractmethod
    def calculate_tax(self, employee_id, tax_year):
        pass

    @abstractmethod
    def get_tax_by_id(self, tax_id):
        pass

    @abstractmethod
    def get_taxes_for_employee(self, employee_id):
        pass

    @abstractmethod
    def get_taxes_for_year(self, tax_year):
        pass
```

- IFinancialRecordService:
  - AddFinancialRecord(employeeId, description, amount, recordType)
  - GetFinancialRecordById(recordId)
  - GetFinancialRecordsForEmployee(employeeId)
  - GetFinancialRecordsForDate(recordDate)

```python
from abc import ABC, abstractmethod
from entity.FINANCIALRECORD import FinancialRecord
from exception.financial_record_exc import
FinancialRecordException
from exception.employee_not_found_exc import
EmployeeNotFoundException

class IFinancialRecordService(ABC):
    @abstractmethod
    def add_financial_record(self, employee_id, description,
amount, record_type):
        pass

    @abstractmethod
    def get_financial_record_by_id(self, record_id):
        pass

    @abstractmethod
    def get_financial_records_for_employee(self, employee_id):
        pass

    @abstractmethod
    def get_financial_records_for_date(self, record_date):
        pass
```

Custom Exceptions:

EmployeeNotFoundException:

• Thrown when attempting to access or perform operations on a non-existing employee.

```
class EmployeeNotFoundException(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message
```

PayrollGenerationException:

• Thrown when there is an issue with generating payroll for an employee.

```
class PayrollGenerationException(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message
```

TaxCalculationException:

• Thrown when there is an error in calculating taxes for an employee.

```
class TaxCalculationException(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message
```

FinancialRecordException:

• Thrown when there is an issue with financial record management.

```
class FinancialRecordException(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message
```

InvalidInputException:

• Thrown when input data doesn't meet the required criteria.

```
class InvalidInputException(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message
```

DatabaseConnectionException:

• Thrown when there is a problem establishing or maintaining a connection with the database.

```
class DatabaseConnectionException(Exception):
    def __init__(self, message):
        self.message = message

    def __str__(self):
        return self.message
```

Test Case: CalculateNetSalaryAfterDeductions

• Objective: Ensure that the system accurately calculates the net salary after deductions (taxes, insurance, etc.).

Test Case: VerifyTaxCalculationForHighIncomeEmployee

• Objective: Test the system's ability to calculate taxes for a high-income employee.

Test Case: ProcessPayrollForMultipleEmployees

• Objective: Test the end-to-end payroll processing for a batch of employees.

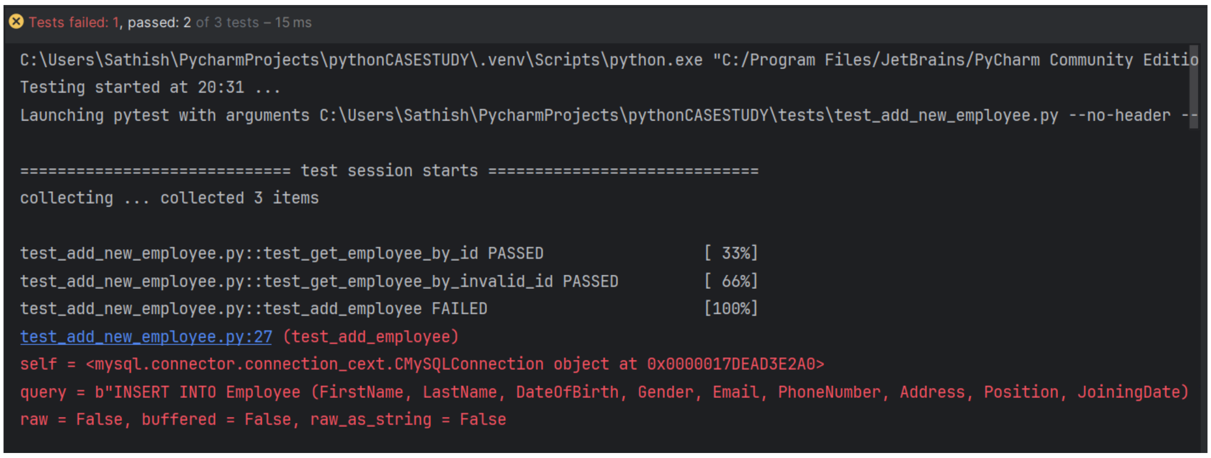
```
import pytest
from datetime import date
from dao.PAYROLL_SERVICE import PayrollService
from exception.employee_not_found_exc import
EmployeeNotFoundException
from exception.payroll_gen_exc import PayrollGenerationException
from util.dbconn import get_connection

@pytest.fixture
def payroll_service():
    db_connection = get_connection()
    return PayrollService(db_connection)

def test_generate_payroll(payroll_service):
    employee_id = 1
    start_date = date(2023, 1, 1)
    end_date = date(2023, 1, 31)

    payroll_service.generate_payroll(employee_id, start_date,
end_date)

    payrolls =
payroll_service.get_payrolls_for_employee(employee_id)
    assert len(payrolls) > 0
    latest_payroll = payrolls[-1]
    assert latest_payroll.employee_id == employee_id
    assert latest_payroll.pay_period_start_date == start_date
    assert latest_payroll.pay_period_end_date == end_date
    assert latest_payroll.basic_salary > 0
    assert latest_payroll.overtime_pay >= 0
    assert latest_payroll.deductions >= 0
    assert latest_payroll.net_salary > 0

def test_generate_payroll_for_invalid_employee(payroll_service):
    invalid_employee_id = 999
    start_date = date(2023, 1, 1)
    end_date = date(2023, 1, 31)

    with pytest.raises(EmployeeNotFoundException):
        payroll_service.generate_payroll(invalid_employee_id,
start_date, end_date)
```

Test Case: VerifyErrorHandlingForInvalidEmployeeData

• Objective: Ensure the system handles invalid input data gracefully.

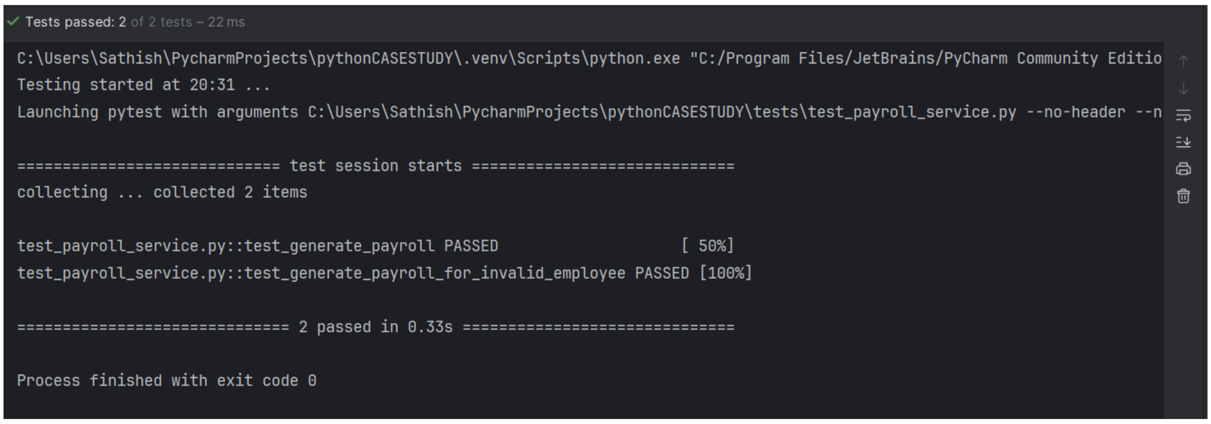
```python
import pytest
from datetime import date
from dao.EMPLOYEE_SERVICE import EmployeeService
from entity.EMPLOYEE import Employee
from exception.employee_not_found_exc import
EmployeeNotFoundException
from util.dbconn import get_connection
@pytest.fixture
def employee_service():
    db_connection = get_connection()
    return EmployeeService(db_connection)

def test_get_employee_by_id(employee_service):
    employee_id = 1

    employee = employee_service.get_employee_by_id(employee_id)

    assert employee.employee_id == employee_id
    assert employee.first_name == "LOGESH"
    assert employee.last_name == "DHAMODARAN"
    assert employee.date_of_birth == date(2002, 10, 22)

def test_get_employee_by_invalid_id(employee_service):
    invalid_employee_id = 999

    with pytest.raises(EmployeeNotFoundException):
        employee_service.get_employee_by_id(invalid_employee_id)

def test_add_employee(employee_service):

    new_employee = Employee(
        first_name="Jane",
        last_name="Smith",
        date_of_birth=date(1985, 9, 23),
        gender="M",
        email="jane.smith@example.com",
        phone_number="1234567890",
        address="123 Main St",
        position="Manager",
        joining_date=date(2022, 1, 1)
    )

    employee_service.add_employee(new_employee)

    added_employee = employee_service.get_employee_by_id(9)
    assert added_employee is not None
```

OUTPUT:



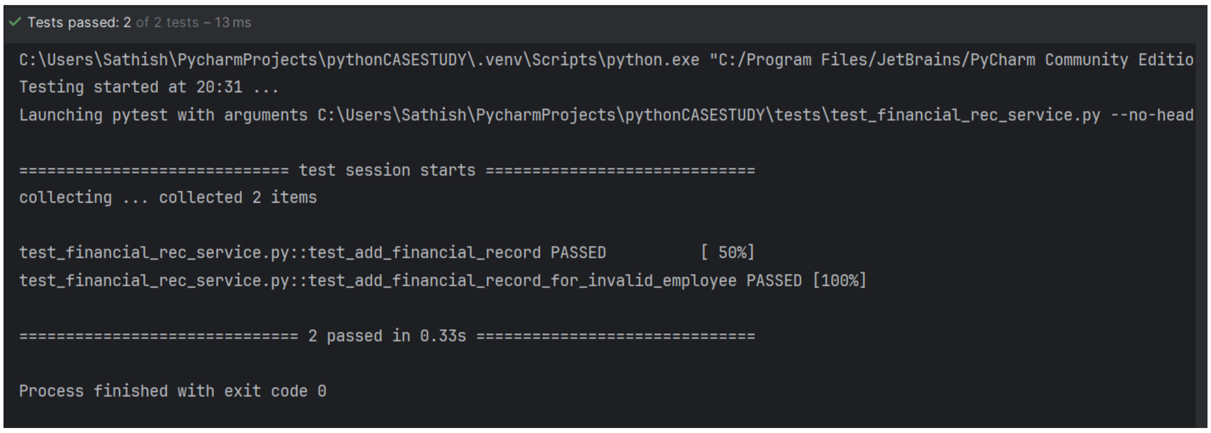
Tests failed: 1, passed: 2 of 3 tests – 15 ms

```
C:\Users\Sathish\PycharmProjects\pythonCASESTUDY\.venv\Scripts\python.exe "C:/Program Files/JetBrains/PyCharm Community Editio
Testing started at 20:31 ...
Launching pytest with arguments C:\Users\Sathish\PycharmProjects\pythonCASESTUDY\tests\test_add_new_employee.py --no-header --

============================ test session starts ============================
collecting ... collected 3 items

test_add_new_employee.py::test_get_employee_by_id PASSED                 [ 33%]
test_add_new_employee.py::test_get_employee_by_invalid_id PASSED         [ 66%]
test_add_new_employee.py::test_add_employee FAILED                       [100%]
test_add_new_employee.py:27 (test_add_employee)
self = <mysql.connector.connection_cext.CMySQLConnection object at 0x0000017DEAD3E2A0>
query = b"INSERT INTO Employee (FirstName, LastName, DateOfBirth, Gender, Email, PhoneNumber, Address, Position, JoiningDate)
raw = False, buffered = False, raw_as_string = False
```



Tests passed: 2 of 2 tests – 22 ms

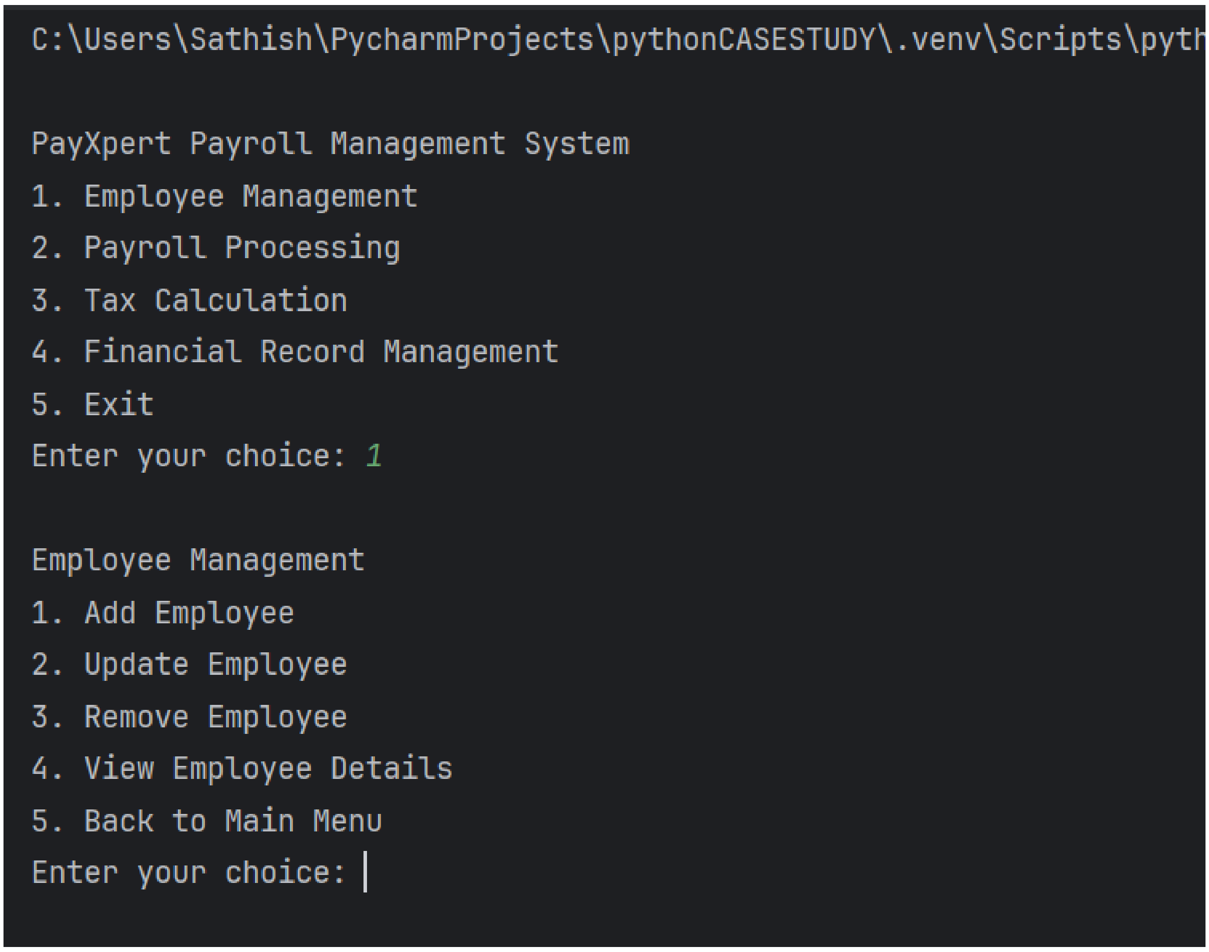
```
C:\Users\Sathish\PycharmProjects\pythonCASESTUDY\.venv\Scripts\python.exe "C:/Program Files/JetBrains/PyCharm Community Editio
Testing started at 20:31 ...
Launching pytest with arguments C:\Users\Sathish\PycharmProjects\pythonCASESTUDY\tests\test_payroll_service.py --no-header --n

============================ test session starts ============================
collecting ... collected 2 items

test_payroll_service.py::test_generate_payroll PASSED                    [ 50%]
test_payroll_service.py::test_generate_payroll_for_invalid_employee PASSED [100%]

============================== 2 passed in 0.33s ==============================

Process finished with exit code 0
```



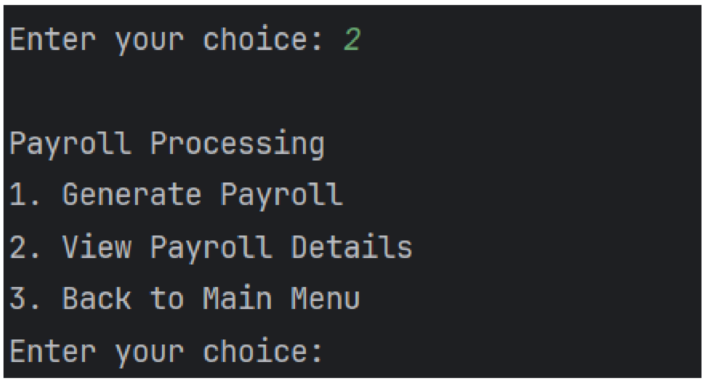
Tests passed: 2 of 2 tests – 13 ms

```
C:\Users\Sathish\PycharmProjects\pythonCASESTUDY\.venv\Scripts\python.exe "C:/Program Files/JetBrains/PyCharm Community Editio
Testing started at 20:31 ...
Launching pytest with arguments C:\Users\Sathish\PycharmProjects\pythonCASESTUDY\tests\test_financial_rec_service.py --no-head

============================ test session starts ============================
collecting ... collected 2 items

test_financial_rec_service.py::test_add_financial_record PASSED          [ 50%]
test_financial_rec_service.py::test_add_financial_record_for_invalid_employee PASSED [100%]

============================== 2 passed in 0.33s ==============================

Process finished with exit code 0
```

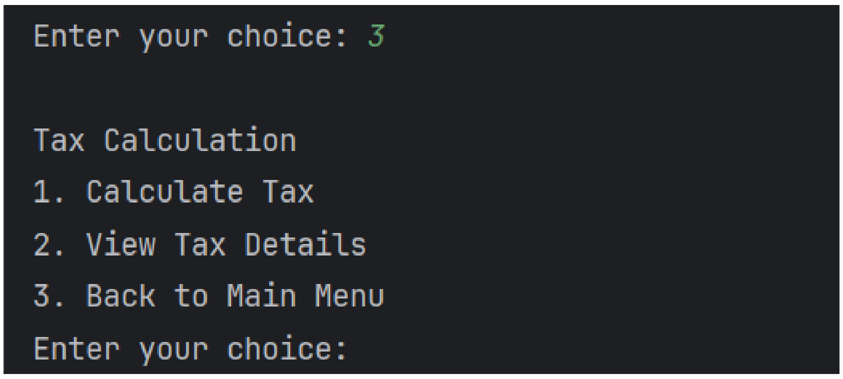
OUTPUT:



```
C:\Users\Sathish\PycharmProjects\pythonCASESTUDY\.venv\Scripts\pyth

PayXpert Payroll Management System
1. Employee Management
2. Payroll Processing
3. Tax Calculation
4. Financial Record Management
5. Exit
Enter your choice: 1

Employee Management
1. Add Employee
2. Update Employee
3. Remove Employee
4. View Employee Details
5. Back to Main Menu
Enter your choice: 
```



```
Enter your choice: 2

Payroll Processing
1. Generate Payroll
2. View Payroll Details
3. Back to Main Menu
Enter your choice:
```



```
Enter your choice: 3

Tax Calculation
1. Calculate Tax
2. View Tax Details
3. Back to Main Menu
Enter your choice:
```