PayXpert

Classes:

• Employee:

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

• Methods: CalculateAge()

• Payroll:

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

• Tax:

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

• FinancialRecord:

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

```
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


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


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


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
```

EmployeeService (implements IEmployeeService):

- Methods:
  - GetEmployeeById
  - GetAllEmployees
  - AddEmployee

- UpdateEmployee
- RemoveEmployee

```
from abc import ABC, abstractmethod
from entity.emp import Employee
from exception_handling.exception 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

class EmployeeService(IEmployeeService):
   def __init__(self, db_connection):
       self.db_connection = db_connection

   def get_employee_by_id(self, employee_id):
       cursor = self.db_connection.cursor()
       query = "SELECT * FROM Employee WHERE EmployeeID = %s"
       cursor.execute(query, (employee_id,))
       result = cursor.fetchone()
       cursor.close()

       if result:
           employee = Employee(
               employee_id=result[0],
               first_name=result[1],
               last_name=result[2],
               date_of_birth=result[3],
               gender=result[4],
               email=result[5],
               phone_number=result[6],
```

```
                address=result[7],
                position=result[8],
                joining_date=result[9],
                termination_date=result[10]
            )
            return employee
        else:
            raise EmployeeNotFoundException(f"Employee with ID {employee_id} not found.")

    def get_all_employees(self):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM Employee"
        cursor.execute(query)
        results = cursor.fetchall()
        cursor.close()

        employees = []
        for result in results:
            employee = Employee(
                employee_id=result[0],
                first_name=result[1],
                last_name=result[2],
                date_of_birth=result[3],
                gender=result[4],
                email=result[5],
                phone_number=result[6],
                address=result[7],
                position=result[8],
                joining_date=result[9],
                termination_date=result[10]
            )
            employees.append(employee)

        return employees

    def add_employee(self, employee_data):
        cursor = self.db_connection.cursor()
            query = "INSERT INTO Employee (FirstName, LastName, DateOfBirth, Gender, Email, PhoneNumber, Address, Position, JoiningDate) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)"
        values = (
            employee_data.first_name,
            employee_data.last_name,
            employee_data.date_of_birth,
            employee_data.gender,
            employee_data.email,
            employee_data.phone_number,
            employee_data.address,
```

```
            employee_data.position,
            employee_data.joining_date
        )
        cursor.execute(query, values)
        self.db_connection.commit()
        cursor.close()

    def update_employee(self, employee_data):
        cursor = self.db_connection.cursor()
        query = "UPDATE Employee SET FirstName = %s, LastName = %s, DateOfBirth
= %s, Gender = %s, Email = %s, PhoneNumber = %s, Address = %s, Position = %s,
JoiningDate = %s, TerminationDate = %s WHERE EmployeeID = %s"
        values = (
            employee_data.first_name,
            employee_data.last_name,
            employee_data.date_of_birth,
            employee_data.gender,
            employee_data.email,
            employee_data.phone_number,
            employee_data.address,
            employee_data.position,
            employee_data.joining_date,
            employee_data.termination_date,
            employee_data.employee_id
        )
        cursor.execute(query, values)
        self.db_connection.commit()
        cursor.close()

    def remove_employee(self, employee_id):
        cursor = self.db_connection.cursor()
        query = "DELETE FROM Employee WHERE EmployeeID = %s"
        cursor.execute(query, (employee_id,))
        self.db_connection.commit()
        cursor.close()
```

PayrollService (implements IPayrollService):

- Methods:
  - GeneratePayroll
  - GetPayrollById
  - GetPayrollsForEmployee
  - GetPayrollsForPeriod

```
from abc import ABC, abstractmethod
from entity.payroll import Payroll
```

```
from exception_handling.exception import PayrollGenerationException
from exception_handling.exception 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

class PayrollService(IPayrollService):
   def __init__(self, db_connection):
       self.db_connection = db_connection

   def generate_payroll(self, employee_id, start_date, end_date):
       cursor = self.db_connection.cursor()

       query = "SELECT * FROM Employee WHERE EmployeeID = %s"
       cursor.execute(query, (employee_id,))
       employee = cursor.fetchone()

       if not employee:
           raise EmployeeNotFoundException(f"Employee with ID {employee_id} not
found.")

       basic_salary = 50000.0
       overtime_pay = 900.0
       deductions = 1000.0
       net_salary = basic_salary + overtime_pay - deductions

            query = "INSERT INTO Payroll (EmployeeID, PayPeriodStartDate,
PayPeriodEndDate, BasicSalary, OvertimePay, Deductions, NetSalary) VALUES (%s,
%s, %s, %s, %s, %s, %s)"
        values = (employee_id, start_date, end_date, basic_salary, overtime_pay,
deductions, net_salary)
       cursor.execute(query, values)
       self.db_connection.commit()
       cursor.close()
```

```
    def get_payroll_by_id(self, payroll_id):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM Payroll WHERE PayrollID = %s"
        cursor.execute(query, (payroll_id,))
        result = cursor.fetchone()
        cursor.close()

        if result:
            payroll = Payroll(
                payroll_id=result[0],
                employee_id=result[1],
                pay_period_start_date=result[2],
                pay_period_end_date=result[3],
                basic_salary=result[4],
                overtime_pay=result[5],
                deductions=result[6],
                net_salary=result[7]
            )
            return payroll
        else:
            return None

    def get_payrolls_for_employee(self, employee_id):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM Payroll WHERE EmployeeID = %s"
        cursor.execute(query, (employee_id,))
        results = cursor.fetchall()
        cursor.close()

        payrolls = []
        for result in results:
            payroll = Payroll(
                payroll_id=result[0],
                employee_id=result[1],
                pay_period_start_date=result[2],
                pay_period_end_date=result[3],
                basic_salary=result[4],
                overtime_pay=result[5],
                deductions=result[6],
                net_salary=result[7]
            )
            payrolls.append(payroll)

        return payrolls

    def get_payrolls_for_period(self, start_date, end_date):
        cursor = self.db_connection.cursor()
          query = "SELECT * FROM Payroll WHERE PayPeriodStartDate >= %s AND
PayPeriodEndDate <= %s"
```

```
        cursor.execute(query, (start_date, end_date))
        results = cursor.fetchall()
        cursor.close()

        payrolls = []
        for result in results:
            payroll = Payroll(
                payroll_id=result[0],
                employee_id=result[1],
                pay_period_start_date=result[2],
                pay_period_end_date=result[3],
                basic_salary=result[4],
                overtime_pay=result[5],
                deductions=result[6],
                net_salary=result[7]
            )
            payrolls.append(payroll)

        return payrolls
```

TaxService (implements ITaxService):

- Methods:
  - CalculateTax
  - GetTaxById
  - GetTaxesForEmployee

```
from abc import ABC, abstractmethod
from entity.tax import Tax
from exception_handling.exception import TaxCalculationException
from exception_handling.exception import EmployeeNotFoundException
from decimal import Decimal


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

class TaxService(ITaxService):
    def __init__(self, db_connection):
        self.db_connection = db_connection

    def calculate_tax(self, employee_id, tax_year):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM Employee WHERE EmployeeID = %s"
        cursor.execute(query, (employee_id,))
        employee = cursor.fetchone()

        if not employee:
            raise EmployeeNotFoundException(f"Employee with ID {employee_id} not
found.")

            query = "SELECT SUM(NetSalary) AS TaxableIncome FROM Payroll WHERE
EmployeeID = %s AND YEAR(PayPeriodEndDate) = %s"
        cursor.execute(query, (employee_id, tax_year))
        result = cursor.fetchone()

        if not result or result[0] is None:
                raise TaxCalculationException(f"No payroll records found for
employee {employee_id} in the year {tax_year}.")

        taxable_income = result[0]

        tax_amount = taxable_income * Decimal('0.2')

            query = "INSERT INTO Tax (EmployeeID, TaxYear, TaxableIncome,
TaxAmount) VALUES (%s, %s, %s, %s)"
        values = (employee_id, tax_year, taxable_income, tax_amount)
        cursor.execute(query, values)
        self.db_connection.commit()
        cursor.close()

    def get_tax_by_id(self, tax_id):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM Tax WHERE TaxID = %s"
        cursor.execute(query, (tax_id,))
        result = cursor.fetchone()
        cursor.close()

        if result:
            tax = Tax(
                tax_id=result[0],
                employee_id=result[1],
                tax_year=result[2],
                taxable_income=result[3],
```

```
            tax_amount=result[4]
        )
        return tax
    else:
        return None

def get_taxes_for_employee(self, employee_id):
    cursor = self.db_connection.cursor()
    query = "SELECT * FROM Tax WHERE EmployeeID = %s"
    cursor.execute(query, (employee_id,))
    results = cursor.fetchall()
    cursor.close()

    taxes = []
    for result in results:
        tax = Tax(
            tax_id=result[0],
            employee_id=result[1],
            tax_year=result[2],
            taxable_income=result[3],
            tax_amount=result[4]
        )
        taxes.append(tax)

    return taxes

def get_taxes_for_year(self, tax_year):
    cursor = self.db_connection.cursor()
    query = "SELECT * FROM Tax WHERE TaxYear = %s"
    cursor.execute(query, (tax_year,))
    results = cursor.fetchall()
    cursor.close()

    taxes = []
    for result in results:
        tax = Tax(
            tax_id=result[0],
            employee_id=result[1],
            tax_year=result[2],
            taxable_income=result[3],
            tax_amount=result[4]
        )
        taxes.append(tax)

    return taxes
```

FinancialRecordService (implements IFinancialRecordService):

• Methods:

- AddFinancialRecord
- GetFinancialRecordById
- GetFinancialRecordsForEmployee

```
from abc import ABC, abstractmethod
from entity.fin import FinancialRecord
from exception_handling.exception import FinancialRecordException
from exception_handling.exception 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

class FinancialRecordService(IFinancialRecordService):
   def __init__(self, db_connection):
       self.db_connection = db_connection

         def   add_financial_record(self,   employee_id,   description,   amount,
record_type):
       cursor = self.db_connection.cursor()

       query = "SELECT * FROM Employee WHERE EmployeeID = %s"
       cursor.execute(query, (employee_id,))
       employee = cursor.fetchone()

       if not employee:
           raise EmployeeNotFoundException(f"Employee with ID {employee_id} not
found.")

             query   =   "INSERT   INTO   FinancialRecord   (EmployeeID,   RecordDate,
Description, Amount, RecordType) VALUES (%s, CURDATE(), %s, %s, %s)"
       values = (employee_id, description, amount, record_type)
       cursor.execute(query, values)
```

```
        self.db_connection.commit()
        cursor.close()

    def get_financial_record_by_id(self, record_id):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM FinancialRecord WHERE RecordID = %s"
        cursor.execute(query, (record_id,))
        result = cursor.fetchone()
        cursor.close()

        if result:
            financial_record = FinancialRecord(
                record_id=result[0],
                employee_id=result[1],
                record_date=result[2],
                description=result[3],
                amount=result[4],
                record_type=result[5]
            )
            return financial_record
        else:
            return None

    def get_financial_records_for_employee(self, employee_id):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM FinancialRecord WHERE EmployeeID = %s"
        cursor.execute(query, (employee_id,))
        results = cursor.fetchall()
        cursor.close()

        financial_records = []
        for result in results:
            financial_record = FinancialRecord(
                record_id=result[0],
                employee_id=result[1],
                record_date=result[2],
                description=result[3],
                amount=result[4],
                record_type=result[5]
            )
            financial_records.append(financial_record)

        return financial_records

    def get_financial_records_for_date(self, record_date):
        cursor = self.db_connection.cursor()
        query = "SELECT * FROM FinancialRecord WHERE RecordDate = %s"
        cursor.execute(query, (record_date,))
        results = cursor.fetchall()
```

```
        cursor.close()

        financial_records = []
        for result in results:
            financial_record = FinancialRecord(
                record_id=result[0],
                employee_id=result[1],
                record_date=result[2],
                description=result[3],
                amount=result[4],
                record_type=result[5]
            )
            financial_records.append(financial_record)

        return financial_records
```

DatabaseContext:

• A class responsible for handling database connections and interactions.

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

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

import configparser

def get_connection_string(property_file):
   config = configparser.ConfigParser()
   config.read(property_file)
   connection_string = config.get('database', 'connection_string')
   return connection_string

import mysql.connector

# Connecting to the MySQL database
db = mysql.connector.connect(
   host="localhost",
```

```
    user="root",
    password="root",
    database="case_study"
)

cursor = db.cursor()

# Creating the Employee table
create_employee_table = """
CREATE TABLE IF NOT EXISTS Employee (
    EmployeeID INT AUTO_INCREMENT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DateOfBirth DATE NOT NULL,
    Gender CHAR(1) NOT NULL,
    Email VARCHAR(100) NOT NULL UNIQUE,
    PhoneNumber VARCHAR(20) NOT NULL,
    Address VARCHAR(200) NOT NULL,
    Position VARCHAR(100) NOT NULL,
    JoiningDate DATE NOT NULL,
    TerminationDate DATE
)
"""

# Creating the Payroll table
create_payroll_table = """
CREATE TABLE IF NOT EXISTS Payroll (
    PayrollID INT AUTO_INCREMENT PRIMARY KEY,
    EmployeeID INT NOT NULL,
    PayPeriodStartDate DATE NOT NULL,
    PayPeriodEndDate DATE NOT NULL,
    BasicSalary DECIMAL(10, 2) NOT NULL,
    OvertimePay DECIMAL(10, 2) NOT NULL,
    Deductions DECIMAL(10, 2) NOT NULL,
    NetSalary DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)
)
"""

# Creating the Tax table
create_tax_table = """
CREATE TABLE IF NOT EXISTS Tax (
    TaxID INT AUTO_INCREMENT PRIMARY KEY,
    EmployeeID INT NOT NULL,
    TaxYear YEAR NOT NULL,
    TaxableIncome DECIMAL(10, 2) NOT NULL,
    TaxAmount DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)
)
```

```
"""

# Creating the FinancialRecord table
create_financial_record_table = """
CREATE TABLE IF NOT EXISTS FinancialRecord (
   RecordID INT AUTO_INCREMENT PRIMARY KEY,
   EmployeeID INT NOT NULL,
   RecordDate DATE NOT NULL,
   Description VARCHAR(200) NOT NULL,
   Amount DECIMAL(10, 2) NOT NULL,
   RecordType VARCHAR(50) NOT NULL,
   FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID)
)
"""

# Execute the table creation queries
cursor.execute(create_employee_table)
cursor.execute(create_payroll_table)
cursor.execute(create_tax_table)
cursor.execute(create_financial_record_table)

# Commit the changes and close the connection
db.commit()
cursor.close()
db.close()
```

Custom Exceptions:

EmployeeNotFoundException:

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

PayrollGenerationException:

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

TaxCalculationException:

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

FinancialRecordException:

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

InvalidInputException:

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

DatabaseConnectionException:

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

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

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

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

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

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

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

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

Unit Testing:

Create NUnit test cases for car rental System are essential to ensure the correctness and

reliability of your system. Below are some example questions to guide the creation of NUnit test

cases for various components of the system:
Test Case: CalculateGrossSalaryForEmployee
• Objective: Verify that the system correctly calculates the gross salary for an employee.

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

```
import pytest
from datetime import date
from dao.fin_record_servics import FinancialRecordService
from exception_handling.exception import EmployeeNotFoundException
from exception_handling.exception import FinancialRecordException
from database.db import get_connection

@pytest.fixture
def financial_record_service():
   db_connection = get_connection()
   return FinancialRecordService(db_connection)

def test_add_financial_record(financial_record_service):
   employee_id = 1
   description = "Salary"
   amount = 5000.0
   record_type = "income"

        financial_record_service.add_financial_record(employee_id,  description,
amount, record_type)

                                                financial_records                =
financial_record_service.get_financial_records_for_employee(employee_id)
```

```
    assert len(financial_records) > 0
    latest_record = financial_records[-1]
    assert latest_record.employee_id == employee_id
    assert latest_record.description == description
    assert latest_record.amount == amount
    assert latest_record.record_type == record_type

def test_add_financial_record_for_invalid_employee(financial_record_service):
    invalid_employee_id = 999
    description = "Salary"
    amount = 5000.0
    record_type = "income"

    with pytest.raises(EmployeeNotFoundException):
                financial_record_service.add_financial_record(invalid_employee_id,
description, amount, record_type)


import pytest
from datetime import date
from dao.emp_service import EmployeeService
from entity.emp import Employee
from exception_handling.exception import EmployeeNotFoundException
from database.db 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 == "Ilakiya"
    assert employee.last_name == "Rangaraju"
    assert employee.date_of_birth == date(2002, 10, 27)

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="Joe",
```

```
        last_name="Shake",
        date_of_birth=date(1979, 8, 31),
        gender="M",
        email="joe@gmail.com",
        phone_number="8796534897",
        address="Chennai",
        position="Manager",
        joining_date=date(2021, 5, 17)
    )

    employee_service.add_employee(new_employee)

    added_employee = employee_service.get_employee_by_id(2)
    assert added_employee is not None

import pytest
from datetime import date
from dao.payroll_service import PayrollService
from exception_handling.exception import EmployeeNotFoundException
from exception_handling.exception import PayrollGenerationException
from database.db 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)

import pytest
from datetime import date
from dao.tax_serv import TaxService
from exception_handling.exception import EmployeeNotFoundException
from exception_handling.exception import TaxCalculationException
from database.db import get_connection

@pytest.fixture
def tax_service():
    db_connection = get_connection()
    return TaxService(db_connection)

def test_calculate_tax(tax_service):
    employee_id = 1
    tax_year = 2024

    tax_service.calculate_tax(employee_id, tax_year)

    taxes = tax_service.get_taxes_for_employee(employee_id)
    assert len(taxes) > 0
    latest_tax = taxes[-1]
    assert latest_tax.employee_id == employee_id
    assert latest_tax.tax_year == tax_year
    assert latest_tax.taxable_income > 0
    assert latest_tax.tax_amount > 0

def test_calculate_tax_for_invalid_employee(tax_service):
    invalid_employee_id = 999
    tax_year = 2023

    with pytest.raises(EmployeeNotFoundException):
        tax_service.calculate_tax(invalid_employee_id, tax_year)
```

Main module

```
from dao.emp_service import EmployeeService
from dao.payroll_service import PayrollService
from dao.tax_serv import TaxService
from dao.fin_record_servics import FinancialRecordService
from entity.emp import Employee
```

```
from database.db import get_connection
from datetime import date
from exception_handling.exception import EmployeeNotFoundException
from exception_handling.exception import PayrollGenerationException
from exception_handling.exception import TaxCalculationException
from exception_handling.exception import FinancialRecordException


def main():
    db_connection = get_connection()

    employee_service = EmployeeService(db_connection)
    payroll_service = PayrollService(db_connection)
    tax_service = TaxService(db_connection)
    financial_record_service = FinancialRecordService(db_connection)

    while True:
        print("\nPayXpert Payroll Management System")
        print("1. Employee Management")
        print("2. Payroll Processing")
        print("3. Tax Calculation")
        print("4. Financial Record Management")
        print("5. Exit")

        choice = input("Enter your choice: ")

        if choice == "1":
            employee_management(employee_service)
        elif choice == "2":
            payroll_processing(employee_service, payroll_service)
        elif choice == "3":
            tax_calculation(employee_service, tax_service)
        elif choice == "4":
                                    financial_record_management(employee_service,
financial_record_service)
        elif choice == "5":
            break
        else:
            print("Invalid choice. Please try again.")

    db_connection.close()


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,
        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'}")


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}")


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}")


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(employee_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()
```

```
C:\Users\HP\AppData\Local\Programs\Python\Python312\python.exe "H:/Case Study-PayXpert/main.py"

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: 4
Enter employee ID: 6
```

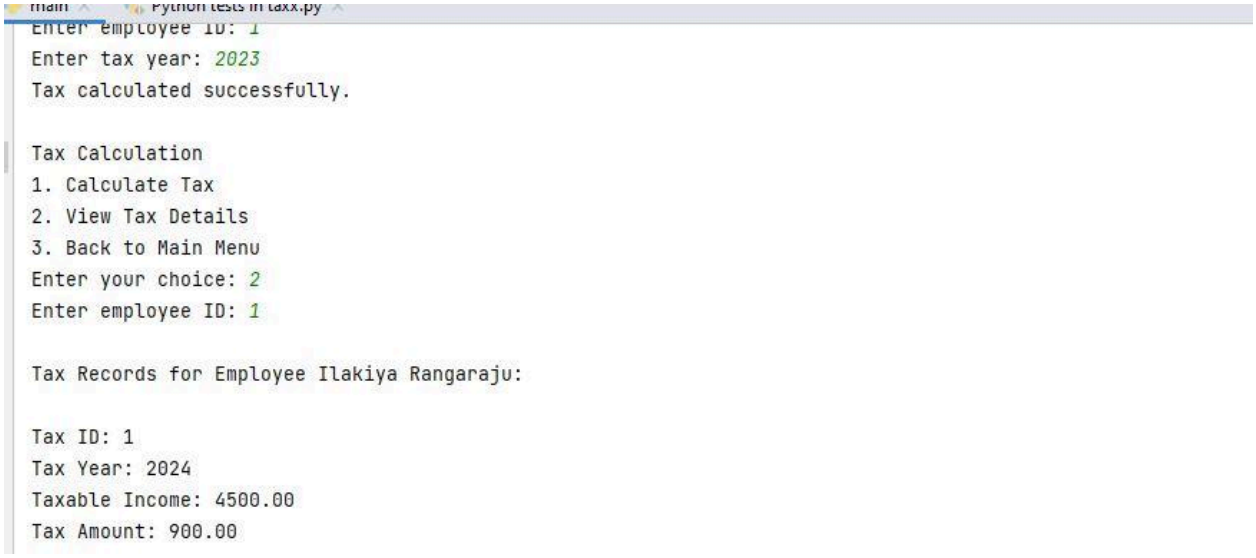
```
Employee Details:
Employee ID: 6
First Name: Malar
Last Name: Vizhi
Date of Birth: 1980-10-31
Gender: F
Email: malar@gmail.com
Phone Number: 9790342951
Address: Chennai
Position: Team Lead
Joining Date: 2022-09-13
Termination Date: N/A

Employee Management
1. Add Employee
2. Update Employee
3. Remove Employee
4. View Employee Details
```

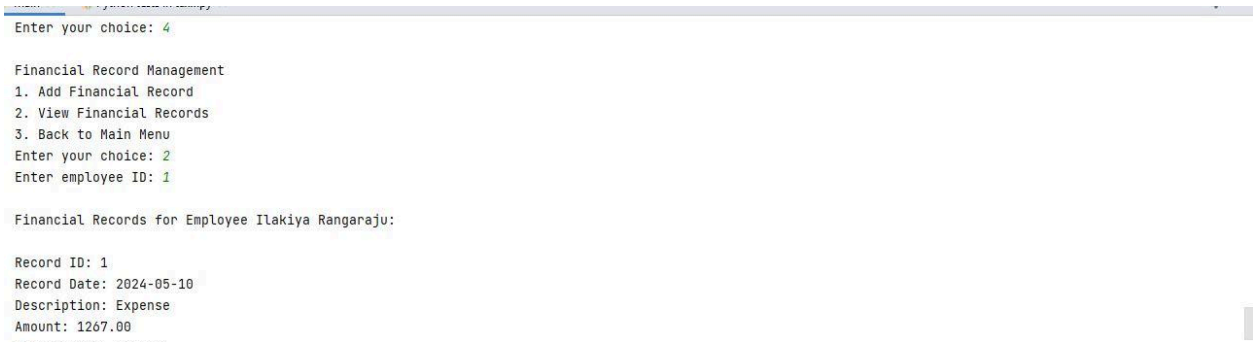
main × Python tests in taxx.py ×

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

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

```
Payroll Processing
1. Generate Payroll
2. View Payroll Details
3. Back to Main Menu
Enter your choice: 2
Enter employee ID: 6
No payroll records found for this employee.

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

Payroll Records for Employee Ilakiya Rangaraju:
```

```
Payroll ID: 1
Pay Period: 2023-07-12 - 2024-09-14
Basic Salary: 5000.00
Overtime Pay: 500.00
Deductions: 1000.00
Net Salary: 4500.00

Payroll ID: 2
Pay Period: 2023-01-01 - 2023-01-31
Basic Salary: 50000.00
Overtime Pay: 900.00
Deductions: 1000.00
Net Salary: 49900.00

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

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

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


main
Python tests in taxx.py
Enter employee ID: 1
Enter tax year: 2023
Tax calculated successfully.
Tax Calculation
1. Calculate Tax
2. View Tax Details
3. Back to Main Menu
Enter your choice: 2
Enter employee ID: 1
Tax Records for Employee Ilakiya Rangaraju:
Tax ID: 1
Tax Year: 2024
Taxable Income: 4500.00
Tax Amount: 900.00


Enter your choice: 4
Financial Record Management
1. Add Financial Record
2. View Financial Records
3. Back to Main Menu
Enter your choice: 2
Enter employee ID: 1
Financial Records for Employee Ilakiya Rangaraju:
Record ID: 1
Record Date: 2024-05-10
Description: Expense
Amount: 1267.00
Record Type: income