Unit Testing Employee Management System

Verification and validation Project

Simon Hunor

Automatics and applied Informatics, 2024

Contents

Abstract

This documentation outlines unit testing procedures for an existing Employee Management System, focusing on the **EmployeeRelationsManager** and **EmployeeManager** components. The tests cover key functionalities such as team leadership, team member associations, and salary calculations. The objective is to enhance system reliability and maintainability through systematic testing practices.

1 Introduction

1.1 Background

In the ever-evolving landscape of software development, the reliability and robustness of systems remain paramount. As organizations increasingly rely on Employee Management Systems to streamline personnel-related processes, ensuring the accuracy and dependability of these systems becomes a critical concern. This documentation addresses this concern by delving into the realm of unit testing for an existing Employee Management System.

1.2 Objectives

The primary objective of this documentation is to establish a comprehensive unit testing framework for the Employee Management System. Through systematic testing procedures, we aim to validate the correctness and reliability of key functionalities within the system, focusing on the **EmployeeRelationsManager** and **EmployeeManager** components.

1.3 Scope

The scope of this documentation encompasses unit testing practices applied to specific scenarios within the Employee Management System. It will provide insights into testing team leadership, team member associations, salary calculations, and the verification of email notification functionalities.

2 System Overview

2.1 EmployeeRelationsManager

2.1.1 Purpose

The **EmployeeRelationsManager** component serves as a crucial module within the Employee Management System, responsible for handling team-related functionalities. It manages relationships between employees, identifying team leaders and members, and facilitating queries related to team structures.

```
class RelationsManager:
    def __init__(self):
        self.employee_list = [
            Employee(id=1, first_name="John", last_name="Doe", base_salary
                                                     =3000,
                     birth_date=datetime.date(1970, 1, 31), hire_date=
                                                               datetime.date(
                                                               1990, 10, 1)),

            Employee(id=2, first_name="Myrta", last_name="Torkelson",
                                                     base_salary=1000,
                     birth_date=datetime.date(1980, 1, 1), hire_date=
                                                               datetime.date(
                                                               2000, 1, 1)),

            # Rest of the local database
        ]

        # Employee.ID=1 is a team lead and 2, 3 are part of the team
        self.teams = {
            1: [2, 3],
            4: [5, 6]
        }
```

2.1.2 Functionality

Team Leadership

The primary functionality of **EmployeeRelationsManager** includes identifying team leaders within the organization. A team leader is an employee responsible for overseeing a group of team members.

```
    def is_leader(self, employee) -> bool:
        return employee.id in self.teams
```

Team Member Associations

This component facilitates the retrieval of team members for a given team leader. It establishes associations between team leaders and their respective team members, enabling efficient team management.

```
    def get_team_members(self, employee: Employee) -> list:
        if self.is_leader(employee):
            member_ids = self.teams[employee.id]
            members = [e.id for e in self.employee_list if e.id in
                                                         member_ids]

            return members
```

2.2 EmployeeManager

2.2.1 Purpose

The EmployeeManager component plays a central role in the Employee Management System, focusing on individual employee-related functionalities. It encompasses tasks such as salary calculation, email notifications, and overall employee management.

2.2.2 Functionality

Salary Calculation

EmployeeManager calculates salaries for employees based on various factors, including base salary, years of service, and leadership roles. The yearly bonus and leader bonus per member attributes influence salary calculations for employees.

```
    def calculate_salary(self, employee: Employee) -> int:
        salary = employee.base_salary

        years_at_company = datetime.date.today().year - employee.hire_date
                                                      .year

        salary += years_at_company * EmployeeManager.yearly_bonus

        if self.relations_manager.is_leader(employee):
            team_members_count = len(self.relations_manager.
                                            get_team_members(employee
                                            ))
            salary += team_members_count * EmployeeManager.
                                            leader_bonus_per_member

        return salary
```

Email Notification

This component handles the generation and notification of employees regarding their salary transfers. It ensures that employees are informed promptly and accurately about their financial transactions.

```
    def calculate_salary_and_send_email(self, employee: Employee) -> str:
        salary = self.calculate_salary(employee)
        message = f"{employee.first_name} {employee.last_name} your salary
                                                      : {salary} has been
                                                      transferred to you."
        return message
```

Notice that the original function only printed the notification, for testing purposes, it has been modified to return the notification instead of printing it directly.

This system overview provides a high-level understanding of the **EmployeeRelations-Manager** and **EmployeeManager** components, laying the foundation for subsequent sections that delve into unit testing and verification procedures.

3 Pytest

3.1 What is pytest?

pytest is a testing framework for Python that simplifies the process of writing and executing tests. It provides a rich set of features, making it easy to design and run comprehensive test suites. Key features of pytest include:

- *Simplicity:* Writing tests with pytest is straightforward and requires minimal boilerplate code.
- *Powerful Assertions:* pytest offers expressive and detailed assertions for efficient test result verification.
- *Fixture Support:* Fixtures allow you to set up and tear down resources for tests, promoting reusability.
- *Test Discovery:* pytest automatically discovers and runs tests, making it easy to maintain and scale test suites.

3.2 Installation for pytest

To run the unit tests for the Employee Management System, you need to have pytest installed. Follow the steps below to install pytest:

1. Open a terminal or command prompt.
2. Run the following command to install pytest using pip:

```
pip install pytest
```

3. Once the installation is complete, you can verify the installation by running:

```
pytest --version
```

3.3 Calculator Model

Let's create a simple example of testing with pytest using a calculator model. Below is an example implementation along with tests:

```
# Calculator Model with its functions.
def add(a, b):
    return a + b


def multiply(a, b):
    return a * b


def substract(a, b):
    return a - b

# Unit tests for the functions
def test_calc_addition():
    output = calculator.add(1, 2)
    assert output == 3

def test_calc_substraction():
    output = calculator.substract(3, 1)
    assert output == 2


def test_calc_multiply():
    output = calculator.multiply(2, 3)
    assert output == 6
```

4 Test Scenarios

4.1 EmployeeRelationsManager

4.1.1 Scenario 1: Team Leader John Doe Existence Check

```
    def test_team_leader_john_doe(self):
        john_doe = Employee(id=1, first_name="John", last_name="Doe",
                                                base_salary=3000,
                            birth_date=date(1970, 1, 31), hire_date=date(
                                                                    1990, 10,
                                                                     1))
        self.assertTrue(self.rm.is_leader(john_doe))
```

The purpose of this test is to verify that the **is_leader** method of the **RelationsManager** correctly identifies John Doe as a team leader. If John Doe is indeed a team leader, the assertion will pass **(True)**. If not, the test will fail **(False)**.

The assertTrue method is part of the unittest framework and is used to assert that a given expression is True. In this case, it checks if the result of **self.rm.is_leader(john_doe)** is **True**.

The test is successful if the assertion passes without raising any exceptions, indicating that John Doe is correctly identified as a team leader by the **is_leader** method.

4.1.2 Scenario 2: Team Members of John Doe Check

```
    def test_team_members_john_doe(self):
        john_doe = Employee(id=1, first_name="John", last_name="Doe",
                                                base_salary=3000,
                            birth_date=date(1970, 1, 31), hire_date=date(
                                                                    1990, 10,
                                                                     1))
        expected_team_members = [2, 3]
        team_members = self.rm.get_team_members(john_doe)
        self.assertEqual(team_members, expected_team_members)
```

The purpose of this test is to ensure that the **get_team_members** method of the **RelationsManager** correctly retrieves the team members of John Doe.

The **assertEqual** method is part of the unittest framework and is used to assert that two values are equal. In this case, it checks if the result of **self.rm.get_team_members(john_doe)** is equal to the expected list [2, 3].

The test is successful if the assertion passes without raising any exceptions, indicating that the team members for John Doe are correctly retrieved.

4.1.3 Scenario 3: Tomas Andre not a Team Member of John Doe Check

```
    def test_tomas_andre_not_team_member_john_doe(self):
        john_doe = Employee(id=1, first_name="John", last_name="Doe",
                                              base_salary=3000,
                            birth_date=date(1970, 1, 31), hire_date=date(
                                                                   1990, 10,
                                                                    1))
        tomas_andre = Employee(id=5, first_name="Tomas", last_name="Andre"
                                              , base_salary=1600,
                               birth_date=date(1995, 1, 1), hire_date=date
                                                                      (2015,
                                                                       1, 1)
                                                                      )
        team_members = self.rm.get_team_members(john_doe)
        self.assertNotIn(tomas_andre.id, team_members)
```

The purpose of this test is to verify that Tomas Andre is not considered a team member of John Doe.

The **assertNotIn** method is part of the unittest framework and is used to assert that a given value is not present in a specified list or iterable.

The test is successful if the assertion passes without raising any exceptions, indicating that Tomas Andre is correctly identified as not being a team member of John Doe.

4.1.4 Scenario 4: Gretchen Walford Salary Check

```
    def test_gretchen_watford_salary(self):
        gretchen_watford = Employee(id=4, first_name="Gretchen", last_name
                                                 ="Watford", base_salary=4000,
                                                  birth_date=date(1960, 1, 1),
                                                  hire_date=date(1990, 1, 1))
        expected_salary = 4000
        self.assertEqual(gretchen_watford.base_salary, expected_salary)
```

The purpose of this test is to ensure that the base salary attribute for Gretchen Watford is correctly set to $4000.

The **assertEqual** method is part of the unittest framework and is used to assert that two values are equal.

The test is successful if the assertion passes without raising any exceptions, indicating that Gretchen Watford's base salary is correctly initialized to $4000.

4.1.5 Scenario 5: Tomas Andre not a Team Leader and Team Members Check

```
    def test_tomas_andre_not_team_leader(self):
        tomas_andre = Employee(id=5, first_name="Tomas", last_name="Andre"
                                                 , base_salary=1600,
                              birth_date=date(1995, 1, 1), hire_date=date
                                                                     (2015,
                                                                      1, 1)
                                                                     )
        is_leader = self.rm.is_leader(tomas_andre)

        self.assertFalse(is_leader)

        team_members = self.rm.get_team_members(tomas_andre)

        self.assertEqual(team_members, None)
```

The purpose of this test is to verify that Tomas Andre is correctly identified as not being a team leader and that attempting to retrieve his team members returns None.

The **assertFalse** method is part of the unittest framework and is used to assert that a given expression is False.

The **assertEqual** method is used to check if the result of attempting to retrieve team members for Tomas Andre is equal to **None**. The test is successful if both assertions pass without raising any exceptions.

4.1.6 Scenario 6: Jude Overcash not in database Check

```
    def test_jude_overcash_not_in_database(self):
        jude_overcash = Employee(id=7, first_name="Jude", last_name="
                                              Overcash", base_salary=2500,
                                              birth_date=date(1985, 1, 1),
                                              hire_date=date(2010, 1, 1))
        all_employees = self.rm.get_all_employees()
        self.assertNotIn(jude_overcash, all_employees)
```

The purpose of this test is to ensure that the employee Jude Overcash is correctly identified as not being in the database.

The **assertNotIn** method is part of the unittest framework and is used to assert that a given value is not present in a specified list or iterable.

The test is successful if the assertion passes without raising any exceptions, indicating that Jude Overcash is not stored in the database.

4.2 EmployeeManager

4.2.1 Non-Team Leader with Hire Date 10.10.1998 and Base Salary $1000 Check

```
    def test_non_leader_salary(self):
        employee = Employee(id=8, first_name="Non", last_name="Leader",
                                                base_salary=1000,
                            birth_date=date(1980, 1, 1), hire_date=date(
                                                                1998, 10,
                                                                 10))
        expected_salary = 3000
        self.assertNotEqual(self.em.calculate_salary(employee),
                                                expected_salary)
```

self.assertNotEqual(self.em.calculate_salary(employee), expected_salary):

This line contains an assertion statement. It checks whether the result of calculating the salary for the employee is not equal to the **expected_salary**.

4.2.2 Scenario 8: Team Leader with 3 Members, Hire Date 10.10.2008, and Base Salary $2000 Check

```
    def test_team_leader_salary(self):
        team_leader = Employee(id=9, first_name="Team", last_name="Leader"
                                                , base_salary=2000,
                               birth_date=date(1980, 1, 1), hire_date=date
                                                                  (2008,
                                                                   10,
                                                                  10))
        self.rm.teams[9] = [10, 11, 12]
        expected_salary = 3600
        self.assertEqual(self.em.calculate_salary(team_leader),
                                                expected_salary)
```

self.assertEqual(self.em.calculate_salary(team_leader), expected_salary): This line contains an assertion statement. It checks whether the result of calculating the salary for the **team_leader** is equal to the **expected_salary**.

4.2.3 Scenario 9: Email Notification Validation Check

```
    def test_salary_calculation_and_email_notification(self):
        employee = Employee(id=1, first_name="John", last_name="Doe",
                                                    base_salary=3000,
                              birth_date=date(1970, 1, 31), hire_date=date(
                                                                  1990, 10,
                                                                   1))

        expected_message = f"{employee.first_name} {employee.last_name}
                                                    your salary: {self.em.
                                                    calculate_salary(employee)}
                                                    has been transferred to you."
        self.assertEqual(self.em.calculate_salary_and_send_email(employee)
                                                    , expected_message)
```

self.assertEqual(self.em.calculate_salary_and_send_email(employee), expected_message): This line contains an assertion statement. It checks whether the result of calling the **calculate_salary_and_send_email** method for the employee matches the **expected_message**.

5 Conclusion

In this documentation, we have presented an overview of the Employee Relations Management system and the associated unit tests. The unit tests were designed to validate various functionalities of the system, including team leadership, salary calculations, and email notifications.

The test scenarios covered a range of cases, from checking team leadership and team members to validating salary calculations and email notifications. Each test aimed to ensure the accuracy and reliability of the system's functionalities.