SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM
KOLOZSVÁRI KAR

Sapientia Hungarian University Of Transylvania

Faculty of Technical and Human Sciences, Targu Mures

Software Testing and Validation

Student:
Molnár Orsolya-Izabella

Supervisor:
Dr. Szántó Zoltán

A Project submitted for the Software testing and validation course

Targu Mures

April 8, 2024

Contents

1 Introduction

1.1 The essential role of software testing in QA

In the intricate tapestry of software development, where lines of code orchestrate complex functionalities, testing stands as an indispensable guardian of quality and reliability. Software testing is a meticulous and multifaceted process aimed at exposing flaws, inconsistencies, and deviations from the desired behavior within a software application. As software grows evermore complex, infiltrating nearly every facet of industry and daily life, the repercussions of untested or poorly tested software range from minor inconveniences to costly disruptions, safety hazards, and even loss of life. [1]

1.2 Types of Software Testing

The discipline of software testing encompasses a vast spectrum of methodologies and techniques, each serving distinct purposes across the software development lifecycle (SDLC).

- **Unit Testing:** At the most granular level, unit testing is the practice of verifying the individual building blocks of code—functions, methods, or classes. Unit tests provide an isolated environment to scrutinize these components, ensuring they operate as intended and produce the correct results under a variety of input conditions.
- **Integration Testing:** Where unit testing focuses on the microcosm, integration testing evaluates the harmonious interplay between interconnected software modules. The goal is to validate if different parts of the system correctly exchange data and function cohesively when combined.
- **System Testing:** System testing takes a holistic approach, treating the entire software application as a ’black box.’ It examines the end-to-end behavior of the system to ensure compliance with functional and non-functional requirements. System testing encompasses various types of tests, including:
  - **Functional Testing:** Validates if the system accurately executes actions in accordance with user specifications.
  - **Performance Testing:** Assesses the system’s speed, responsiveness, and stability under varying loads and stress conditions.
  - **Security Testing:** Identifies vulnerabilities and seeks to prevent unauthorized access or data breaches.
  - **Usability Testing:** Evaluates the user experience, focusing on ease of use, intuitiveness, and accessibility.
- **Regression Testing:** Regression testing provides a safety net against unintended consequences. When changes or updates are made to existing code, regression tests are executed to ensure earlier functionalities remain intact and no new defects have been introduced.

[2]

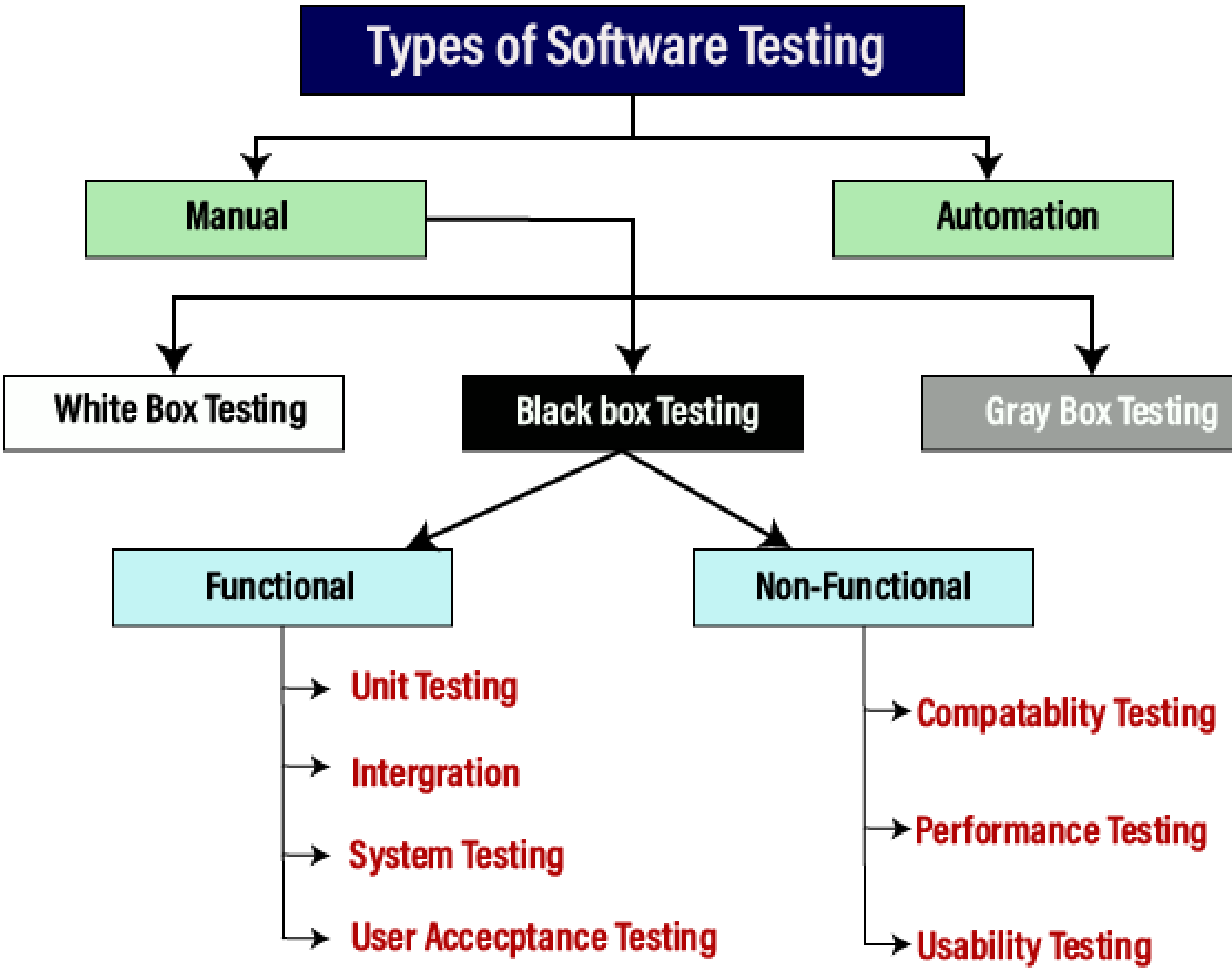


Figure 1: Types of Software Testing

2 What is Unit Testing?

Unit testing is a cornerstone of software development, laying the foundation for the creation of robust and reliable code. At its heart, unit testing centers around isolating and verifying the smallest testable components of a software application. These units are typically individual functions, methods, or classes within the codebase.

2.1 The purpose of Unit Testing

- **Catching error early:** Unit tests pinpoint defects at the most granular level of the code, enabling developers to identify and fix bugs early in the development process. This saves time, effort, and significantly reduces costs that would escalate if bugs made it through to later stages of testing or even into production.
- **Code confidence:** A comprehensive suite of unit tests instills confidence that individual code units are behaving correctly. This allows developers to modify or refactor existing code with greater assurance that changes won't introduce unexpected consequences or break existing functionality.
- **Design encouragement:** The act of writing unit tests often encourages developers to create code that is modular, testable, and loosely coupled. This leads to a more maintainable and adaptable codebase overall.
- **Living documentation:** Unit tests describe the expected behavior of code components. This serves as a form of "living documentation," making the code easier to understand and reducing ambiguity for new developers joining the project.

2.2 Characteristics of Well-Written Unit Tests

- **Isolation:** A good unit test exercises a single unit of code without dependencies on external components like databases, file systems, or network interactions. Mocking techniques are often used to replace those dependencies with controlled test doubles.
- **Fast:** Unit tests should execute blazingly fast, taking mere milliseconds to run. This encourages developers to run tests frequently and integrate them into the development workflow.
- **Repeatable:** Unit tests should produce consistent results every time they're executed, regardless of environment or external factors.
- **Self-Verifying:** Unit tests should automatically determine if they pass or fail, requiring minimal human intervention.
- **Thourough:** A robust set of unit tests aims to achieve high code coverage, exercising all crucial execution paths and various input scenarios (both valid and invalid) within the unit being tested.

[3]

3 About the project

This project is a simple Employee Management System implemented in [4]**Python**. It includes functionalities such as calculating employee salary, sending salary notifications and displaying employee information.

4 Description of the selected Unit Testing Suite

The unit testing suite for this Employee Management System is designed to ensure that each individual component of the system functions as expected. It is implemented using the [5]**pytest** framework, a popular testing tool in Python known for its simplicity and ease of use.

4.1 Motivations

I chose this framework because:

- It requires less boilerplate code compared to other testing frameworks.
- Despite its simplicity, pytest is also very powerful. It supports parameterized testing, where you can run the same test function with different sets of data.
- pytest provides detailed reports when tests fail. It shows the exact point of failure and the values of all variables at the time of failure, which can be very helpful for debugging.
- pytest can easily integrate with other testing libraries and frameworks. For example, it can run tests written for **unittest** or **nose**.

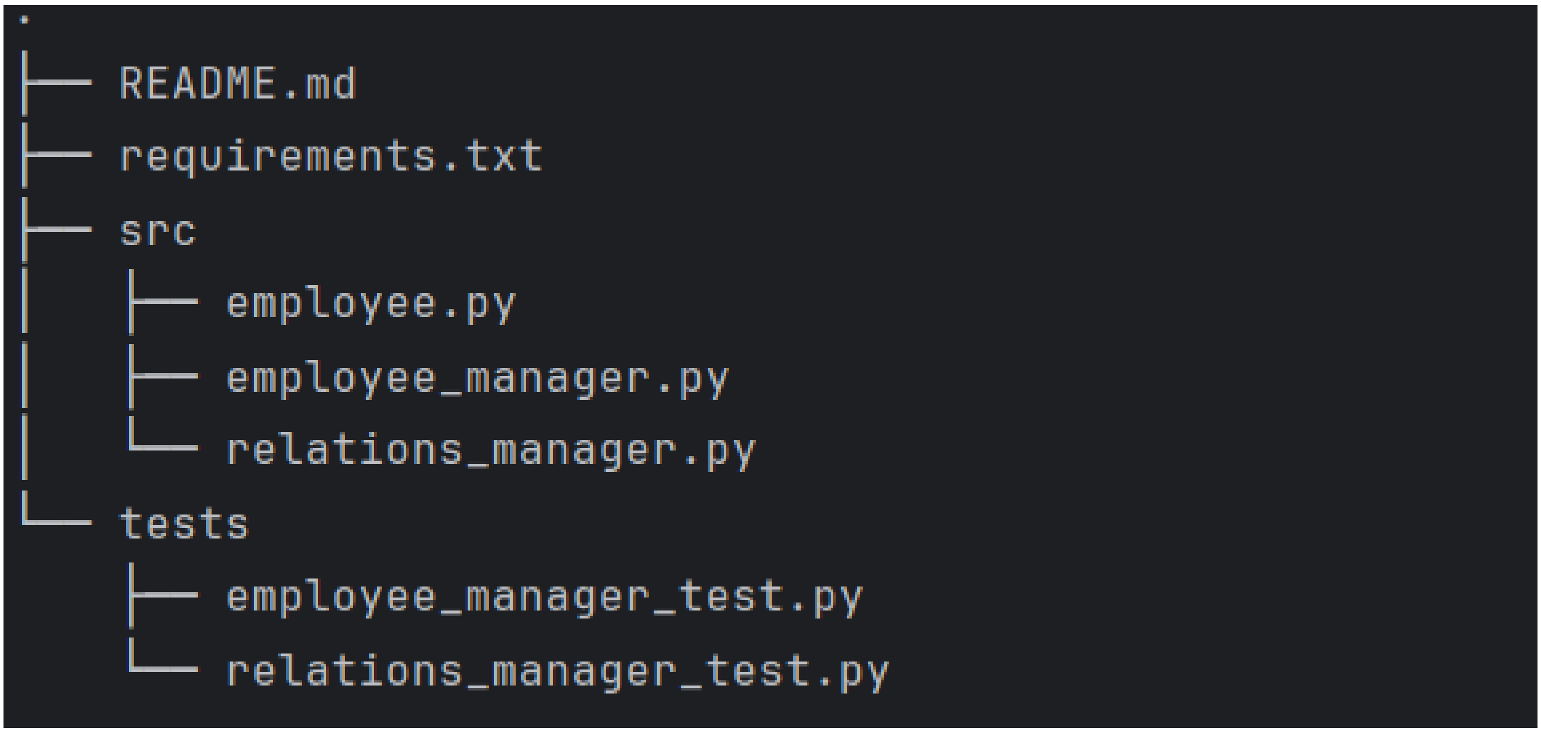
4.2 Installation

The installation process for this project is straightforward and involves installing the necessary Python dependencies. These dependencies are listed in a file called **requirements.txt**. To install the dependencies, you need to have Python and pip (Python's package installer) installed on your system. If you don't have Python installed, you can download it from the official website. Pip is included by default from Python version 3.4 onwards. Once you have Python and pip installed, you can install the project dependencies by navigating to the project directory in your terminal and running the following command:

```
pip install -r requirements.txt
```

This command tells pip to install the packages listed in the **requirements.txt** file. After running this command, all the necessary dependencies for the project should be installed, and you should be able to run the project and its tests.

5 Codebase description



```
.
├── README.md
├── requirements.txt
├── src
│   ├── employee.py
│   ├── employee_manager.py
│   └── relations_manager.py
└── tests
    ├── employee_manager_test.py
    └── relations_manager_test.py
```

Figure 2: Project Structure

- README.md: This file contains information about the project, including a description of the project, how to install dependencies, and how to run tests.
- requirements.txt: This file lists the Python dependencies required for the project.
- src/: This directory contains the source code of the project.

  1. employee.py: The employee.py file contains the definition of a Python class named Employee. This class is a data model that represents an employee in an organization. The class is defined using Python's dataclasses module, which provides a decorator and functions for adding special methods to classes.

     ```
     from dataclasses import dataclass
     import datetime
     ```

     The @dataclass decorator is used to automatically add special methods to the Employee class, including __init__, __repr__, and others. These methods are used for initializing the class, representing the class as a string, and comparing instances of the class, respectively.

     ```
     @dataclass
     ```

     The Employee class has six attributes: id, first_name, last_name, birth_date, base_salary, and hire_date. Each attribute is typed, meaning that a type hint is provided for each attribute. This is not required in Python, but it can make the code easier to understand and debug.

```python
class Employee:
    id: int
    first_name: str
    last_name: str
    birth_date: datetime.date
    base_salary: int
    hire_date: datetime.date
```

The id attribute represents the unique identifier of an employee. The first_name and last_name attributes represent the first and last name of an employee, respectively. The birth_date attribute represents the birth date of an employee, and it is of type **datetime.date**. The base_salary attribute represents the base salary of an employee. The hire_date attribute represents the date when the employee was hired, and it is also of type datetime.date.

2. employee_manager.py: The employee_manager.py file contains the definition of a Python class named EmployeeManager. This class is responsible for managing employees and their salaries in an organization.

```python
class EmployeeManager:
    yearly_bonus = 100
    leader_bonus_per_member = 200
```

The EmployeeManager class has two class-level attributes: yearly_bonus and leader_bonus_per_member. These attributes represent the yearly bonus that each employee receives and the additional bonus that a team leader receives for each member of their team, respectively. The __init__ method initializes an instance of the EmployeeManager class. It takes a RelationsManager instance as an argument and assigns it to the relations_manager attribute. This RelationsManager instance is used to manage relationships between employees.

```python
def __init__(self, relations_manager: RelationsManager):
    self.relations_manager = relations_manager
```

The calculate_salary method calculates the salary of an employee. It takes an Employee instance as an argument and returns the calculated salary. The salary is calculated based on the employee's base salary, their years at the company, and whether they are a team leader.

```python
def calculate_salary(self, employee: Employee) -> int:
```

The calculate_salary_and_send_email method calculates the salary of an employee and sends them an email with the calculated salary. It takes an Employee instance as an argument and does not return anything. The email sending functionality is not implemented in this method; instead, it prints a message to the console.

```python
def calculate_salary_and_send_email(self, employee: Employee)
```

In the if __name__ == '__main__': block at the end of the file, an instance of RelationsManager and EmployeeManager are created, and the calculate_salary_and_send_email method is called for a specific employee.

```python
if __name__ == '__main__':
```

3. relations_manager.py: contains the definition of a Python class named RelationsManager. This class is responsible for managing the relationships between employees in an organization.

```
class RelationsManager:
    def __init__(self):
```

The __init__ method initializes an instance of the RelationsManager class. It sets up a list of Employee instances, representing the employees in the organization. Each Employee instance is created with an id, first_name, last_name, base_salary, birth_date, and hire_date.

```
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)),
    ...
]
```

The __init__ method also sets up a dictionary called teams, where the keys are the ids of team leaders and the values are lists of ids of the team members.

```
self.teams = {
    1: [2, 3],
    4: [5, 6]
}
```

The is_leader method checks if an Employee instance is a team leader. It does this by checking if the id of the Employee instance is a key in the teams dictionary.

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

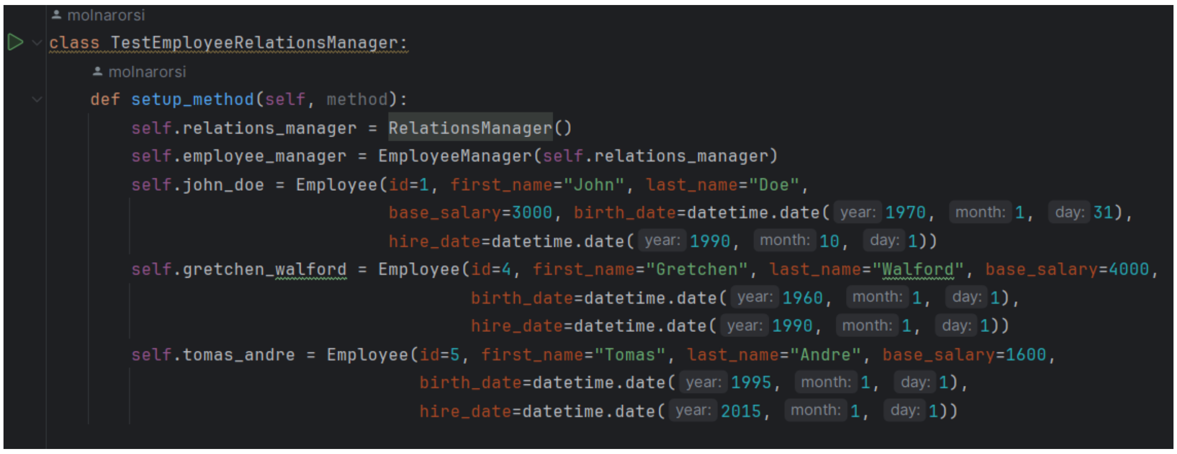
The get_all_employees method returns the list of all Employee instances.

```
def get_all_employees(self) -> list:
    return self.employee_list
```

The get_team_members method returns a list of team members for a given Employee instance, if the Employee instance is a team leader. It does this by getting the list of ids associated with the Employee instance's id in the teams dictionary, and then finding the Employee instances with those ids in the employee_list.

```
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 membe
  return members
```

- tests/: This directory contains the test cases for the project.



```
  molnarorsi
class TestEmployeeRelationsManager:
      molnarorsi
    def setup_method(self, method):
        self.relations_manager = RelationsManager()
        self.employee_manager = EmployeeManager(self.relations_manager)
        self.john_doe = Employee(id=1, first_name="John", last_name="Doe",
                                 base_salary=3000, birth_date=datetime.date( year: 1970,  month: 1,  day: 31),
                                 hire_date=datetime.date( year: 1990,  month: 10,  day: 1))
        self.gretchen_walford = Employee(id=4, first_name="Gretchen", last_name="Walford", base_salary=4000,
                                         birth_date=datetime.date( year: 1960,  month: 1,  day: 1),
                                         hire_date=datetime.date( year: 1990,  month: 1,  day: 1))
        self.tomas_andre = Employee(id=5, first_name="Tomas", last_name="Andre", base_salary=1600,
                                    birth_date=datetime.date( year: 1995,  month: 1,  day: 1),
                                    hire_date=datetime.date( year: 2015,  month: 1,  day: 1))
```

Figure 3: TestEmployeeRelationsManager class

1. employee_manager_test.py: This file contains test cases for the Employee-Manager class.
2. relations_manager_test.py: This file contains test cases for the Relations-Manager class.

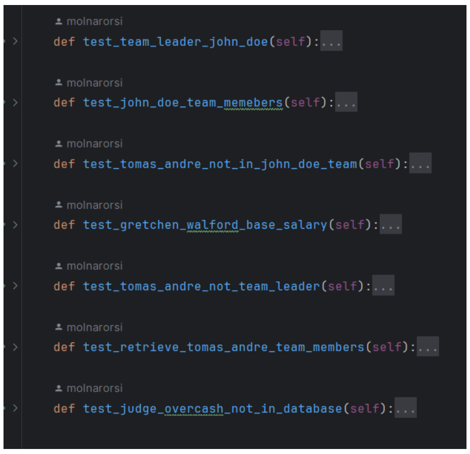
6 Testing

6.1 Relation Manager Tests

The test suite begins by importing necessary modules and adjusting the system path to include the parent directory. This is done to allow importing of modules from the src directory.

```
import datetime
import pytest
import sys
import os
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__),
from src.employee import Employee
from src.employee_manager import EmployeeManager
from src.relations_manager import RelationsManager
```

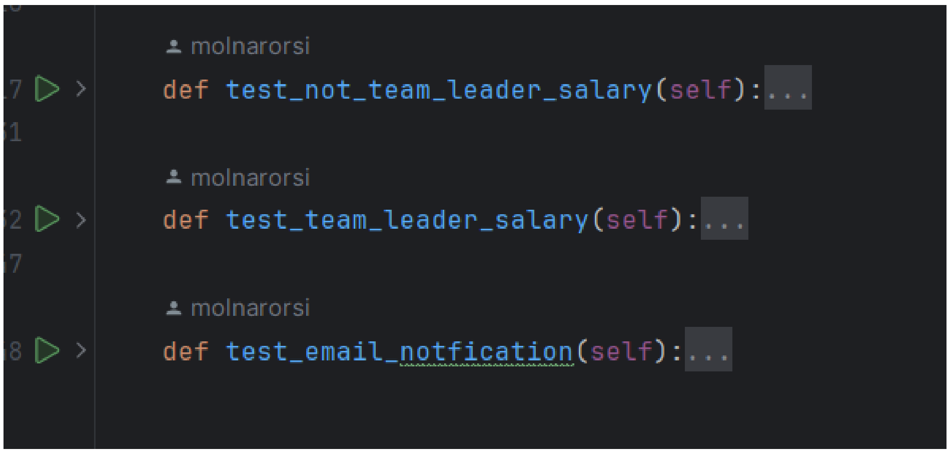
The TestEmployeeRelationsManager class is defined to contain all the test cases. The setup_method function is used to set up the testing environment before each test case. It initializes instances of RelationsManager, EmployeeManager, and Employee. The test cases are methods within the TestEmployeeRelationsManager class. Each method represents a single test case.



```
molnarorsi
def test_team_leader_john_doe(self):...

molnarorsi
def test_john_doe_team_memebers(self):...

molnarorsi
def test_tomas_andre_not_in_john_doe_team(self):...

molnarorsi
def test_gretchen_walford_base_salary(self):...

molnarorsi
def test_tomas_andre_not_team_leader(self):...

molnarorsi
def test_retrieve_tomas_andre_team_members(self):...

molnarorsi
def test_judge_overcash_not_in_database(self):...
```

Figure 4: Test cases in RelationsManager

- The test_team_leader_john_doe method tests whether John Doe is a team leader. It retrieves all employees and checks if John Doe is a team leader using the is_leader method of relations_manager. The test passes if John Doe is a team leader.
- The test_john_doe_team_members method tests whether the team members of John Doe are as expected. It retrieves the team members of John Doe using the get_team_members method of relations_manager and checks if they match the expected team members. The test passes if the actual team members match the expected team members.
- The test_tomas_andre_not_in_john_doe_team method tests whether Tomas Andre is not in John Doe's team. It retrieves the team members of John Doe and checks if Tomas Andre is not in the team. The test passes if Tomas Andre is not in John Doe's team.
- The test_gretchen_walford_base_salary method tests whether the base salary of Gretchen Walford is as expected. It retrieves the base salary of Gretchen Walford and checks if it matches the expected base salary. The test passes if the actual base salary matches the expected base salary.
- The test_tomas_andre_not_team_leader method tests whether Tomas Andre is not a team leader. It checks if Tomas Andre is a team leader using the is_leader method of relations_manager. The test passes if Tomas Andre is not a team leader.
- The test_retrieve_tomas_andre_team_members method tests whether Tomas Andre does not have any team members. It retrieves the team members of Tomas Andre using the get_team_members method of relations_manager and checks if he does not have any team members. The test passes if Tomas Andre does not have any team members.
- The test_judge_overcash_not_in_database method tests whether Jude Overcash is not in the database. It retrieves all employees and checks if Jude Overcash is not among them. The test passes if Jude Overcash is not in the database.



```
molnarorsi
def test_not_team_leader_salary(self):...

molnarorsi
def test_team_leader_salary(self):...

molnarorsi
def test_email_notfication(self):...
```

Figure 5: Test cases in EmployeeManager

6.2 Employee Manager Tests

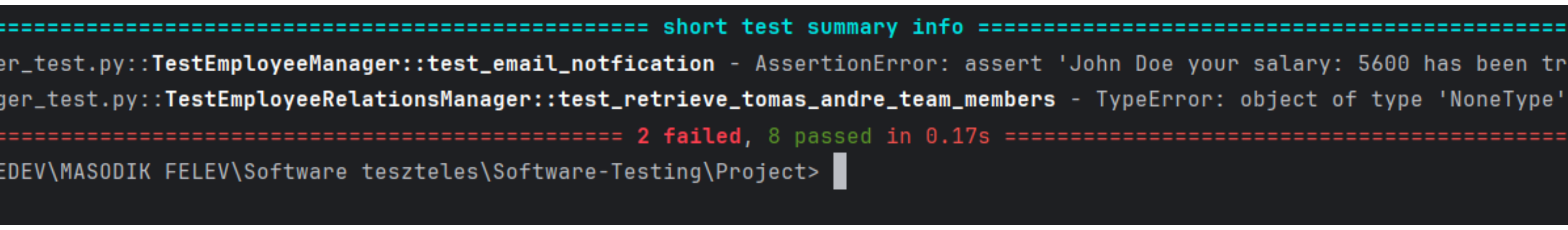
The TestEmployeeManager class is defined to contain all the test cases. The setup_method function is used to set up the testing environment before each test case. It initializes instances of RelationsManager and EmployeeManager.

```
class TestEmployeeManager:
 def setup_method(self):
  self.rm = RelationsManager()
  self.em = EmployeeManager(relations_manager=self.rm)
```

The test cases are methods within the TestEmployeeManager class. Each method represents a single test case.

- The test_not_team_leader_salary method tests the salary calculation for an employee who is not a team leader. It creates an Employee instance and calculates the expected salary. Then, it uses the calculate_salary method of EmployeeManager to calculate the actual salary. The test passes if the actual salary matches the expected salary.
- The test_team_leader_salary method tests the salary calculation for a team leader. It creates an Employee instance representing a team leader and adds team members to the leader using the teams attribute of RelationsManager. Then, it calculates the expected salary and uses the calculate_salary method of EmployeeManager to calculate the actual salary. The test passes if the actual salary matches the expected salary.
- The test_email_notification method tests the email notification functionality of EmployeeManager. It creates an Employee instance and calculates the expected email message. Then, it uses the calculate_salary_and_send_email method of EmployeeManager to get the actual email message. The test passes if the actual email message matches the expected email message.

Here is the output after running the pytest tests command:



```
================================================== short test summary info ==================================================
er_test.py::TestEmployeeManager::test_email_notfication - AssertionError: assert 'John Doe your salary: 5600 has been tr
ger_test.py::TestEmployeeRelationsManager::test_retrieve_tomas_andre_team_members - TypeError: object of type 'NoneType'
================================================ 2 failed, 8 passed in 0.17s ================================================
EDEV\MASODIK FELEV\Software teszteles\Software-Testing\Project>
```

Figure 6: Test Results

7 Conclusion

In conclusion, pytest is a very useful testing method and as shown in my documentation, it is easy to use. Its installation does not require effort, a test can be created by automatic caching. The functions, which contain the test, must have the word "test" in their name. The codebase I tested on is a rudimentary and simple one, but fit for purpose and it is a perfect way to illustrate pytest. As it is not perfect, it is you may run into errors if you want to run the given tests, such as dates, the format of the dates in the employee data does not match the format of the format specified in the caller text. Another possible problem I have noticed is the that when calculating the salary, since we always use the current year the output is different from the expected value. So, when testing the given code base, most of the tests were successful, but nevertheless not perfect and problems may arise when performing more complex tests

List of Figures

References

[1] J. Pan, “Software testing,” *Dependable Embedded Systems*, vol. 5, 1999.

[2] Javatpoint. (2024) Software testing tutorial. [Online]. Available: https://www.javatpoint.com/software-testing-tutorial

[3] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Manning Publications, 2020.

[4] Python Software Foundation, *The Python Language Reference*. [Online]. Available: https://docs.python.org/3/reference/

[5] pytest developer community, “pytest: helps you write better programs.” [Online]. Available: https://docs.pytest.org/en/8.0.x/