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.

```
1     def is_leader(self, employee) -> bool:
2         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$_{t}eam_{m}embers(self, employee : Employee) -> list : if self.is_{l}eader(employee) : member_{i}ds = self.teams[employee.id] members = [e.id for e in self.employee_{l}ist if e.id in member_{i}ds]$

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 **EmployeeRelationsManager** 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

```
1     def test_tomas_andre_not_team_member_john_doe(self):
2         john_doe = Employee(id=1, first_name="John", last_name="Doe", base_salary=3000,
3                             birth_date=date(1970, 1, 31), hire_date=date(1990, 10, 1))
4         tomas_andre = Employee(id=5, first_name="Tomas", last_name="Andre", base_salary
          ↪  =1600,
5                                birth_date=date(1995, 1, 1), hire_date=date(2015, 1, 1))
6         team_members = self.rm.get_team_members(john_doe)
7         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

```
1     def test_jude_overcash_not_in_database(self):
2         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))
3         all_employees = self.rm.get_all_employees()
4         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

```
1     def test_non_leader_salary(self):
2         employee = Employee(id=8, first_name="Non", last_name="Leader", base_salary=
          ↪  1000,
3                             birth_date=date(1980, 1, 1), hire_date=date(1998, 10, 10))
4         expected_salary = 3000
5         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

```
1     def test_team_leader_salary(self):
2         team_leader = Employee(id=9, first_name="Team", last_name="Leader", base_salary
          ↪  =2000,
3                                birth_date=date(1980, 1, 1), hire_date=date(2008, 10, 10
                                 ↪  ))
4         self.rm.teams[9] = [10, 11, 12]
5         expected_salary = 3600
6         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

```
1     def test_salary_calculation_and_email_notification(self):
2         employee = Employee(id=1, first_name="John", last_name="Doe", base_salary=3000,
3                             birth_date=date(1970, 1, 31), hire_date=date(1990, 10, 1))
4
5         expected_message = f"{employee.first_name} {employee.last_name} your salary: {
          ↪  self.em.calculate_salary(employee)} has been transferred to you."
6         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 Database Integration

5.1 Overview of Database Integration

5.1.1 Purpose of Integrating Database

The integration of a database into our Employee Management System marks a pivotal enhancement in how data is stored, retrieved, and managed within the application. The primary purpose behind this strategic move was to transition from a static, in-memory data management approach to a dynamic, persistent storage solution. This change not only facilitates scalability by accommodating a growing volume of data but also enhances data integrity and security. By leveraging a database, the application gains the ability to handle complex queries, relationships, and transactions more efficiently, laying a foundation for future expansions and functionalities.

5.1.2 Benefits of Database Integration

The shift to a database-driven architecture brings forth several key advantages, contributing to the overall robustness and efficiency of the Employee Management System:

- *Persistence of Data:* Unlike in-memory storage, which is volatile and limited to the application's lifecycle, a database ensures that all employee and team information persists safely across sessions. This means that data is not lost when the application restarts or crashes, providing a reliable storage solution.

- *Scalability:* As the organization grows, so does the amount of data that needs to be managed. Databases are designed to handle large volumes of data, making it easier to scale the application without compromising performance or stability.

- *Data Integrity and Consistency:* Database management systems enforce data integrity constraints, such as unique identifiers for employees and referential integrity between teams and their members. This automatic enforcement helps prevent data anomalies and ensures that the data remains consistent and accurate over time.

- *Concurrent Access:* Databases are built to handle multiple concurrent users and applications accessing and modifying the data simultaneously. This capability is crucial for enterprise applications where different parts of the system or different users need to interact with the data at the same time.

- *Advanced Query Capabilities:* With a database, the application can perform complex queries and aggregations, enabling more sophisticated features such as reporting, filtering employees based on various criteria, and analyzing team structures. SQL (Structured Query Language) offers a powerful and flexible way to retrieve and manipulate data.

- *Security:* Databases provide robust security features, including access control, encryption, and secure data transmission. This ensures that sensitive employee information is protected from unauthorized access and breaches.

- *Maintenance and Backup*: Modern databases come with tools for backup, recovery, and maintenance, reducing the risk of data loss and making it easier to manage data health and performance over time.

Integrating a database into the Employee Management System signifies a significant step towards a more mature, reliable, and scalable application. It lays the groundwork for future

enhancements, including integration with other systems, introduction of new features, and adaptation to changing business needs.

5.2 What is SQLAlchemy?

SQLAlchemy is a powerful and flexible SQL toolkit and **Object-Relational Mapping (ORM)** library for **Python**. It provides a full suite of well-known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language. **SQLAlchemy** abstracts away many of the common tasks associated with database interactions, allowing developers to work with database tables in terms of Python classes and objects. It supports a wide range of SQL database engines, including **SQLite**, **PostgreSQL**, **MySQL**, and **Oracle**, making it highly versatile for various database applications.

SQLAlchemy operates in two primary modes:

- *Core:* At its core, **SQLAlchemy** offers a schema-centric SQL generation and execution library. This mode is particularly useful for those who prefer to interact with their database using SQL queries directly but enjoy the convenience of Pythonic code and data structures.

- *ORM:* The **ORM** (Object-Relational Mapping) layer allows developers to map Python classes to database tables, and instances of those classes to rows in their corresponding tables. This enables the manipulation of database data using Pythonic operations, reducing the need to write SQL queries manually for basic CRUD (Create, Read, Update, Delete) operations.

5.2.1 Key Features of SQLAlchemy

- *Data Model:* Define database schemas and objects using Python classes and syntax.
- *Querying:* Use Python expressions to build SQL queries dynamically.
- *Session Management:* Automate and manage database sessions and transactions.
- *Database Engine Support:* Compatible with a wide variety of database engines.
- *Migrations:* Tools like Alembic (built on top of SQLAlchemy) support database schema migrations.

5.2.2 How to Install SQLAlchemy in Python

You can install **SQLAlchemy** using pip, Python's package installer. Ensure you have Python and pip installed on your system before proceeding. To install SQLAlchemy, open your terminal (Command Prompt, PowerShell, or terminal emulator in Linux/macOS) and execute the following command:

```
pip install SQLAlchemy
```

This command downloads **SQLAlchemy** from the **Python Package Index (PyPI)** and installs it in your Python environment. After the installation is complete, you can start using **SQLAlchemy** in your Python projects by importing it:

```
1 from sqlalchemy import create_engine, Column, Integer, String, MetaData, Table
2 from sqlalchemy.orm import sessionmaker
```

This basic example shows how to import some of the fundamental components of **SQLAlchemy**. Depending on your specific use case (Core or ORM), you might need to import additional modules or classes.

To verify that **SQLAlchemy** has been installed correctly, you can check its version:

```
import sqlalchemy
print(sqlalchemy.__version__)
```

This command prints the version of **SQLAlchemy** installed in your environment, confirming that the installation was successful.

5.3 Modifications to RelationsManager

The **RelationsManager** class underwent significant modifications to support database integration, allowing for dynamic data management through **SQLAlchemy** sessions instead of relying solely on static, in-memory data structures. This section details these changes and their implications for the system.

5.3.1 Adjusting to Use SQLAlchemy Session

The primary adjustment made to the RelationsManager class involved incorporating an **SQLAlchemy** Session object into its initialization. This session object enables the **RelationsManager** to interact with the database, executing queries and transactions directly on the database rather than manipulating in-memory Python lists and dictionaries.

Initializing RelationsManager with a Session

```
class RelationsManager:
    def __init__(self, session):
        self.session = session

```

init is a special method in Python classes. It's called automatically when a new instance of the class is created. This method is commonly used for initializing the new object's state, setting up its attributes, or performing any setup necessary when an object is created.

session is a parameter passed to the **init** method. In this context, **session** is expected to be an instance of **SQLAlchemy** Session. This object is central to **SQLAlchemy's ORM** (Object-Relational Mapping) functionality, enabling database sessions through which you can query and manipulate the database.

5.3.2 Functinoality

These two functions are part of the **RelationsManager** class, which interacts with a database to manage relationships within an organization, such as teams and their leaders. These methods assume the use of **SQLAlchemy**, a popular Object-Relational Mapping (ORM) library for Python, to interact with the database.

```
 def is_leader(self, employee: Employees) -> bool:
        # Query the database to check if an employee is a leader
        return self.session.query(Teams).filter(Teams.leaderId == employee.employeeId).
        ↪  count() > 0
```

self.session.query(Teams): Begins a query on the Teams table in the database. **.filter(Teams.leaderId == employee.employeeId):** Filters the Teams records to find rows where the **leaderId** column matches the **employeeId** of the given employee. **.count() > 0:** Counts the number of records that match the filter. If the count is greater than 0, it implies the given employee is a leader of at least one team, and **True** is returned. Otherwise, **False** is returned.

```
def get_team_members(self, employee: Employees) -> list:
        # Query the database to get team members if an employee is a leader
        if self.is_leader(employee):
            team = self.session.query(Teams).filter(Teams.leaderId == employee.
            ↪  employeeId).first()
            return [member.employeeId for member in team.members] if team else []
        return []
```

First, it checks if the given employee is a leader by calling **self.is_leader(employee)**. If the employee is a leader, it proceeds to query the Teams table for the team led by this employee, using **.filter(Teams.leaderId == employee.employeeId.first())** to find the first (and presumably only) team record where the employee is listed as the leader. Once the team is found, it constructs a list of **employeeIds** for each member in that team by iterating over team.members (assuming team.members is a relationship defined in the **ORM** model that links a team to its members). If no team is found, or the team has no members, an empty list is returned. If the initial check finds that the employee is not a leader, it immediately returns an empty list.

Together, these methods provide essential functionality for managing team leadership relationships within an organization, utilizing the database to store and query this information efficiently.

5.4 Modifications to EmployeeManager

This Python class, **EmployeeManager**, is designed to manage employee-related operations, including calculating salaries and generating salary notification messages. It utilizes an instance of RelationsManager and interacts with an Employees ORM model which represents an employee in a database.

Initializing EmployeeMananger

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

Initializes a new instance of **EmployeeManager**. Requires a instance as a parameter, which it assigns to the instance variable **self.relations_manager**. This **RelationsManager** is expected to provide functionality to determine team leadership and membership.

5.4.1 Functionality

```
1     def calculate_salary(self, employee: Employees) -> int:
2
3         salary = employee.baseSalary
4
5         # Calculate years at company using the correct attribute names
6         years_at_company = datetime.date.today().year - employee.hireDate.year
7
8         salary += years_at_company * EmployeeManager.yearly_bonus
9
10        # Check if the employee is a leader using the corrected RelationsManager method
11        if self.relations_manager.is_leader(employee):
12            # Get the list of team member IDs and calculate the leader bonus
```

```
            team_members_count = len(self.relations_manager.get_team_members(employee))
            salary += team_members_count * EmployeeManager.leader_bonus_per_member

        return salary
```

Calculates the total salary for a given employee. The salary calculation takes into account the base salary, years at the company, and additional bonuses if the employee is a team leader. The employee parameter is expected to be an instance of the Employees **ORM** model. The method returns an integer representing the total calculated salary.

```
    def calculate_salary_and_send_email(self, employee: Employees) -> str:
        salary = self.calculate_salary(employee)
        # Formulate the message using the correct attribute names
        message = f"{employee.firstName} {employee.lastName}, your salary: {salary}
        ↪   has been transferred to you."

        return message
```

Generates a salary notification message for the given employee. First calculates the employee's salary using **calculate_salary**. Then, it formulates a message string stating that the salary has been transferred to the employee, including the employee's name and the calculated salary. The method returns this message string, which could be used to notify the employee, via email or another messaging service.

5.5 Database Schema Description

A basic structure for interacting with a database using **SQLAlchemy**. Representing the definition of database models (Employees and Teams) and a singleton class (DB) for database connection management.

5.5.1 SQLAlchemy Imports

```
from sqlalchemy import create_engine, Column, Integer, String, Date, ForeignKey
from sqlalchemy.orm import declarative_base, relationship, Session
```

Imports necessary components from **SQLAlchemy**, including **create_engine** for database connectivity, Column, Integer, String, Date, ForeignKey for defining table columns, **declarative_base** for model base class, relationship for defining relationships between models, and Session for handling database sessions.

5.5.2 Base Model Declaration

```
Base = declarative_base()
```

Base = declarative_base(): Creates a base class for declarative class definitions. All entity classes should inherit from this base.

5.5.3 Employees Model

```
class Employees(Base):
    __tablename__ = "employees"

    employeeId = Column(Integer, primary_key=True)
    firstName = Column(String)
    lastName = Column(String)
    birthDate = Column(Date)
    baseSalary = Column(Integer)
    hireDate = Column(Date)
```

```
10          team_id = Column(Integer, ForeignKey("teams.teamId"), nullable=True)
11          team = relationship("Teams", back_populates="members")
```

A class **Employees** is defined, inheriting from Base. It represents a table in the database with the name "employees". The class defines several columns **(employeeId, firstName, lastName, birthDate, baseSalary, hireDate)** that correspond to the columns in the database table. **employeeId** is the **primary key**. There's a **team_id** column defined with a foreign key relationship to the **"teams.teamId"** column, implying that each employee belongs to a team. A relationship to the Teams class is defined, indicating an ORM-level relationship between Employees and Teams, allowing for easy access to an employee's team.

5.5.4 Teams Model

```
1   class Teams(Base):
2       __tablename__ = "teams"
3
4       teamId = Column(Integer, primary_key=True)
5       leaderId = Column(Integer)
6       members = relationship("Employees", back_populates="team", cascade=
        ↪  "all, delete-orphan")
```

Similarly, a **Teams** class is defined for the "teams" table, with a **primary key teamId** and a **leaderId** column to store the ID of the team leader. A **relationship** to the Employees class is defined to indicate the team members, with **back_populates** used to form a **bi-directional** relationship. This allows for accessing the team from an employee instance and vice versa. The cascade="all, delete-orphan" option ensures that deletions are cascaded and orphaned employees (without a team) are automatically deleted.

5.5.5 DB Singleton Class

```
class DB:
    __instance = None

    def __init__(self):
        if DB.__instance is not None:
            raise Exception("This class is Singleton!")
        else:
            DB.__instance = self

        self.engine = None
        self.connection = None
        self.metadata = None
        self.DB_NAME = "employees.db"

    def get_instance(self):
        return self.__instance

    def connect_database(self):
        self.engine = create_engine(f"sqlite:///{self.DB_NAME}", echo=True)
        self.metadata = Base.metadata
        self.metadata.bind = self.engine
        self.connection = self.engine.connect()

    def init_database(self):
        Base.metadata.create_all(self.engine)
```

A **DB** class is defined to manage the database connection. It's designed as a singleton class, meaning **only one** instance of this class can exist in the application at any time. The **__init__** method checks if an instance already exists and raises an exception if an attempt is made to create another instance. Otherwise, it initializes class attributes for managing the database connection. The **get_instance** method returns the singleton instance of the class. **connect_database** initializes the connection to the database using **create_engine**, with

the **DB_NAME** attribute specifying the database file ("employees.db"). The connection and metadata are stored as instance attributes. **init_database** creates all tables defined by the base class in the database, effectively initializing the database schema.

5.5.6 How It Works Together

Models **Employees** and **Teams** represent the structure of the database tables and their relationships. These models can be used to perform ORM operations, like querying the database or inserting new records, in an object-oriented manner. The DB class manages the database connection and ensures that only one connection is active throughout the application lifecycle. It also provides a method to initialize the database schema based on the models.

5.6 Migrating Local Data to Database

```python
import datetime
from sqlalchemy.orm import Session
from database import DB, Employees, Teams
from relations_manager import RelationsManager

def main():
    db = DB()
    db.connect_database()
    db.init_database()

    with Session(db.engine) as session:
        manager = RelationsManager()

        for employee_data in manager.employee_list:
            employee = Employees(
                employeeId=employee_data.id,
                firstName=employee_data.first_name,
                lastName=employee_data.last_name,
                birthDate=employee_data.birth_date,
                baseSalary=employee_data.base_salary,
                hireDate=employee_data.hire_date,
            )
            session.add(employee)

        for team_id, member_ids in manager.teams.items():
            team = Teams(leaderId=team_id, members=[])
            for member_id in member_ids:
                employee = session.query(Employees).get(member_id)
                team.members.append(employee)

            session.add(team)

        session.commit()

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

This method illustrates how existing in-memory data (e.g., employees and their team associations) can be transferred into the database, ensuring a smooth transition and minimal disruption to the application's functionality.

5.7 Test Scenarios

This part defines a set of unit tests for an Employee Management System that utilizes **SQLAlchemy** for database operations. It features two primary components: **TestEmployeeRelationsManager** and **TestEmployeeManager**, each with tests that verify specific functionalities related to managing employees and their relationships within an organization.

5.7.1 Overall Structure

```
1    import unittest
```

The code uses **unittest**, a built-in Python module for writing and running tests. It defines two test classes that inherit from **unittest.TestCase**, allowing the use of a variety of assertions to test the application's logic.

5.7.2 Database Setup with SQLAlchemy

create_engine('sqlite:///WithDB/employees.db'): This creates a connection to a SQLite database named employees.db located in the **WithDB** directory. SQLite is chosen here for simplicity and ease of setup. **Session = sessionmaker(bind=cls.engine):** This creates a **sessionmaker**, which is a factory for producing session objects that are bound to the database engine. Sessions are used to manage transactions.

5.7.3 Class TestEmployeeRelationsManager

This class tests functionalities related to the **RelationsManager**, which handles the logic for determining employee roles (like team leadership) and relationships (such as team membership).

setUpClass and tearDownClass Methods

```
    @classmethod
    def setUpClass(cls):
        cls.engine = create_engine('sqlite:///WithDB/employees.db')
        Session = sessionmaker(bind=cls.engine)
        cls.session = Session()

    @classmethod
    def tearDownClass(cls):
        cls.session.close()
```

These class-level methods run once before any tests and once after all tests in the class have been run, respectively. They are used here to set up a database connection before tests and close it afterward, ensuring that all tests operate on a consistent and isolated database environment.

setUp Method

```
    def setUp(self):
        self.rm = RelationsManager(self.session)
```

Prepares the test environment before each individual test method runs. It instantiates a **RelationsManager** object with the current database session, ensuring that each test has a fresh instance.

Test Methods

```
    def test_team_leader_john_doe(self):
        john_doe = self.session.query(Employees).filter_by(firstName="John", lastName=
        ↪  "Doe").first()
        self.assertIsNotNone(john_doe, "John Doe does not exist in the database")
        self.assertTrue(self.rm.is_leader(john_doe),
        ↪  "John Doe is not marked as a leader but should be")
```

test_team_leader_john_doe: Checks if John Doe is correctly identified as a team leader.

```
    def test_team_members_john_doe(self):
        john_doe = self.session.query(Employees).filter_by(firstName="John", lastName=
        ↪  "Doe").first()
        expected_team_members = self.rm.get_team_members(john_doe)
        self.assertIsNotNone(expected_team_members, "John Doe should have team members"
        ↪  )
        # Use the IDs directly since get_team_members returns a list of IDs
        self.assertCountEqual(expected_team_members, [2, 3])
```

test_team_members_john_doe: Verifies that the correct team members are associated with John Doe.

```
    def test_tomas_andre_not_team_member_john_doe(self):
        john_doe = self.session.query(Employees).filter_by(firstName="John", lastName=
        ↪  "Doe").first()
        tomas_andre = self.session.query(Employees).filter_by(firstName="Tomas",
        ↪  lastName="Andre").first()
        team_member_ids = self.rm.get_team_members(john_doe)  # Expecting list of IDs
        self.assertNotIn(tomas_andre.employeeId, team_member_ids)
```

test_tomas_andre_not_team_member_john_doe: Ensures Tomas Andre is

not mistakenly identified as a member of John Doe's team.

```
1    def test_gretchen_watford_salary(self):
2        gretchen_watford = self.session.query(Employees).filter_by(firstName="Gretchen"
         ↪  , lastName="Watford").first()
3        self.assertIsNotNone(gretchen_watford,
         ↪  "Gretchen Watford does not exist in the database")
4        self.assertEqual(gretchen_watford.baseSalary, 4000,
         ↪  "Gretchen Watford's salary does not match the expected")
```

test_gretchen_watford_salary: Confirms Gretchen Watford's salary matches the expected amount in the database.

```
1    def test_tomas_andre_not_team_leader(self):
2        tomas_andre = self.session.query(Employees).filter_by(firstName="Tomas",
         ↪  lastName="Andre").first()
3        self.assertIsNotNone(tomas_andre, "Tomas Andre does not exist in the database")
4        self.assertFalse(self.rm.is_leader(tomas_andre),
         ↪  "Tomas Andre is incorrectly marked as a leader")
```

test_tomas_andre_not_team_leader: Validates that Tomas Andre is not incorrectly marked as a team leader.

5.7.4 Class TestEmployeeManager

This class tests functionalities related to the **EmployeeManager**, which is responsible for calculating salaries and generating salary notification messages.

setUp Method

```
    def setUp(self):
        self.rm = RelationsManager(self.session)
        self.em = EmployeeManager(self.rm)
```

Similar to **TestEmployeeRelationsManager**, it sets up the environment for each test, creating **RelationsManager** and **EmployeeManager** instances.

Test Methods

```
    def test_non_leader_salary(self):
        # Example for an existing employee
        employee = self.session.query(Employees).filter_by(firstName="Tomas", lastName=
        ↪  "Andre").first()
        self.assertIsNotNone(employee, "Employee does not exist in the database")
        calculated_salary = self.em.calculate_salary(employee)
        # Use the correct expected salary as per your business logic
        expected_salary = 3000  # Replace with the correct expected value
        self.assertNotEqual(calculated_salary, expected_salary)
```

test_non_leader_salary: Tests the salary calculation for a non-leader employee to ensure it does not mistakenly equal the expected salary for leaders.

```
    def test_team_leader_salary(self):
        # Example for an existing employee
        team_leader = self.session.query(Employees).filter_by(firstName="Gretchen",
        ↪  lastName="Watford").first()
        self.assertIsNotNone(team_leader, "Team Leader does not exist in the database")
        calculated_salary = self.em.calculate_salary(team_leader)
        expected_salary = 7800  # Replace with the correct expected value
        self.assertEqual(calculated_salary, expected_salary)
```

test_team_leader_salary: Verifies that a team leader's salary is correctly calculated, considering both the base salary and bonuses.

```
    def test_salary_calculation_and_email_notification(self):
        employee = self.session.query(Employees).filter_by(firstName="John", lastName=
        ↪  "Doe").first()
        self.assertIsNotNone(employee,
        ↪  "Employee John Doe does not exist in the database")
        expected_message = f"{employee.firstName} {employee.lastName}, your salary: {
        ↪  self.em.calculate_salary(employee)} has been transferred to you."
        actual_message = self.em.calculate_salary_and_send_email(employee)
        self.assertEqual(actual_message, expected_message)
```

test_salary_calculation_and_email_notification: Checks the correctness of the generated email message about salary disbursement for an employee.

5.7.5 Key Points

- **Integration with SQLAlchemy:** The tests demonstrate how to integrate SQLAlchemy with unit testing, showing interactions with a real database.
- **Session Management:** It emphasizes the importance of managing database sessions, including setup and teardown to ensure clean testing environments.

- **Testing Strategies:** The examples highlight how to write tests for both data integrity (e.g., correct salaries and roles) and application logic (e.g., email notification content).

This suite of tests ensures that both the relationship management and salary calculations in the Employee Management System work as expected against a database, illustrating a comprehensive approach to testing database-driven applications.

6 Conclusion

In this documentation, we have detailed the architecture and testing framework of the Employee Relations Management system, with a specific focus on the integration of database functionalities. This integration represents a pivotal enhancement, transitioning from an in-memory model to a robust, persistent storage solution using **SQLAlchemy**. The unit tests have been designed to validate the system's functionalities, encompassing both the original logic and the new database interactions.

The test scenarios examined a broad spectrum of functionalities, including team leadership identification, salary computation, and the generation of email notifications, all within the context of a database-driven environment. For each functionality, tests were crafted to not only verify the correctness of the system's operations but also to ensure the integrity and persistence of data within the database. This approach highlighted the system's adaptability and reliability in managing data relationships and transactions effectively.

In conclusion, the evolution of the Employee Relations Management system, underscored by the shift to a database-centric architecture, significantly enhances its functionality and scalability. The detailed unit tests underscore our commitment to maintaining a high standard of quality and reliability, ensuring that the system not only meets but exceeds the operational requirements in managing employee relations and associated data.