The University of Azad Jammu and Kashmir

Lab Task # 08

Course Instructor: Engr. Sidra Rafique

Semester: Fall-2024

Session: 2022-2026

Submission Date: Nov 11,2024

Submitted By: Syeda Urwa Ajmal

Roll No: 2022-SE-16

Course Name: SC&D

Code: SE-3102

Repository Pattern

Contents

The Repository Pattern in Python

Introduction

The Repository Pattern is a design pattern used to abstract data access logic, providing a separation between the business logic and the data storage layer. It acts as a mediator between the domain and data mapping layers, ensuring that the domain logic remains independent of how data is persisted or retrieved. This pattern promotes clean

architecture, testability, and maintainability by encapsulating the data access code and presenting a simplified interface to the business logic.

Key Features of the Repository Pattern

• Abstraction: Hides the implementation details of data storage.

• Decoupling: Separates data access logic from business logic.

• Consistency: Provides a consistent API for data operations regardless of the underlying data source.

• Testability: Simplifies unit testing by allowing the repository to be mocked.

Code Explanation

1. Define a Model

The model represents the data structure. In this example, the User class defines the attributes of a user (user_id, name, and email):

```
class User:
    def __init__(self, user_id, name, email):
        self.user_id = user_id
        self.name = name
        self.email = email

    def __repr__(self):
        return f"User({self.user_id}, {self.name}, {self.email})"
```

```python
#Define a Model:
class User:
    def __init__(self, user_id, name, email):
        self.user_id = user_id
        self.name = name
        self.email = email

    def __repr__(self):
        return f"User({self.user_id}, {self.name}, {self.email})"
```

2. Repository Interface

The interface defines the operations supported by the repository. It ensures consistency and defines the methods to be implemented by any repository class:

```
from abc import ABC, abstractmethod

class UserRepository (ABC):
    @abstractmethod
    def add_user (self, user: User):
        pass

    @abstractmethod
    def get_user_by_id (self, user_id: int) -> User:
        pass

    @abstractmethod
    def list_users(self) -> list [User]:
        pass

    @abstractmethod
    def remove_user (self, user_id: int):
        pass
```

```python
#Repository Interface:
from abc import ABC, abstractmethod

class UserRepository(ABC):
    @abstractmethod
    def add_user(self, user: User):
        pass

    @abstractmethod
    def get_user_by_id(self, user_id: int) -> User:
        pass

    @abstractmethod
    def list_users(self) -> list[User]:
        pass

    @abstractmethod
    def remove_user(self, user_id: int):
        pass
```

3. Repository Implementation

The repository implementation contains the logic for interacting with the data storage. In this example, an in-memory implementation is provided for simplicity:

```
class InMemoryUserRepository(UserRepository):
    def __init__(self):
        self.users = {}

    def add_user(self, user: User):
        self.users[user.user_id] = user

    def get_user_by_id(self, user_id: int) -> User:
        return self.users.get(user_id)

    def list_users(self) -> list[User]:
        return list(self.users.values())

    def remove_user(self, user_id: int):
        self.users.pop(user_id, None)
```

```python
#Repository Implementation (In-Memory):
class InMemoryUserRepository(UserRepository):
    def __init__(self):
        self.users = {}

    def add_user(self, user: User):
        self.users[user.user_id] = user

    def get_user_by_id(self, user_id: int) -> User:
        return self.users.get(user_id)

    def list_users(self) -> list[User]:
        return list(self.users.values())

    def remove_user(self, user_id: int):
        self.users.pop(user_id, None)
```

4. Usage Example

Here's how the repository can be used to manage user data:

```
# Instantiate the repository
repo = InMemoryUserRepository()

# Add users
repo.add_user(User(1, "Alice", "alice@example.com"))
repo.add_user(User(2, "Bob", "bob@example.com"))

# Fetch and list users
print(repo.get_user_by_id(1))  # Output: User(1, Alice, alice@example.com)
print(repo.list_users())       # Output: [User(1, Alice, alice@example.com), User(2, Bob,
bob@example.com)]

# Remove a user
repo.remove_user(1)
print(repo.list_users())       # Output: [User(2, Bob, bob@example.com)]
```

```python
#Repository Implementation (In-Memory):
class InMemoryUserRepository(UserRepository):
    def __init__(self):
        self.users = {}

    def add_user(self, user: User):
        self.users[user.user_id] = user

    def get_user_by_id(self, user_id: int) -> User:
        return self.users.get(user_id)

    def list_users(self) -> list[User]:
        return list(self.users.values())

    def remove_user(self, user_id: int):
        self.users.pop(user_id, None)
```

Output of code:

```
User(1, Alice, alice@example.com)
[User(1, Alice, alice@example.com), User(2, Bob, bob@example.com)]
[User(2, Bob, bob@example.com)]
```

Advantages

• Flexibility: Easily switch the data source (e.g., from an in-memory list to a database) without altering business logic.

• Readability: Clean and organized data access code.

• Testability: Repositories can be mocked for unit tests.

Design Principles of the Repository Pattern

• Abstraction: Encapsulates the data access logic.

• Separation of Concerns: Decouples business logic from data storage.

• Consistency: Ensures a uniform API for interacting with data sources.

• Flexibility: Enables switching data storage mechanisms seamlessly.