ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

EXERCISE 06

DATA ACCESS OBJECTS (DAO)

AND REPOSITORY PATTERN

SOFTWARE ANALYSIS AND DESIGN

21120280 – Lý Minh Khuê

Thành phố Hồ Chí Minh – 2024

TABLE OF CONTENTS

CONCEPT OF DAO

Data access object (DAO)- one of the Structural Design Pattern is introduced to help with the **isolation** the **business layer** from the **persistence layer(repository layer)** using an abstract API.
The main idea behind this is that instead of interacting directly with the persistence layer, we communicate through a intermidiary layer called **DAO** that perform all the **CRUD** operations. This allows both the business layer and persistence layer to evolve seperately without any restriction of knowing each other.
With **DAO**, we have our components:

- **Data Access Object Interface** - This interface defines the standard operations to be performed on data source for a model object(s).
- **Data Access Object concrete class** - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
- **Model Object or Value Object** - This object is simple POJO containing get/set methods to store data retrieved using DAO class.
- **DataSource** – a place where data is stored.

For example:

```java
public class User {
    private int id;
    private String username;
    private String email;

    public User(int id, String username, String email) {
        this.id = id;
        this.username = username;
        this.email = email;
    }

    public void setId(int i) { this.id = i; }

    public int getId() { return this.id; }

    public void setUsername(String u) { this.username = u; }

    public String getUsername() { return this.username; }

    public void setEmail(String e) { this.email = e; }

    public String getEmail() { return this.email; }


    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", email='" + email + '\'' +
                '}';
    }
}
```

Figure 1. User class

This class define Entity **'User'**.

The overrided **toString()** function is for details of the **User** object when using **System.out.println()**

```java
1 implementation
public interface UserDAO {
    1 implementation
    List<User> getAllUsers();
    1 implementation
    User getUserById(int id);
    1 implementation
    void addUser(User user);
    1 implementation
    void updateUser(User user);
    1 implementation
    void deleteUser(int id);
}
```

Figure 2. UserDAO interface

Define the interface for the **DAO**.

```java
public class UserDAOImpl implements UserDAO {
    private final Map<Integer, User> users = new HashMap<>();
    private int nextId = 1;

    @Override
    public List<User> getAllUsers() { return new ArrayList<>(users.values()); }

    @Override
    public User getUserById(int id) { return users.get(id); }

    @Override
    public void addUser(User user) {
        user.setId(nextId++);
        users.put(user.getId(), user);
    }

    @Override
    public void updateUser(User user) { users.put(user.getId(), user); }

    @Override
    public void deleteUser(int id) { users.remove(id); }
}
```

Figure 3. UserDAOImpl class

This is the DAO concrete class that implement every abstract method in DAO interface. This allows you to interact with the datasource.

The datasource in this example is a HashMap<Integer, User> that is stored in an in-memory storage, in which the unique user's IDs are used as keys to store the object User

```java
public class Main {
    public static void main(String[] args) {
        UserDAO userDAO = new UserDAOImpl();

        // Adding users
        User user1 = new User( id: 1, username: "abcd1234", email: "abcd1234@gmail.com");
        userDAO.addUser(user1);

        User user2 = new User( id: 2, username: "xyz", email: "xyz@gmail.com");
        userDAO.addUser(user2);

        // Getting all users
        System.out.println("All users:");
        userDAO.getAllUsers().forEach(System.out::println);

        // Updating user
        user2.setEmail("updated_email@gmail.com");
        userDAO.updateUser(user2);

        // Getting user by ID
        System.out.println("\nUser with ID 2:");
        System.out.println(userDAO.getUserById(2));

        // Deleting user
        userDAO.deleteUser( id: 2);
        System.out.println("\nAfter deletion:");
        userDAO.getAllUsers().forEach(System.out::println);
    }
}
```

Figure 4. DAO Main class

```
All users:
User{id=1, username='abcd1234', email='abcd1234@gmail.com'}
User{id=2, username='xyz', email='xyz@gmail.com'}

User with ID 2:
User{id=2, username='xyz', email='updated_email@gmail.com'}

After deletion:
User{id=1, username='abcd1234', email='abcd1234@gmail.com'}
```

Figure 5. DAO Main result

This example shows how the pattern uses a DAO interface to perform CRUB operations on objects, which is motivation of the DAO design pattern.

ADVANTAGES OF THE DAO PATTERN

1. **Abstraction and Encapsulation:**
   The persistence logic is abstracted to a separate layer known as the Data Access Layer which enables the application to adapt safely in Persistence mechanism.
2. **Loosely-coupling:**
   DAO design pattern also minimizes coupling between components of an application. By using this design pattern, your View Layer is entirely independent of DAO layer, with only the Service layer relying on it which is also abstracted through the use of DAO interface.
3. **Flexibility and Adaptability:**
   With its reliance on interfaces, it also comes well with the object-oriented design principle "programming for interface than implementation" which results in flexible and quality code.
4. **Enhanced Testability:**
   DAO design pattern allows JUnit test to run faster as it allows to create Mock and can be performed without the need of connecting to a database. It improves testing because it simplifies testing procedures as the issues can be addressed by examining the code rather than having to troubleshoot database connectivity.

REPOSITORY PATTERN

"repository is a mechanism for encapsulating storage, retrieval, and search behavior, which emulates a collection of objects."

-Eric Evans' book *Domain-Driven Design*

Repository pattern is also one of the design patterns that provides a way to manage data access logic in a centralized location. It acts as a intermediary layer between business logic layer and data access layer. It separates the logic that retrieves the data and maps it to the entity model from the business logic that operates on the model.

```
1 implementation
public interface UserRepository  {
    1 implementation
    User findById(int id);
    1 implementation
    List<User> findAll();
    1 implementation
    void save(User user);
    1 implementation  Rename usages
    void delete(int id);
}
```

Figure 6. UserRepository Interface

```
public class UserRepositoryImpl implements UserRepository {
    private final Map<Integer, User> users = new HashMap<>();
    private int nextId = 1;

    @Override
    public List<User> findAll() {
        return new ArrayList<>(users.values());
    }

    @Override
    public User findById(int id) {
        return users.get(id);
    }

    @Override
    public void save(User user) {
        if (user.getId() == 0) { // Assuming id of 0 indicates a new user
            user.setId(nextId++);
        }
        users.put(user.getId(), user);
    }

    @Override
    public void delete(int id) { users.remove(id); }
}
```

Figure 7. UserRepositoryImpl Class

```java
public class Main {
    public static void main(String[] args) {
        UserRepository userRepository = new UserRepositoryImpl();

        // Adding users
        User user1 = new User( id: 1, username: "abcd1234", email: "abcd1234@gmail.com");
        userRepository.save(user1);

        User user2 = new User( id: 2, username: "xyz", email: "xyz@gmail.com");
        userRepository.save(user2);

        // Getting all users
        System.out.println("All users:");
        userRepository.findAll().forEach(System.out::println);


        // Getting user by ID
        System.out.println("\nUser with ID 2:");
        System.out.println(userRepository.findById(2));

        // Deleting user
        userRepository.delete( id: 2);
        System.out.println("\nAfter deletion:");
        userRepository.findAll().forEach(System.out::println);
    }
}
```

Figure 8. Repository Main class

```
All users:
User{id=1, username='abcd1234', email='abcd1234@gmail.com'}
User{id=2, username='xyz', email='xyz@gmail.com'}

User with ID 2:
User{id=2, username='xyz', email='xyz@gmail.com'}

After deletion:
User{id=1, username='abcd1234', email='abcd1234@gmail.com'}
```

Figure 9. Repository Main result

REPOSITORY PATTERN AND DAO PATTERN

So with all these definitions, is it the same as **DAO** pattern as mentioned earlier? Well, basically, we can see both of them as the same. But Repository is the highest abstraction, offering a collection interface to retrieving entities, which means a DAO can be a repository. This abstraction becomes a necessity when you want to protect your domain code from underlying database technology.

A **DAO(**in other words – **object used to access data)** allows for a simpler way to get data from storage, hiding the queries.
Repository deals with data too and hides queries and all that but, a repository deals with **business/domain models**.
A repository will use a DAO to get the data from the storage and uses that data to restore a **business model**.
Both patterns mean the same (they store data and they abstract the access to it and they are both expressed closer to the domain model and hardly contain any DB reference), but the way they are used can be slightly different, DAO being a bit more flexible/generic, while Repository is a bit more specific and restrictive to a type only.
Also, a Repository is generally a narrower interface. It should be simply a collection of objects, with a **Get(id)**, **Find(ISpecification)**, **Add(Entity)**.
A method like **Update** is appropriate on a DAO, but not a Repository - when using a Repository, changes to entities would usually be tracked by separate UnitOfWork.

REFERENCES

- DAO

https://gpcoder.com/4935-huong-dan-java-design-pattern-dao/
Design Patterns: Data Access Object (oracle.com)

- Repository

https://stackoverflow.com/questions/8550124/what-is-the-difference-between-dao-and-repository-patterns
https://www.linkedin.com/pulse/what-repository-pattern-alper-sara%C3%A7/
https://deviq.com/design-patterns/repository-pattern
https://www.baeldung.com/java-dao-vs-repository