DAO Class in Java

Data Access Object patterns, often known as **DAO** patterns, are used to divide high level business services from low level data accessing APIs or actions. The members of the Data Access Object Pattern are listed below.

Data Access Object Interface: The Data Access Object Interface specifies the common operations to be carried out on a model object (s).

Concrete Data Access Object class: This class implements the aforementioned interface. This class is in charge of obtaining data from a data source, which could be a database, XML, or another type of storage system.

Model or Value Object: This object is a straightforward POJO with get/set methods for storing data obtained using the DAO class.

Implementation

A student object will be created and used as a model as well as a value object.

Data Access Object Interface is called StudentDao.

The concrete class StudentDaoImpl implements the Data Access Object Interface. StudentDao will be used by DaoPatternDemo, our demo class, to show how to use the Data Access Object pattern.

Given Problem

Assuming that we have a web application project that utilize MySQL database to follow the requirements from customer. So, we will use driver of MySQL to interact with database. But in other beautiful day, the customer want to use additional database such as PostgreSQL, then, we have to modify our code to compatible with this database. It makes our layers that has tightly coupling with persistence layer when we change to other database.

Therefore, what is solution to prevent the tightly coupling of other layers with persistence layer?

Solution with DAO pattern

The DAO pattern is a structural pattern that allow us to isolate the application/business layer from the persistence layer (usually a relational database, but it could be any other persistence mechanism) using an abstract API.

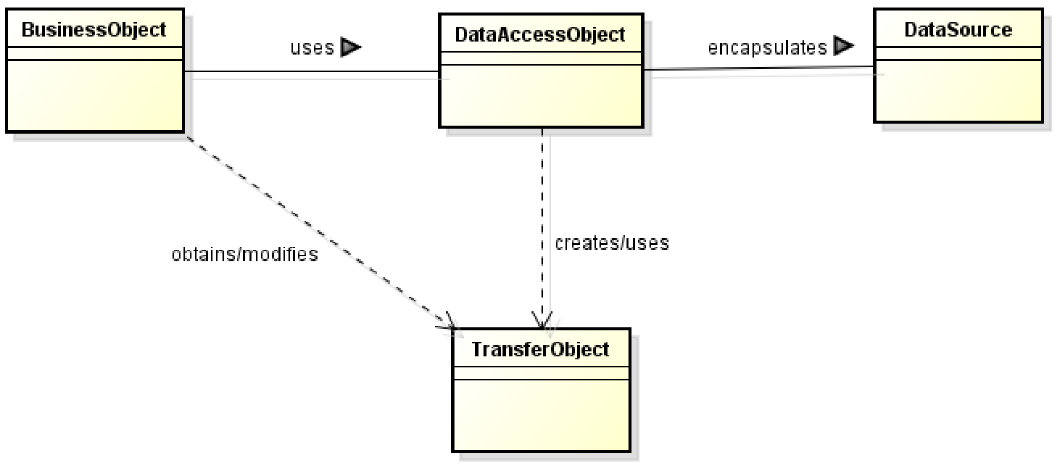
DAO pattern is based on design principles such as **abstraction** and **encapsulation**. It will protect the rest of application from many operations in persistence layer. For example, change database from MySQL to PostgreSQL, change storage by file to database.

The DAO pattern is commonly used pattern to persist domain objects into a database. The most common form of a DAO pattern is a class that contains CRUD methods for a particular domain entity type.

So, DAO class is an intermediate layer to help other layers to communicate with persistence layers regardless of any storage mechanism such as file, database management, …, then, performs some operations such as CRUD.

For example:

DAO pattern can be represented with many ways such as Java Persistence API (JPA), Enterprise Java Bean (EJB), Object-Relational Mapping (ORM) with many specific implementations such as Hibernate, iBATIS, Spring JPA, …



JAVA PROGRAM

Now, we will use Java language to describe the DAO pattern. And based on these code, we will analyze the advantages and disadvantages of DAO pattern.

When to use

- When we need to change the current database to the other database such as Oracle, MySQL, MariaDB, SQL Server, MongoDB.
- We want to separate a data source’s client interface from its data access mechanism.
- We want to adapt a specific data resource’s access API to a generic client interface.
- In a larger project, different teams work on different parts of the application: the DAO pattern allows clean separation of concerns.

Advantages and Disadvantages

1. Advantages

    - It separeates the domain logic that use it from any particular persistence mechanism or APIs –> loose coupling between layers.

    - The interface methods signature are independent of the content of the Domain class. When we add some fields to the Domain class, we do not need to change the DAO interface nor its caller.

    - Testing our service that calls the DAO: We can write a mock DAO that behaves just as we need it in the test (Ex: simulate that there is no DB connection, which is hard to reproduce automatically).

    - Generate some layers around our DAO: We could use Aspect Oriented Programming - AOP to generate caching or transaction handling around our DAO methods. In this case, we have an object that implements the DAOs interface but has nothing to do with the original implementation.

- Switching the DB technology: if we switch from MySQL to DB2, we just need to write another implementation of the interface and switch the MySQL DAO and the DB2 DAO.

2. Disadvantages

    - Potential disadvantages of using DAO include leaky abstraction, code duplication, and abstraction inversion.

      A leaky abstraction is an abstraction that leaks details that it is supposed to abstract away.

    - When application requires multiple DAOs, one might find oneself repeating essentially the same create, read, update, and delete code for each DAO

      –> Solution: implementing a generic DAO that handles these common operations.

    - Any changes to the interface require edits in not just one implementation, but in multiple implementations in multiple classes. When the Open/Closed rule is violated here, it really blows up.

Wrapping up

- Some assumptions behind the DAO implementation:

    - All database access in the system is made through a DAO to achieve encapsulation.

    - Each DAO instance is responsible for one primary domain object or entity. If a domain object has an independent lifecycle, it should have its own DAO.

    - The DAO is responsible for creations, reads (by primary key), updates, and deletions – that is CRUD – on the domain object.

    - The DAO may allow queries based on criteria other than the primary key. We refer to these as *finder methods* or *finders*. The return value of a finder is normally a collection of the domain object for which the DAO is responsible.

  - The DAO is not responsible for handling transactions, sessions, or connections. These are handled outside the DAO to achieve flexibility.

- A typical DAO implementation has the following components:
  - A DAO factory class.
  - A DAO interface.
  - A concrete class that implements the DAO interface.
  - Data transfer object (sometimes called **Value Objects**)

- The concrete DAO class contains logic for accessing data from a specific data source.

- Use a DAO pattern when we need a DAO, whereas we do not need a repository. A DAO can basically be used as a messaging system, between the application and the database.

  So, if we need to generate a report, which is often a rendering of read-only data, or update any logging tables, then a DAO should suffice. No need for managing such transaction in session, just quick data dumps and updates.

  For managing actual domain entities, and not **Value Object**, use a repository.

- In the absence of an ORM frameowork, the DAO pattern handles the impedance mismatch that a relational database has with object-oriented techniques. In DDD, we inject Repositories, not DAO’s in domain entities.

- DAO’s are related to peristence, and persistence is infrastructure, not domain. The main problem is that we have lots of different concerns polluting the domain. According to DDD, an object should be distilled unitl nothing remains that does not relate to its meaning or support its role in interactions. And that’s exactly the problem the Repository pattern tries to solve.

Why Do We Need the DAO Design Pattern?

- **Structure:** The source is structured and easily navigable by centralising all database-related code in the EmployeeDao.
- **Consistency:** The utilisation of a defined interface, known as the DAO interface, guarantees a uniform method of engaging with the database, hence minimising the probability of mistakes arising from disparate techniques.
- **Easy Maintenance:** The EmployeeDaoImpl class allows for easy maintenance by facilitating changes to the database structure or technology with minimal impact on the rest of the application.
- **Facilitating Testing:** Unit testing becomes easier due to the presence of a well-defined set of methods to test within the EmployeeDao.
- **Scalability:** With the growth of the application, it is possible to incorporate more DAO interfaces and implementations, thus providing an architecture that can handle increased demands and is easy to maintain.

What Are the Key Components of the DAO Design Pattern?

The DAO pattern facilitates the attainment of separation of concerns, enhancing code organisation, maintainability, and adaptability to alterations in data storage or access logic through the utilisation of the following primary components:

1. BusinessObject

The BusinessObject serves as a representation of the client's data. The object necessitates access to the data source in order to acquire and store data. Business Objects serve as the fundamental elements in your program and are frequently employed to contain business logic. They can engage with Transfer Objects for the purpose of transferring data.

2. DataAccessObject

The DataAccessObject serves as the central component of this design pattern. The DataAccessObject serves as an abstraction layer for the BusinessObject, allowing seamless access to the data source by hiding the details of the underlying data access technology. It offers a consistent interface for executing CRUD operations on entities.

DAOs serve as a means to encapsulate the code that is special to the database. They enable the rest of the application to interact with data entities without needing to worry about the details of how the data is saved or retrieved.

3. Data source

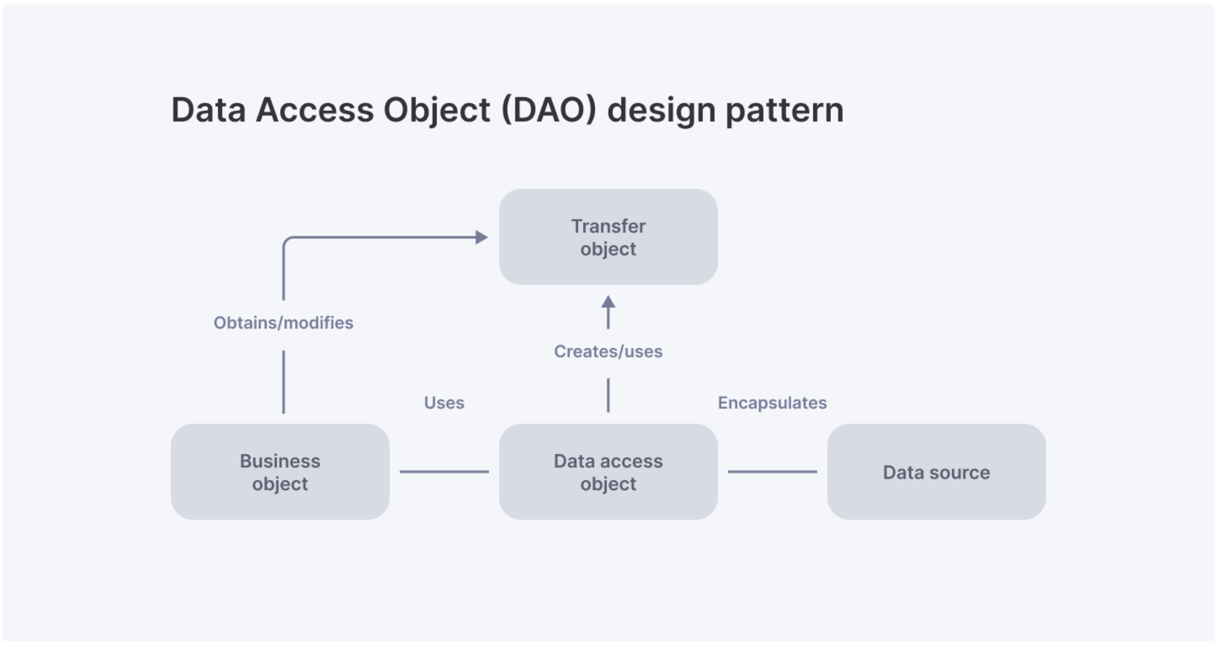
This is an implementation of a data source. The Data Source is responsible for handling the connection to the underlying database. It is utilised by the DAOs to acquire connections and perform queries. It simplifies the process by hiding specific information such as connection pooling, database URL, and passwords.

4. TransferObject

This is a Transfer Object that serves as a data carrier. It is employed for transmitting data between a client and a server, or between several levels of an application. The DataAccessObject may also accept data from the client in a Transfer Object to modify the data in the data source. Transfer Objects facilitate the reduction of method calls between the client and server by consolidating various data fields into a singular object.

How components interact with one another:

- The Business Object contains the fundamental business logic and communicates with the movement Object to facilitate the movement of data.
- The DAO interfaces with both the Business Object and the Transfer Object. The system utilises the Transfer Object to facilitate the exchange of data between the database and the application. It may also collaborate with Business Objects to transform the data from the transfer format to the internal representation used by the application.
- The DAO utilises the Data Source to acquire database connections required for conducting queries and modifications.



A Simple Implementation

Step 1:

Value Object creation.

Step 2:

Data Access Object Interface creation.

Step 3:

Construct a class that implements the aforementioned interface.

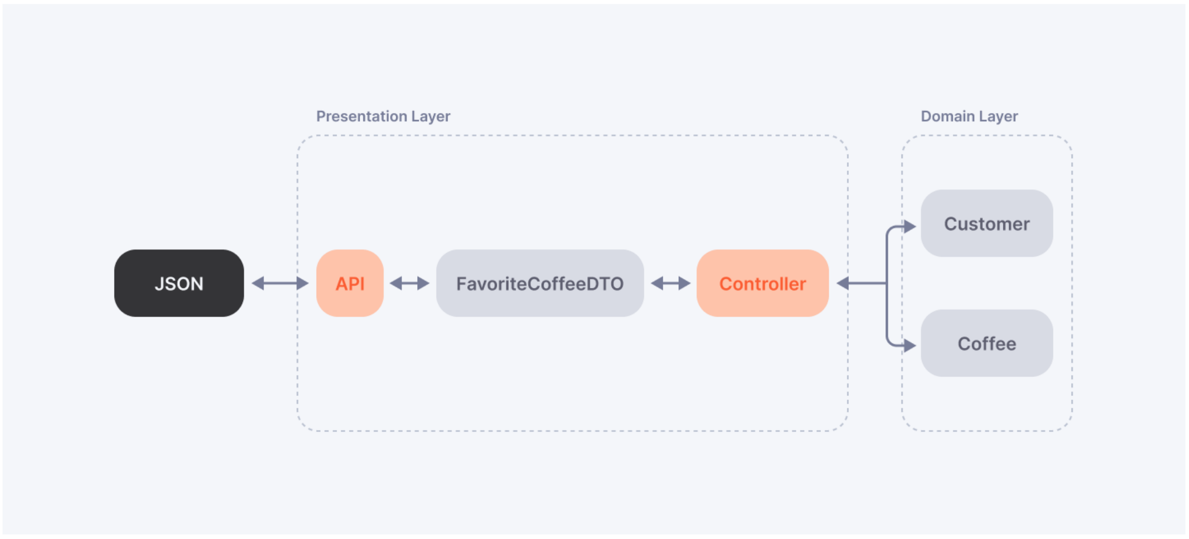
Step 4:

Utilize the EmployeeDao to illustrate how to use the Data Access Object pattern.

Implementing a DTO

It is possible to make a DTO out of a POJO or a Java Bean. The most important thing about a DTO is that it keeps things like the display layer and the domain model separate.

To show this, let's look at a small rest service. Let's say we have a coffee shop where people buy coffee and talk. They both belong to different domains in the system. If I want to find out what a customer's favourite coffee is, I'll make an API that gives me the FavoriteCoffeDTO info as a whole.



The code looks like this:

```
public class Coffee {
   private Long id;
   private String name;
   private List<String> ingredients;
   private String preparation;
}
public class Customer {
   private Long id;
   private String firstName;
   private String lastName;
   private List<Coffee> coffees;
}
public class FavoriteCoffeeDTO {
   private String name;
   private List<String> coffees;
}
```

Key Characteristics and Benefits

Data Transfer Objects (DTOs) include certain essential attributes that render them valuable for transferring data across different layers inside a Java application:

- **Immutability:** DTOs are commonly constructed with the characteristic of being immutable, which implies that their attributes are unable to be altered once they have been formed. This feature ensures that the program is thread-safe and minimises the risk of data corruption.
- **Serializability:** This refers to the ability of DTOs to be readily transformed into a format that can be stored or communicated over a network. This enables seamless transmission of data across different levels, from a service layer to a display layer.
- **Simplicity:** DTOs are intentionally devoid of any business logic in order to maintain simplicity and minimise their weight. This enables its utilisation for data transport without introducing any additional intricacy to the program.
- **Accessors and mutators:** Data Transfer Objects (DTOs) commonly consist of accessors and mutators for each property, enabling the retrieval and modification of their properties.
- **Separation of Data Handling Concerns:** DTOs normally do not include methods for storing or retrieving data from a database, therefore they are not concerned with persistence. This effectively segregates the issues related to transferring data from the issues related to persisting data.
- **Tailored Data Representation:** DTOs are specifically developed to offer an appropriate representation of data that caters to the specific requirements of each tier. For instance, the service layer may necessitate a more intricate

depiction of data compared to the presentation layer, which may just require a fraction of the data.

DTOs enhance the speed, scalability, and maintainability of Java programs by segregating data transfer problems from business logic concerns.

The Pattern

DTOs, which stand for "Data Transfer Objects," are things that move data from one process to another so that there are fewer method calls.