**DAO Design Pattern**

With DAO [design pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial), we have following components on which our design depends:

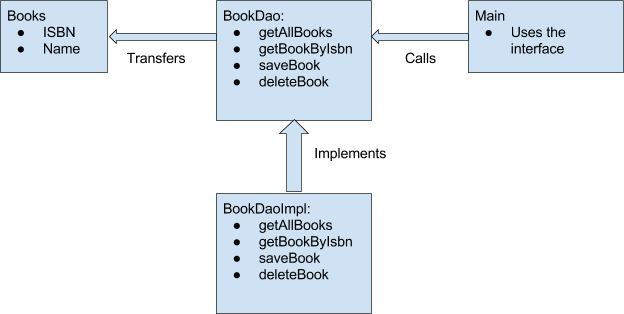
* The model which is transferred from one layer to the other.
* The [interfaces](https://www.journaldev.com/1601/interface-in-java) which provides a flexible design.
* The interface implementation which is a concrete implementation of the persistence logic.

**Implementing DAO pattern**

With above mentioned components, let’s try to implement the DAO pattern. We will use 3 components here:

1. The Book model which is transferred from one layer to the other.
2. The BookDao interface that provides a flexible design and API to implement.
3. BookDaoImpl concrete class that is an implementation of the BookDao interface.

Let us put this logic into a diagram:



**DAO Pattern model Class**

Now, let’s put up our model object.

public class Books {

private int isbn;

private String bookName;

public Books() {

}

public Books(int isbn, String bookName) {

this.isbn = isbn;

this.bookName = bookName;

}

// getter setter methods

}

It is a simple object with just 2 properties to keep things simple.

**DAO Pattern Interface**

Let’s define the interface to access the data associated with it at persistence level.

import java.util.List;

public interface BookDao {

List<Books> getAllBooks();

Books getBookByIsbn(int isbn);

void saveBook(Books book);

void deleteBook(Books book);

}

**DAO Pattern Implementation**

Next, we create a concrete class implementing the above interface.

import java.util.ArrayList;

import java.util.List;

public class BookDaoImpl implements BookDao {

//list is working as a database

private List<Books> books;

public BookDaoImpl() {

books = new ArrayList<>();

books.add(new Books(1, "Java"));

books.add(new Books(2, "Python"));

books.add(new Books(3, "Android"));

}

@Override

public List<Books> getAllBooks() {

return books;

}

@Override

public Books getBookByIsbn(int isbn) {

return books.get(isbn);

}

@Override

public void saveBook(Books book) {

books.add(book);

}

@Override

public void deleteBook(Books book) {

books.remove(book);

}

}

**Using DAO Pattern**

Finally, we put this implementation to use in our main() method:

public class AccessBook {

public static void main(String[] args) {

BookDao bookDao = new BookDaoImpl();

for (Books book : bookDao.getAllBooks()) {

System.out.println("Book ISBN : " + book.getIsbn());

}

//update student

Books book = bookDao.getAllBooks().get(1);

book.setBookName("Algorithms");

bookDao.saveBook(book);

}

}

**Advantages of DAO pattern**

There are many advantages for using DAO pattern. Let’s state some of them here:

1. While changing a persistence mechanism, service layer doesn’t even have to know where the data comes from. For example, if you’re thinking of shifting from using MySQL to MongoDB, all changes are needed to be done in the DAO layer only.
2. DAO pattern emphasis on the low coupling between different components of an application. So, the View layer have no dependency on DAO layer and only Service layer depends on it, even that with the interfaces and not from concrete implementation.
3. As the persistence logic is completely separate, it is much easier to write Unit tests for individual components. For example, if you’re using JUnit and Mockito for testing frameworks, it will be easy to mock the individual components of your application.
4. As we work with interfaces in DAO pattern, it also emphasizes the style of “work with interfaces instead of implementation” which is an excellent [OOPs](https://www.journaldev.com/12496/oops-concepts-java-example) style of programming.