SWE 4302
Object Oriented Concepts II Lab

Lab-11

Group-B

Mohammod Mahbub Ur Rahman
220042148
BSc. in Software Engineering

Task 1

Leaky Encapsulation: In the “Book”, “Cart”, and “User” classes have private attributes but We can access those values by the get, set methods. This makes the encapsulation leaky. Anyone can access the private attributes thus making our attributes private meaningless.

Insufficient Modularization: In the “User” class there are some attributes that relate to his address. We can further modularize it by taking those attributes and putting them in a separate class that will handle all address related work.

Needless Repetition: The “DBConnection” and “DBUtil” classes do the same thing, just their structures are different.

Task 2

Solve:

Leaky Encapsulation: To solve this, we can delete some setter and getter methods. like in the user class the get method for the password should be deleted as by using this anyone can the password of the user. The use of the password should be limited to the User class only. It is also valid for some other attributes.

we can also make some attributes public as some of them are made private unnecessarily such as the firstName, lastName of the user class.

finally If we make an attributes Private we need to make sure that no one can access it unauthorized and make it read/write only

```
private String password;
private String firstName;
private String lastName;
private void setPassword(String password) {
   this.password = password;
}

//no get password.
```

Insufficient Modularization: To solve this we need to take the address related data to another class.

```
public class User implements Serializable {
       Address address;
}

public class Address {
   private String addressLine1;
   private String city;
   private String state;
   private String country;
   private String address;
       //other relevant method
}
```

Needless Repetition: To solve this we need to merge those two classes into one cause they do the same thing.

```
public class DBConnection {  no usages   Dr-Lepic

    private static Connection connection;  3 usages

    public static Connection getDatabaseConnection() {  no usages   Dr-Lepic

        if (connection == null) {
            try {

                Class.forName(DatabaseConfig.DRIVER_NAME);

                connection = DriverManager.getConnection(DatabaseConfig.CONNECTION_STRING,
                        DatabaseConfig.DB_USER_NAME,
                        DatabaseConfig.DB_PASSWORD);
            } catch (SQLException | ClassNotFoundException e) {

                e.printStackTrace();

            }
        }

        return connection;
    }

}
```

Task 3

To remove the use of static we can follow the following way.

1. Create an interface that stores the methods that return the relevant query as a string.

```
interface QueryProvider {
   String getAllBooksQuery();
   String getBookByIdQuery();
   String deleteBookByIdQuery();
   String addBookQuery();
   String updateBookQtyByIdQuery();
}
```

2. Implement the interface in a class to provide the implementation.

```
class FirstQueryProvider implements QueryProvider {
   public String getAllBooksQuery() {
       return "SELECT * FROM " + BooksDBConstants.TABLE_BOOK;
   }
   public String getBookByIdQuery() {
       return "SELECT * FROM " + BooksDBConstants.TABLE_BOOK + "
WHERE " + BooksDBConstants.COLUMN_BARCODE + " = ?";
   }
... ...
.
.
.
}
```

3. Now create a FirstQueryProvider type object in the required class and store it in a QueryProvider type reference.

```
QueryProvider queryProvider = new FirstQueryProvider();
```

now call the relevant methods when needed.

How is it maintaining runtime polymorphism?

let’s say we have another class that implements the QueryProvider. The implementation is slightly different from the First one. now if we change the object type to the another class-

```
QueryProvider queryProvider = new SecondQueryProvider();
```

then the same methods we used before will give me different results. By this it maintains the runtime polymorphism.

Task 4

The two classes that implement the singleton pattern are “DBConnection” and “DBUtil”.

Here the “DBUtil” class’s implementation is more memory-intensive because in this class the static connection type object is created when the program is compiled. so it is created even before we need to use it. It takes up memory space without needing it.

On the other hand, “DBConnection” class’s connection type object is created when the object is needed thus saving our memory space.

Task 5

To handle multiple books in the “Cart” class and calculate the total price of the cart we can do the following things.

1. Store the books in an arraylist of “Book”

```
private ArrayList<Book> books;
```

write/edit the methods to add and remove book.

```
public void addBook(Book book) {
   books.add(book);
}
public void removeBook(int index) {
   books.remove(index);
}
```

2. Iterate through the arraylist and sum the price of the books in the list and return the price.

```
public double getTotalPrice(){
   double totalPrice = 0;
   for (Book book : books) {
       totalPrice += book.getPrice();
   }
   return totalPrice;
}
```