In order to make sure that the Database Manager worked as intended, Junit testing was conducted for each of its methods. The ZOMB+E methodology was not followed because it was not particularly relevant for this type of methods.

These are the Test Cases elaborated for each method:

setUp()

- Create an Order object and add two items to it: a Coffee and a Tea, both which exist in the database, and an extra, Honey, which will be added to the tea. This will be the order used in each method.

getInstance()

- The only two possible returns of the getInstance() method are null or a new object of the Singleton class. Therefore, the criteria for this test to pass is that the return of this method is not Null, because it means that an instance of the Singleton was returned.

Create()

- Should never throw anything, just create an Order object.
- After creating the order and by knowing the respective orderId, if we search for the id, the object returned should be **equal** to the one created previously.

ReadById()

- Should not throw anything if the ID exists.
- Should return a null element if the ID does not exist.

UpdateStatus()

- Should not throw an exception if the order ID exists.
- If the order id does not exist, nothing happens.
- If successful, it should be possible to read the order by id and see if its status is equal to the new status defined. (assertEquals).
    - Pending
    - Completed
    - Unpaid
- It should not be possible to update to any other status. This would throw an SQL Exception because of a violation of type.

UpdateComment()

- Should not throw an exception if the order ID exists.
- If the order id does not exist, nothing happens.
- If successful, it should be possible to read the order by id and see if its comment is equal to the new comment defined. (assertEquals).

AddItemToOrder()

- Should not throw an exception if both the Order and Item exist.
- Should throw an SQLexception if the OrderId does not exist.

- If successful, it should be possible to fetch the Order by Id and look for the item in its item list. (assertEquals)

Delete()

- It should not throw an exception if Order ID exists.
- Nothing happens if the order id does not exist.
- If the order id exists, we could try to look for the order by id and a null element.

Some observations worth making:

When, in the tests, an order is read from the database through the method *"readById",* the Id is hard-coded in the test, and the Id of each other is kept track of throughout the test class. This can be observed in the following screenshot:

```
@Test  void GupdateStatus1()
    throws SQLException // 3
{
  ConcreteOrderDAO.getInstance().create(order);
  ConcreteOrderDAO.getInstance().updateStatus( orderId: 3, status: "completed");

  Order orderUpdated = ConcreteOrderDAO.getInstance().readById(3)

  assertEquals( expected: "completed", orderUpdated.getStatus());
}
```

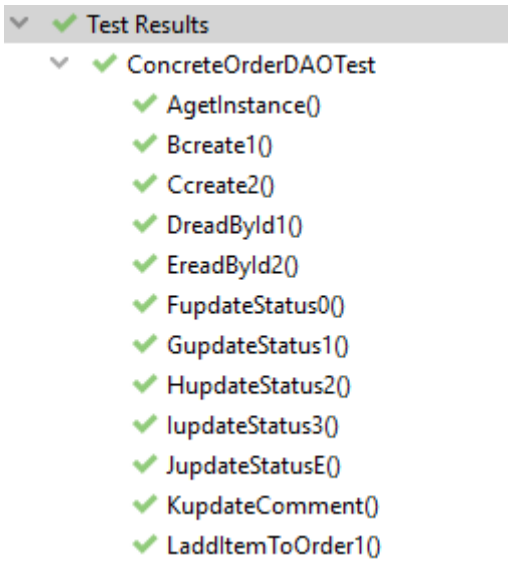
It is important to reset the database every time the tests are run so that the order indexes add up.

The first time that all of the tests were run multiple times, the number of tests passed & failed were varying. This was because the order in which the tests were ran was arbitrary by default on Junit, when actually, the tests elaborated were designed to be ran in order, due to the index of the orders created. If the order was random, the indexes would not correspond to the orders intended and the tests would be inaccurate. Therefore, it was necessary to force the order to become alphabetic for the tests to run correctly. This was done in the following manner:

```
@TestMethodOrder(MethodOrderer.MethodName.class)
class ConcreteOrderDAOTest
{
```

(continues on next page)

Then, the names of the methods were changed to start with A, B, C…



This way, they would run in the desired order (alphabetically).

Some test methods are worth going into a bit more detail.

```
@BeforeEach void setUp()
{
  order = new Order( paidWithCash: false);
  coffee = new Item( id: 1,  name: "Coffee",  type: "coffee",  price: 20,  description: ":)");
  tea = new Item( id: 2,  name: "Tea",  type: "tea",  price: 7,  description: "lovely herbal tea");
  honey = new Extra( name: "honey");

  order.addItem(coffee);
  order.addItem(tea);
  order.addExtraToItem(tea, honey);
}
```

For the setup(), it was relevant to add items to an Order. It's important that these item objects exist in the database, although in practice, the only field from the Item order that is sent to the database is the "id". In essence: it's crucial that the id provided in the Object is the same as the one existing in the database.

The extra must also exist in the database. As of this sprint, there was no functionality for adding items and extras to the database through the Java application, so the "dummy" items were added manually in DataGrip for the purpose of testing.

```java
@Test  void Ccreate2() throws SQLException // 2
{
  ConcreteOrderDAO.getInstance().create(order);
  Order createdOrder = ConcreteOrderDAO.getInstance().readById(2);

  System.out.println("Order: " + order);
  System.out.println("Created order: " + createdOrder);
  assertEquals(order.toString(), createdOrder.toString());
}
```

Another method worth talking about was create() (2). AssertEquals was returning false even if all of both object's fields were the exact same because it was comparing their references. Overriding the equals method in Order did not work. So the solution found was to run "assertEquals" in the toString() of both Order objects, which only looks at their "contents". This way, it was possible to compare the order provided to the method with the one created in the Database and see if they were the same.

As for the rest of the methods, they are straight-forward and clear enough by reading their descriptions in the Test Cases on the first page.