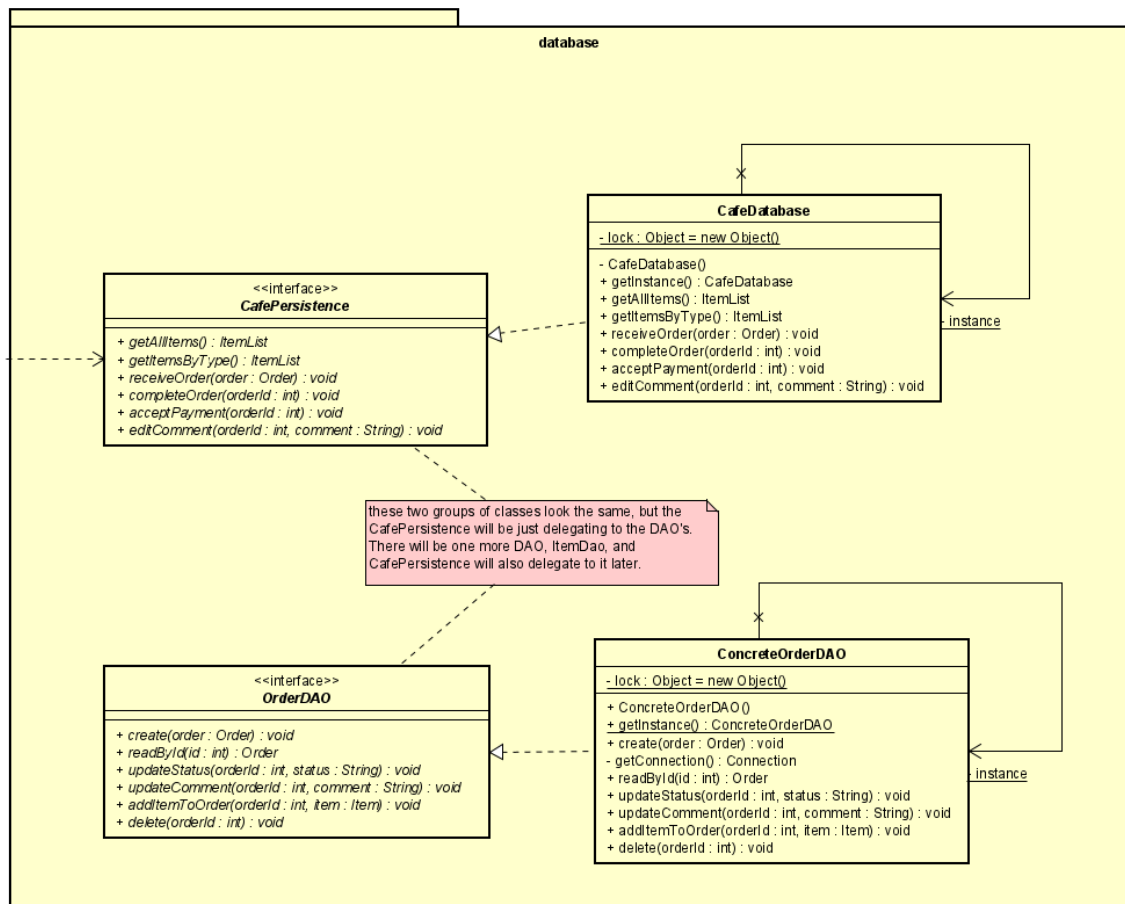


DATABASE ADAPTER DOCX – SPRINT 2 – ADD ITEM TO ORDER

In order for the system to communicate with the Database, it needed a class to handle these communications. The class that was elaborated is a Singleton class called CafeDatabase. This class implements an interface called CafePersistence. However, this class just serves to delegate methods to order Data Access Object Classes in which the communication will be implemented. For the Sprint 2 Product Backlog Item, it was only relevant to have an Order DAO. However, later, when there is an Item DAO, the CafePersistence Interface will also have methods to deal with adding and removing Items and Extras from the Database. But for now, it was not relevant.



A good way to start documenting it is by showing how it looks like as a Class Diagram:



Both concrete classes are Singletons. This is because it doesn't make sense for there to be more than one instance of a Database Manager. The CafeDatabase will not be doing any "work", just delegating to other Singletons.



The only method directly relevant to the Product Backlog Item at hand is "receiveOrder()", which delegates to the "Create" method in the DAO. It is relevant to say that the DAO only has "CRUD" methods, while the CafeDatabase has method names which make more sense regarding the system context. Still, they will only be delegating to the DAO CRUD methods. So, only the "C" is relevant now: create an order so that it will be registered in the database.

CAFEDATABASE:

```
42    @Override public void receiveOrder(Order order)
43      {
44          try {
45              ConcreteOrderDAO.getInstance().create(order);
46          }
47          catch (Exception e) {
48              e.printStackTrace();
49          }
50      }
51  }
52
```

As mentioned before, only a delegation to the OrderDAO happens. However, in this class, it is possible to go into more detail.

ORDERDAO:

```
37    @Override public void create(Order order) throws SQLException
38      {
39          String comment = order.getComment();
40          LocalDateTime dateTime = order.getDateTime().getLocalDateTime();
41          double price = order.getPrice();
42          String status = order.getStatus();
43          ArrayList<Item> items = order.getItemList().getAllItems();
44
45          try (Connection connection = getConnection())
46          {
47              PreparedStatement statement = connection.prepareStatement(
48                  sql: "INSERT INTO order_(comment, datetime, price, status) VALUES (?, ?, ?, ?);",
49                  PreparedStatement.RETURN_GENERATED_KEYS);
50              statement.setString( parameterIndex: 1, comment);
51              statement.setTimestamp( parameterIndex: 2, Timestamp.valueOf(dateTime));
52              statement.setDouble( parameterIndex: 3, price);
53              statement.setString( parameterIndex: 4, status);
54              statement.executeUpdate();

```

The first thing that happens inside the method is the fetching of the information inside the provided Order object, because it will then be inserted into the Update statement.

After all the information is acquired, a statement is prepared. This statement inserts a new Order row into the Order table. The statement is then executed.

However, that is not all. Because an order does not only exist on its own – it has items, and these items may, in turn, have extras which were added to them. Inside the database, this implies the addition of rows to the tables “ItemInOrder” and “ExtraInItemInOrder”.

Please observe the following screenshot, which is a continuation of the “create” method.

```

55      ResultSet keys = statement.getGeneratedKeys();
56      int orderId = 0;
57      if (keys.next())
58      {
59          orderId = keys.getInt( columnIndex: 1);
60      }
61      else
62      {
63          throw new SQLException("No keys generated");
64      }
65      for (Item item : items)
66      {
67          int itemId = item.getId();
68          ArrayList<Extra> extras = item.getExtras();
69          PreparedStatement itemInOrderStatement = connection.prepareStatement(
70              sql: "INSERT INTO iteminorder(order_id, item_id) VALUES (?,?)");
71          itemInOrderStatement.setInt( parameterIndex: 1, orderId);
72          itemInOrderStatement.setInt( parameterIndex: 2, itemId);
73          itemInOrderStatement.executeUpdate();

```

The first thing that happens is the fetching of the Order id. This is necessary because the order object provided does not have an Id: the Java class does not have a field for the order Id because it is auto generated when the Order is inserted into the database. Therefore, it is necessary to get it when the order object is created. This is done by getting the “generated keys” by the statement that was executed. The id is then stored in “orderId”.

After that, the method goes through every item in the order’s item list and gets the information relevant to the insert: the item’s id, and also its extras, which will be dealt with in the following screenshot. The method then inserts both the order and item’s ids into the table “ItemInOrder”, and the Item is then associated with the order. Now, an order exists and it has items. But what about the extras that might have been added to these items?

```

74      for (Extra extra : extras)
75      {
76          System.out.println("entered statement.");
77          PreparedStatement extrainiteminorderStatement = connection.prepareStatement(
78              sql: "INSERT INTO extrainiteminorder(extra_id, item_id, order_id) VALUES (?,?,?)");
79          extrainiteminorderStatement.setString( parameterIndex: 1, extra.getName());
80          extrainiteminorderStatement.setInt( parameterIndex: 2, itemId);
81          extrainiteminorderStatement.setInt( parameterIndex: 3, orderId);
82          extrainiteminorderStatement.executeUpdate();

```

The exact same thing happens. For every extra in each item, a row is added to the “ExtraInItemInOrder” table.

And that is how the Database Manager handled the creating of an order by the system. It is to note that the method will be called all the way from the View to the ViewModel to the Model to the Client, who remotely calls it from the server, which then delegates to the Database Manager. So it is a method that goes all the way back to the view.