IMS PROJECT READ ME - Waleed Naseem

Project Description

The overall objective of the project is to;

To create an application with the utilisation of supporting tools, methodologies and technologies that encapsulate all fundamental modules covered during training.

Git Hub

Downloaded the Repo via GitBash

Need to fill in this part. - How I used GitBash with it / when I used it.

Starting On The Code

I took my first proper look at the code and domain for the project.

I read the scope and tried to familiarize what I want to do in my head, and then break it down as steps that I would do in Java and SQL to meet the application demands - I would then pass this information on to Jira and use a scrum board.

Jira

https://evilvillainy.atlassian.net/jira/software/projects/QIMS/boards/9/roadmap?shared=&atlOrigin=eyJpIjoiNDhmYzBhZWNlNmNhNDhkMzIiODc5YTAyNWQ3ZjdjOGIiLCJwIjoiaiJ9

I firstly made 3 epics on a Scrum Board as I thought these were the 3 most important things for this project.

I then broke down what the application needs to do into user stories.

After defining user stories, I created child issues, which contained the task.

I then used scrum poker online, to get story points and assign them to the task. (Ideally, would have been a lot more interesting had there been more than one person working on this project)

As I worked through the coding, I updated the board as I went along.

<u>What the application needs to do</u>

I have broken what the application needs to do down to explain any issues I faced and or explain my code and little bits I learned along the way.

The application needs to be an inventory management system that needs to be able to:

- Add an item to the system.
- View all items in the system.
- Update an item in the system.
- Delete an item in the system.
- Create order in the system.
- View all orders in the system.
- Delete an order in the system.
- Add an item to order.
- Calculate a cost for an order.
- Delete an item in an order.

I have not broken down the ‘Customer’ functions as this was given to us from the start. These include;

- Add a customer to the system
- View all customers in the system
- Update a customer in the system
- Delete a customer in the system.

I will now show my IMS system running and showing the above functions all working.

<u>The Logger ‘Issue’</u>

I needed a refresh on what *LOGGER* is and how it is used.

The purpose of the logging system is to provide the user with facilities to log information and retrieve it later.

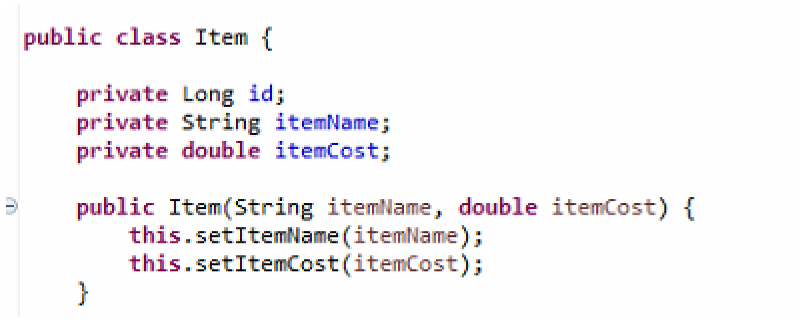
When first looking at the code, I did not fully understand what the *LOGGER* function did; however, through trial and error I figured out how to use the function efficiently. I used this function throughout the code to output to the console (and user).

‘Items’

- Add an item to the system
- View all items in the system
- Update an item in the system
- Delete an item in the system

To have items in the system, we added an SQL Table for Items. This included id (primary key) itemName and itemCost.

I then copied the code given to us for the customer functions (customerDAO.java, customerController.java and customer.java) and replaced the variables with our item specific values. This meant changing item.java to String itemName and double itemCost. Id was kept from customer.java

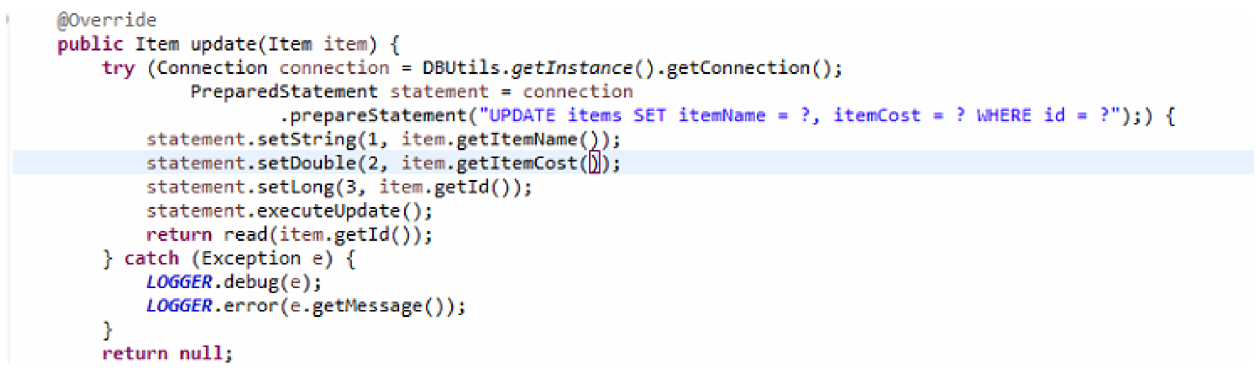


```
public class Item {

    private Long id;
    private String itemName;
    private double itemCost;

    public Item(String itemName, double itemCost) {
        this.setItemName(itemName);
        this.setItemCost(itemCost);
    }
```

ItemDAO.java was changed to the correct SQL statements to pull from the items table instead of the customer table and to update the correct fields within the table. This includes using double instead of string for the itemCost field, instead of both fields being strong in the customerDAO.java. Double was also used instead of int so we can assign decimal values to the item cost.



```
    @Override
    public Item update(Item item) {
        try (Connection connection = DBUtils.getInstance().getConnection();
                PreparedStatement statement = connection
                        .prepareStatement("UPDATE items SET itemName = ?, itemCost = ? WHERE id = ?");) {
            statement.setString(1, item.getItemName());
            statement.setDouble(2, item.getItemCost());
            statement.setLong(3, item.getId());
            statement.executeUpdate();
            return read(item.getId());
        } catch (Exception e) {
            LOGGER.debug(e);
            LOGGER.error(e.getMessage());
        }
        return null;
```

^^ Item update function in itemDAO.java used as one example of how the function was changed from customerDAO.java

‘Order‘ - The good, the bad, the ugly.

- Create an order in the system.
- View all orders in the system.
- Delete an order in the system
- Add an item to an order.
- Delete an item in an order.

The base for these functions was taken from the item functions, however, these need to be modified heavily.

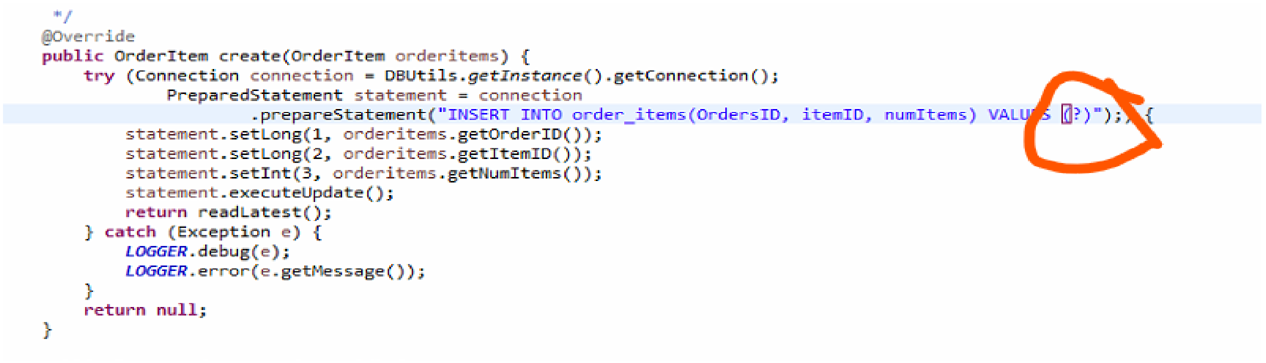
We first implemented the tables required into SQL. These tables were orders and order_items. Orders were used for the creation of orders and linking them to a customer using the customerID (foreignkey from customers). Order_items was used to link the orderID (foreignkey from orders table) to an itemID (foreignkey from items table) and numItems to know how many items and what item was placed in the order. Once this was done we could begin coding in Java.

I began with order.java, this class had 2 variables, id and customerID both with the data type of long. I then made OrderItem.java which has 3 variables, orderID, itemID and numItems. I had orderID and itemID as long data type while numItems as int, as the quantity of items placed in an order would not surpass 32bits in practicality, whereas the ID could.

After this, the DAO’s were made. As there are two tables I am manipulating I would need two DAO’s one for each table. So I made OrderDAO.java and OrderItemsDAO.java.

OrderItemsDAO proved to be quite a struggle in comparison to all the other DAO’s, a lot of time was spent researching on QA community, StackOverflow, YouTube and the java documentation to find the relevant information needed to create this DAO.

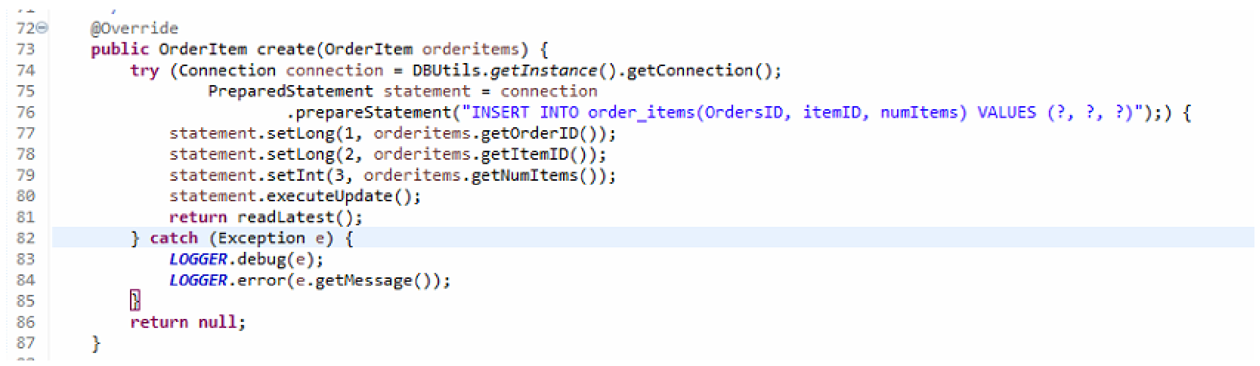
One problem I faced for quite a while was inserting an order into the order_items table when also adding to the orders table simultaneously. I found the issue to be within the SQL query I was sending within OrderItemsDAO.java



```
     */
    @Override
    public OrderItem create(OrderItem orderitems) {
        try (Connection connection = DBUtils.getInstance().getConnection();
                PreparedStatement statement = connection
                        .prepareStatement("INSERT INTO order_items(OrdersID, itemID, numItems) VALUES (?)");) {
            statement.setLong(1, orderitems.getOrderID());
            statement.setLong(2, orderitems.getItemID());
            statement.setInt(3, orderitems.getNumItems());
            statement.executeUpdate();
            return readLatest();
        } catch (Exception e) {
            LOGGER.debug(e);
            LOGGER.error(e.getMessage());
        }
        return null;
    }
```

Broken code ^ (OrderItemsDAO.java)

This was fixed by adding two more question marks into the VALUES parentheses as I specified 3 statements to be inserted however only 1 value was expected. Once that was changed to 3 values expected the problem was fixed.



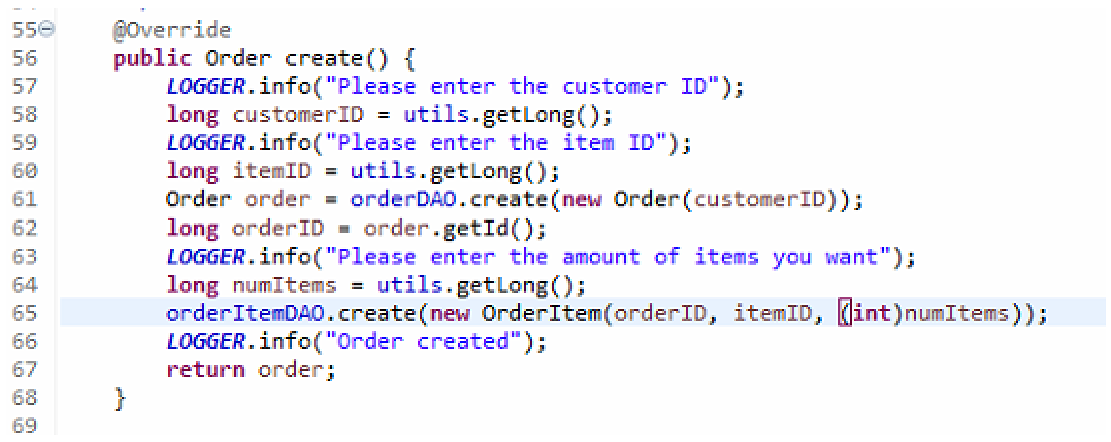
```
@Override
public OrderItem create(OrderItem orderitems) {
    try (Connection connection = DBUtils.getInstance().getConnection();
            PreparedStatement statement = connection
                    .prepareStatement("INSERT INTO order_items(OrdersID, itemID, numItems) VALUES (?, ?, ?)");) {
        statement.setLong(1, orderitems.getOrderID());
        statement.setLong(2, orderitems.getItemID());
        statement.setInt(3, orderitems.getNumItems());
        statement.executeUpdate();
        return readLatest();
    } catch (Exception e) {
        LOGGER.debug(e);
        LOGGER.error(e.getMessage());
    }
    return null;
}
```

Fixed code ^ (OrderItemsDAO.java)

This code was not all done in a one by one fashion as there were quite a few moving pieces within it. So code had to be edited and redone to work due to errors occurring. This happened quite a few times when fitting in the final piece, OrderController.java.

An OrderItemController was not needed as everything we wanted the user to be able to do would interface through OrderController which would then be able to modify both orders and order_items table through the OrderDAO and OrderItemsDAO classes.

In OrderController, I ran into some difficulties with the create function however through some research it was quickly resolved. The issue I was having was trying to create a new OrderItem however, utils did not have a getInt function for numItems. Here I was posed with two options, either implement a getInt function into utils or convert the long into an int value. I chose to convert the long into an int value using (int)variable.



```
55      @Override
56      public Order create() {
57          LOGGER.info("Please enter the customer ID");
58          long customerID = utils.getLong();
59          LOGGER.info("Please enter the item ID");
60          long itemID = utils.getLong();
61          Order order = orderDAO.create(new Order(customerID));
62          long orderID = order.getId();
63          LOGGER.info("Please enter the amount of items you want");
64          long numItems = utils.getLong();
65          orderItemDAO.create(new OrderItem(orderID, itemID, (int)numItems));
66          LOGGER.info("Order created");
67          return order;
68      }
69
```

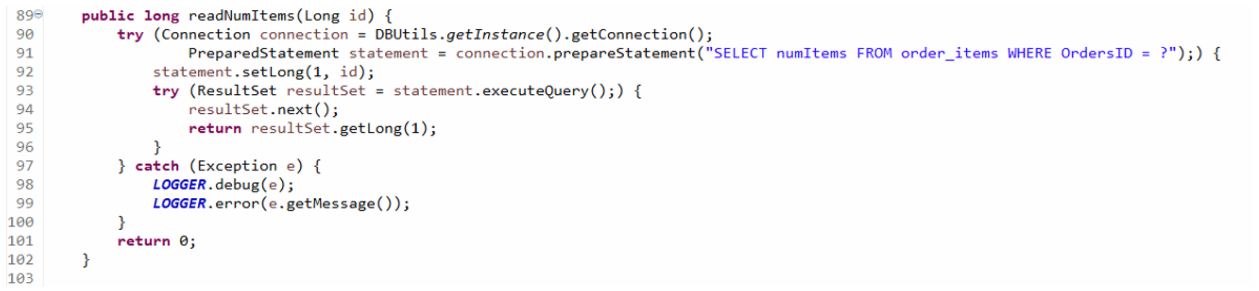
^ OrderController.java fixed code for create function. Line 65.

Create, and delete commands had to interface with both the orders table and the order_items table in SQL, while the update command (to add or delete an item from an order, aka change the quantity) only had to interface with the order_items table.

‘Calculate’

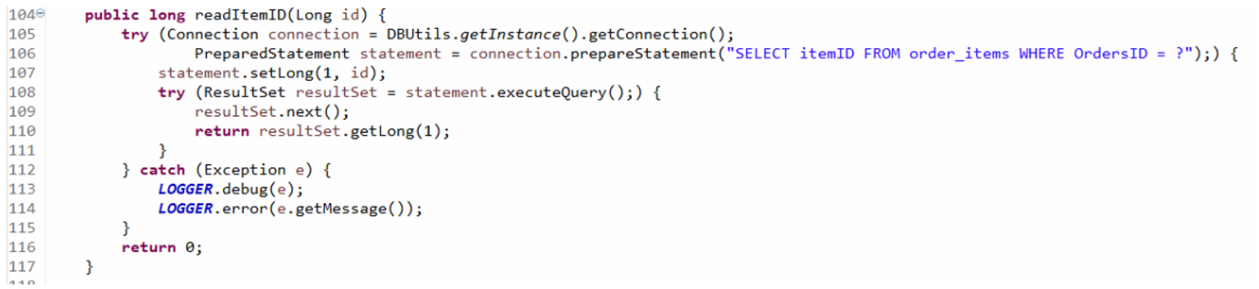
- Calculate a cost for an order.

Calculating the cost for the order required interfacing with two tables however it wasn’t the orders and order_items table and instead had to interface with the order_items table and items table. This was needed as the relevant information required was itemCost(in items table), orderID and numItems(both found in orders_table). In order to ease the process of reading the information of these tables, three functions were implemented in the OrderItemsDAO.java to return the required value from each field.



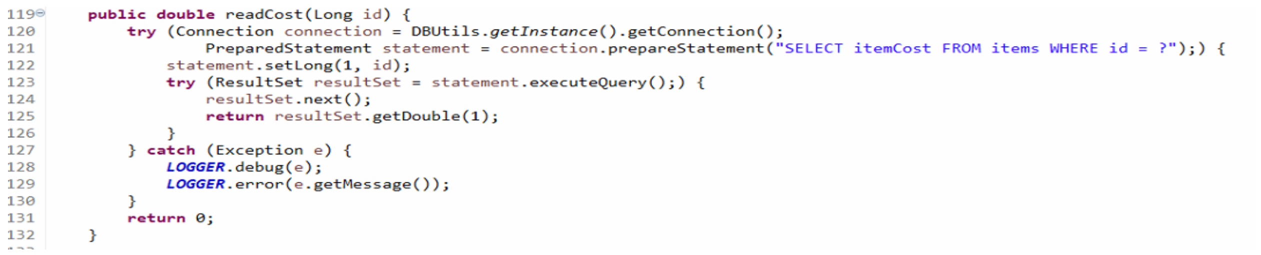
```java
    public long readNumItems(Long id) {
        try (Connection connection = DBUtils.getInstance().getConnection();
                PreparedStatement statement = connection.prepareStatement("SELECT numItems FROM order_items WHERE OrdersID = ?");) {
            statement.setLong(1, id);
            try (ResultSet resultSet = statement.executeQuery();) {
                resultSet.next();
                return resultSet.getLong(1);
            }
        } catch (Exception e) {
            LOGGER.debug(e);
            LOGGER.error(e.getMessage());
        }
        return 0;
    }
```

^OrderItemsDAO.java gets the value of numItems for a given orderID



```java
    public long readItemID(Long id) {
        try (Connection connection = DBUtils.getInstance().getConnection();
                PreparedStatement statement = connection.prepareStatement("SELECT itemID FROM order_items WHERE OrdersID = ?");) {
            statement.setLong(1, id);
            try (ResultSet resultSet = statement.executeQuery();) {
                resultSet.next();
                return resultSet.getLong(1);
            }
        } catch (Exception e) {
            LOGGER.debug(e);
            LOGGER.error(e.getMessage());
        }
        return 0;
    }
```

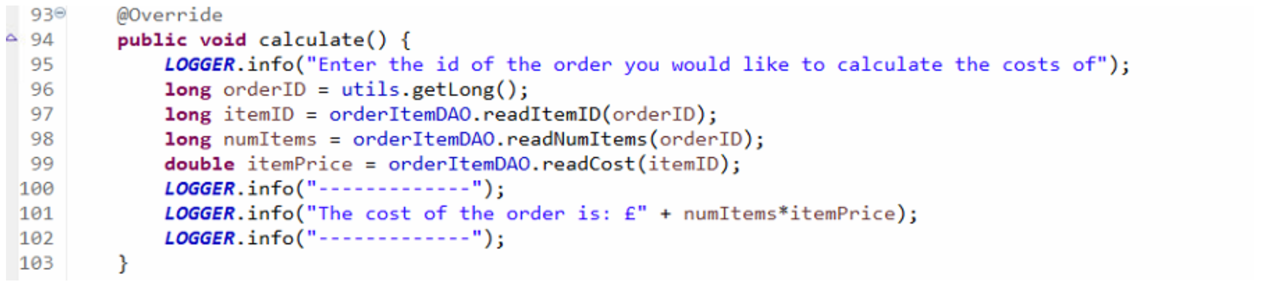
^OrderItemsDAO.java gets the value of the itemID for a given orderID



```java
    public double readCost(Long id) {
        try (Connection connection = DBUtils.getInstance().getConnection();
                PreparedStatement statement = connection.prepareStatement("SELECT itemCost FROM items WHERE id = ?");) {
            statement.setLong(1, id);
            try (ResultSet resultSet = statement.executeQuery();) {
                resultSet.next();
                return resultSet.getDouble(1);
            }
        } catch (Exception e) {
            LOGGER.debug(e);
            LOGGER.error(e.getMessage());
        }
        return 0;
    }
```

^OrderItemsDAO.java gets the price of an item for a given itemID.

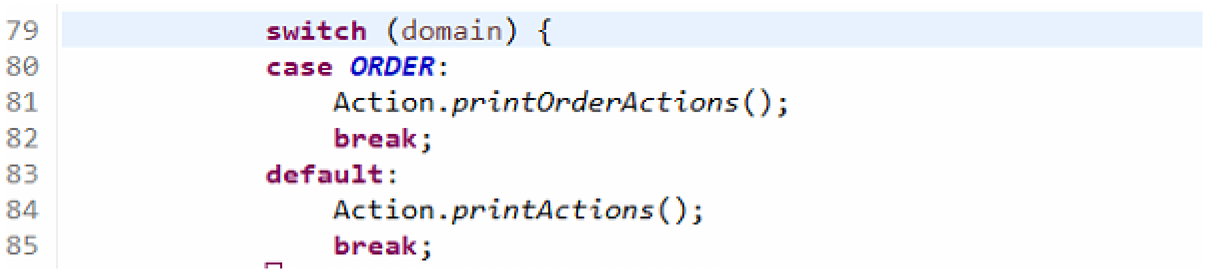
These are all then used in the calculate function in OrderController in order to output the cost to the user given an orderID



```java
    @Override
    public void calculate() {
        LOGGER.info("Enter the id of the order you would like to calculate the costs of");
        long orderID = utils.getLong();
        long itemID = orderItemDAO.readItemID(orderID);
        long numItems = orderItemDAO.readNumItems(orderID);
        double itemPrice = orderItemDAO.readCost(itemID);
        LOGGER.info("-------------");
        LOGGER.info("The cost of the order is: £" + numItems*itemPrice);
        LOGGER.info("-------------");
    }
```

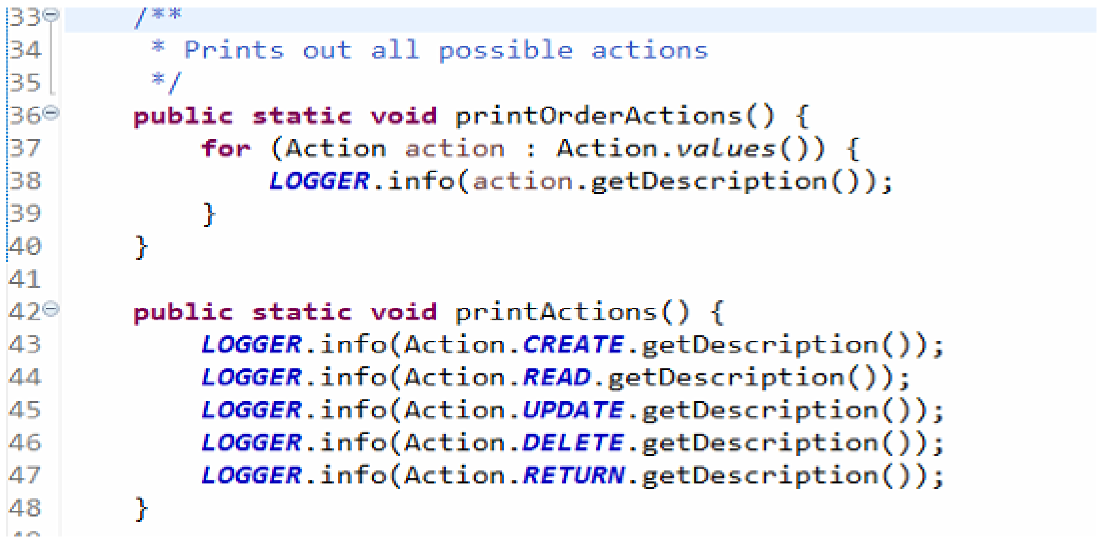
^OrderController.java calculate function

Finally, this all had to be implemented into the IMS and action java files. As order had one extra function compared to controller and item. This was achieved through using a switch and an extra method in the action class.



```java
            switch (domain) {
            case ORDER:
                Action.printOrderActions();
                break;
            default:
                Action.printActions();
                break;
```

^IMS.java calls printOrderActions() when ORDER is stated by the user. But printActions is called every other time.



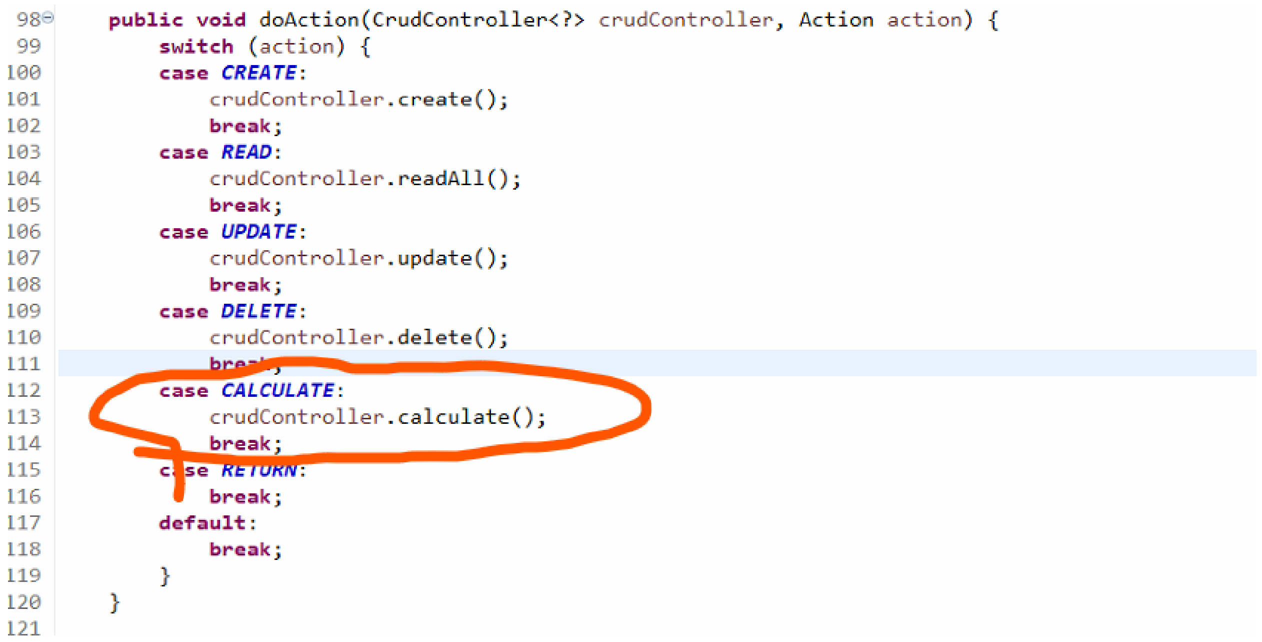
```java
    /**
     * Prints out all possible actions
     */
    public static void printOrderActions() {
        for (Action action : Action.values()) {
            LOGGER.info(action.getDescription());
        }
    }

    public static void printActions() {
        LOGGER.info(Action.CREATE.getDescription());
        LOGGER.info(Action.READ.getDescription());
        LOGGER.info(Action.UPDATE.getDescription());
        LOGGER.info(Action.DELETE.getDescription());
        LOGGER.info(Action.RETURN.getDescription());
    }
```

^Action.java both functions used in the case switch above.

The printOrderActions method prints all actions (includes calculate) while the normal printActions prints out all the actions except calculate as only the order function uses it.

Finally the CALCULATE Option was implemented into IMS.java as well:



```
    public void doAction(CrudController<?> crudController, Action action) {
        switch (action) {
        case CREATE:
            crudController.create();
            break;
        case READ:
            crudController.readAll();
            break;
        case UPDATE:
            crudController.update();
            break;
        case DELETE:
            crudController.delete();
            break;
        case CALCULATE:
            crudController.calculate();
            break;
        case RETURN:
            break;
        default:
            break;
        }
    }
```

^IMS.java case CALCULATE

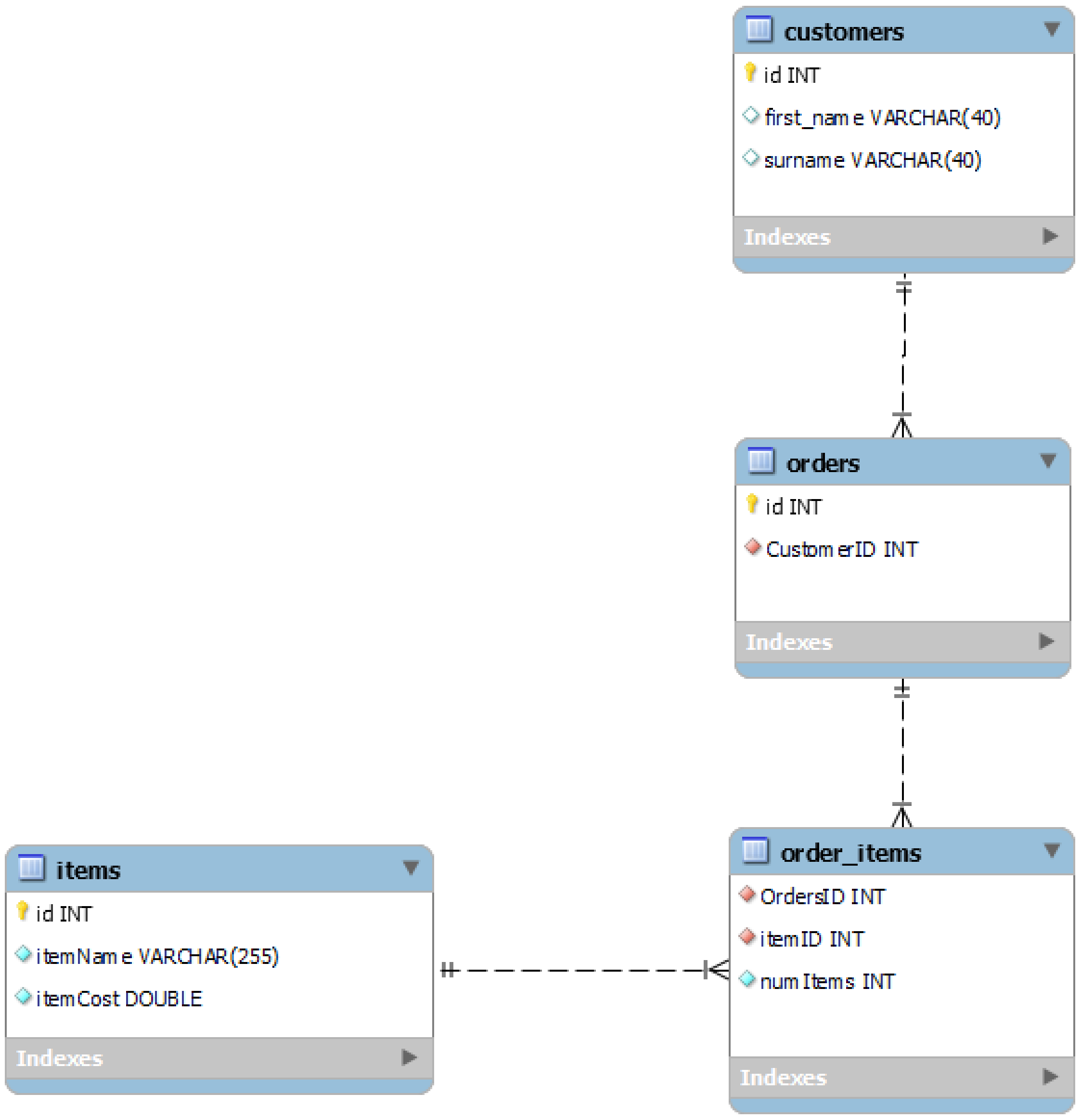
<u>Considerations</u>

- A customer needs to have a name.
- An item needs to have a name and a value.
- An order needs to have a customer and contains items.

This was all implemented when creating the SQL tables, and DAO and Domains in Java.

I couldn’t figure out how to implement different items within the same order, however, to still solve the problem, I decided to have multiples of the same item within an order so that the user can still add or remove an item from the order by updating the quantity of items in the order through the order update console menu.

EDR DIAGRAM


customers
id INT
first_name VARCHAR(40)
surname VARCHAR(40)
Indexes
orders
id INT
CustomerID INT
Indexes
items
id INT
itemName VARCHAR(255)
itemCost DOUBLE
Indexes
order_items
OrdersID INT
itemID INT
numItems INT
Indexes

Risk Matrix For IMS Project - Waleed Naseem

Impact

Likelihood

	1	2	3
1	Low	Low	Medium
2	Low	Medium	High
3	Medium	High	High

Risk Assessment For IMS Project - Waleed Naseem

ID	Risk Description	Likelihood Of Risk Occurring	Impact If The Risk Occurs	Severity (Based on impact and likelihood)	Owner (person who manages the risk)	Mitigating Action
1	Code Issues; ● Poorly written code. ● Difficult to read code. ● Program not running. ● Only some functions work.	2	3	High	Dev Team	Poor quality code can have an impact on software development. This can be mitigated by the following few steps; ● Testing the code frequently. ● Using coding best practices. ● Resolving bugs and logical errors when they're found. ● Ensuring Git-Hub is used to upload data on a forked branch.
2	Scope Creep; ● Unrealistic expectations. ● Complete change of project goals and outcomes.	1	2	Low	Stake Holder	In the case of this project, there will be no Scope Creep as it has been defined and will be unchanged till presented. Project scope changing can cause major risks in software development, it refers to a project scope completely morphing into something different. It can delay projects and cause you to miss deadlines and or extend timeframes.
3	Low productivity; ● Lack of understanding of project scope. ● Lack of experience in writing code. ● Reasonable timeframes for the project.	2	3	High	Dev Team	In the case of this project, as it is my first one there is a strong possibility of low productivity, majorly caused by being frustrated at certain code not working and being burnt out because of this. It can also be an issue for major companies, sometimes teams may struggle due to delays/employee burnout. This can be mitigated by; ● Finding a great motivational leader that can keep everyone motivated and manage the team. ● Ensuring the team communicates effectively about project details and problems. ● Ensure a good paced plan is made to lower stress amongst the team.

4	Stakeholder Issues; ● Lack of clarity for expectations. ● Lack of communication.	2	2	Medium	Stake Holder	In this project, just like scope creep, it's not something to worry about as our outcomes required have been predefined by stakeholders. However, in the workplace, issues can arrive with stakeholders, it's a huge risk as low engagement from them can lead to inaccurate expectations. To mitigate these issues we can; ● Communicate with shareholders more often and effectively. ● Involve stakeholders in team meetings.
5	Aggressive Deadlines; ● Undeliverable deadlines. ● Issues keeping up with demands.	2	3	High	Stake Holder	In this project again this won't be an issue as it's a deadline that is fixed and will not change. However, in the workplace these deadlines can be mitigated by; ● Using a Kanban Board and sticking to it. ● Ensuring that communication is up to date with stakeholders and other dev teams. ● Keeping up to date with project timelines.
6	Acts of God; ● Extreme weather events. ● Death. ● Natural disasters.	1	3	Medium	Stake Holder	This is very unlikely to affect this project. However, there is always an underlying chance of an event occurring. There are no mitigating circumstances for this particular risk as they are unpredictable.

Unit Tests

Integration Tests

Coding Style Tests