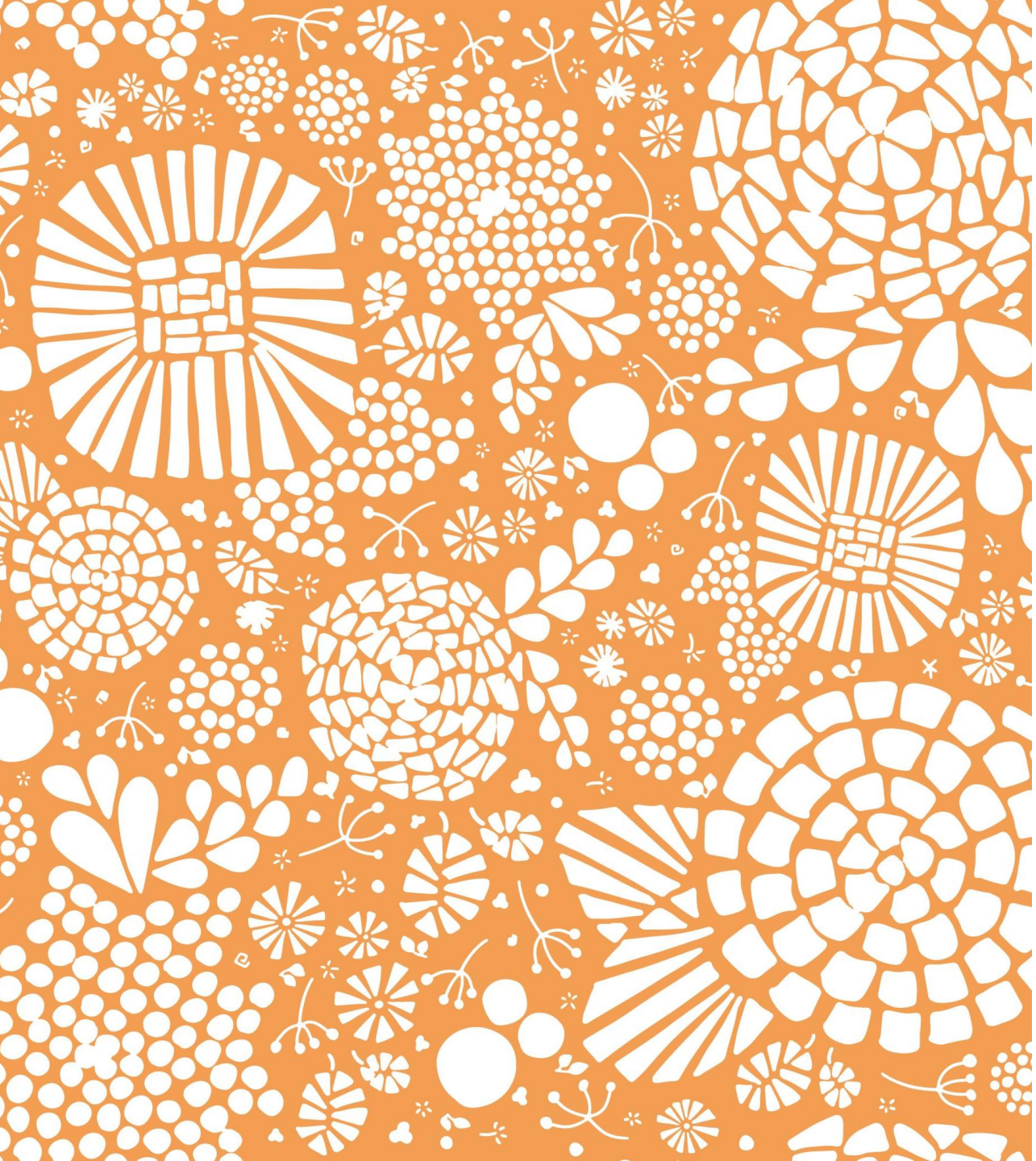


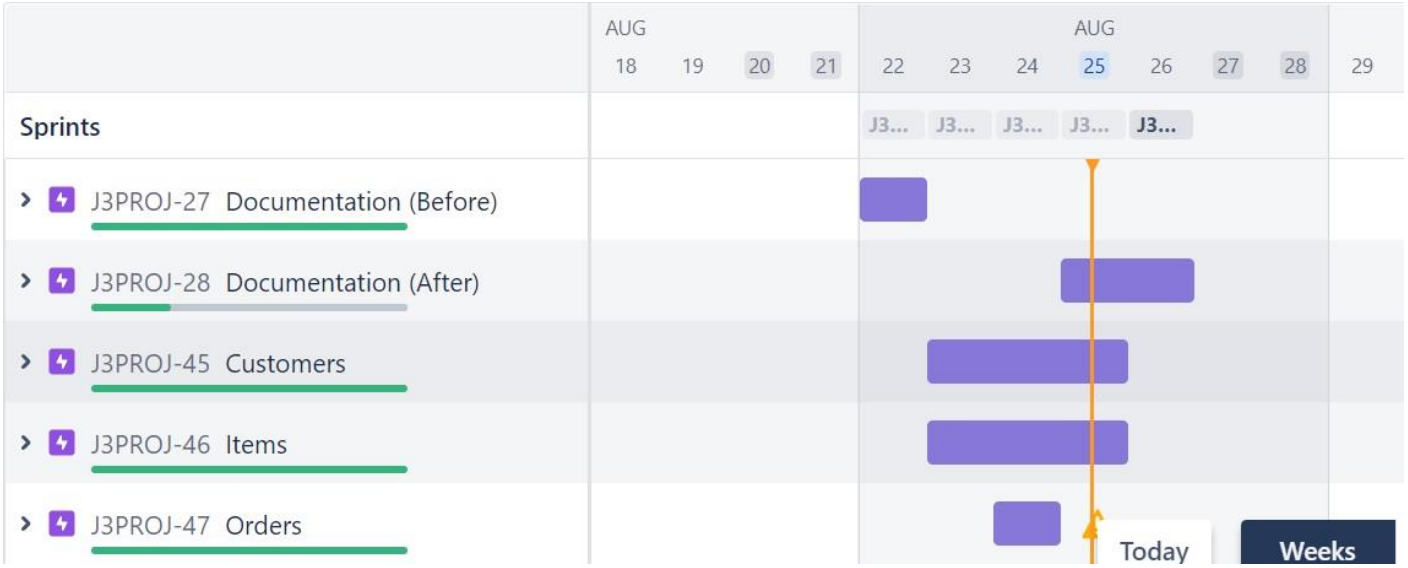
IMS for an online bar service

JADE FOSTER

Introduction

- First of all I broke the project into three main sections: Customers, Items and Orders.

- Used this as my 3 epics for the project, associated them with user stories and then broke all the tasks down necessary to do this in a Jira board, and organised them into sprints based on story point estimates.



Introduction

- I knew items would take a similar format to customers, but orders would be more complicated due to the many to many relationship, so we have to keep track of not just the orders but the items in the orders too.

- So I began by creating the my SQL schema and initial data, choosing the context of an online bar service, so customers could make orders and staff could keep track of them.

- Then I worked my way up from checking over the customer relevant classes, then to item and finally to orders. The extra functionality in orders took up the majority of the time, as trying to implement them in the existing structure of the program was challenging at first.

Consultant Journey

- Jade, 21, graduated this summer with a physics degree from the University of Oxford

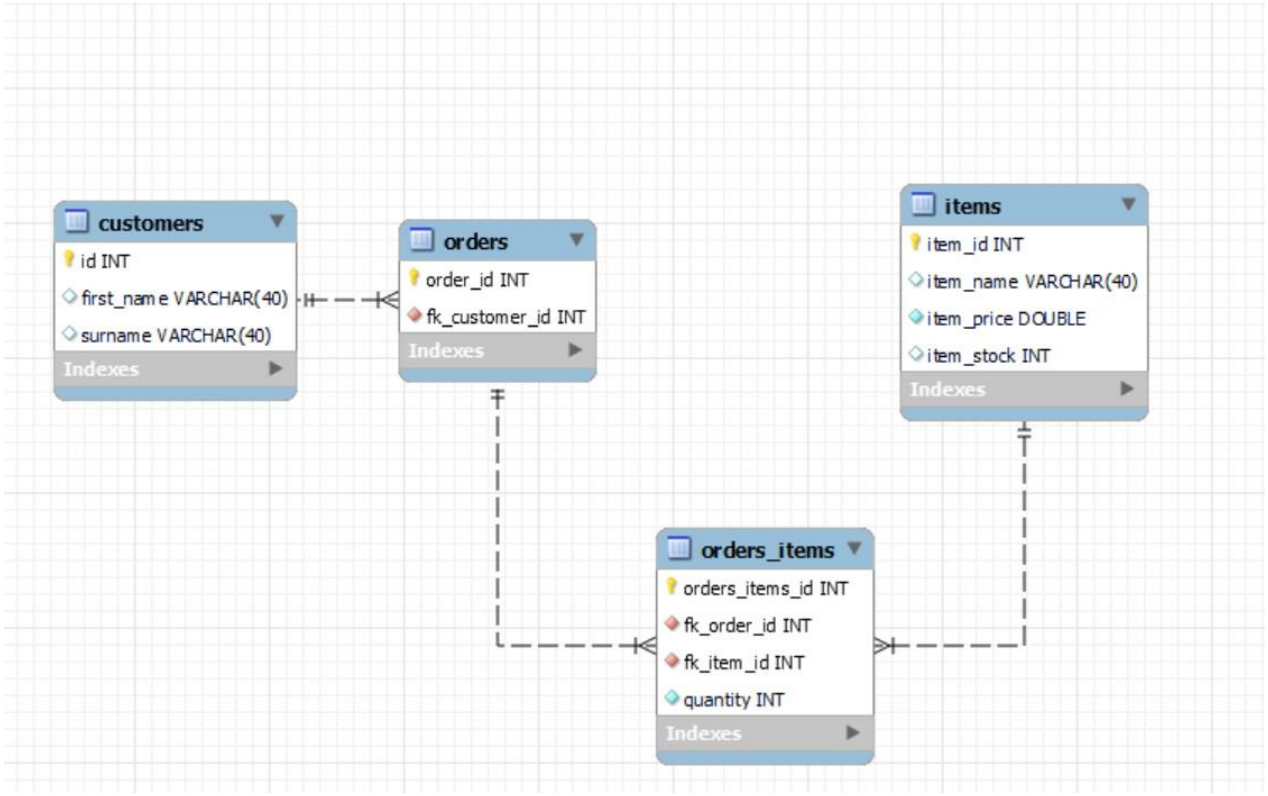
- My background before the QA academy was mainly python and MATLAB and other scientific based programming languages, so it was fun to experiment with a different styled language

- This project uses Java 18 and SQL, with Git for versioning, Maven for dependency management and JUnit for testing.

- This project also used Jira, which I have never used for a full project before, and I found it very helpful to use in breaking down my time.

- Testing and Mockito is still quite new to me so that part has taken me a little longer, as it doesn't come as naturally to me as normal coding does.

Database Schema



Continuous Integration

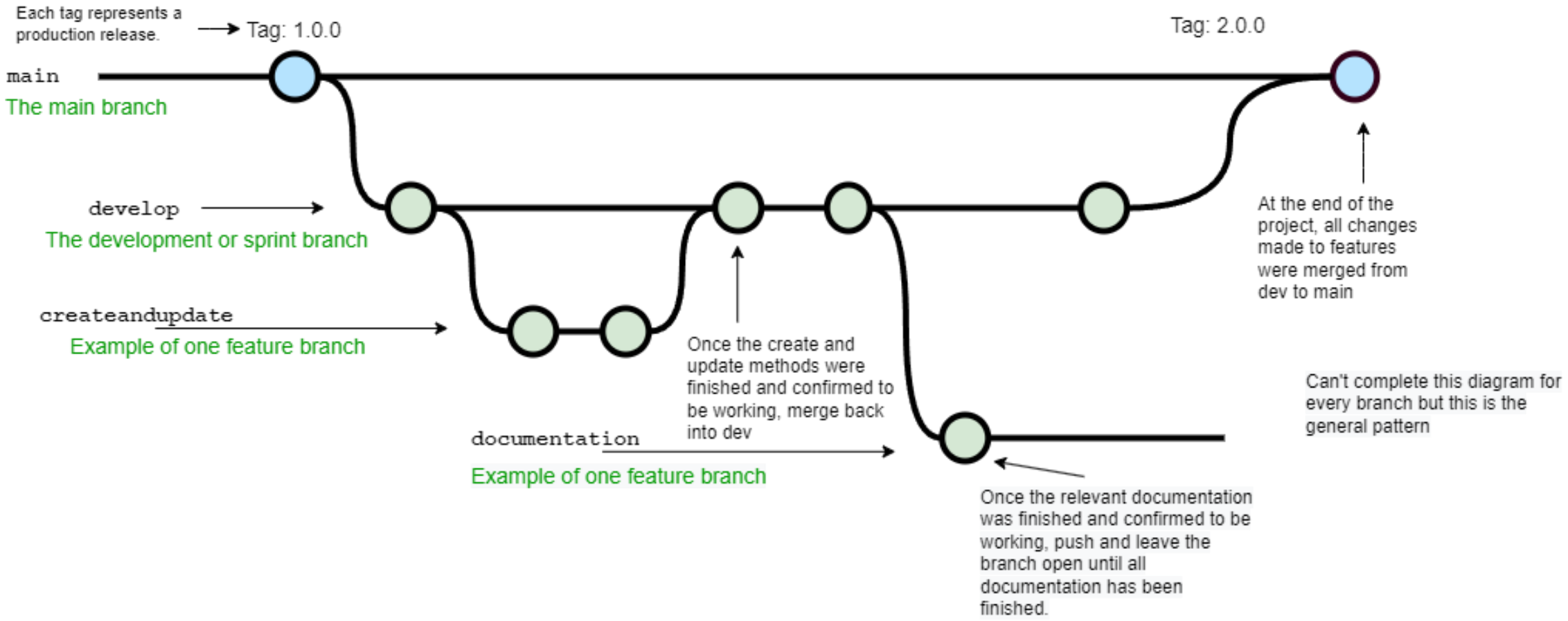
- In terms of version control, I used the branch model in Git, pushing all of my features to the dev branch, and then I will push my dev branch to main once I am finished at the end of today.

- Continuous Integration refers to the automated integration of code from many contributors into a single software project.

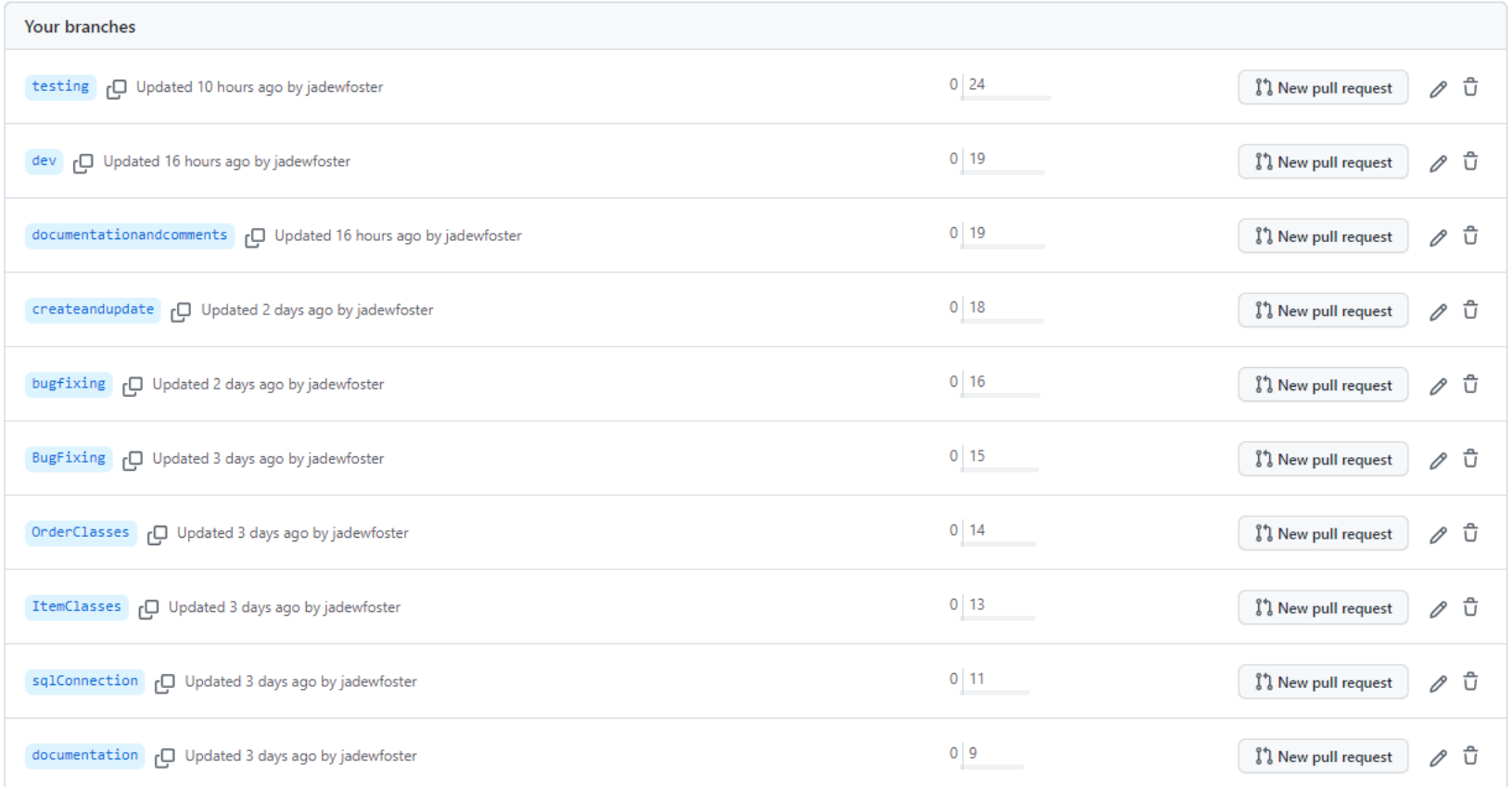
- As only myself and the original writer were involved, clearly this was not the case, however the style of branching meant that if this were a project with many contributors, it would be continuously integrated.

- This allows new code to be made easily and frequently using automated testing tools to test before integrating with the main branch. I chose to use a dev branch as I was saving testing until the end, therefore the master branch is always ready to deploy.

Git Branching Diagram for IMS Project



Branches in Git



In hindsight, my branch naming was not consistent and there were overlaps in content of some branches, but the features are generally separated to their own branch

Testing

Testing is currently at 66.7% coverage

- Main testing is in DAO and controller, as the getters and setters and constructors get tested via these methods.

- In the Order/Item/Customer DAO tests, unit testing is used and mainly assertequals

- In the Order/Item/Customer Controller tests, integration testing is used via Mockito objects.

- I will give an example of each now

Item update testing

```
    @Test
    public void testUpdate() {
        Item updated = new Item(1L, "Drink", 10.00, 100);

        Mockito.when(this.utils.getLong()).thenReturn(1L);
        Mockito.when(this.utils.getString()).thenReturn(updated.getItemName());
        Mockito.when(utils.getDouble()).thenReturn(updated.getPrice());
        Mockito.when(utils.getInteger()).thenReturn(updated.getStock());
        Mockito.when(this.dao.update(updated)).thenReturn(updated);

        assertEquals(updated, this.controller.update());

        Mockito.verify(this.utils, Mockito.times(1)).getLong();
        Mockito.verify(this.utils, Mockito.times(1)).getString();
        Mockito.verify(this.utils, Mockito.times(1)).getDouble();
        Mockito.verify(this.utils, Mockito.times(1)).getInteger();
        Mockito.verify(this.dao, Mockito.times(1)).update(updated);
    }
```

Order readAll testing

```
@Test
public void testReadAll() {
    List<Order> orders = new ArrayList<>();
    orders.add(new Order(1L, 1L, 1));

    Mockito.when(dao.readAll()).thenReturn(orders);

    assertEquals(orders, controller.readAll());

    Mockito.verify(dao, Mockito.times(1)).readAll();
}
```

Demonstration – User Story 1

As a staff member,

I want to be able to add a customer to the system,

So that I can make accounts for people over the phone and keep track of who our customers are.

Demonstration – User Story 2

As a staff member,

I want to be able to update items,

So I can change attributes such as stock and price

Demonstration – User Story 3

As a staff member, I want to be able to read all orders, So that I can keep the staff up to date with what items need to be made.

AND

As a customer, I want to be able to see the total cost my order, So that I know how much payment to make.

Acceptance Criteria Examples

1. Given the user is taken to the Order page,

When entering the corresponding shortcut,

Then I can create, read, update and delete the entries. As well as updating the items in an order.

2. Given the user enters a new order,

When entering the corresponding customer id, item id and quantity,

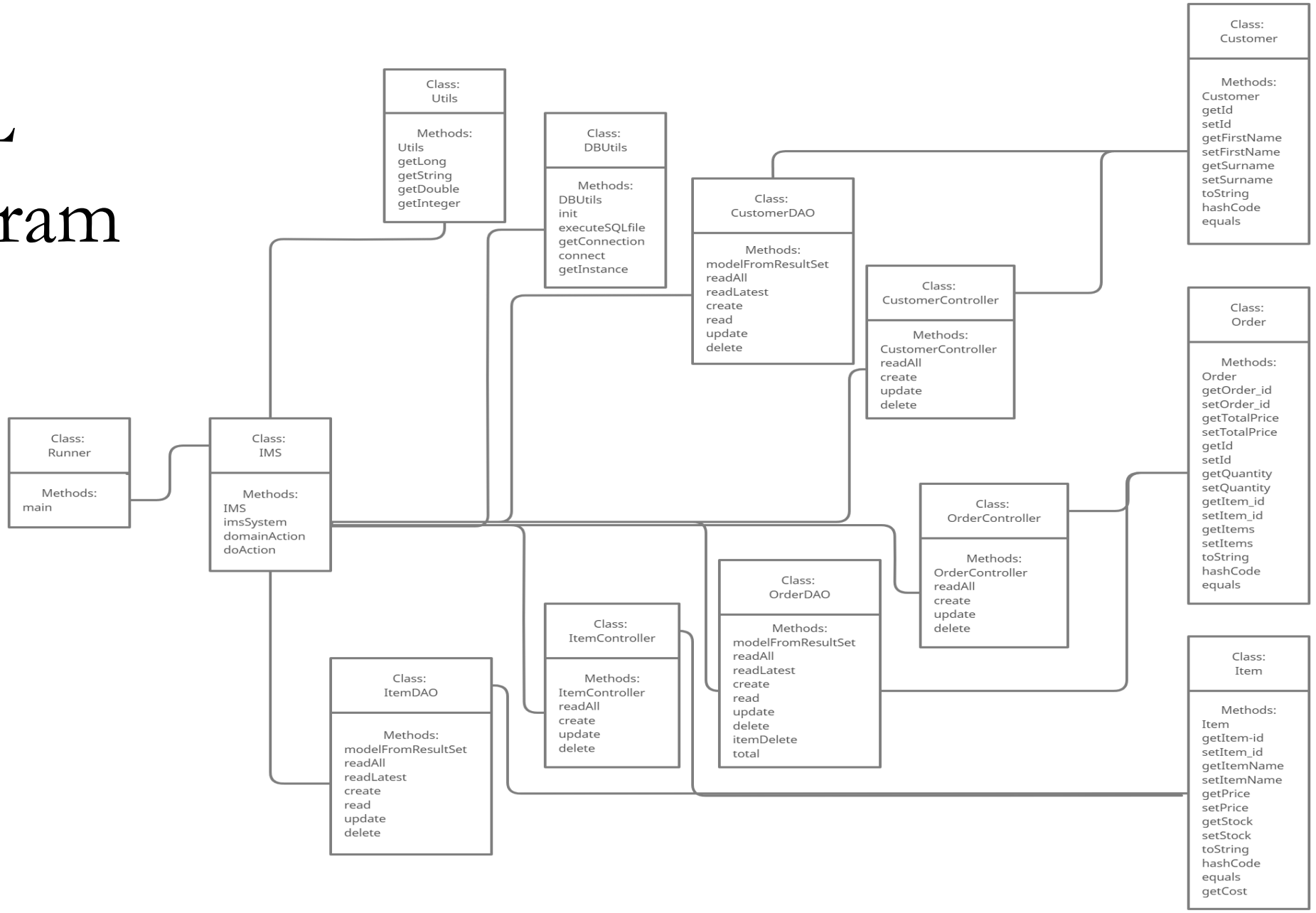
Then the inventory of orders and orders_items is updated with this item.

3. Given the user reads orders,

When entering READ,

Then all orders and their items are printed, as well as the total cost of each order.

UML Diagram


Class:
Runner
Methods:
main
Class:
IMS
Methods:
IMS
imsSystem
domainAction
doAction
Class:
Utils
Methods:
Utils
getLong
getString
getDouble
getInteger
Class:
DBUtils
Methods:
DBUtils
init
executeSQLfile
getConnection
connect
getInstance
Class:
CustomerDAO
Methods:
modelFromResultSet
readAll
readLatest
create
read
update
delete
Class:
CustomerController
Methods:
CustomerController
readAll
create
update
delete
Class:
Customer
Methods:
Customer
getId
setId
getFirstName
setFirstName
getSurname
setSurname
toString
hashCode
equals
Class:
Order
Methods:
Order
getOrder_id
setOrder_id
getTotalPrice
setTotalPrice
getId
setId
getQuantity
setQuantity
getItem_id
setItem_id
getItems
setItems
toString
hashCode
equals
Class:
OrderController
Methods:
OrderController
readAll
create
update
delete
Class:
OrderDAO
Methods:
modelFromResultSet
readAll
readLatest
create
read
update
delete
itemDelete
total
Class:
ItemController
Methods:
ItemController
readAll
create
update
delete
Class:
ItemDAO
Methods:
modelFromResultSet
readAll
readLatest
create
read
update
delete
Class:
Item
Methods:
Item
getItem-id
setItem_id
getItemName
setItemName
getPrice
setPrice
getStock
setStock
toString
hashCode
equals
getCost

Sprint 1 - Monday

Completed initial documentation inc risk assessment, Jira board and created database tables

Retrospective:

- Had a small glitch where I lost my risk assessment and had to do it over from the beginning

- Decided to redo my Jira board in later sprints to line up with the MVP model rather than adding extra features earlier in the week

- Database schema and entry was done very quickly and did not have to be changed later

Sprint 2 - Tuesday

Altered small details in customer classes, and used them as a template for the item classes. (inc DAO and Controller)

Retrospective:

- I think the aesthetic changes worked well, the lines to separate each request make it much easier to follow for the user

- Item classes went well too, no major differences from the structure of customer due to having the same relationship.

- Decided to add a stock attribute to the item class, which required a getInteger() method in utils.

- Then just adding item controller attribute into IMS.java file, instantiating object and adding active = this.item in the switch case statement

Sprint 3 - Wednesday

Mostly working on the orders functionality, specifically the total price and delete item/delete order case statements.

Retrospective:

- Took much longer than I needed to on this due to very silly errors and being returned null values from readLatest() due to an issue with resultSet.next() being called twice.

- Realised that prepare statements don't work with multiple queries in SQL, so had to separate them. This fixed my issues with the create and update methods for orderDAO.

- I also had an issue with calculating the total as initially it wasn't counting across identical order_ids, and instead was just outputting the first value. Also imported decimal format to allow rounding to 2dp for price.

- Decided to add a switch case statement to DELETE for order so that the user could choose to delete by order or by item

SQL multi-line query fix

```
@Override
public Order create(Order order) {
    try (Connection connection = DBUtils.getInstance().getConnection()) {
        try (PreparedStatement statement = connection
                .prepareStatement("INSERT INTO orders(fk_customer_id) VALUES (?)")) {
            statement.setLong(1, order.getId());
            statement.executeUpdate();
        }
        try (PreparedStatement statement = connection.prepareStatement(
                "SELECT order_id INTO @newid FROM orders WHERE fk_customer_id = ? ORDER BY order_id DESC
            statement.setLong(1, order.getId());
            statement.executeQuery();
        }
        try (PreparedStatement statement = connection.prepareStatement(
                "INSERT INTO orders_items(fk_order_id, fk_item_id, quantity) VALUES (@newid, ?, ?)")) {
            statement.setLong(1, order.getItem_id());
            statement.setInt(2, order.getQuantity());
            statement.executeUpdate();
        }
    } catch (SQLException e) {
        LOGGER.debug(e);
        LOGGER.error(e.getMessage());

    }

    LOGGER.info(readLatest());
    return order;
}
```

Switch statement for deleting orders

```
@Override
public int delete() {
    LOGGER.info("-------------------------------------------------------------------------------------------------\n"
            + "Please select whether you would like to delete the entire order, or an item from the order");
    LOGGER.info("DEL ORDER/DEL ITEM");
    String action = utils.getString();
    action = action.toUpperCase();
    switch (action) {


        case "DEL ORDER":
            LOGGER.info("-------------------------------------------------------------------------------------------------
                    + "Please enter the order id of the order you would like to delete:");
            Long order_id = utils.getLong();
            orderDAO.delete(order_id);
            LOGGER.info("------------------------------Order has successfully been deleted!-------------------------
            break;


        case "DEL ITEM":
            LOGGER.info("-------------------------------------------------------------------------------------------------
                    + "Please enter the order id you would like to delete an item from:");
            Long order_id2 = utils.getLong();
            LOGGER.info("Please enter the item id you would like to delete from the order:");
            Long item_id = utils.getLong();
            orderDAO.itemDelete(order_id2, item_id);
            LOGGER.info("------------------------------Item from order has successfully been deleted!----------------
            break;
```

Sprint 4 - Thursday

Testing and last aesthetic touches to main

Retrospective:

- Was receiving a JUnit error in testing, and took a long google search to find that it was due to an outdated dependency for equalsverifier. I updated the dependency and chose to update JUnit and Mockito to more current versions at the same time.

- Struggled with testing and lots of things giving me failures when I have verified that they work directly into SQL through running the program

Sprint 5 - Friday

My sprint for the rest of today, which is to finish off testing and complete the build before sending it off!

Conclusion

- Project works and updates SQL accordingly, however many functions in the order classes aren't as smooth or clean as I'd like them to be, and specifically TotalPrice

- Testing has been a bit struggle as we only learnt Mockito very recently, and I have struggled to test the more completed methods, specifically in OrderController. A lot of time has been spent on testing due to silly things holding back the coverage like the outdated equalsVerifier in the dependencies.

- If I had more time I would have added better access to the price function, more features such as updating stock levels according to customers etc.

Thank you for listening!