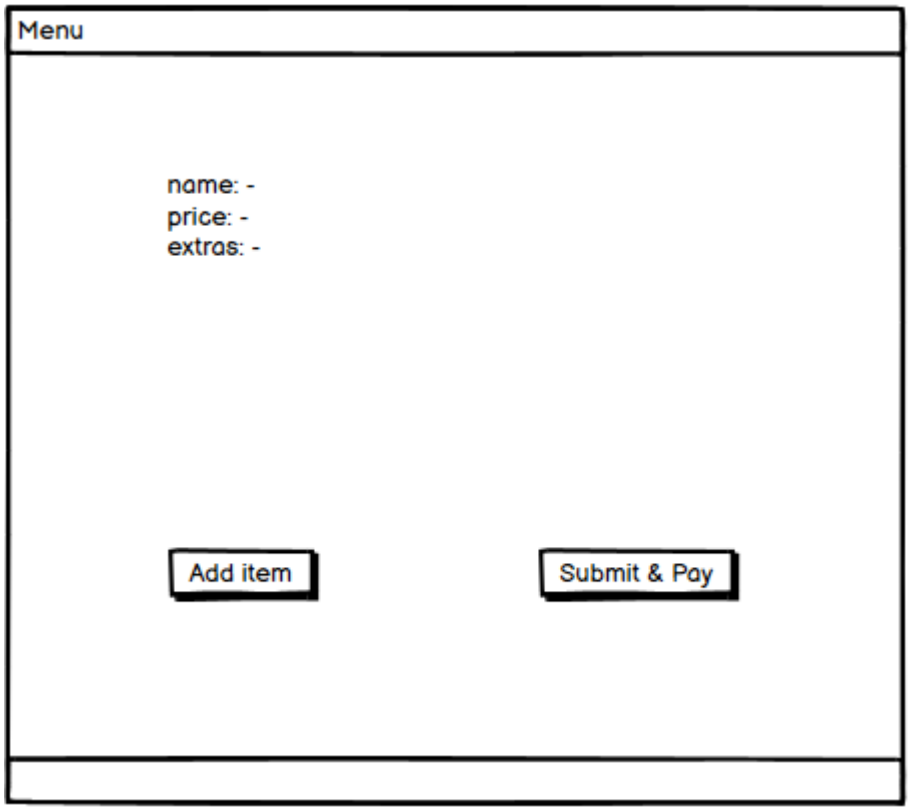


GUI Sketch in case we find this interesting

MVVM

In order to increase the flexibility of the UI, the MVVM architecture was chosen when designing the backbone of the system. This aligns perfectly with UP, as it makes it easier to add extra views and functionality as the system grows bigger, without interfering with already existing classes in the view and viewmodel packages. The responsibilities are divided into three different packages:

1. Model

The Model stores all the data the system is dealing with. It's only responsibility is to provide the data (in this system the items together with their prices, descriptions and possible extras). It does not know the ViewModel, nor the View.

2. View

The view is the part the user interacts with (the fxml files, the viewcontrollers, and the view factory). It's responsibilities include making the data from the Model presentable, and interacting with the user. Every component that will be updated has a binding to a Viewmodel property, where the methods are called. Therefore, the View knows the ViewModel, but not the Model.

3. ViewModel

The ViewModel acts as a connector between the View and the Model, taking the input from the view, then accessing and updating data in the model. In this system the ViewModel would take an order that was created in the View and send it to the Model to be submitted. This means that the ViewModel knows the Model but not the View.

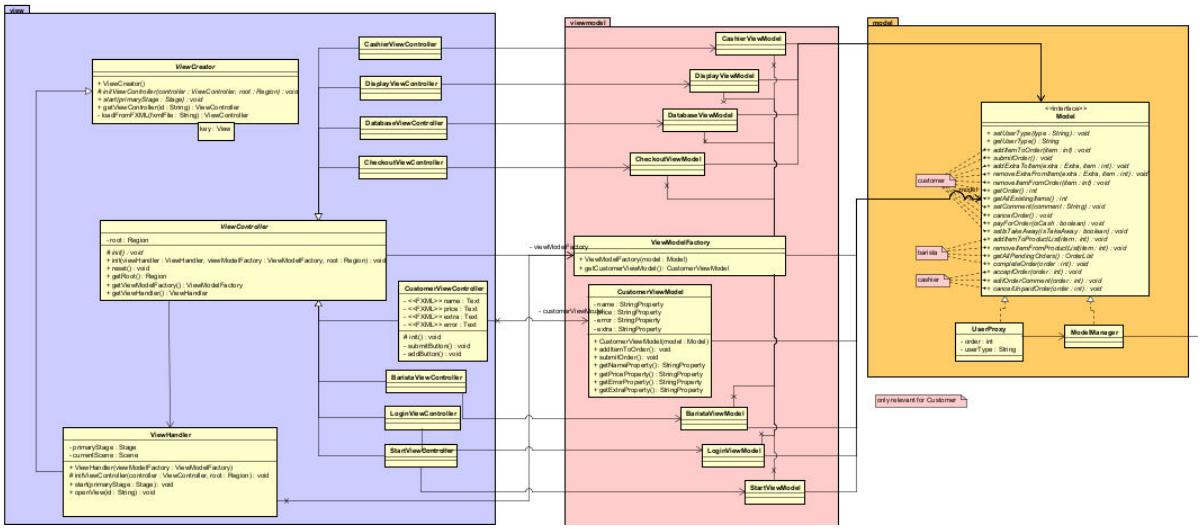
TESTING MVVM

During the testing of the MVVM, the V-model has been followed. JUnit was the first step for making sure all the methods inside the CustomerViewModel class were working correctly, since the viewmodel provides the functionality for the view while accessing and updating data in the model. After testing the constructor, the ZOMB+E was used for the rest of the methods. M, B and E was not tested for the method addItemToOrder(), since only one object can be used at a time, and the Exception was tested when adding a null item to the order. Similarly, submitOrder() was only tested for one, since only one order object can be added at once. Unit testing was very helpful: it highlighted the need for exceptions to be thrown when an empty order was being submitted. The rest of the methods are getters and have been tested to return the expected values.

```
59  @Test void constructor()
60  {
61    CustomerViewModel customerViewModel = new CustomerViewModel(model);
62    assertNotNull(customerViewModel);
63  }
64
65  @Test void addItemToOrder_0() { assertDoesNotThrow(() -> model.addItemToOrder(item)); }
69
70  @Test void addItemToOrder_Z()
71  {
72    assertThrows(Exception.class, () -> model.addItemToOrder(null));
73  }
74
75  @Test void submitOrder_0()
76  {
77    model.addItemToOrder(item);
78    assertDoesNotThrow(() -> model.submitOrder());
79  }
80
81  @Test void submitOrder_Z()
82  {
83    assertThrows(NullPointerException.class, () -> model.submitOrder());
84  }
85
86  @Test void getNameProperty_0() { assertEquals(name.get(),  actual: "Latte Macchiato"); }
90
```

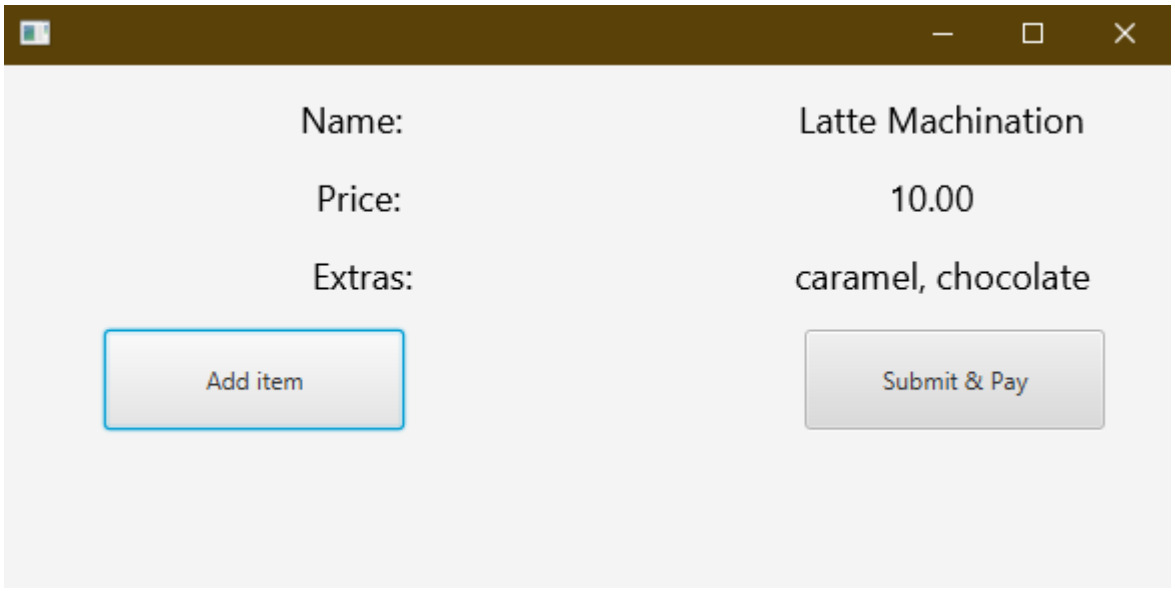
This is the only class that has been unit tested, followed by the integration test as there is a necessity to run the JavaFX components and the methods inside the ViewController class are private, thus inaccessible inside a Test class. As a result of that, the test is done by simply creating system outputs in the places where it is necessary to track the completion of the tested method. Those system outputs are visible inside the console.

MVVM class diagram



To understand the complexity of the MVVM and therefore testing it, the MVVM part of the class diagram is shown. The method is being called in the ViewController and should be delegated up to the Model.

Adding item window



In this GUI the adding of the item to the order and submitting it will occur. When the “Add item” and “Submit & Pay” buttons are pressed the CustomerViewController delegates the methods to the CustomerViewModel and then to the Model. The completion of this is assessed by a proper system output which is shown below in a console. Moreover the methods with those system outputs from the ModelManager class are also shown below.

Console output

```
"C:\Program Files\Java\jdk-11.0.2\bin\java.exe" ...
maj 05, 2022 1:06:04 PM com.sun.javafx.application.PlatformImpl startup
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @2c4a2673'
I am adding an item: Item{id=6, name='Latte Machination', type='coffee', price=10.0, description='', extras=[]}
I am submitting order
```

Adding output in the ModelManager

```
@Override public void addItemToOrder(Item item)
{
  if (order==null)
  {
    order= new Order( paidWithCash: false);
  }
  System.out.println("I am adding an item: " + item);
  order.addItem(item);
}
```

Submitting output in the ModelManager

```
@Override public void submitOrder()
{
  try
  {
    System.out.println("I am submitting order");
    client.receiveOrder(order);
  }
  catch (RemoteException e)
  {
    e.printStackTrace();
  }
}
```

TESTING RMI

The RMI testing will be conducted by the usage of integration testing as there is a need to check the compatibility between the client side and server side. This will be done by displaying outputs in the console from the server side, if the method runs without any errors. When the method from the RemoteCafeServer interface on the client side is called, if the connection is correct, the RemoteServer invokes a method and the logic is executed. Below there are shown accurate code snippets.

Console output

```
"C:\Program Files\Java\jdk-11.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2020.2\lib\idea_rt.jar=62559:C:\P
Registry started...
Server started...
I have correctly received this:
Order{itemList=ItemList{items=[Item{id=6, name='Latte Machination', type='coffee', price=10.0, description='', extras=[]}]}, comment='', dateTime=05/05/
```

Tested method

```
@Override public void receiveOrder(Order order) throws RemoteException
{
  try {
    System.out.println("I have correctly received this:");
    System.out.println(order);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

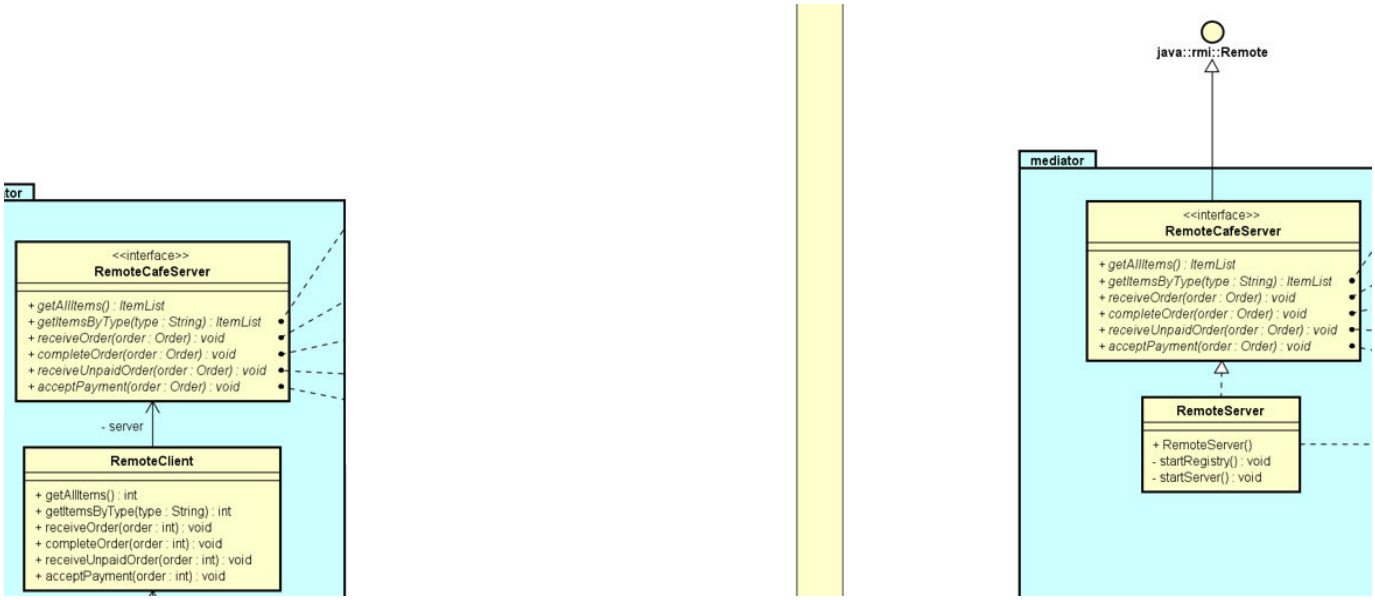
Order toString method

```
@Override public String toString()
{
  return "Order{" + "itemList=" + itemList + ", comment='" + comment + '\''
      + ", dateTime=" + dateTime + ", price=" + price + ", status='" + status
      + '\'' + '}';
}
```

DOCUMENTATION OF IMPLEMENTATION RMI

In order to get a better understanding of the RMI implementation it is crucial to show a Class Diagram for this part. RemoteCafeServer interface contains all the essential methods for the system, which would be implemented by RemoteServer and later called by RemoteClient using the interface.

RMI Class Diagram



RemoteCafeServer interface

```
public interface RemoteCafeServer extends Remote
{
  ItemList getAllItems() throws RemoteException;
  ItemList getItemsByType(String type) throws RemoteException;
  void receiveOrder(Order order) throws RemoteException;
  void completeOrder(Order order) throws RemoteException;
  void acceptPayment(Order order) throws RemoteException;
}
```

The constructor of RemoteServer class

```
public class RemoteServer implements RemoteCafeServer

{

  public RemoteServer()
      throws RemoteException, MalformedURLException, SQLException
  {
    startRegistry();
    startServer();
  }
```

StartRegistry and Server methods

```
private void startRegistry() throws RemoteException
{
  try
  {
    Registry reg = LocateRegistry.createRegistry( port: 1099);
    System.out.println("Registry started...");
  }
  catch (java.rmi.server.ExportException e)
  {
    System.out.println("Registry already started? " + e.getMessage());
  }
}

private void startServer() throws RemoteException, MalformedURLException
{
  UnicastRemoteObject.exportObject( obj: this,  port: 0);
  Naming.rebind( name: "Cafe",  obj: this);
  System.out.println("Server started...");
}
```

Those two methods start the registry and server with the specified name. This allows the client to connect to the server, which happens in the following way.

```
public class RemoteClient
{
  private RemoteCafeServer server;

  public RemoteClient()
      throws MalformedURLException, NotBoundException, RemoteException
  {
    server = (RemoteCafeServer)
        Naming.lookup( name: "rmi://localhost:1099/Cafe");
    //UnicastRemoteObject.exportObject(this,0);
    //This will be needed when we will be doing the RMI Observer
  }
```

Method in the RemoteClient

```
public void receiveOrder(Order order) throws RemoteException
{
  server.receiveOrder(order);
}
```

There is no logic inside the RemoteClient class as all of the methods are calling the according method from the interface. Therefore, all of the work is happening on the server side and the requested information is returned.

Method in the RemoteServer

```
@Override public void receiveOrder(Order order) throws RemoteException
{
  try {
    ConcreteOrderDAO.getInstance().create(order);
    System.out.println("I have correctly received this:");
    System.out.println(order);
  }
  catch (SQLException e) {
    e.printStackTrace();
  }

}
```