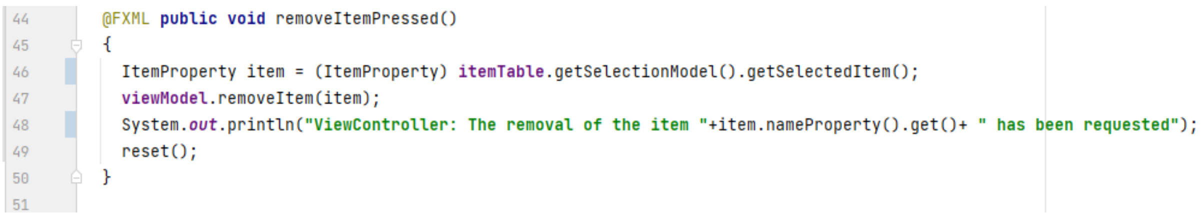
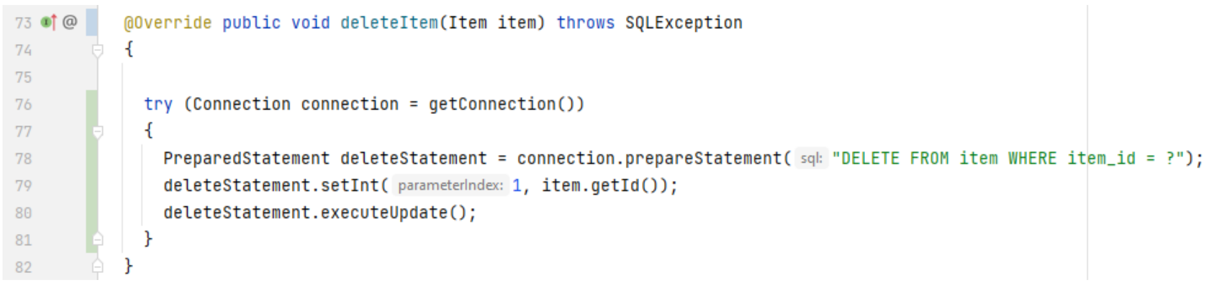
Remove Item From Product List

The removal of an item from the list of products is done by the admin selecting an item from the list in the DatabaseView, which has been implemented as seen in the following code snippet:



```java
@FXML public void removeItemPressed()
{
  ItemProperty item = (ItemProperty) itemTable.getSelectionModel().getSelectedItem();
  viewModel.removeItem(item);
  System.out.println("ViewController: The removal of the item "+item.nameProperty().get()+ " has been requested");
  reset();
}
```

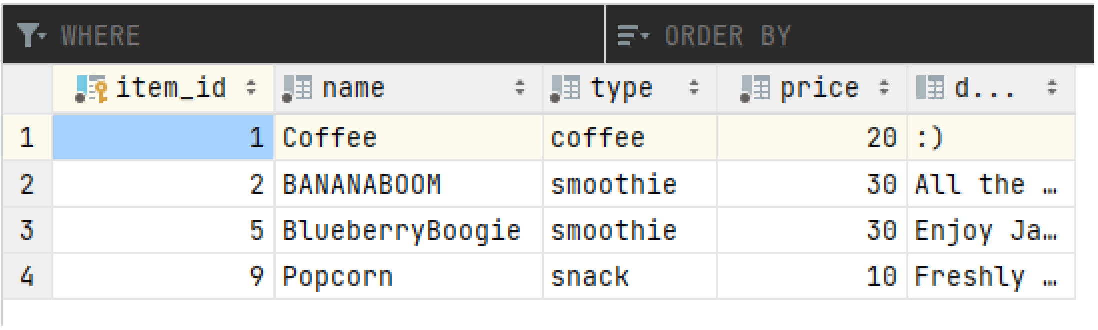
With respect to the MVVM structure, instead of depending on the model packages' Item class, an ItemProperty class has been implemented and used for this method.
This item is passed on to the DatabaseViewModel, which calls the model, sending the item to the server through an RMI connection. The item is then further sent from the CafeDatabase to the ConcreteItemDAO where the item is removed by the method deleteItem(), using the ID of the item as that is unique, to prevent removing an item without the user's intention.



```java
@Override public void deleteItem(Item item) throws SQLException
{

  try (Connection connection = getConnection())
  {
    PreparedStatement deleteStatement = connection.prepareStatement( sql: "DELETE FROM item WHERE item_id = ?");
    deleteStatement.setInt( parameterIndex: 1, item.getId());
    deleteStatement.executeUpdate();
  }
}
```

Integration Testing (REMOVE ITEM FROM PL):

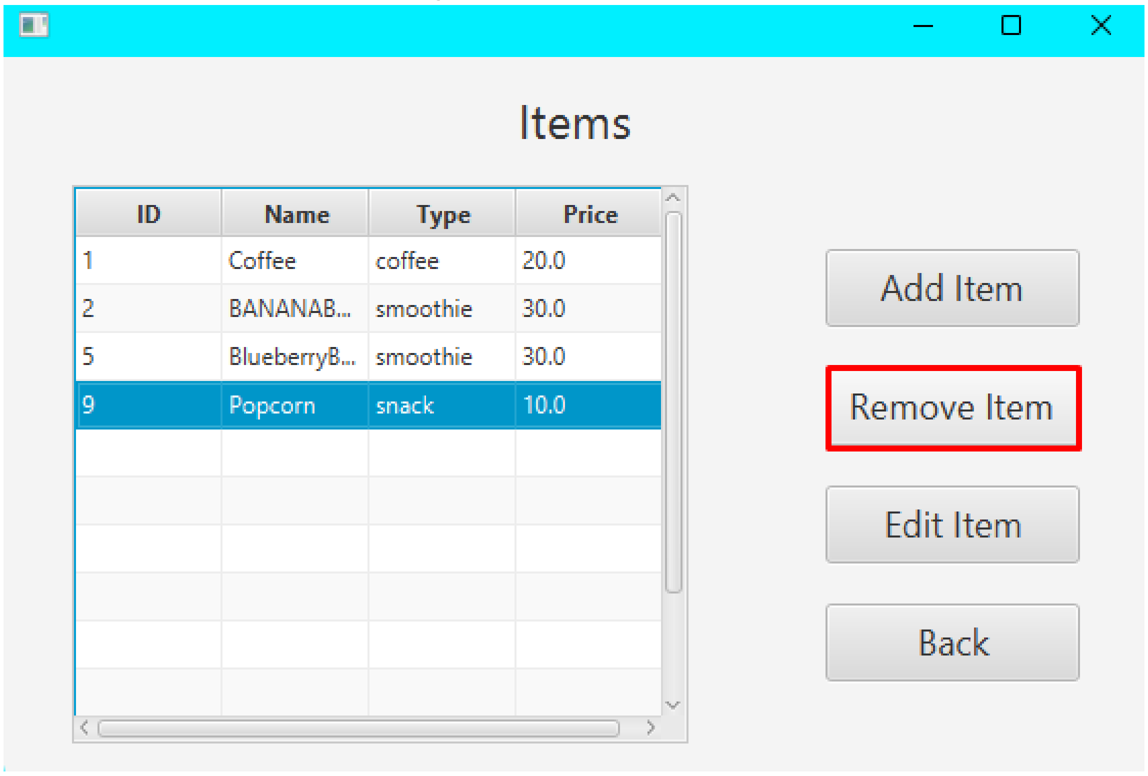
Database items before removing the item "Popcorn"



WHERE ORDER BY

	item_id	name	type	price	d...
1	1	Coffee	coffee	20	:)
2	2	BANANABOOM	smoothie	30	All the …
3	5	BlueberryBoogie	smoothie	30	Enjoy Ja…
4	9	Popcorn	snack	10	Freshly …

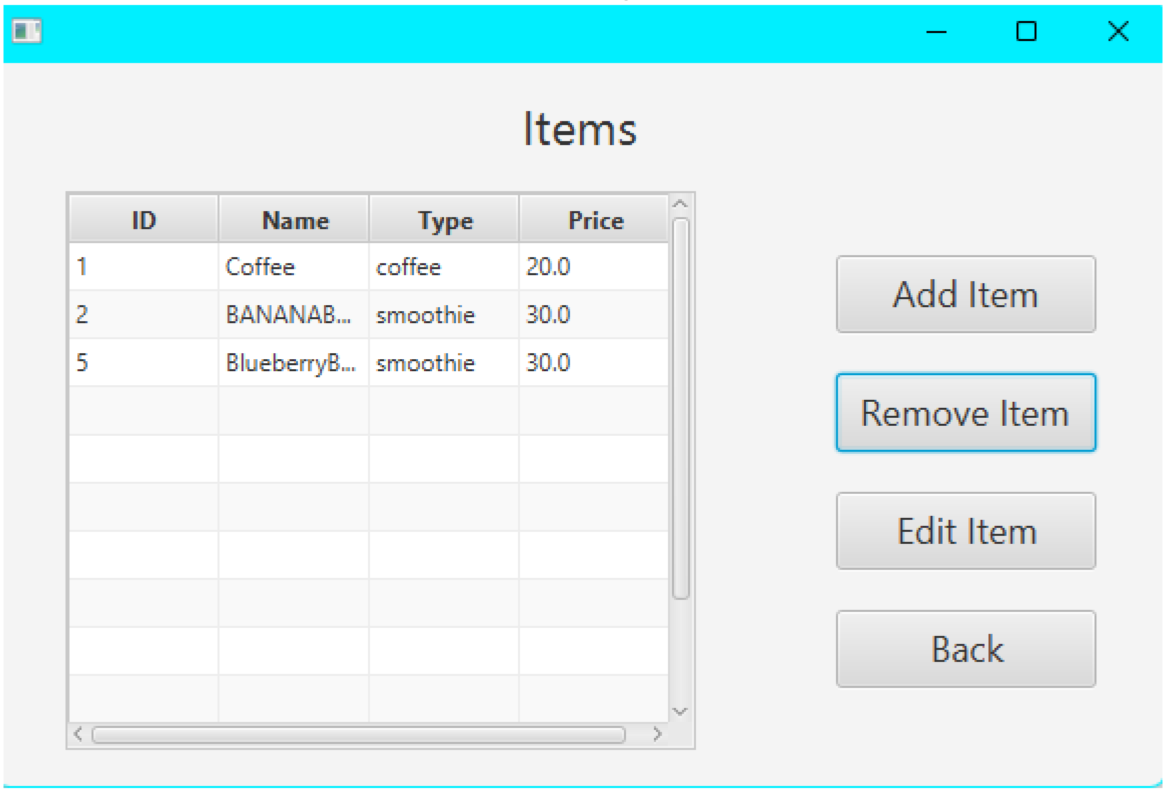
The admin clicking "Remove Item" for the item "Popcorn"



Items

ID	Name	Type	Price
1	Coffee	coffee	20.0
2	BANANAB...	smoothie	30.0
5	BlueberryB...	smoothie	30.0
9	Popcorn	snack	10.0

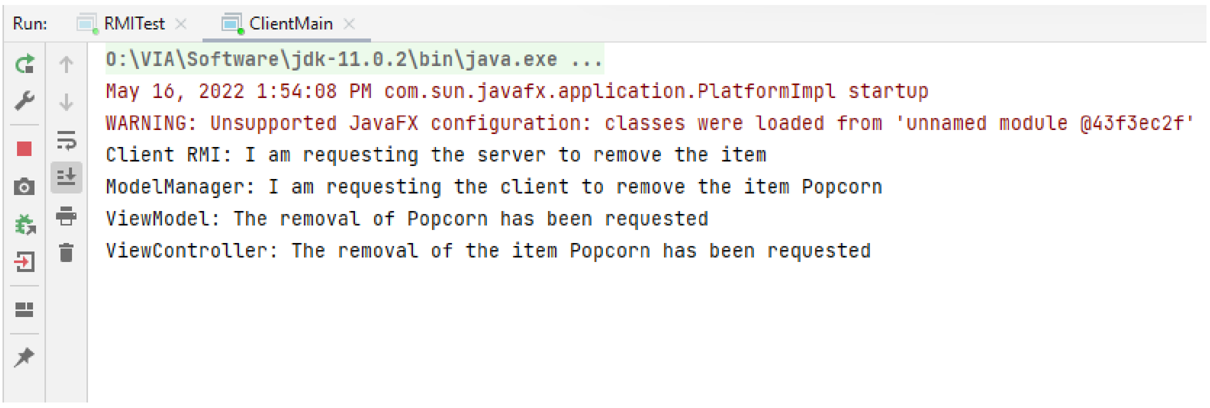
The "Popcorn" item is no longer present in the view



Items

ID	Name	Type	Price
1	Coffee	coffee	20.0
2	BANANAB...	smoothie	30.0
5	BlueberryB...	smoothie	30.0

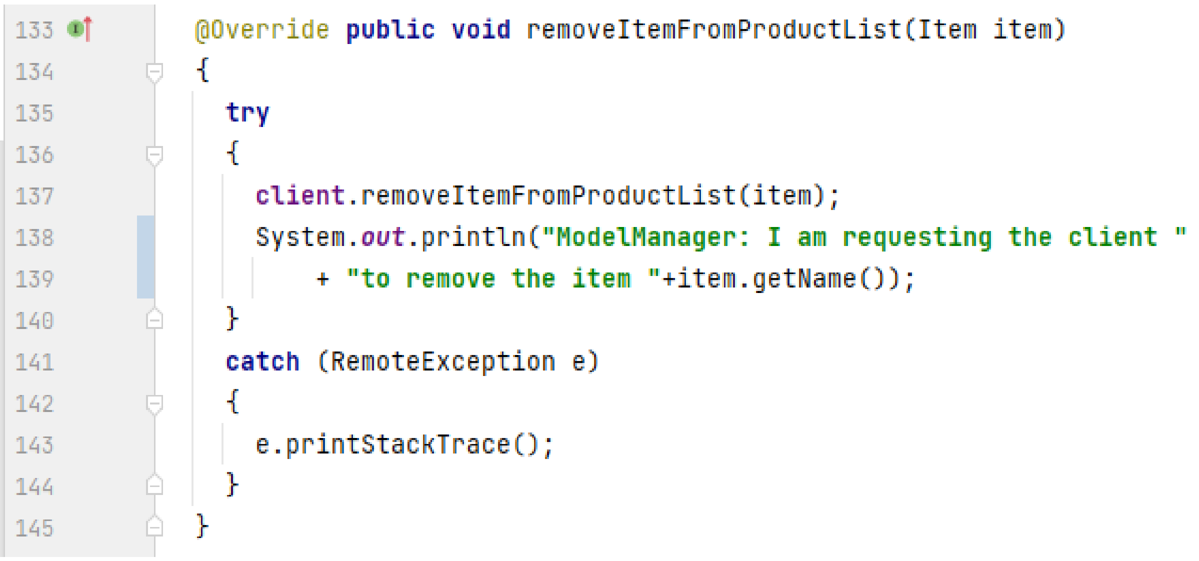
Printouts on the Client Side



```
O:\VIA\Software\jdk-11.0.2\bin\java.exe ...
May 16, 2022 1:54:08 PM com.sun.javafx.application.PlatformImpl startup
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @43f3ec2f'
Client RMI: I am requesting the server to remove the item
ModelManager: I am requesting the client to remove the item Popcorn
ViewModel: The removal of Popcorn has been requested
ViewController: The removal of the item Popcorn has been requested
```

Noticeably, the print outs are shown in the opposite of the expected order. The reason for this is that the printouts are called after the methods are executed, as shown in the following code snippet:

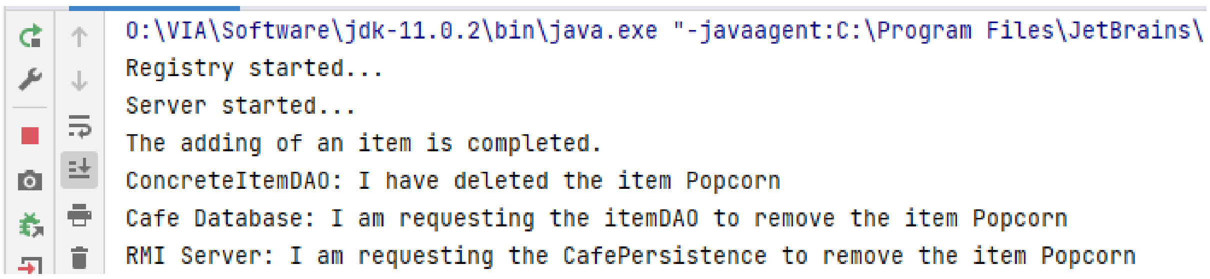
The print outs following the methods to be tested



```
@Override public void removeItemFromProductList(Item item)
{
  try
  {
    client.removeItemFromProductList(item);
    System.out.println("ModelManager: I am requesting the client "
        + "to remove the item "+item.getName());
  }
  catch (RemoteException e)
  {
    e.printStackTrace();
  }
}
```

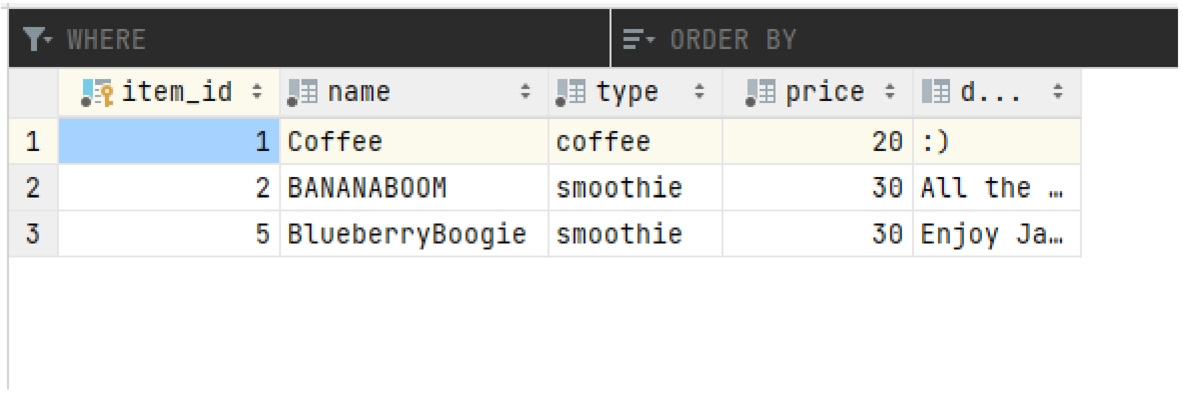
The item goes all the way to the RMI, and then the printouts are called in each of the classes, in a backwards order. This way, the print outs are only done once the methods have been successfully executed; the same is true for the print outs on the server side.

Printouts on the Server Side



```
O:\VIA\Software\jdk-11.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\
Registry started...
Server started...
The adding of an item is completed.
ConcreteItemDAO: I have deleted the item Popcorn
Cafe Database: I am requesting the itemDAO to remove the item Popcorn
RMI Server: I am requesting the CafePersistence to remove the item Popcorn
```

Database items after removing the item “Popcorn”

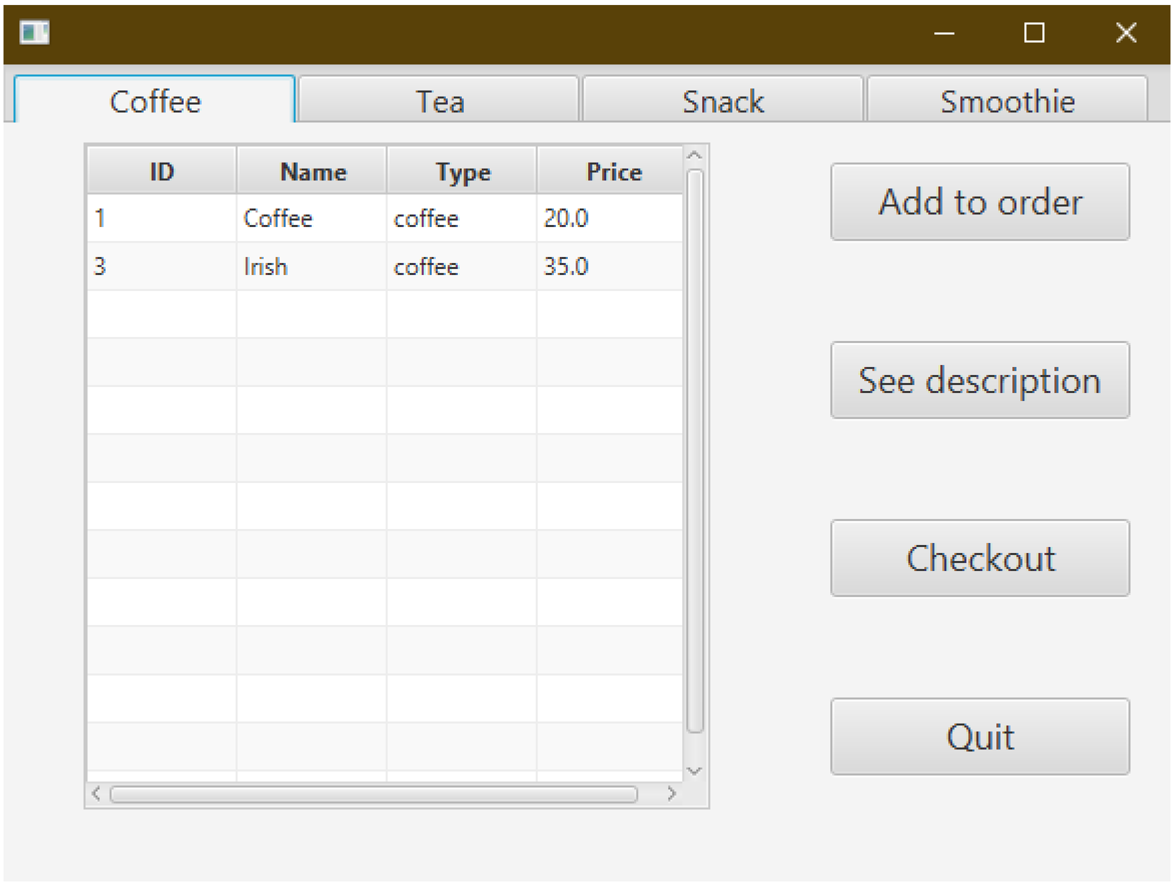

WHERE
ORDER BY

	item_id	name	type	price	d...
1	1	Coffee	coffee	20	:)
2	2	BANANABOOM	smoothie	30	All the …
3	5	BlueberryBoogie	smoothie	30	Enjoy Ja…

CustomerView documentation - design and implementation

Coming from the product backlog, one of the most crucial parts of the problem domain is for the customer to see all of the available items in the Campus Cafe. In order to solve this issue, the design of the proper GUI window has to be done. All of the necessary elements relevant for this use case need to be incorporated into the design process.

CustomerView window

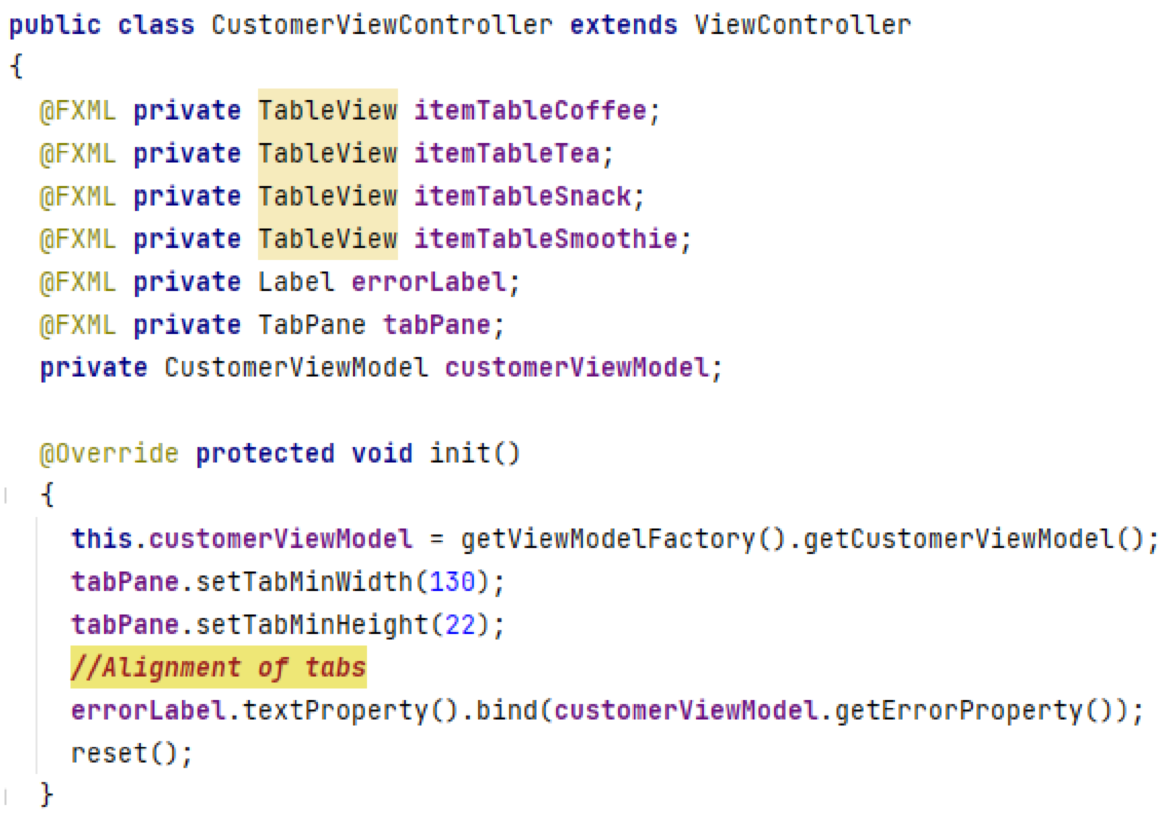


ID	Name	Type	Price
1	Coffee	coffee	20.0
3	Irish	coffee	35.0

After the analysis part of the system, it was known that items are divided into four different groups: coffee, tea, snack and smoothie. Therefore, displaying the items sorted by the group is an accurate solution. To achieve this, a JavaFX component is used - TabPane. Each tab contains a TableView presenting items of the same type. Besides that, there are four buttons which are relevant for the purpose of this view.

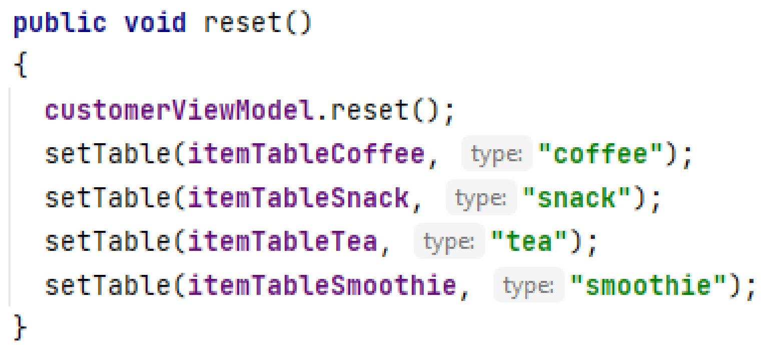
The implementation of the Customer View starts with extending the ViewController superclass. All of the FXML elements are introduced, four TableViews, one for each type, label for error and a TabPane. In order to delegate methods and follow the MVVM architecture pattern, there is an association with the responding View Model. The overridden method init() gets the view model, aligns the TabPane, binding with the error property from View Model occurs and lastly the reset() method is called.

Fields and init method

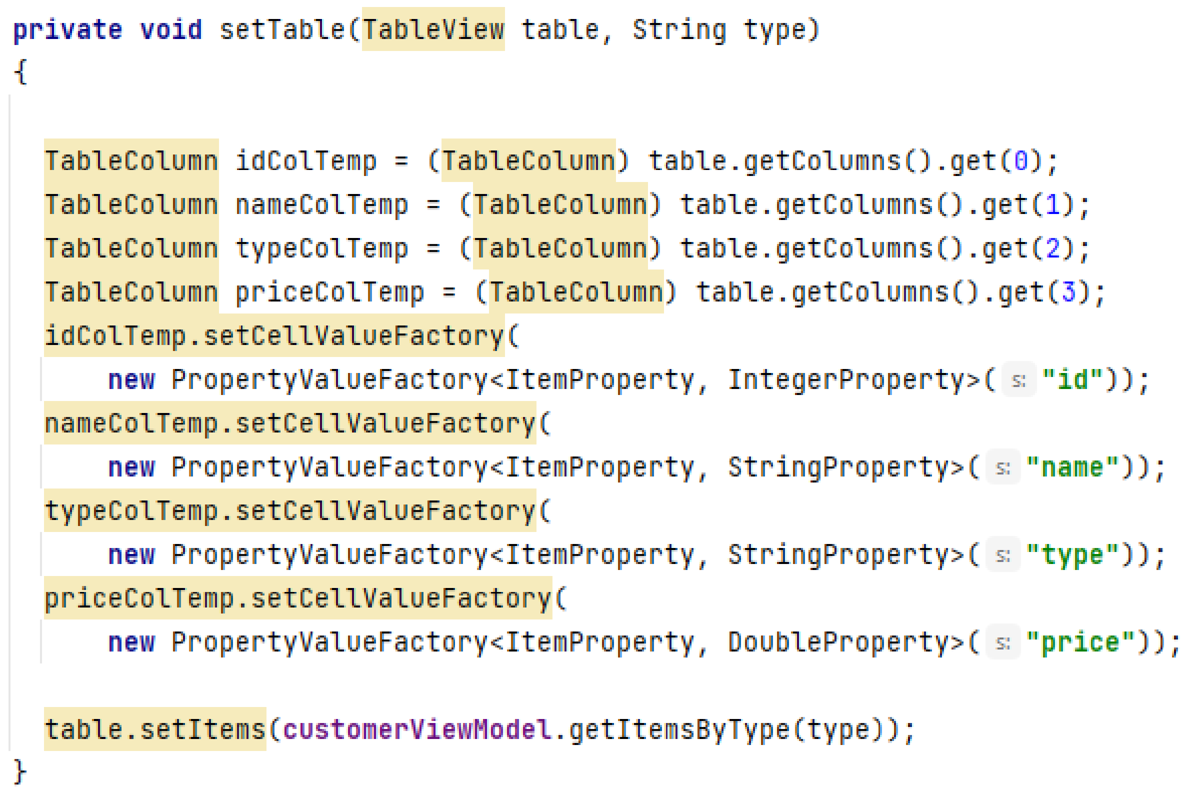


```java
public class CustomerViewController extends ViewController
{
  @FXML private TableView itemTableCoffee;
  @FXML private TableView itemTableTea;
  @FXML private TableView itemTableSnack;
  @FXML private TableView itemTableSmoothie;
  @FXML private Label errorLabel;
  @FXML private TabPane tabPane;
  private CustomerViewModel customerViewModel;

  @Override protected void init()
  {
    this.customerViewModel = getViewModelFactory().getCustomerViewModel();
    tabPane.setTabMinWidth(130);
    tabPane.setTabMinHeight(22);
    //Alignment of tabs
    errorLabel.textProperty().bind(customerViewModel.getErrorProperty());
    reset();
  }
```

The reset method not only delegates the reset to the View Model, but also calls the setTable method for each TableView. The setTable method takes two arguments and this is the TableView to be filled with information and the String type of the items to be queried for. Initial step inside the method is to get the columns from the table and set their Value Property according to the ItemProperty. After that, the TableView's items are obtained from the View Model and set. The ViewModel's getItemsByType method returns an ObservabaleList.



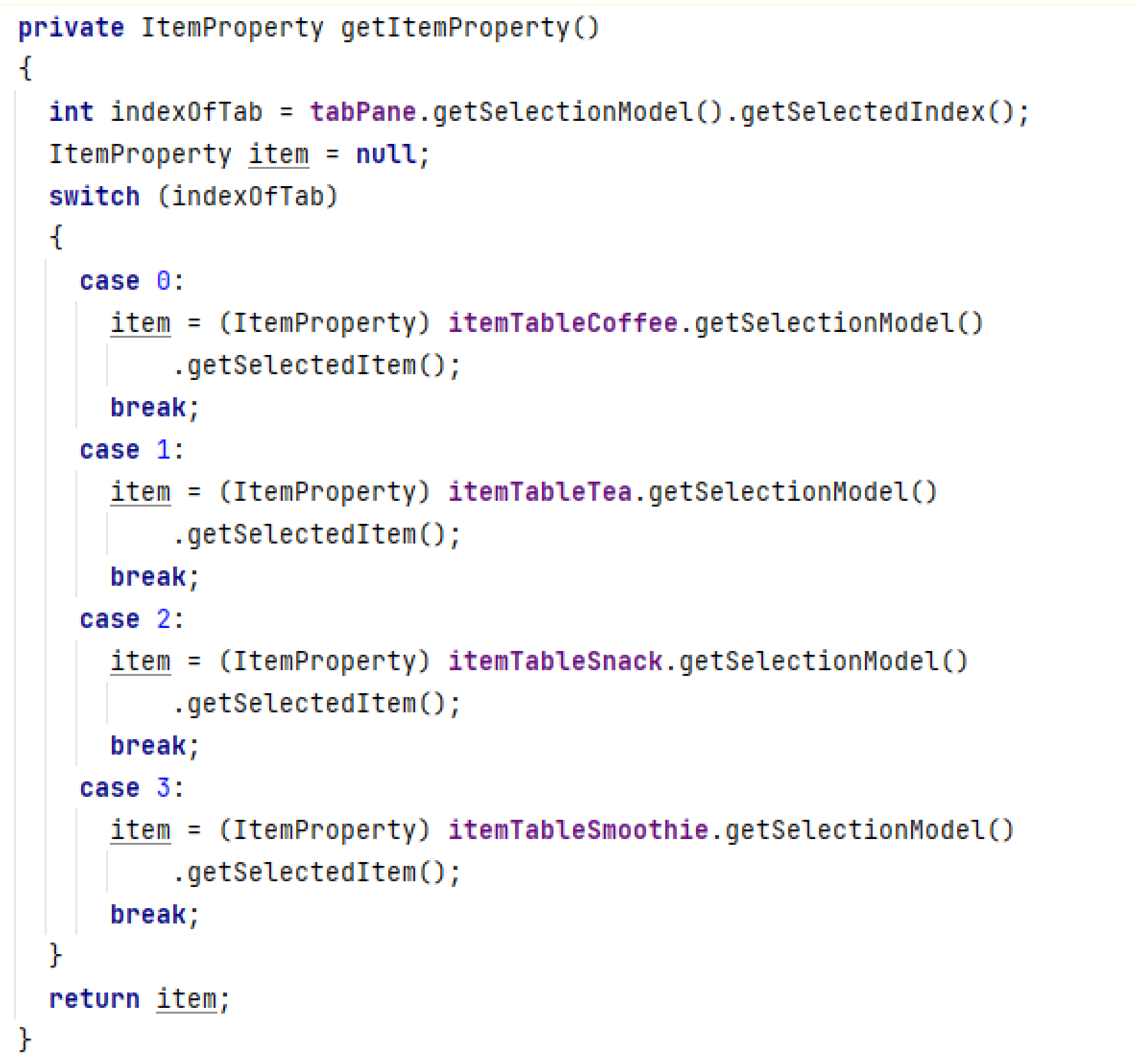
```
public void reset()
{
  customerViewModel.reset();
  setTable(itemTableCoffee,  type: "coffee");
  setTable(itemTableSnack,  type: "snack");
  setTable(itemTableTea,  type: "tea");
  setTable(itemTableSmoothie,  type: "smoothie");
}
```



```
private void setTable(TableView table, String type)
{

  TableColumn idColTemp = (TableColumn) table.getColumns().get(0);
  TableColumn nameColTemp = (TableColumn) table.getColumns().get(1);
  TableColumn typeColTemp = (TableColumn) table.getColumns().get(2);
  TableColumn priceColTemp = (TableColumn) table.getColumns().get(3);
  idColTemp.setCellValueFactory(
      new PropertyValueFactory<ItemProperty, IntegerProperty>( s: "id"));
  nameColTemp.setCellValueFactory(
      new PropertyValueFactory<ItemProperty, StringProperty>( s: "name"));
  typeColTemp.setCellValueFactory(
      new PropertyValueFactory<ItemProperty, StringProperty>( s: "type"));
  priceColTemp.setCellValueFactory(
      new PropertyValueFactory<ItemProperty, DoubleProperty>( s: "price"));

  table.setItems(customerViewModel.getItemsByType(type));
}
```

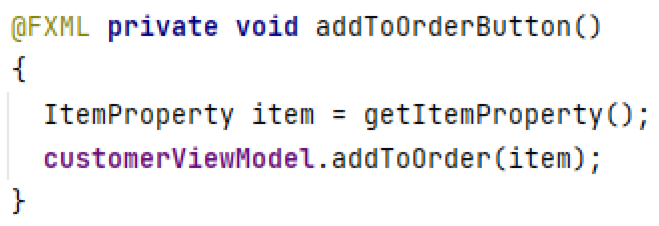
With a focus on choosing the correct item to perform actions, it is necessary to know which tab is actually selected by the user. This is possible by getting the selected index of the tab which allows to make a switch statement to create an ItemProperty from the selected item from the appropriate TableView.

getItemProperty method

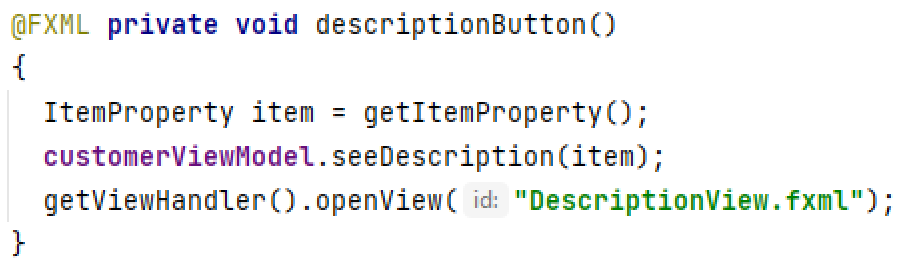


```java
private ItemProperty getItemProperty()
{
    int indexOfTab = tabPane.getSelectionModel().getSelectedIndex();
    ItemProperty item = null;
    switch (indexOfTab)
    {
        case 0:
            item = (ItemProperty) itemTableCoffee.getSelectionModel()
                    .getSelectedItem();
            break;
        case 1:
            item = (ItemProperty) itemTableTea.getSelectionModel()
                    .getSelectedItem();
            break;
        case 2:
            item = (ItemProperty) itemTableSnack.getSelectionModel()
                    .getSelectedItem();
            break;
        case 3:
            item = (ItemProperty) itemTableSmoothie.getSelectionModel()
                    .getSelectedItem();
            break;
    }
    return item;
}
```

Adding an item to the order happens by getting the selected item and delegating this to the ViewModel. This is the same case for the descriptionButton.

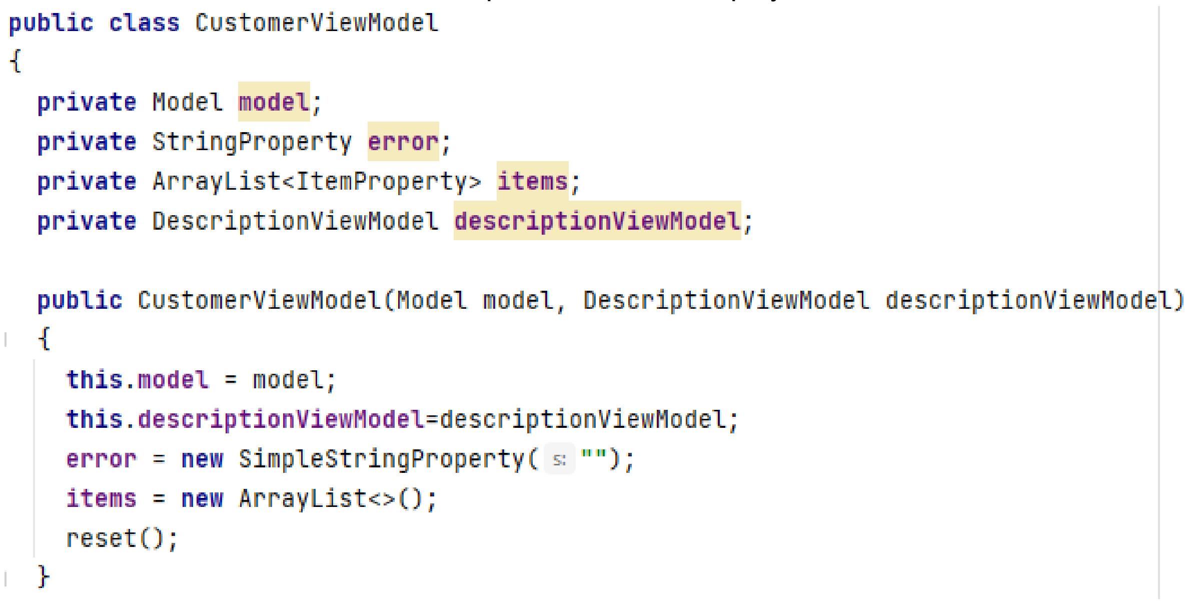


```java
@FXML private void addToOrderButton()
{
  ItemProperty item = getItemProperty();
  customerViewModel.addToOrder(item);
}
```

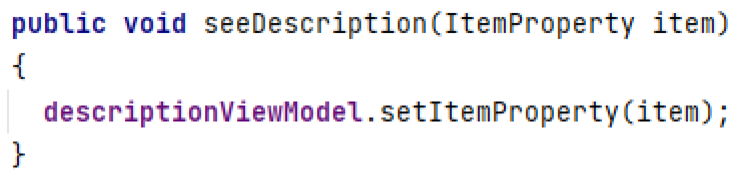


```java
@FXML private void descriptionButton()
{
  ItemProperty item = getItemProperty();
  customerViewModel.seeDescription(item);
  getViewHandler().openView( id: "DescriptionView.fxml");
}
```

It should be noted that the CustomerViewModel has an association with the DescriptionViewModel and it is initialised in the constructor. This is essential in order to call the method setItemProperty from the mentioned ViewModel as this is the way to transfer the information about which item's description should be displayed.

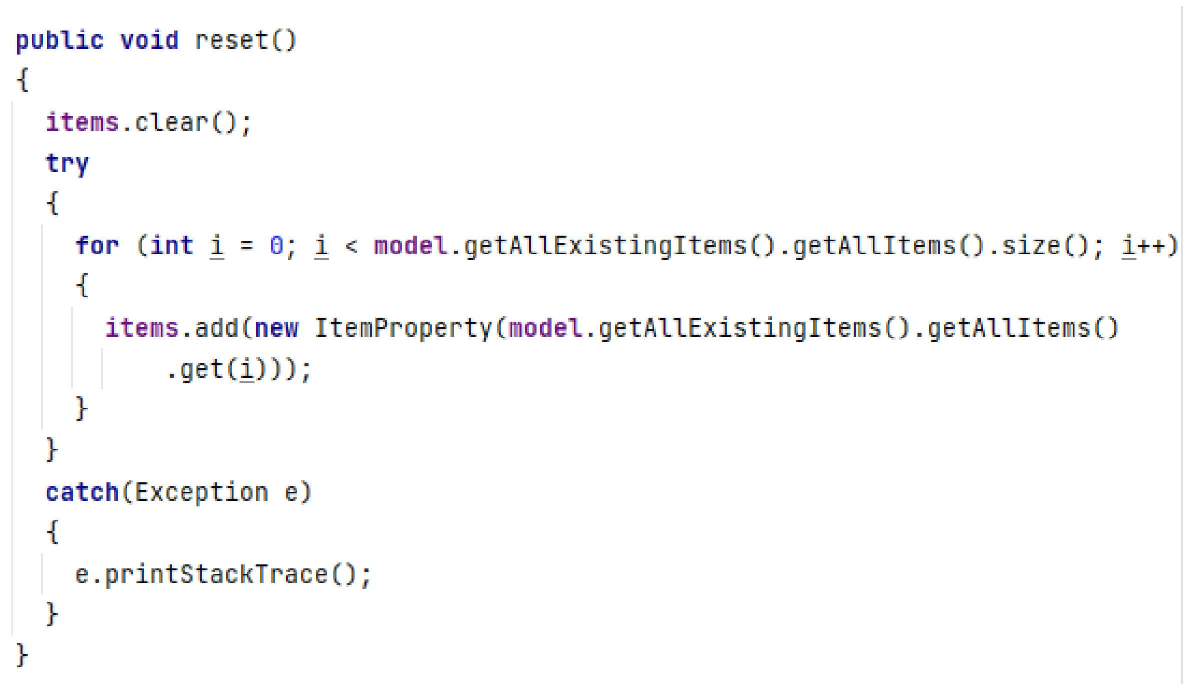


```java
public class CustomerViewModel
{
  private Model model;
  private StringProperty error;
  private ArrayList<ItemProperty> items;
  private DescriptionViewModel descriptionViewModel;

  public CustomerViewModel(Model model, DescriptionViewModel descriptionViewModel)
  {
    this.model = model;
    this.descriptionViewModel=descriptionViewModel;
    error = new SimpleStringProperty( s: "");
    items = new ArrayList<>();
    reset();
  }
```



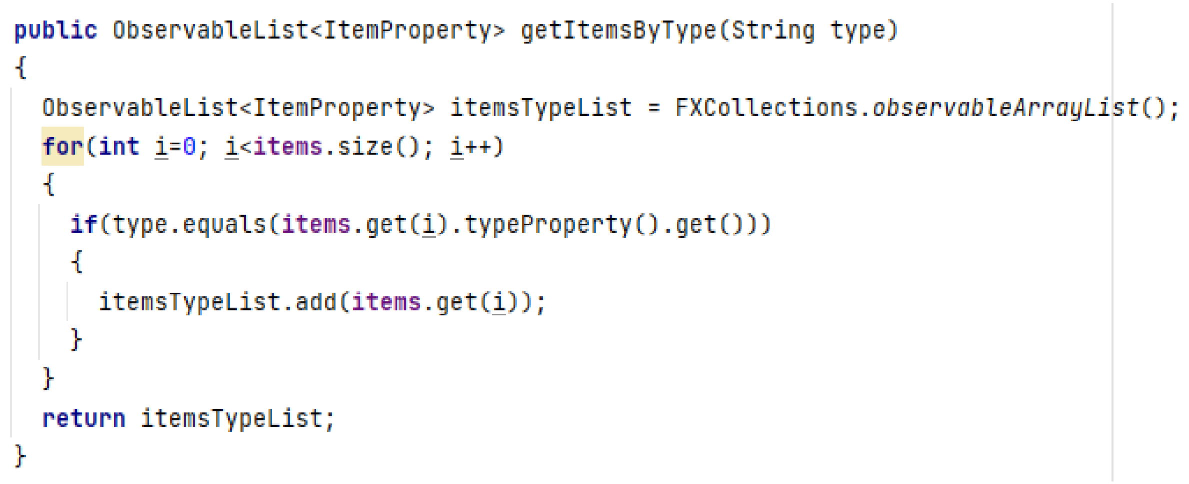
```java
public void seeDescription(ItemProperty item)
{
  descriptionViewModel.setItemProperty(item);
}
```

The CustomerViewModel contains the ArrayList of type ItemProperty and inside the reset method it retrieves the information from the model. Each time the reset is called, the ArrayList is cleared for the purpose of not having duplicates.



```java
public void reset()
{
    items.clear();
    try
    {
        for (int i = 0; i < model.getAllExistingItems().getAllItems().size(); i++)
        {
            items.add(new ItemProperty(model.getAllExistingItems().getAllItems()
                    .get(i)));
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
```

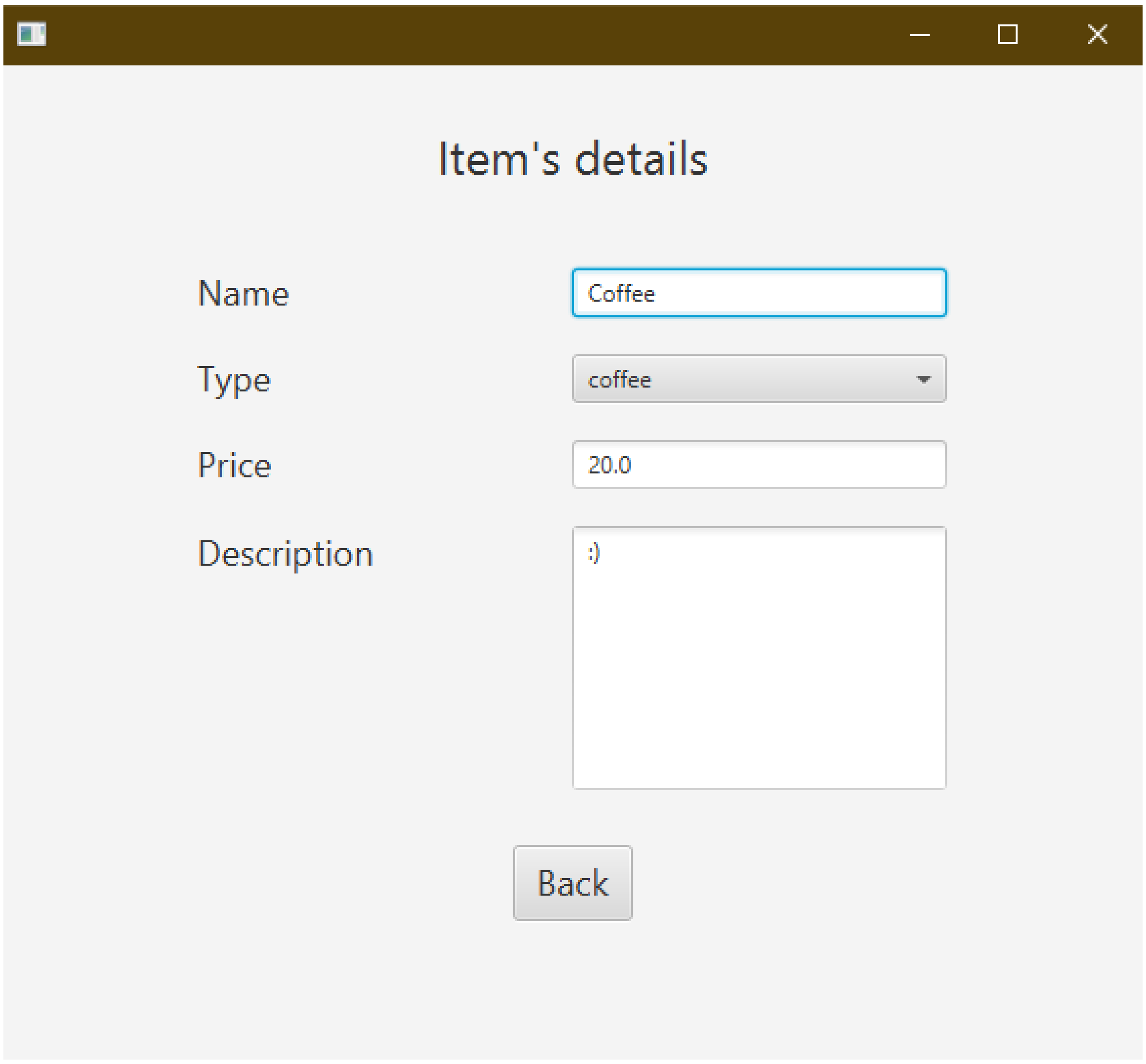
The getItemsByType method returns an Observable List of ItemProperty containing items of the specified type. The FXCollections' observable Array List was used.



```java
public ObservableList<ItemProperty> getItemsByType(String type)
{
    ObservableList<ItemProperty> itemsTypeList = FXCollections.observableArrayList();
    for(int i=0; i<items.size(); i++)
    {
        if(type.equals(items.get(i).typeProperty().get()))
        {
            itemsTypeList.add(items.get(i));
        }
    }
    return itemsTypeList;
}
```

Description View - design and implementation

The Description view is essential for a user to see the description of an item as this one of product backlog items. The view is simple and the only action that a user can do is to go back to the CustomerView.


Item's details

Name: Coffee

Type: coffee

Price: 20.0

Description: :)

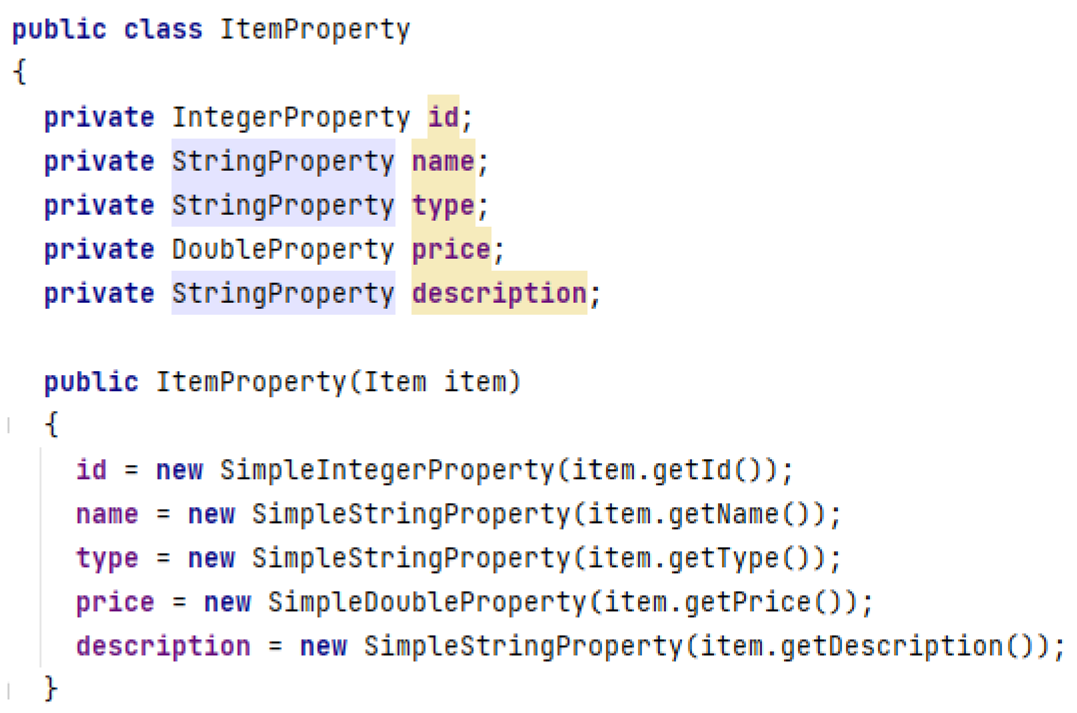
Inside the ViewModel a method setItemProperty called from the CustomerViewModel, sets the properties to the values of the ItemProperty.

```java
public void setItemProperty(ItemProperty item)
{
  name.set(item.nameProperty().get());
  chosen = (item.typeProperty());
  price.set(String.valueOf(item.priceProperty().get()));
  description.set(item.descriptionProperty().get());
}
```

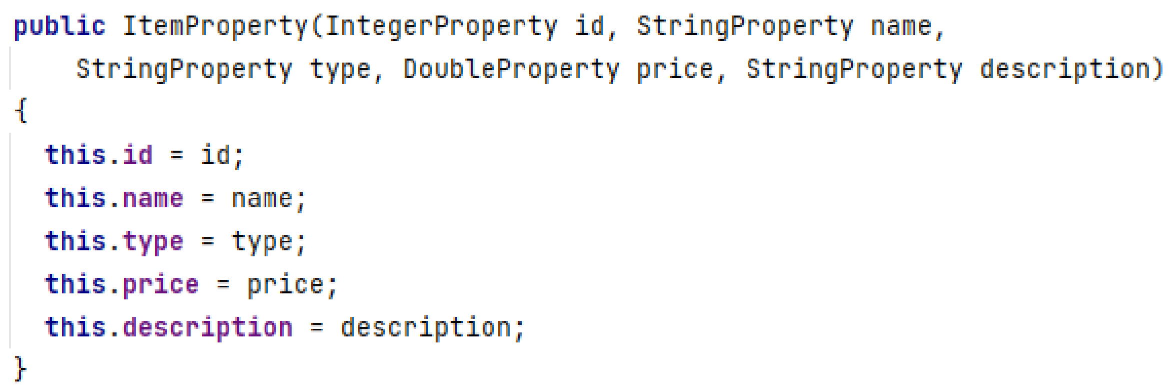
ItemProperty and OrderProperty - design and implementation

In some of the cases inside the system both the View Model and View Controller need to create, for instance, an Item object in order to interact with the Model. Nevertheless, creation of objects from the utility package by those classes violates the MVVM design pattern which clearly states that the View Model should only communicate with the Model and View Controller should only communicate with the View Model. Therefore, there is a need to introduce a Property class containing the relevant information from the classes from the utility package.

The ItemProperty class has relevant fields from the Item class stored as responding Property objects. There are two constructors, one taking the Item object and translating it into the properties. The second one is setting the values from the given properties.

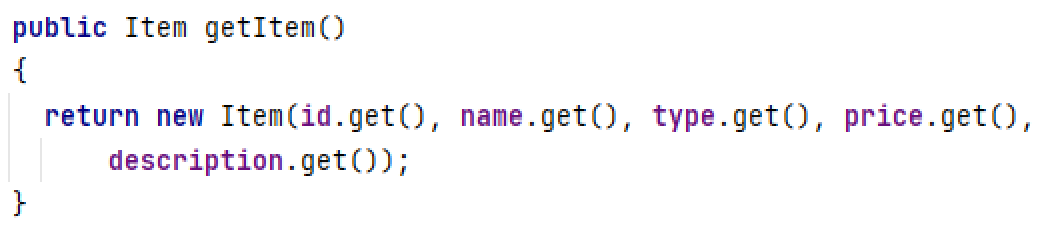


```
public class ItemProperty
{
  private IntegerProperty id;
  private StringProperty name;
  private StringProperty type;
  private DoubleProperty price;
  private StringProperty description;

  public ItemProperty(Item item)
  {
    id = new SimpleIntegerProperty(item.getId());
    name = new SimpleStringProperty(item.getName());
    type = new SimpleStringProperty(item.getType());
    price = new SimpleDoubleProperty(item.getPrice());
    description = new SimpleStringProperty(item.getDescription());
  }
```



```
public ItemProperty(IntegerProperty id, StringProperty name,
    StringProperty type, DoubleProperty price, StringProperty description)
{
  this.id = id;
  this.name = name;
  this.type = type;
  this.price = price;
  this.description = description;
}
```

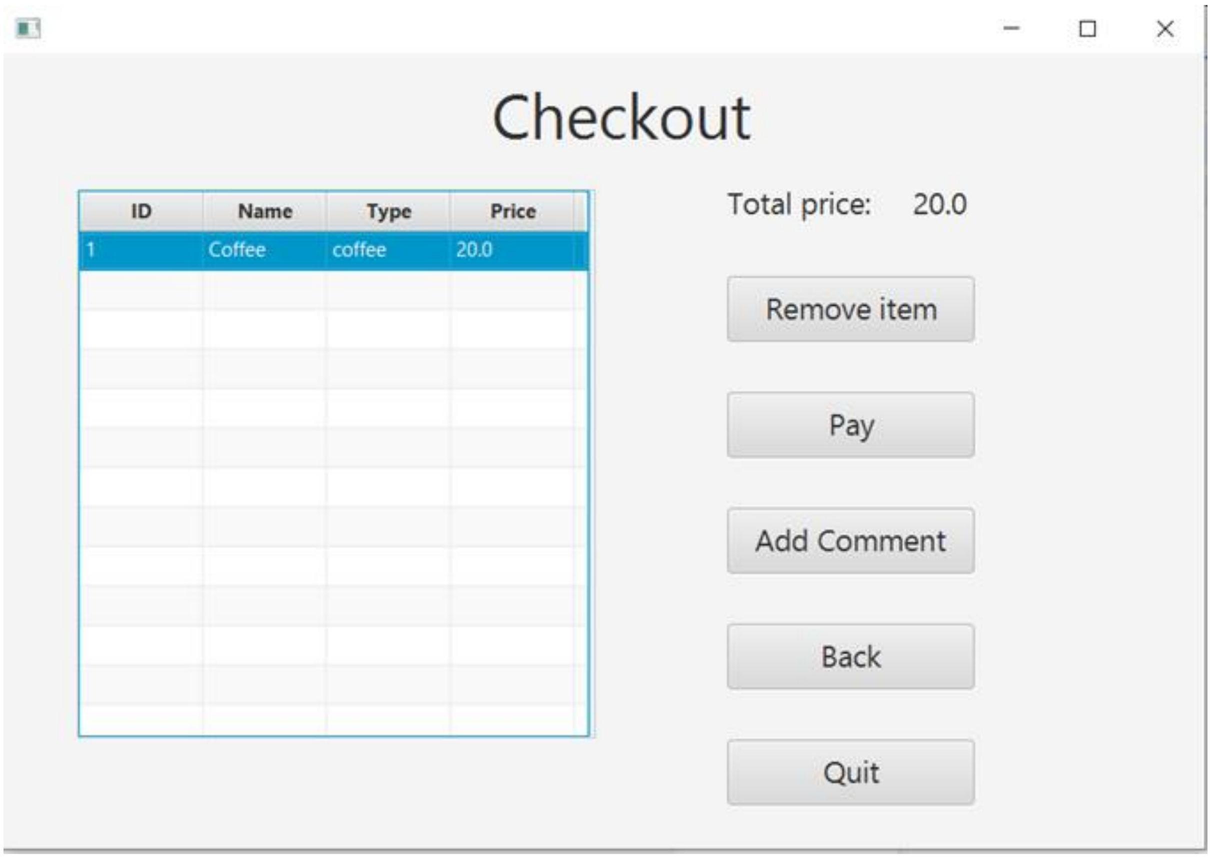
Method worth mentioning is getItem which creates a new Item object from the given properties.



```java
public Item getItem()
{
  return new Item(id.get(), name.get(), type.get(), price.get(),
      description.get());
}
```

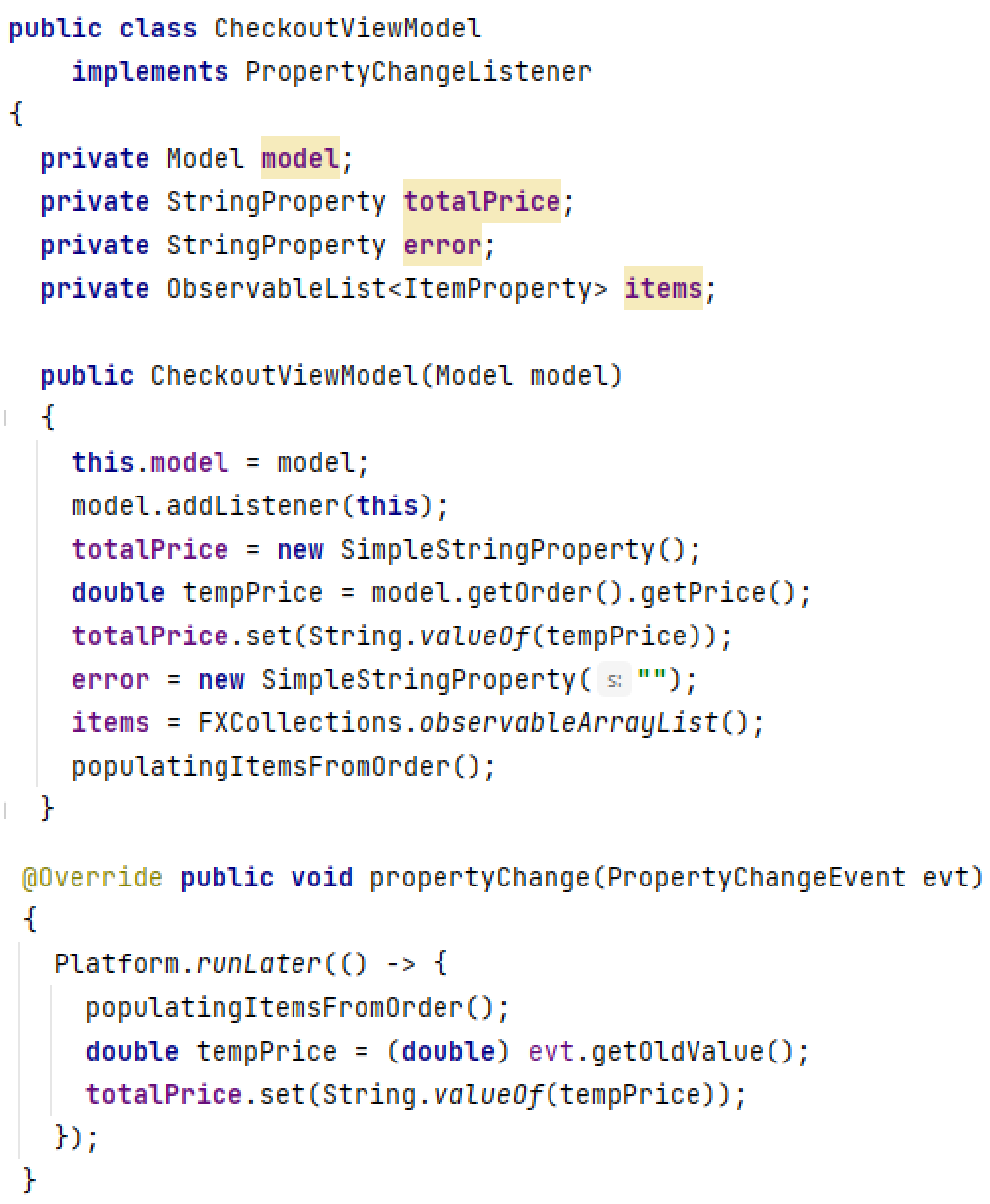
Checkout View - design and implementation

The checkout view is crucial to fulfil the customer's need to see the summary of the order before payment, the possibility of removing an item and adding comment. In this view the TableView was used, a label with a price and four relevant buttons.



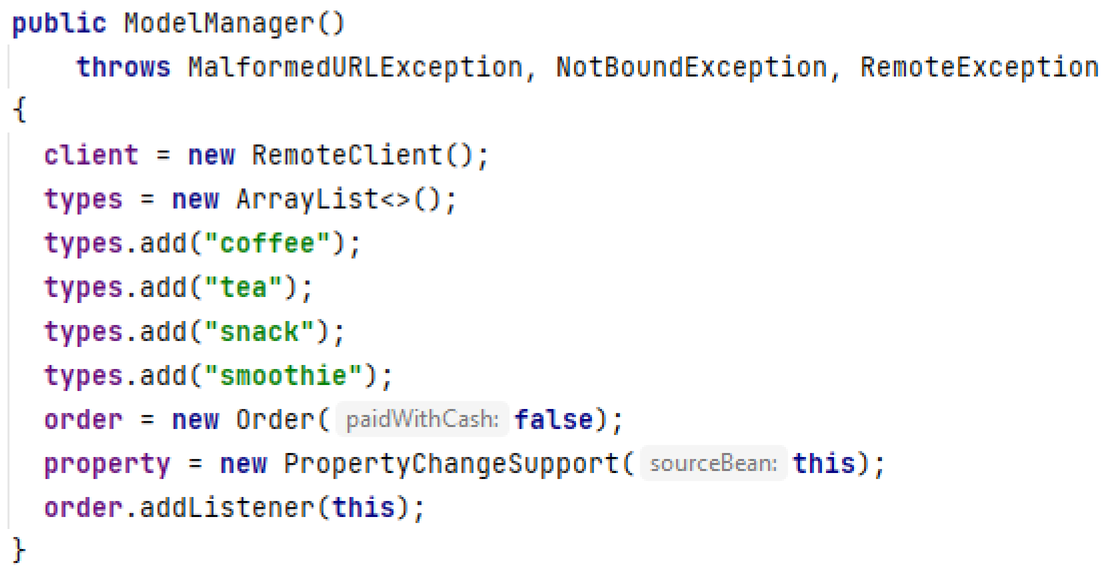
ID	Name	Type	Price
1	Coffee	coffee	20.0

The CheckoutViewModel is implementing PropertyChangeListener following the Observer pattern. It is listening to the events fired by the model and then refreshing the ObservableList and updating the total price which is StringProperty bound with the Label from the CheckoutViewController.

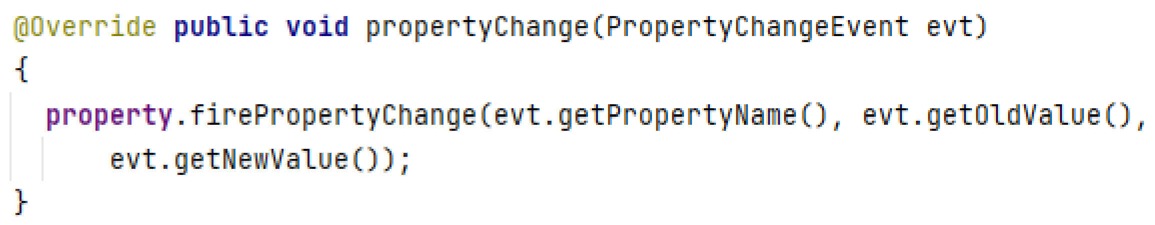


```java
public class CheckoutViewModel
    implements PropertyChangeListener
{
  private Model model;
  private StringProperty totalPrice;
  private StringProperty error;
  private ObservableList<ItemProperty> items;

  public CheckoutViewModel(Model model)
  {
    this.model = model;
    model.addListener(this);
    totalPrice = new SimpleStringProperty();
    double tempPrice = model.getOrder().getPrice();
    totalPrice.set(String.valueOf(tempPrice));
    error = new SimpleStringProperty( s: "");
    items = FXCollections.observableArrayList();
    populatingItemsFromOrder();
  }

 @Override public void propertyChange(PropertyChangeEvent evt)
 {
   Platform.runLater(() -> {
     populatingItemsFromOrder();
     double tempPrice = (double) evt.getOldValue();
     totalPrice.set(String.valueOf(tempPrice));
   });
 }
```

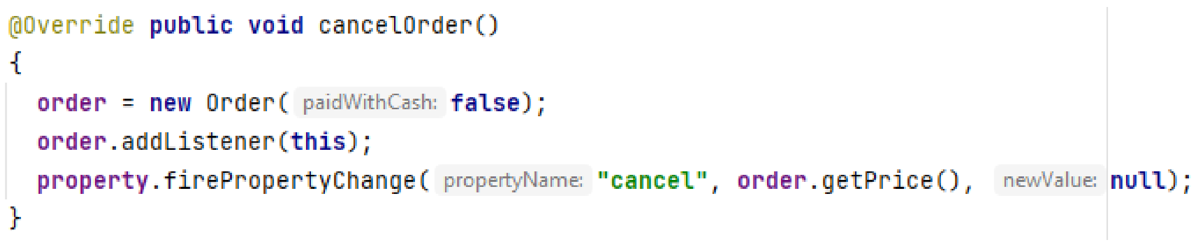
The ModelManager is both a listener and a subject. It is added as a listener to the Order class and the propertyChange() method sends the event further to the ViewModel. Besides that, it also fires an event when the order is cancelled. The information sent is the total price of the order.



```
public ModelManager()
    throws MalformedURLException, NotBoundException, RemoteException
{
  client = new RemoteClient();
  types = new ArrayList<>();
  types.add("coffee");
  types.add("tea");
  types.add("snack");
  types.add("smoothie");
  order = new Order( paidWithCash: false);
  property = new PropertyChangeSupport( sourceBean: this);
  order.addListener(this);
}
```

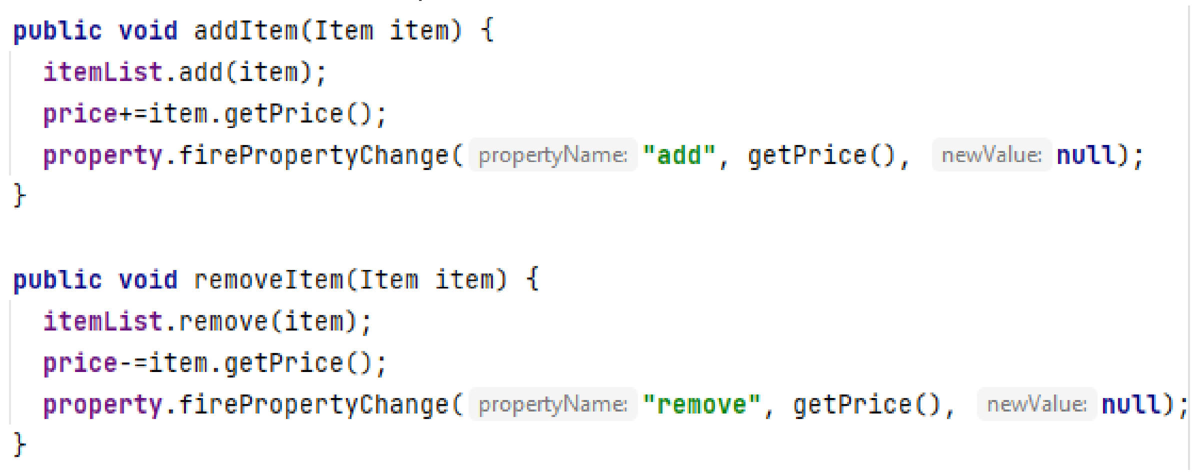


```
@Override public void propertyChange(PropertyChangeEvent evt)
{
  property.firePropertyChange(evt.getPropertyName(), evt.getOldValue(),
      evt.getNewValue());
}
```



```
@Override public void cancelOrder()
{
  order = new Order( paidWithCash: false);
  order.addListener(this);
  property.firePropertyChange( propertyName: "cancel", order.getPrice(),  newValue: null);
}
```

The class order is firing an event every time the item is added or removed from the order. The information sent is the total price of the order.

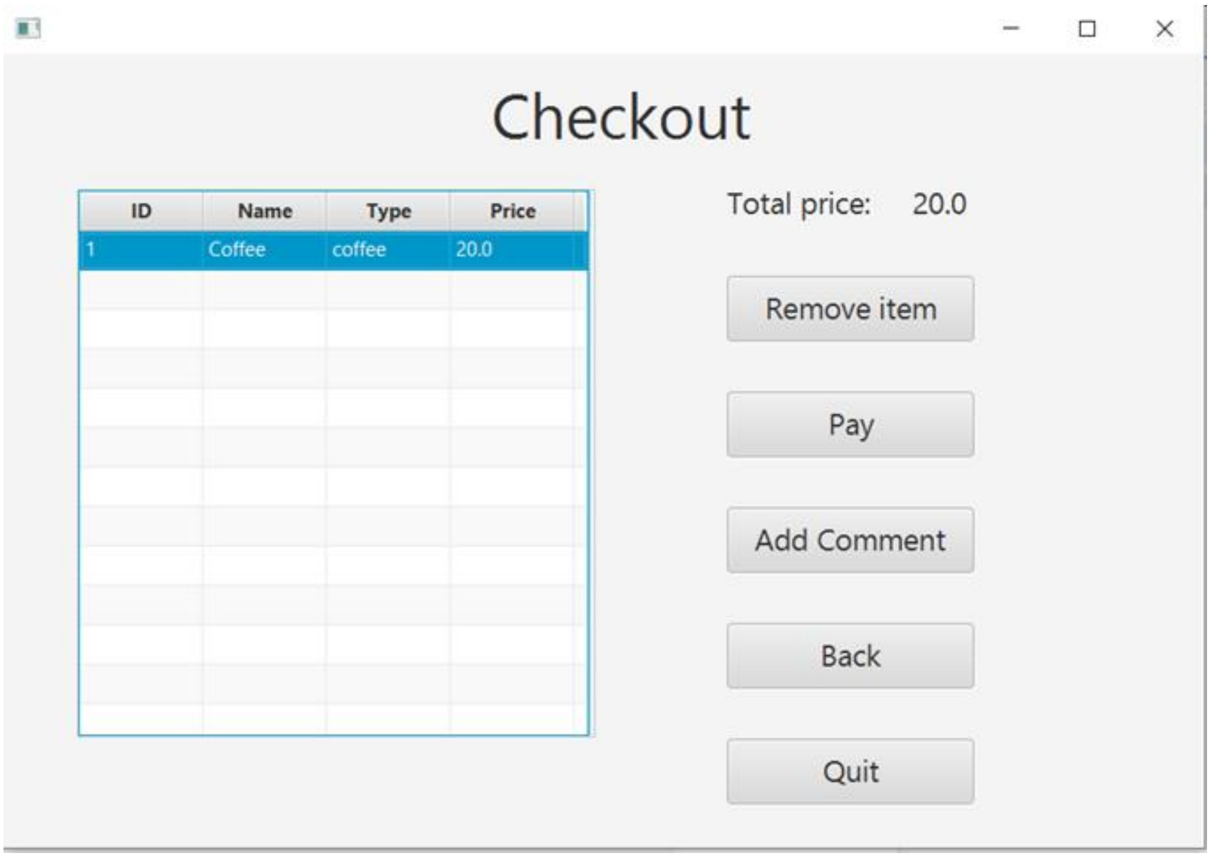


```
public void addItem(Item item) {
  itemList.add(item);
  price+=item.getPrice();
  property.firePropertyChange( propertyName: "add", getPrice(),  newValue: null);
}

public void removeItem(Item item) {
  itemList.remove(item);
  price-=item.getPrice();
  property.firePropertyChange( propertyName: "remove", getPrice(),  newValue: null);
}
```

Documentation of Checkout Oder Integration testing

For the purpose of testing the Check out case the integration test was organised. This will test if the existing methods are working and if the information is being transferred correctly from the view to the database, if needed, going through the right steps.

Checkout View

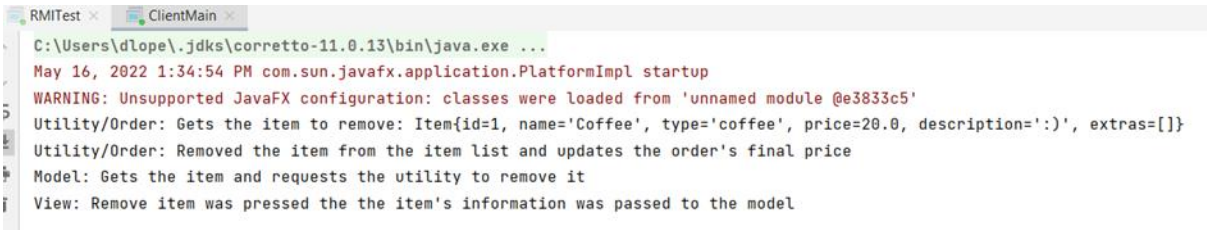


ID	Name	Type	Price
1	Coffee	coffee	20.0

Remove Item

In the “Remove Item” case only client side is being used.

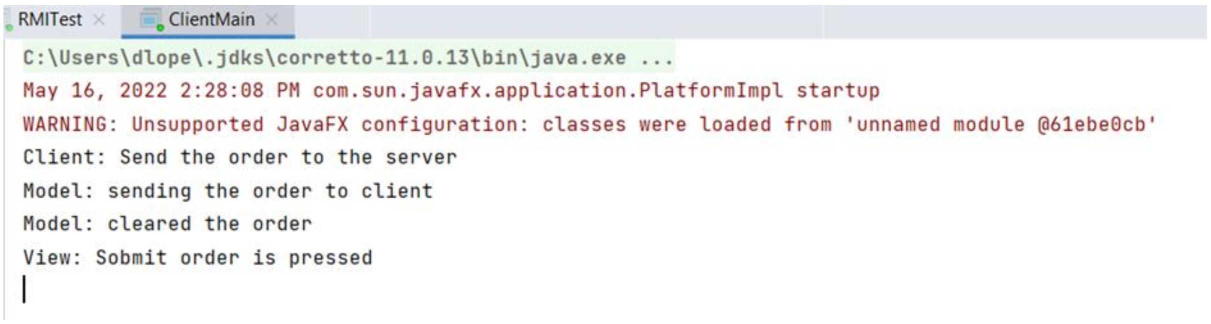
As it can be seen following the printouts, the item is removed from the order with success.



```
C:\Users\dlope\.jdks\corretto-11.0.13\bin\java.exe ...
May 16, 2022 1:34:54 PM com.sun.javafx.application.PlatformImpl startup
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @e3833c5'
Utility/Order: Gets the item to remove: Item{id=1, name='Coffee', type='coffee', price=20.0, description=':)', extras=[]}
Utility/Order: Removed the item from the item list and updates the order's final price
Model: Gets the item and requests the utility to remove it
View: Remove item was pressed the the item's information was passed to the model
```

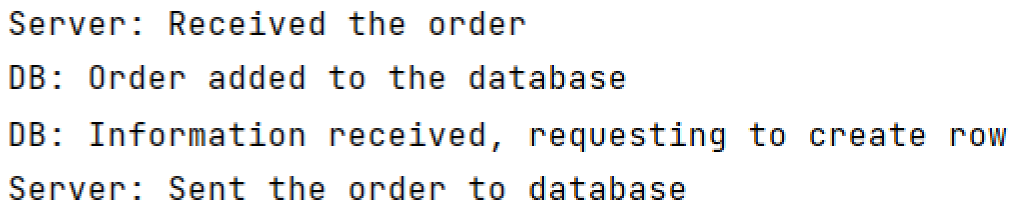
Pay

Client-Side: It can be easily noticed by the printouts that the client effectively sent the order list to the server.



```
C:\Users\dlope\.jdks\corretto-11.0.13\bin\java.exe ...
May 16, 2022 2:28:08 PM com.sun.javafx.application.PlatformImpl startup
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @61ebe0cb'
Client: Send the order to the server
Model: sending the order to client
Model: cleared the order
View: Sobmit order is pressed
```

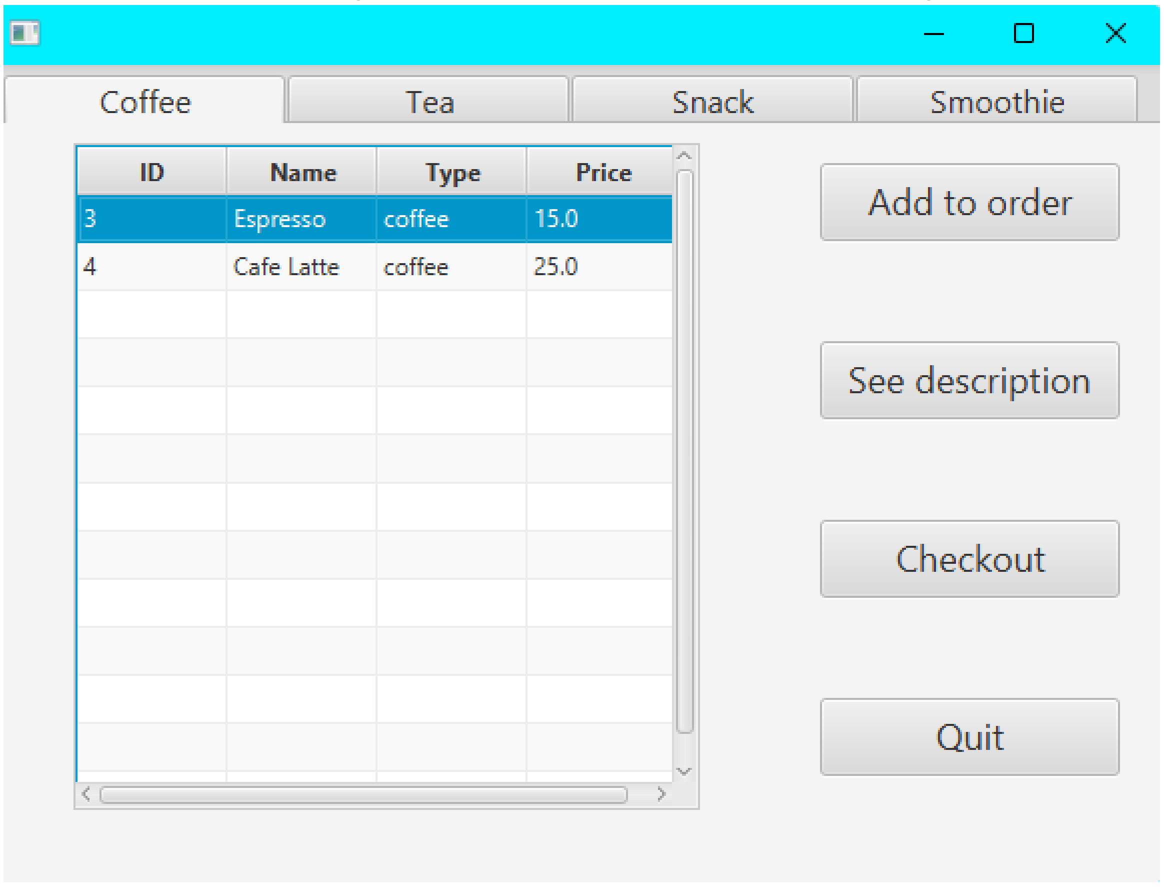
Server-Side: It’s noticeable that the server got the information from the client and properly handled the information and added it to the database.



```
Server: Received the order
DB: Order added to the database
DB: Information received, requesting to create row
Server: Sent the order to database
```

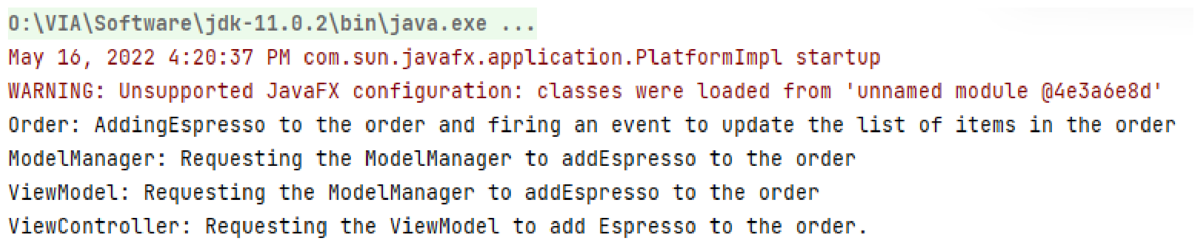
Add item to the Order Integration Testing

Selecting an item in the CustomerView before adding it



ID	Name	Type	Price
3	Espresso	coffee	15.0
4	Cafe Latte	coffee	25.0

Print outs representing the course of the Item object from the ViewController until the Order



```
O:\VIA\Software\jdk-11.0.2\bin\java.exe ...
May 16, 2022 4:20:37 PM com.sun.javafx.application.PlatformImpl startup
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @4e3a6e8d'
Order: AddingEspresso to the order and firing an event to update the list of items in the order
ModelManager: Requesting the ModelManager to addEspresso to the order
ViewModel: Requesting the ModelManager to addEspresso to the order
ViewController: Requesting the ViewModel to add Espresso to the order.
```

The item is added to the observable list of items in the CheckOutViewModel, making it visible in the CheckOutView. It cannot be seen in the database yet, as the order only gets to the database once it has been submitted.