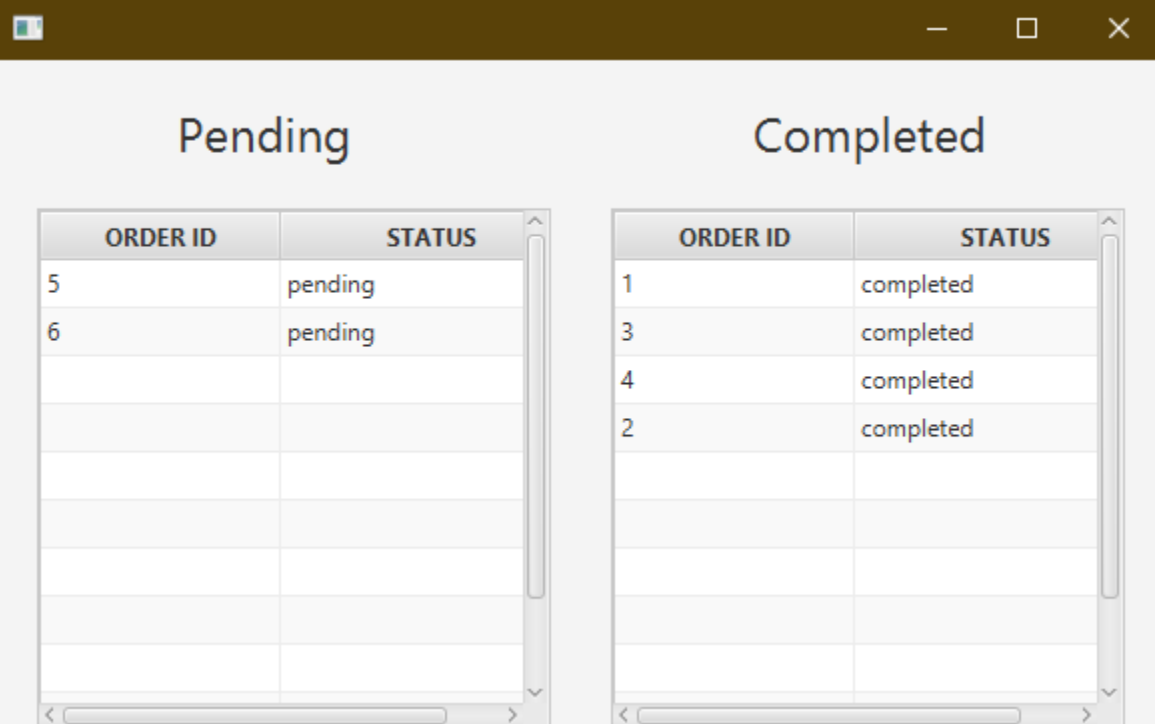


## Display View - design

To follow the Product Backlog, the customer has to be able to see whether the order is still pending or completed. This uncomplicated view incorporates two FXML TableViews for each status. Aiming to inform the customer, the order ID is also displayed.

*Display window*



Pending		Completed	
ORDER ID	STATUS	ORDER ID	STATUS
5	pending	1	completed
6	pending	3	completed
		4	completed
		2	completed

## Display View - implementation

The main concern of this case is that the table should reflect the changes made by both baristas and cashiers in real time. Therefore, it is essential to include the Observer pattern both on the Client and Server side. As a result of that, the understanding of the Observer pattern used in the system is crucial to proceed to ViewModel and ViewController.

*RemoteCafeServer interface*

```
public interface RemoteCafeServer extends Remote, RemoteSubject<String, String>
{
    ItemList getAllItems() throws RemoteException, SQLException;
    ItemList.getItemsByType(String type) throws RemoteException, SQLException;
    void receiveOrder(Order order) throws RemoteException;
    void completeOrder(Order order) throws RemoteException;
    void receiveUnpaidOrder(Order order) throws RemoteException;
    void acceptPayment(Order order) throws RemoteException;
    void addItemToProductList(Item item) throws RemoteException;
    ArrayList<Order> getAllPendingOrders() throws RemoteException;
    ArrayList<Order> getAllCompletedOrders() throws RemoteException;
    void removeItemFromProductList(Item item) throws RemoteException, SQLException;
    boolean addListener(GenericListener<String, String> listener,
        String... propertyNames) throws RemoteException;
    boolean removeListener(GenericListener<String, String> listener,
        String... propertyNames) throws RemoteException;
}
```

The fundamental part of the Observer pattern is that the RemoteCafeServer is extending the RemoteSubject interface. The proper methods are implemented and PropertyChangedHandler is introduced and created. It is crucial that both the interface signature and the PropertyChangedHandler have the same type of values to be fired later.

```
private CafePersistence cafePersistence;
private PropertyChangedHandler<String, String> property;

public RemoteServer()
    throws RemoteException, MalformedURLException, SQLException
{
    cafePersistence = CafeDatabase.getInstance();
    property = new PropertyChangedHandler<> (source: this);
    startRegistry();
    startServer();
}
```

The events are fired every time the order is received or completed and correctly stored inside the database. The event is firing two null values as this method is used only to notify that a change has occurred. Consequently, there is no need to send anything.

#### *Firing events inside the RemoteServer*

```
@Override public void receiveOrder(Order order) throws RemoteException
{
    cafePersistence.receiveOrder(order);
    property.firePropertyChange( propertyName: "pending", value1: null, value2: null);
}

@Override public void completeOrder(Order order) throws RemoteException
{
    cafePersistence.completeOrder(order.getId());
    property.firePropertyChange( propertyName: "completed", value1: null, value2: null);
}
```

Following step is the RemoteClient class that is a listener as it is implementing the RemoteListener for the RMI connection with the server and its fired events. Besides that, it is also a subject in the Observer pattern, because of the UnnamedPropertyChangeSubject interface implementation. In the constructor this object is added as a listener to the changes fired by the server. The propertyChange method is firing the event further with the information received from the server's event.

#### *RemoteClient class*

```
public class RemoteClient implements RemoteListener<String, String>,
    UnnamedPropertyChangeSubject
{
    private RemoteCafeServer server;
    private PropertyChangeSupport property;

    public RemoteClient() throws MalformedURLException, NotBoundException, RemoteException
    {
        server = (RemoteCafeServer) Naming.lookup( name: "rmi://localhost:1099/Cafe");
        UnicastRemoteObject.exportObject( obj: this, port: 0);
        server.addListener( listener: this);
        property = new PropertyChangeSupport( sourceBean: this);
    }
}
```

#### *PropertyChange method*

```
@Override public void propertyChange(ObserverEvent<String, String> event)
    throws RemoteException
{
    property.firePropertyChange( propertyName: "change", event.getValue1(), event.getValue2());
}
```

After that the event is caught inside the ModelManager and sent further to the DisplayViewModel without any changes. As a consequence, code snippets from the ModelManager are redundant.

Inside the DisplayViewModel there are two Observable Lists variables for each status of the order. The reset method is filling them with the correct orders obtained from the Model which is indirectly fetching this information from database.

*DisplayViewModel class*

```
public class DisplayViewModel implements PropertyChangeListener  
  
{  
    private Model model;  
    private ObservableList<OrderProperty> pendingList;  
    private ObservableList<OrderProperty> completedList;  
  
    public DisplayViewModel(Model model)  
    {  
        this.model=model;  
        model.addListener(this);  
        pendingList= FXCollections.observableArrayList();  
        completedList= FXCollections.observableArrayList();  
        reset();  
    }  
}
```

#### *Reset method from DisplayViewModel class*

```
public void reset()
{
    pendingList.clear();
    for(int i=0; i<model.getAllPendingOrders().size(); i++)
    {
        pendingList.add(new OrderProperty(model.getAllPendingOrders().get(i)));
    }
    completedList.clear();
    for(int i=0; i<model.getAllCompletedOrders().size(); i++)
    {
        completedList.add(new OrderProperty(model.getAllCompletedOrders().get(i)));
    }
}
```

To get a better insight on how the Model is receiving the information it is essential to understand the connection with the database. *Statement.setString(1, status)* is setting a String “?” from the query the line above to the argument from the method signature. Due to this, when the statement is executed it is searching for the orders with the user specified status. The next steps are translating the result into the returnable Array List of type Order.

#### *getOrderByStatus method from ConcreteOrderDAO class*

```
@Override public ArrayList<Order> getOrdersByStatus(String status)
    throws SQLException
{
    ArrayList<Order> returnOrders = new ArrayList<>();
    ArrayList<Integer> returnOrdersIds = new ArrayList<>();
    try (Connection connection = getConnection()) {
        PreparedStatement statement = connection.prepareStatement( sql: "SELECT * FROM order_ WHERE status = ?");
        statement.setString( parameterIndex: 1,status);
        ResultSet orderResultSet = statement.executeQuery();
        while (orderResultSet.next()) {
            int id = orderResultSet.getInt( columnLabel: "order_id");
            returnOrdersIds.add(id);
        }

        for (int i : returnOrdersIds) {
            returnOrders.add(readById(i));
        }
    }
    return returnOrders;
}
```

Shifting back to the DisplayViewModel class, when the event is fired by the Model and its name is “change”, the reset method is called. Consequently, the ObservableLists are loaded from the Model once again to reflect the changes made by baristas and cashiers.

```
@Override public void propertyChange(PropertyChangeEvent evt)
{
    Platform.runLater()->{
        if(evt.getPropertyName().equals("change"))
        {
            reset();
        }
    });
}
```

Last class in this chain is DisplayViewController which contains two TableView elements from the FXML file. The reset method is setting the items of the TableView with the according Observable List from the DisplayViewModel. On account of usage of the Observable List object, when the list is updated inside the DisplayViewModel, the TableViews are automatically modified.

```
public class DisplayViewController extends ViewController
{
    @FXML private TableView<OrderProperty> tableViewCompleted;
    @FXML private TableView<OrderProperty> tableViewPending;
    private DisplayViewModel displayViewModel;

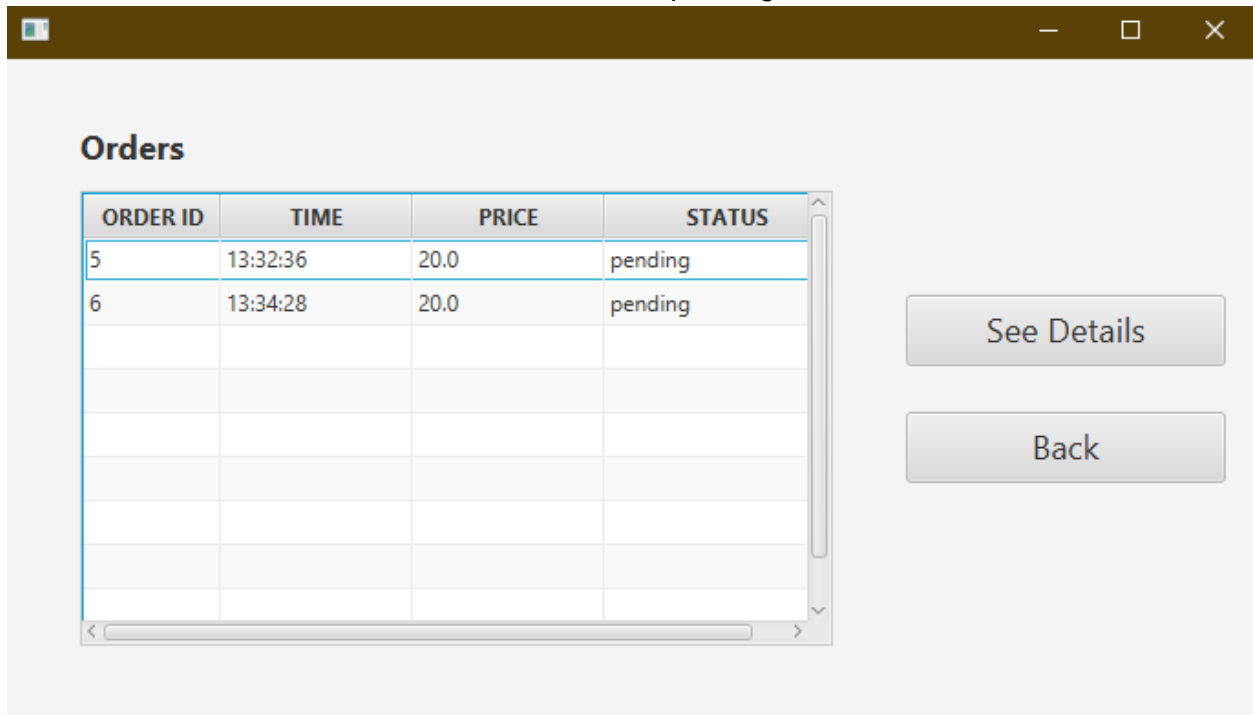
    @Override protected void init()
    {
        displayViewModel = getViewModelFactory().getDisplayViewModel();
        setColumns(tableViewCompleted);
        setColumns(tableViewPending);
        reset();
    }

    public void reset()
    {
        displayViewModel.reset();
        tableViewPending.setItems(displayViewModel.getPendingList());
        tableViewCompleted.setItems(displayViewModel.getCompletedList());
    }
}
```

## Display View - testing

In order to test the displaying of pending and completed orders the integration test is conducted. The BaristaView with two pending orders is shown and the same orders are visible in the DisplayView. Therefore, it is working correctly. The Observer pattern is tested in the way if the change of status of order is reflected in the GUI window. When the order with id "6" is completed, it is accurately demonstrated inside the DisplayView.

*BaristaView with two pending orders*



*DisplayView with two pending orders*

Pending		Completed	
ORDER ID	STATUS	ORDER ID	STATUS
5	pending	1	completed
6	pending	3	completed
		4	completed
		2	completed

*DisplayView with the order id "6" completed*

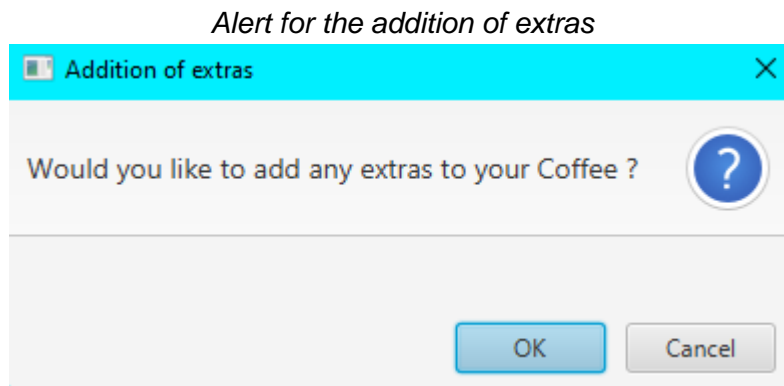
Pending		Completed	
ORDER ID	STATUS	ORDER ID	STATUS
5	pending	1	completed
		3	completed
		4	completed
		2	completed
		6	completed

FUCKING SOLID DOCUMENTATION KAMIL <#



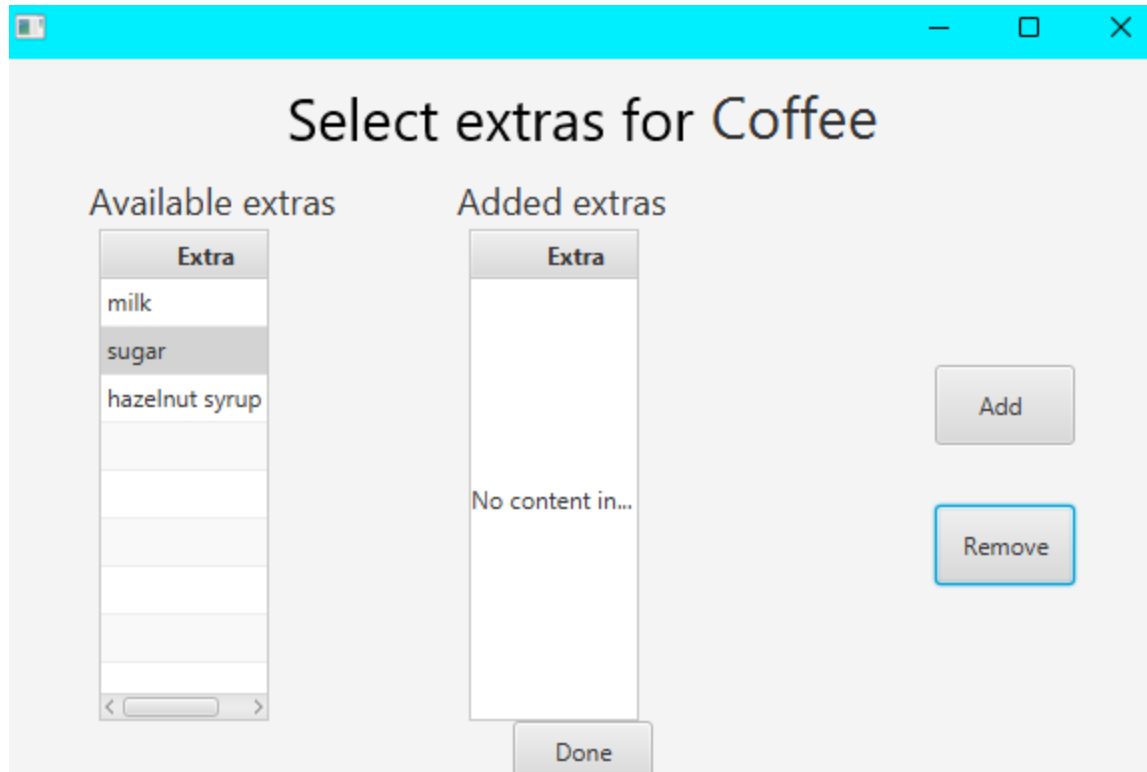
## DESIGN OF THE EXTRA VIEW

Following the product backlog, the customer's wish is to be prompted for extras to be added to his item. Therefore, whenever the customer wants to add an item to his order, an alert will show up asking if he would like to add any extras to the current item.



By clicking Cancel, the item will be added to the order without extras. On the other hand, by clicking OK, the ExtraView is opened.

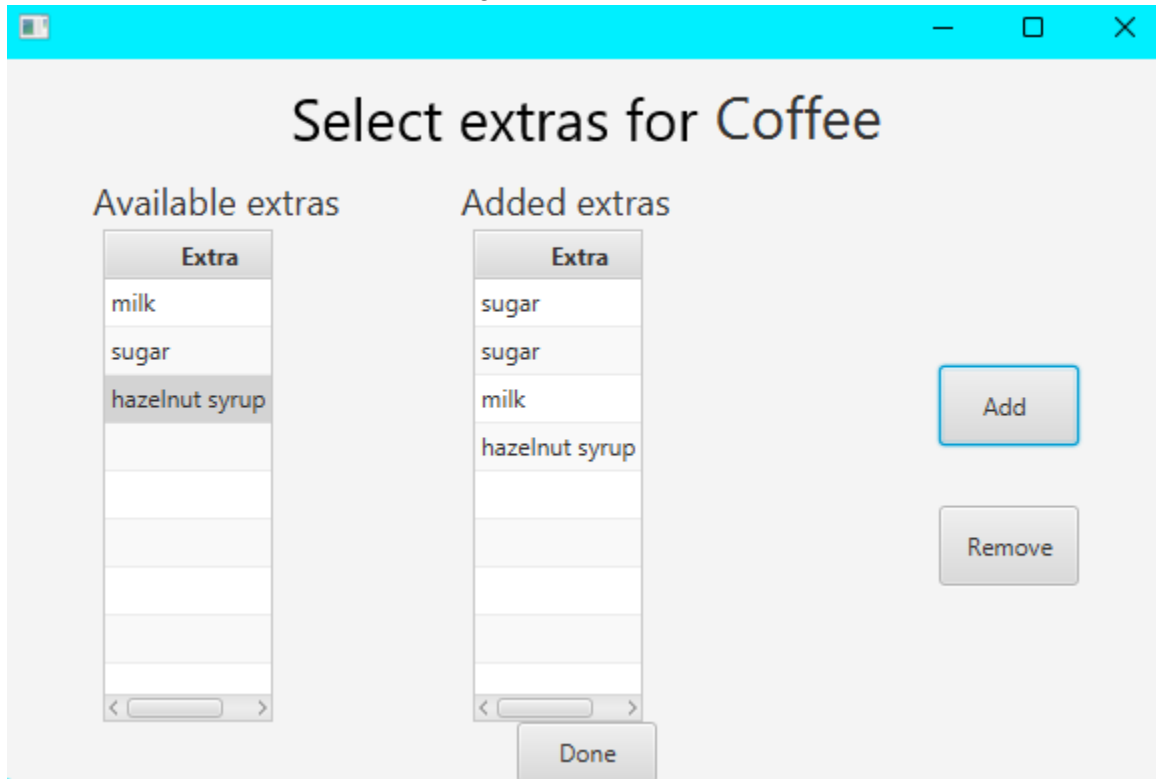
*ExtraView*



Two tables have been added to this view. The table on the left contains all the extras that are available to be added to the current item type the customer is adding to his order - this will be different for coffees, snacks, smoothies and teas. The table on the right will be filled by the user by selecting an item from the left and clicking add. Finally, by pressing the Done button, the item is added to the order with all of its extras.

## INTEGRATION TESTING OF EXTRA VIEW

## Adding four extras in the ExtraView



## Print outs after adding extras in the ExtraView

```
May 19, 2022 1:37:05 PM com.sun.javafx.application.PlatformImpl startup
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @571aeebe'
ViewController: Adding extrasugar to the item.
ViewController: Adding extrasugar to the item.
ViewController: Adding extramilk to the item.
ViewController: Adding extrahazelnut syrup to the item.
```

## Print outs after pressing Done in the ExtraView

```
0:\VIA\Software\jdk-11.0.2\bin\java.exe ...
May 19, 2022 1:37:05 PM com.sun.javafx.application.PlatformImpl startup
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @571aeebe'
ViewController: Adding extrasugar to the item.
ViewController: Adding extrasugar to the item.
ViewController: Adding extramilk to the item.
ViewController: Adding extrahazelnut syrup to the item.
Model manager: The item with its extras: Item{id=1, name='Coffee', type='coffee', price=20.0, description:''}, extras=[Extra{name='sugar'}]
Model manager: The item with its extras: Item{id=1, name='Coffee', type='coffee', price=20.0, description:''}, extras=[Extra{name='sugar'}, Extra{name='sugar'}]
Model manager: The item with its extras: Item{id=1, name='Coffee', type='coffee', price=20.0, description:''}, extras=[Extra{name='sugar'}, Extra{name='sugar'}, Extra{name='milk'}]
Model manager: The item with its extras: Item{id=1, name='Coffee', type='coffee', price=20.0, description:''}, extras=[Extra{name='sugar'}, Extra{name='sugar'}, Extra{name='milk'}, Extra{name='hazelnut syrup'}]
Item{id=1, name='Coffee', type='coffee', price=20.0, description:''}, extras=[Extra{name='sugar'}, Extra{name='sugar'}, Extra{name='milk'}, Extra{name='hazelnut syrup'}]
Model: I have added the item Item{id=1, name='Coffee', type='coffee', price=20.0, description:''}, extras=[Extra{name='sugar'}, Extra{name='milk'}, Extra{name='hazelnut syrup'}] to the order
ViewModel: I have asked the model to add the item to the order, with the following extras: sugar, sugar, milk, hazelnut syrup.
ViewController: Requesting the ViewModel to add the item to the order
```

*extrainiteminorder table in the database for the newly added item*

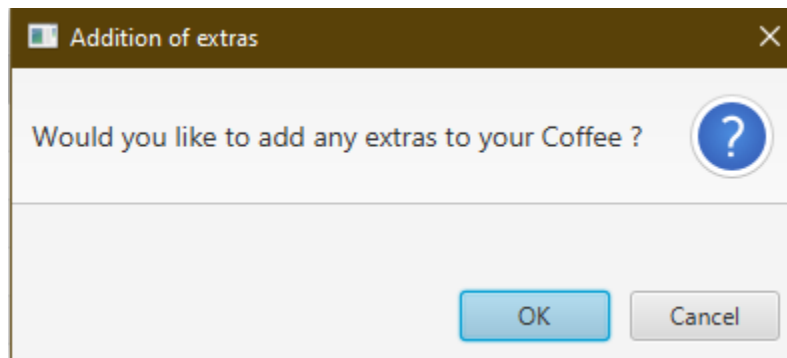
extra_id	item_in_order_id	item_id	order_id
1 sugar	20	1	15

The integration testing shows that a few things are missing: Even though the model adds the extras to the item correctly, the database only takes the first extra that has been added to the item. On the other hand, there is no way for the barista to know what extras have been added to the item since the extras are not displayed in any of the views.

## Extra View - implementation

To enter the ExtraView, there is an alert inside the CustomerViewController, which asks about the addition of extras to the selected item, that has to be confirmed.

*The alert about extras*



The confirmation method return type is boolean as this is reflecting whether the user chose to add an extra or not.

```
private boolean confirmation(String name)
{
    Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
    alert.setTitle("Addition of extras");
    alert.setHeaderText("Would you like to add any extras to your "+name+" ?");
    Optional<ButtonType> result = alert.showAndWait();

    return (result.isPresent()) && (result.get() == ButtonType.OK);
}

@FXML private void addToOrderButton()
{
    ItemProperty item = getItemProperty();
    if (confirmation(item.getItem().getName()))
    {
        customerViewModel.setItemForExtras(item);
        getViewHandler().openView( id: "ExtraView.fxml");
    }
    else
    {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Adding item...");
        alert.setHeaderText(item.getItem().getName() + " is added to your order.");
        alert.show();
        customerViewModel.addToOrder(item);
    }
}
```

The method `setItemForExtras` is passing the `ItemProperty` to the `CustomerViewModel` which is then calling the `ExtraViewModel` as there is an association between those two classes. The association between two ViewModels is a violation of the Single Responsibility Principle, because the `CustomerViewModel` is not only implementing the functionality for its View, but also communicating with different ViewModel. This issue could be solved by introducing a class to handle the communication between the ViewModels. Its name could be `CustomerHandler`.

Shifting to the ExtraViewModel, the ItemProperty variable set by CustomerViewModel is used to fill the availableExtras ObservableList with extras applicable for the item. It is possible by calling the Model to retrieve the correct data. The model is fetching the information from the database which is explained later.

#### *ExtraViewModel*

```
private ItemProperty currentItem;
private ObservableList<ExtraProperty> availableExtras;
private ObservableList<ExtraProperty> addedExtras;

public ExtraViewModel(Model model)
{
    this.model = model;
    availableExtras = FXCollections.observableArrayList();
    addedExtras = FXCollections.observableArrayList();
}

public void reset()
{
    setList(availableExtras, currentItem.typeProperty().get());
}

public void setList(ObservableList<ExtraProperty> extraList, String type)
{
    extraList.clear();
    for (int i = 0; i < model.getAllExtrasByType(type).size(); i++)
    {
        extraList.add(new ExtraProperty(model.getAllExtrasByType(type).get(i)));
    }
}
```

The addedExtras ObservableList is used to allow the customer to modify the extras - add and remove them. When the customer clicks the “done” button, the extras from the ObservableList are added to the item. After that the item is added to the order. The last step after the button is pressed is that the ExtraViewController is requesting to open the CustomerView.

*Manipulation of addedExtras list*

```
public void addExtraToItem(ExtraProperty extra)
{
    addedExtras.add(extra);
}

public void removeExtraFromItem(ExtraProperty extra)
{
    addedExtras.remove(extra);
}

public void addItemToOrder()
{
    if (addedExtras.size() > 0)
    {
        for (int i = 0; i < addedExtras.size(); i++)
        {
            model.addExtraToItem(addedExtras.get(i).getExtra(),
                                currentItem.getItem());
        }
    }
    System.out.println(currentItem.getItem());
    model.addItemToOrder(currentItem.getItem());
}
```

*getExtrasByType(String type) method in ConcreteExtraDAO class*

```
34  @Override public ArrayList<Extra> getExtrasByType(String type) throws SQLException
35  {
36      ArrayList<Extra> extrasList = new ArrayList<>();
37
38      try (Connection connection = getConnection())
39      {
40          PreparedStatement statement = connection.prepareStatement(
41              sql: "SELECT * FROM extraavailablefortype WHERE type = ?");
42          statement.setString( parameterIndex: 1, type);
43          ResultSet orderResultSet = statement.executeQuery();
44          while (orderResultSet.next())
45          {
46              String name = orderResultSet.getString( columnLabel: "extra_id");
47              extrasList.add(new Extra(name));
48          }
49      }
50      return extrasList;
51  }
```

The purpose of the above shown method is to retrieve the extras for the item type passed in as a parameter. This is done by connecting to the database, and adding all the extras with the same type as in the parameter one by one to an ArrayList created inside this method.