

CASHIER VIEW DOCUMENTATION

DESIGN CHOICES

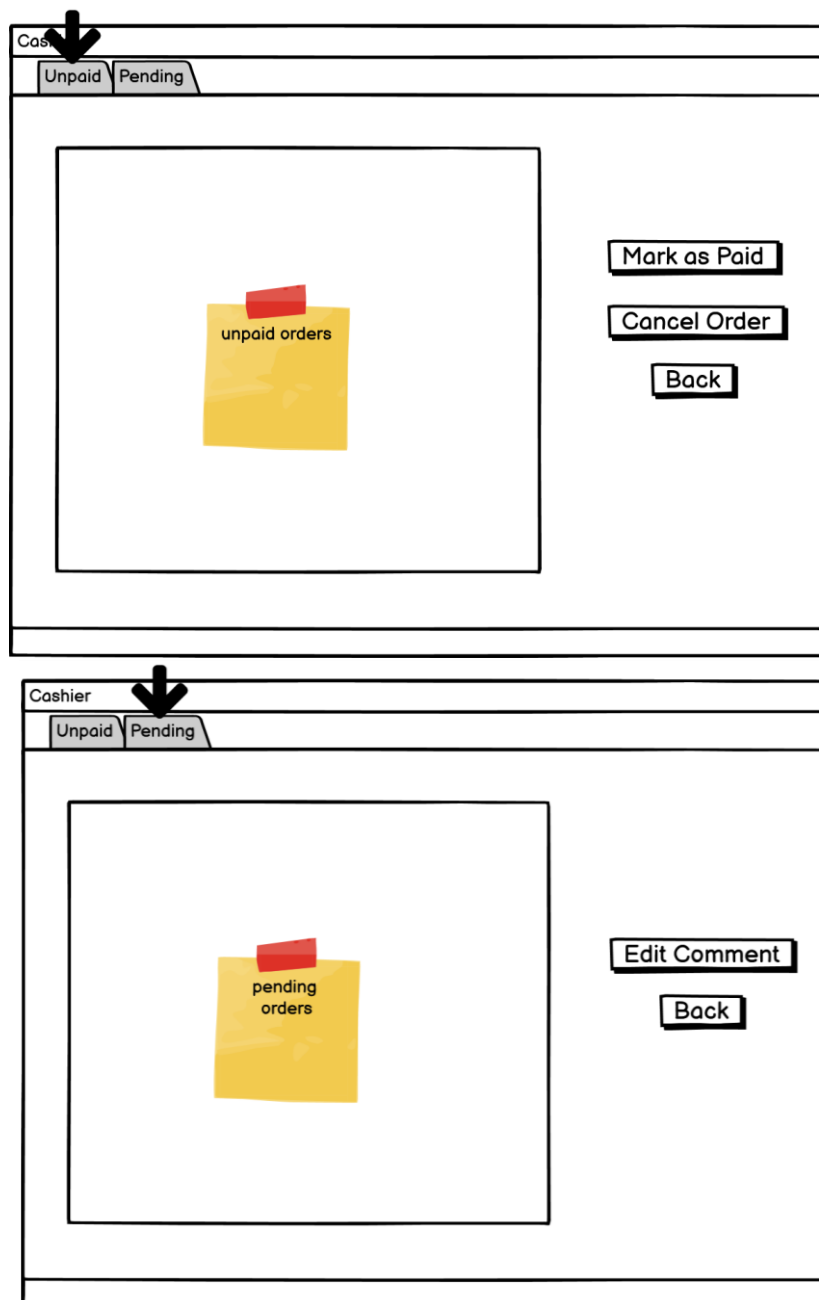
The Cashier View serves to fulfill the Cashier actor Product Backlog items. These are:

9. As a cashier, I want to accept an order when a payment is in cash so that it will appear for baristas.

14. As a cashier, I want to be able to edit an order's comment so that wishes from customers can be fulfilled.

15. As a cashier, I want to cancel orders when they are not paid for so that they will not use unnecessary resources.

The first draft for the view was the following:



The Cashier View features two tabs because a Cashier needs access to both Unpaid and Pending orders. In each view, there are the buttons necessary to fulfill each Product Backlog item:

UNPAID ORDERS:

- Mark them as paid (unpaid -> pending)
- Cancel them (delete order)

PENDING ORDERS:

- Edit comment

The Unpaid Order management is simple – everything occurs in the same view. The cashier can select an order and press a button in order to either change its state or delete it.

However, the Pending Order management required the elaboration of another view to edit the order's Comment.

The new view's appearance follows:

Diagram illustrating the 'Edit Comment' view structure:

- Top bar: Edit Comment
- Centered text: Edit Comment
- Centered text: <ORDER NUMBER>
- Text area (labeled comment):
- Buttons: Cancel, Confirm

There are not many elements to it. The most relevant ones are the Text Area to edit the comment and the button Confirm, which sends the changes to the server and confirms them.

The Edit Comment View and the Cashier View share a resource in order to have access to the same Selected Order object. This will be explained in detail in the Implementation part of this document.

IMPLEMENTATION

Starting out with the Cashier View Controller, it is a default View Controller.

The instance variables are:

```
16 public class CashierViewController extends ViewController
17 {
18     @FXML private TableView unpaidTable;
19     @FXML private TableView pendingTable;
20     @FXML private TabPane tabPane;
21     private SelectionModel unpaidOrderSelection;
22     private SelectionModel pendingOrderSelection;
23     private CashierViewModel viewModel;
```

There is an fx ID for both tables, an fx ID for the tabPane, a View Model and a Selection Model for both tables so that it is possible to fetch the selected Order in each.

The reason why there is not an fx ID for each Table Column is because it would be ineffective to manually populate them. For this purpose, there is a method, previously implemented in Customer View, which can populate a table's columns when provided with the TableView and the distinguishing feature between them. In this case, what distinguishes the Order tables is the status of the orders. So, for this view, the method looks like this:

```
42 @ private void setTable(TableView table, String status)
43 {
44
45     TableColumn idColTemp = (TableColumn) table.getColumns().get(0);
46     TableColumn timeColTemp = (TableColumn) table.getColumns().get(1);
47     TableColumn priceColTemp = (TableColumn) table.getColumns().get(2);
48     TableColumn statusColTemp = (TableColumn) table.getColumns().get(3);
49     idColTemp.setCellValueFactory(
50         new PropertyValueFactory<OrderProperty, IntegerProperty>(s: "id"));
51     timeColTemp.setCellValueFactory(
52         new PropertyValueFactory<OrderProperty, StringProperty>(s: "time"));
53     priceColTemp.setCellValueFactory(
54         new PropertyValueFactory<OrderProperty, StringProperty>(s: "price"));
55     statusColTemp.setCellValueFactory(
56         new PropertyValueFactory<OrderProperty, DoubleProperty>(s: "status"));
57
58     switch (status) {
59         case "unpaid":
60             table.setItems(viewModel.getUnpaidOrders());
61             break;
62         case "pending":
63             table.setItems(viewModel.getPendingOrders());
64             break;
65     }
66 }
67 }
```

What happens in this method is that, instead of populating the Columns by their id, they are fetched from the TableView, they are provided with a Cell Value Factory with the corresponding type of information and then populated with the right Objects fetched from the View Model depending on the status provided. This method could be reused in the case of the addition of more tabs to the View, or even reused in another view.

As for the methods related to the buttons, it might be worth sharing about one for each type of order: one for Unpaid orders, one for Pending.

The method “markAsPaidPressed()” serves to change an order’s status from “unpaid” to “pending”. This method looks simple in the view but actually branches all the way to the database. What happens here is that the selected order is fetched and the functionality is then delegated to the View Model. The View Model gains “knowledge” of the selected order and then calls the method to mark it as paid.

```
69 @FXML private void markAsPaidPressed() {  
70     OrderProperty order = (OrderProperty) unpaidOrderSelection.getSelectedItem();  
71     viewModel.setSelectedUnpaidOrder(order);  
72     viewModel.markOrderAsPaid();  
73     reset();  
74 }
```

What happens next, in the View Model, is a delegation of functionality to the Model. But first, it can be observed that the Selected order is not fetched from the View Model itself, but from an instance variable called “handler”. Before moving on to the rest of the integration discussion, it is worth looking into the purpose of this “handler”.

```
public void markOrderAsPaid()  
{  
    model.acceptPayment(handler.getSelectedUnpaidOrder().getOrder());  
}
```

These are the instance variables of the CashierViewModel. At the bottom, there is one of type *CashierHandler*. This class is a shared resource that will be accessed by both CashierViewModel and EditCommentViewModel.

```
9 public class CashierViewModel  
10 {  
11     private Model model;  
12     private ObservableList<OrderProperty> unpaidOrders;  
13     private ObservableList<OrderProperty> pendingOrders;  
14     private CashierHandler handler;
```

The class `CashierHandler` is simple: it has an `OrderProperty` of each possible selected order in `Cashier View`, as well as `Setters` and `Getters` for them (not shown). This serves for decoupling of classes: by sharing this resource, the `View Models` involved do not need to depend on each other or “worry” about setting the selected `Order` – all of this is taken care of in the `Handler` class.

```
6 public class CashierHandler
7 {
8     private OrderProperty selectedUnpaidOrder;
9     private OrderProperty selectedPendingOrder;
```

The way that the `View Models` and `Handler` are instantiated in the `View Model Factory` might also be worth looking at.

```
23 public ViewModelFactory(Model model)
24 {
25     this.cashierHandler = new CashierHandler();
26     this.cashierViewModel = new CashierViewModel(model, cashierHandler);
27     this.editCommentViewModel = new EditCommentViewModel(model, cashierHandler);
```

Since both `View Models` need access to this class, the `Handler` is created first, and the `View Models` will then take it as an argument for their constructor when being created in the `Factory`. This way, the classes do not depend on each other and the `Factory` takes care of the association between the `View Models` and the `Handler`.

Going back to the integration:

The method “`markOrderAsPaid()`” in the `View Model` calls the method “`acceptPayment()`” in the `Model`.

The `Model` then delegates to the “`client`”, which is the class implementing the `Remote Interface` for `RMI`.

```
@Override public void acceptPayment(Order order)
{
    try
    {
        client.acceptPayment(order);
    }
}
```

Afterwards, the “`client`” calls the method in the “`server`”, which is the class implementing the `Remote Interface` on the server side.

```
62 public void acceptPayment(Order order) throws RemoteException
63 {
64     server.acceptPayment(order);
65 }
```

On the server side, what happens is a delegation to the “Persistence” class, which interacts with the database.

```
94  @Override public void acceptPayment(Order order) throws RemoteException
95  {
96      cafePersistence.acceptPayment(order.getId());
97  }
```

At last, what happens in the Persistence class is the changing of this order’s status from unpaid to pending.

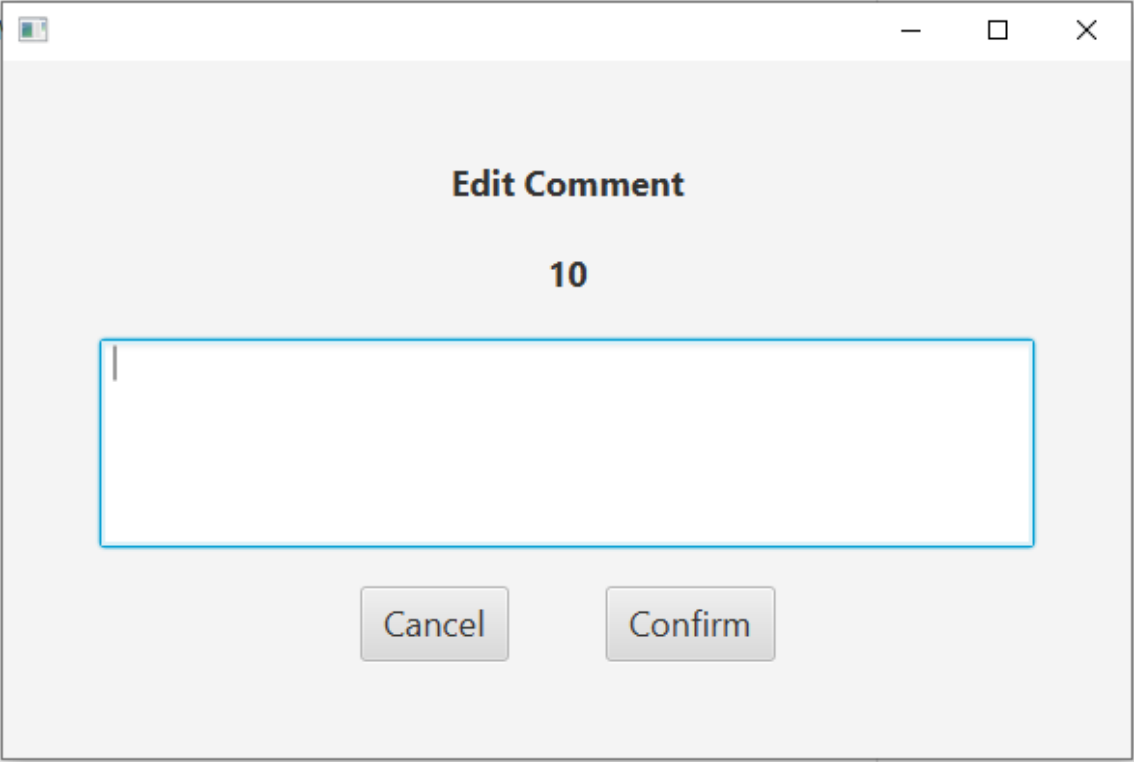
```
82  @Override public void acceptPayment(int orderId)
83  {
84      try
85      {
86          orderDAO.updateStatus(orderId, status: "pending");
87      }
```

After this, the order will disappear from the Cashier’s Unpaid Orders tab and appear in the Pending Orders tab.

Speaking of which, it is time to look into the Pending Order’s button: Edit Comment.

This button opens another view:

If the pending order number 10 was selected, what will appear is the following.



The screenshot shows a Java Swing window with a light gray background. At the top, there are standard window controls (minimize, maximize, close). The title of the window is "Edit Comment". Below the title, the number "10" is displayed in a bold, black font. In the center of the window, there is a large, empty text input field with a blue border. At the bottom of the window, there are two buttons: "Cancel" and "Confirm", both with a light gray background and a thin border.

It is then possible for the cashier to edit the comment and press Confirm. What happens next is similar to the Integration that was explained for the Unpaid Order above, except that the method called is “editComment” instead of “acceptPayment”.

Just like the Cashier View Model, the current view also has a CashierHandler instance that will take care of fetching which order was selected in the previous View and bring its information to the present view.

```
9 public class EditCommentViewModel
10 {
11     private Model model;
12     private StringProperty orderNumber;
13     private StringProperty commentArea;
14     private CashierHandler handler;
```

Having said all of this, it should be clear how the Cashier View came to fruition in order to fulfill the Cashier Actor Product Backlog Items successfully.