SPRINT 3 – ADD ITEM VIEW DOCUMENTATION

DESIGN

After looking at the Product Backlog item related to this print, it was inferred that the window necessary to fulfill this use case needed to contain fields for all the information about a new item. These would be the instance variables in the Item object, except the id, which is auto-generated by the database, and the extras, which are added by the customer when he orders.

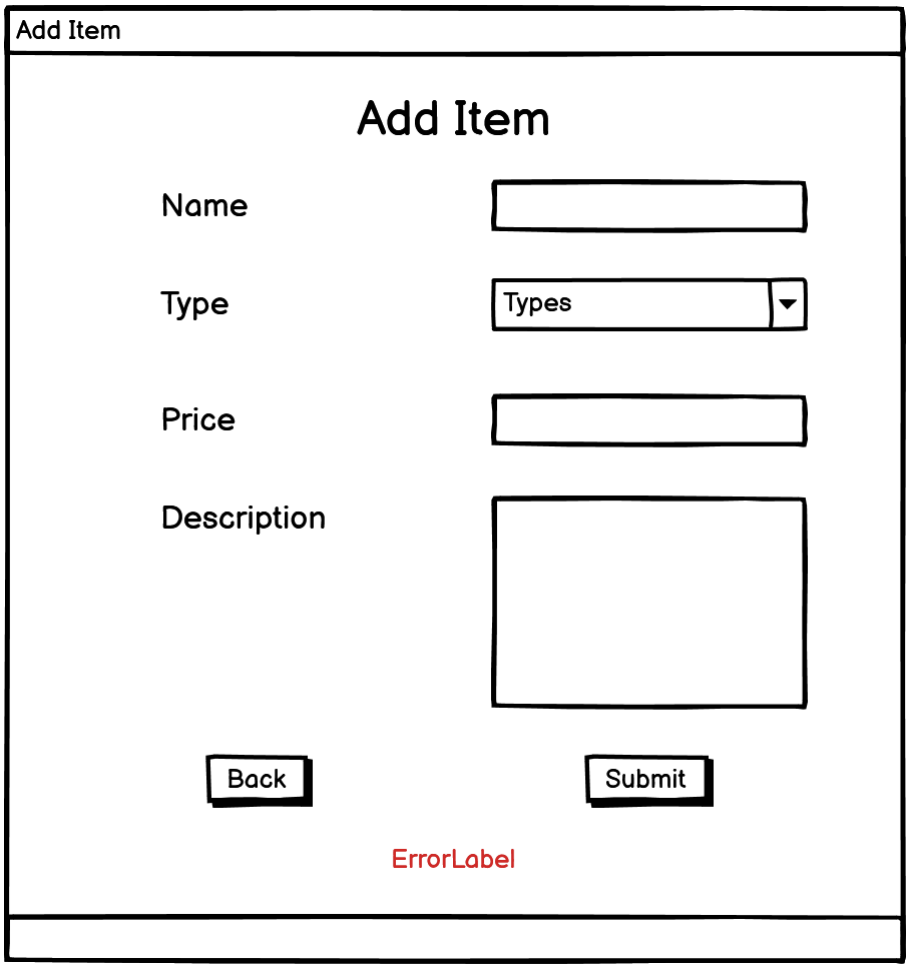
```
public class Item implements Serializable
{
  private int id;
  private String name;
  private String type;
  private double price;
  private String description;
  private ArrayList<Extra> extras;
```

For this purpose, a new constructor without id or extras was created:

```
public Item(String name, String type, double price,
    String description)
{
  this.id = -1;
  this.name = name;
  this.type = type;
  this.price = price;
  this.description = description;
  this.extras = new ArrayList<>();
}
```

The id is set as -1 because it is not relevant. For the purpose of this window, the Item object serves only as a container of information to be sent to the database.

After having inferred all of this, a design for the window was elaborated. It is in the following page.



The name is a short text field, the Types are a choice-box with the same types as the ones existing in the database, the Price is also a text field and the Description, which might be longer, is a text area.

There are then two buttons for navigation: Back, which would take the User back to the Database View, and Submit, which would check the fields and attempt to add an item to the database.

An error label is also present for troubleshooting of problems while testing.

As for implementation-oriented details, some checks will need to be made, mainly in the Price field. It will be necessary to check whether the price is valid: it must be a number and it must not be unreasonably high. Other than that, there is not much that can go wrong in this window.

IMPLEMENTATION

For the window to work, it needs both a Controller and a View Model. The controller has all of the injectable FXML fields, which are initially bound to View Model properties, and as for methods, it delegates everything that is functionality to the View Model.

```
public class AddItemViewController extends ViewController
{
  @FXML private TextField nameField;
  @FXML private ChoiceBox<String> typeChoiceBox;
  @FXML private TextField priceField;
  @FXML private TextArea descriptionArea;
  @FXML private Label errorLabel;
  private AddItemViewModel addItemViewModel;

  @Override protected void init()
  {
    addItemViewModel = getViewModelFactory().getAddItemViewModel();
    nameField.textProperty().bindBidirectional(addItemViewModel.nameProperty());
    priceField.textProperty()
        .bindBidirectional(addItemViewModel.priceProperty());
    descriptionArea.textProperty()
        .bindBidirectional(addItemViewModel.descriptionProperty());
    errorLabel.textProperty()
        .bindBidirectional(addItemViewModel.errorProperty());
    typeChoiceBox.getItems().addAll(addItemViewModel.getTypes());
    //Putting items inside the choice box
    typeChoiceBox.valueProperty()
        .bindBidirectional(addItemViewModel.chosenProperty());
    //Binding the chosen item with the view model
  }
```

As for the View Model:

```
14  public class AddItemViewModel
15  {
16    private Model model;
17    private StringProperty name;
18    private ArrayList<String> types;
19    private StringProperty price;
20    private StringProperty description;
21    private StringProperty error;
22    private StringProperty chosen;
```

These are all the instance variables. It is worth nothing that “types” refers to all the possible types in the ChoiceBox and “chosen” refers to the type that is selected.

The types are hard-coded in the clear() method:

```
36      public void clear()
37      {
38        name.set("");
39        price.set("");
40        error.set("");
41        description.set("");
42        types.add("coffee");
43        types.add("tea");
44        types.add("snack");
45        types.add("smoothie");
46      }
```

It could have been more dynamic to have put them as a list inside the model and fetched it from the Model to the View Model.

Other than that, something else worth mentioning is the submit() method when the Submit button is pressed.

```
78      public void submit()
79      {
80        try
81        {
82          if (name.get().equals("") || description.get().equals("") || chosen.get()
83              .equals("") || price.get().equals(""))
84          {
85            throw new IllegalArgumentException();
86          }
87          double itemPrice = Double.parseDouble(price.get());
88          Item item = new Item(name.get(), chosen.get(), itemPrice,
89              description.get());
90          System.out.println(item);
91          model.addItemToProductList(item);
92          clear();
93        }
94        catch (NumberFormatException e)
95        {
96          error.set("The price has to be a number with a dot.");
97        }
98        catch (IllegalArgumentException e)
99        {
100         error.set("The fields cannot be empty.");
101       }
102
103     }
```

Here are the checks for everything that can go wrong in this window. If any of the fields are empty, an exception is thrown and caught right after at the end of the method. When something goes wrong, the error label is set to describe what the problem was. The other thing that can go wrong is when parsing the Double from the Price field. Since this is a text

field, it is possible for a user to insert numbers or undesired characters there. This would make it impossible to parse a Double. The exception possibly thrown by this method is Number Format Exception, which is also dealt with in a catch block by telling the user how the Price field should be filled. This exception would also be thrown if the value inserted featured a comma instead of a dot (e.g., “9,10” instead of “9.10”).