

Restaurant Helper

1. Objectives

The aim of this application is to create a simple system that can help manage a restaurant, targeting three types of users: administrators that can alter the restaurant menu, waiters that receive customers' orders and chefs that need to be notified what dishes to prepare next.

In order to achieve this aim, we need to touch upon the following sub-goals:

- to create a model that simulates a restaurant scenario, involving multiple users
- to set up a system of notifications alerting the chef when new orders are received (or when orders change)
- to provide a graphical interface to the target users that allows them to interact with the application
- to save data locally between sessions, so that it is easy to continue from a previous point

These goals will be treated simply in the class design section of Chapter 3, and will then be clarified in Chapter 4.

2. Problem Analysis

This chapter aims to describe the requirements of the problem as well as how the end-users will interact with the application, together with any assumptions made about the input of such end-users.

2.1. Functional Requirements

The finished application must provide the following functionality:

- Provide an interactive interface to the target users, designed for each to suit their particular goals

- Permit administrators to add, delete or edit items on the menu, including simple foods and complex courses that are composed, in turn, of simpler ones
- Permit waiters to add, delete or edit orders, including specifically what dishes have been ordered, as well as providing them with receipt-generating functionality on demand
- Permit chefs to see what dishes are required, as well as allowing them to mark as completed dishes that they have finished cooking and notifying them automatically when new items must be prepared (or when an order has been changed or cancelled in a way that affects their work)
- Store restaurant data such as the menu and current orders on shutdown, then reload them on startup to allow persistency of the menu and writing down of orders ahead of time (for example, the night before when they are due)

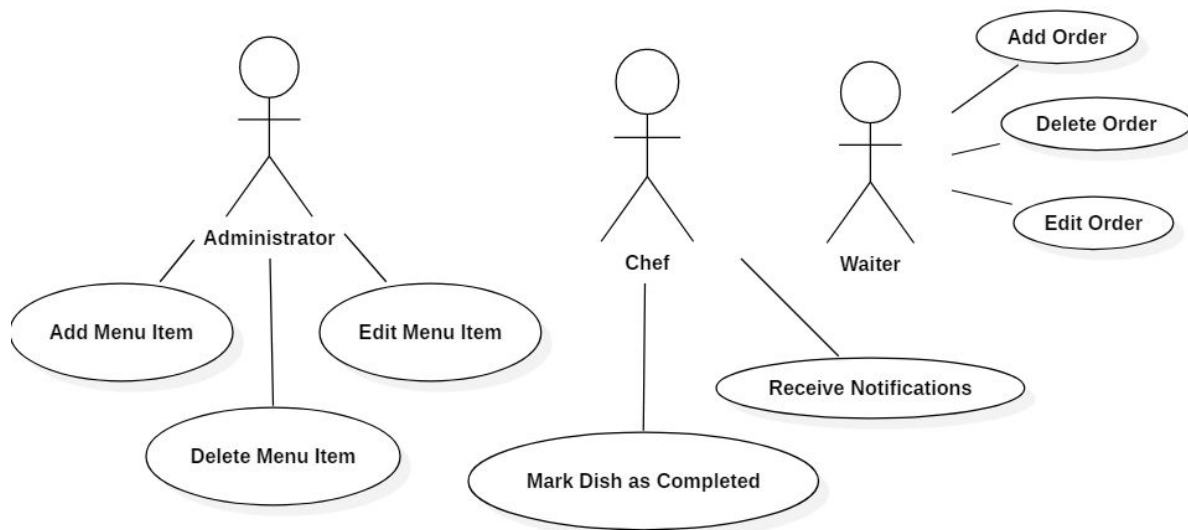
2.2. Assumptions

We have not made any assumptions about the input users provide. In the case of incorrect input the application (most probably will) simply output an error message stating what went wrong, then wait for further user input.

We do, however, make the assumption that users are fairly familiar with the application and do not have need for anything more than titles and very simple tips to successfully understand how to operate the application.

2.3. Use Cases

The following diagram shows the possible use cases for this application:



Use case: add a new order

Actor: waiter

Preconditions:

- The application is running
- There is at least one item on the menu

Main success scenario:

1. The waiter selects that he/she wants to add a new order
2. The program provides a form for the waiter to fill in
3. The waiter correctly fills in the relevant data and selects the requested dishes
4. The system stores the order
5. The system notifies the chef which dishes need to be prepared

Alternative sequences:

A. Wrong input

3. The waiter inputs wrong data, for instance an invalid table number or an order with no dishes
4. The system notices the error and outputs a message to the waiter explaining what was wrong

5. The use case returns to step 2

Use case: edit a menu item

Actor: administrator

Preconditions: none

Main success scenario:

1. The administrator starts the application
2. The system loads the locally-stored configuration data
3. The system displays the current menu in a table
4. The administrator selects the item he/she wishes to edit
5. The system provides a form for editing the item
6. The administrator inputs the desired changes
7. The system stores the altered item and updates the displayed table
8. The administrator closes the application
9. The system saves the new data into the configuration file

Alternative sequences:

A. Missing configuration file

2. The system cannot locate the configuration file
3. The system creates a new, empty configuration and warns the user that the configuration was not found
4. Either the use case continues from step 3, allowing the administrator to add some items first, or the use case fails until the configuration file is found.

B. Cancelled edit

6. The administrator decides to cancel the operation
7. The system does not commit the changes, keeping the previous version of the item
8. Either the use case continues to step 8 or it returns to step 4 and the administrator chooses another item.

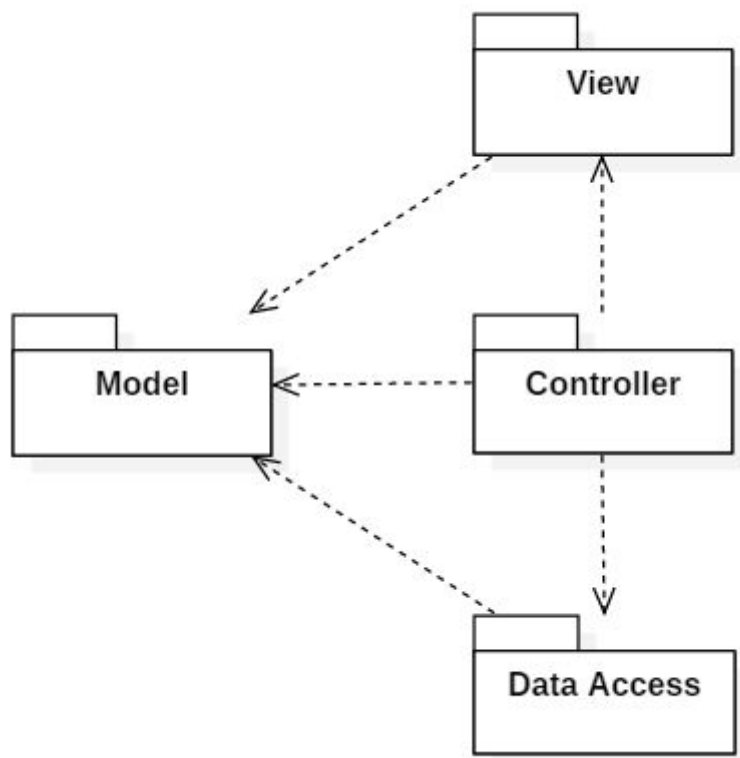
3. Design

This chapter deals with how the the problem is approached from an object-oriented perspective, how it will be split into packages, what

interfaces and data structures will be used and the algorithms that will be employed.

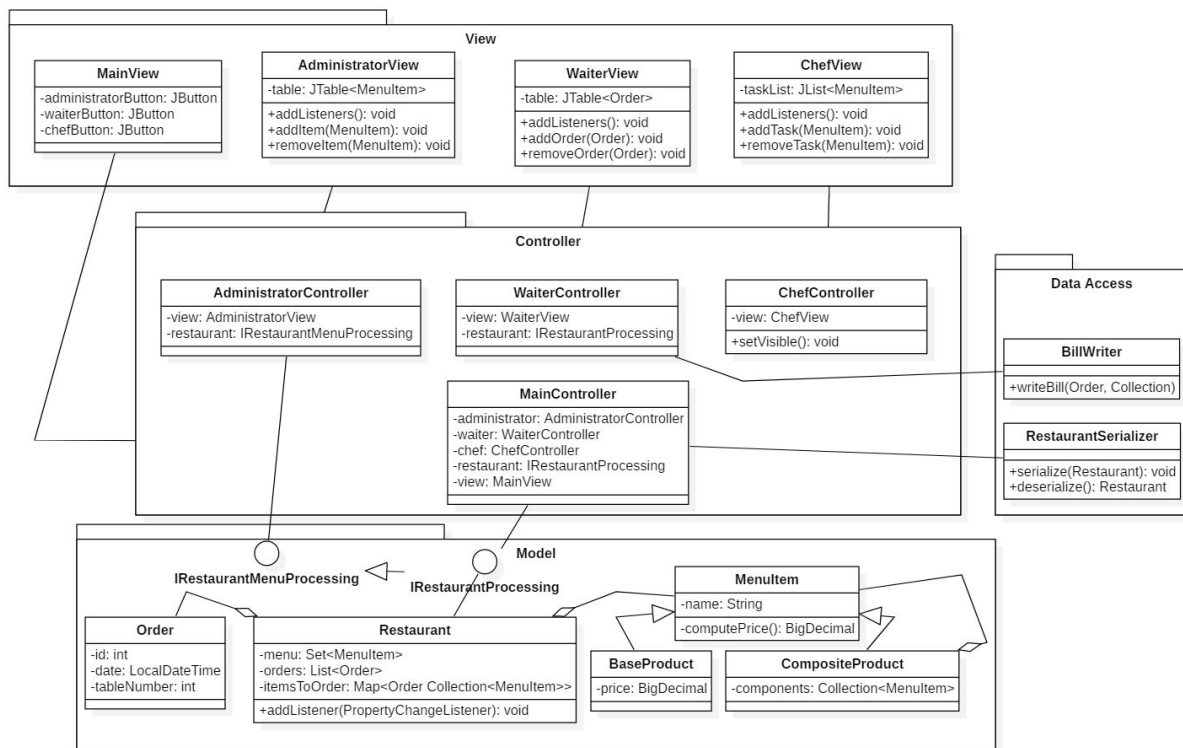
3.1. Package Diagram

The application, due to its small size, will be built on a simple MVC package structure, as shown in the diagram below:



3.2. Class Diagram

In this chapter, we will discuss what kind of classes we wish to implement in each package and explain our reasoning. Since the application is relatively small, we will include an application-wide class diagram below:



The important parts are as follows:

- There is a separate view for each user, and a controller for each. This is because each user has quite a bit of unique functionality required, and putting them all together into a single class would have made it far too cumbersome.
- We choose to store the menu of a restaurant as a *Set* - because duplicate entries make no sense in the context of a menu -, but the orders as a *List* - because it would not be nonsensical to have two (almost) identical orders at once.
- There are two interfaces instead of only one - that is because it is logically coherent to only have access to the menu, without the ordering-related functionality - and one extends the other, because order-related functionality cannot work in the absence of a menu.
- Most importantly, *there is no Observer and Observable*.

Nevertheless, our application aims to obtain the same functionality using listeners (more specifically `PropertyChangeListeners`). The reason for this is that as of Java 9, `Observable` has been marked as deprecated - not because the design pattern's idea is bad, but because the actual implementation carries along difficulties, such as having to inherit `Observable` and having little control over which

Observers receive what. We chose to circumvent those difficulties by using listeners: the `IRestaurantProcessing` interface allows connecting listeners to the underlying structure. When initialised, the `ChefController` connects a such listener to our dataset, which allows it to be notified of changes even though it does not store any reference to the restaurant later.

4. Implementation

This chapter describes the classes and interfaces in more detail, covering the most important methods and aspects of each.

Due to constraints on the length of this documentation, we will only include the most relevant areas of code here, while attaching the entire code separately.

4.1. The Model package

The Model package deals mostly with entities. In this subchapter we will handle the data classes quickly and allocate most of our time to the `Restaurant` class and its associated interfaces.

It can be noted that we make ample use of the Lombok plug-in in this section, to avoid having large amounts of boilerplate code.

4.1.1. *BaseProduct*

```
@NoArgsConstructor
@EqualsAndHashCode(callSuper = true)
public class BaseProduct extends MenuItem {

    private static final long serialVersionUID = -5747622277188617506L;

    @Setter
    private BigDecimal price;

    // We need this, because Lombok doesn't know how to call super like this
    public BaseProduct(String name, BigDecimal price) {
        super(name);
        this.price = price;
    }
}
```

Notable in the above code:

- We declare a *serialVersionUID*. This is important for serialization, so that the JVM can tell that the objects it's deserializing are of the correct type and version as the ones it expects.
- We explicitly declare an all-argument constructor, although Lombok can do that already. This is because the Lombok plug-in cannot handle inheritance scenarios such as the call to a non-empty `super()` in the first line of the constructor.

2.1.2. *CompositeProduct*

```
@Override
public String toString() {
    String stringForm = "CompositeProduct [name=" + name + ", subproducts=";

    StringJoiner products = new StringJoiner(", ", "{", "}");
    for(MenuItem item : subproducts) {
        products.add(item.toString());
    }
    stringForm += products.toString() + "]";

    return stringForm;
}
```

A simple implementation of `toString()` using a `StringJoiner` to concatenate and delineate subproducts.

2.1.3. *MenuItem*

```
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
public abstract class MenuItem implements Serializable {

    private static final long serialVersionUID = 8003098423871292672L;

    @Getter @Setter
    protected String name;

    public abstract BigDecimal computePrice();
    public abstract String toString();
}
```

The abstract class that most objects try to use when dealing with menu entries. It is interesting to note that `computePrice()` has vastly different implementations in the two child classes: for `CompositeProduct`

it is calculated, whereas in the case of BaseProduct it is basically a getter method. Notice how MenuItem implements Serializable.

4.1.4. Order

```
@Data
@AllArgsConstructor
@EqualsAndHashCode
public class Order implements Serializable {
    private static final long serialVersionUID = -8445679034099576539L;

    private int id;
    private LocalDateTime date;
    private int tableNumber;
}
```

A simple data class, due to following the guidelines presented in the lab. Note the usage of the “@Data” Lombok annotation.

We feel that it is somewhat inappropriate that the Order class does *not* contain a collection of MenuItem objects, given that each Order is associated to exactly one such collection anyway. Moreover, most of the data one usually wants to access about an Order is actually about this collection, instead.

4.1.5. Restaurant

```
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class Restaurant implements Serializable, IRestaurantProcessing {

    private static final long serialVersionUID = -4926370622648439074L;

    @ToString.Exclude
    private transient PropertyChangeSupport observableSupport = new PropertyChangeSupport(this);
```

The last line of code in the above excerpt is the most important: in order to use listeners instead of the Observable pattern, we need something playing the role of the array of Observers. That something is the PropertyChangeSupport, which is used as a gateway for all listener-related tasks. Notice how it is excluded from the Lombok toString() method and how it is marked “transient” so that it is not serialized.

```
// We need the method below, since observableSupport is lost in serialization
private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    observableSupport = new PropertyChangeSupport(this);
}

public void addChangeListener(PropertyChangeListener l) {
    observableSupport.addPropertyChangeListener(l);
}
```

Because we do not serialize the PropertyChangeSupport, it is vital that we create a new instance of it when deserializing an object, since the constructor does not get run in these circumstances.

Also, below is portrayed an example of adding a listener, much like the addObserver() method in Observable.

```
public void editOrderContents(Order ord, Collection<MenuItem> newItems) {
    Collection<MenuItem> oldItems = itemsPerOrder.get(ord);
    itemsPerOrder.replace(ord, newItems);

    observableSupport.firePropertyChange("orderProducts", oldItems, newItems);
}
```

Above is the equivalent of notifyObservers(): the firePropertyChange() method. It has the added benefit of being able to specify a property name, so that only those listeners waiting on that property are notified. Moreover, it allows us to easily pass both the old and the new value to observers, so that they can easily update.

4.1.6. IRestaurantMenuProcessing

```
public interface IRestaurantMenuProcessing {
    public Set<MenuItem> getMenu();
    public void addToMenu(MenuItem item);
    public void removeFromMenu(MenuItem item);
    public void editInMenu(MenuItem oldItem, MenuItem newItem);
    public void addChangeListener(PropertyChangeListener l);
}
```

This interface exists to underline the fact that it is logically consistent to work with only the menu of a restaurant, entirely ignoring orders. This allows us to mask the order-handling mechanisms from those classes that don't actually have any need of them.

4.1.7. *IRestaurantProcessing*

```
public interface IRestaurantProcessing extends IRestaurantMenuProcessing {  
    public Collection<Order> getOrders();  
    public Map<Order, Collection<MenuItem>> getItemsPerOrder();  
    public void addOrder(Order ord, Collection<MenuItem> items);  
    public void removeOrder(Order ord);  
    public void editOrder(Order oldOrd, Order newOrd);  
    public void editOrderContents(Order ord, Collection<MenuItem> newItems);  
}
```

This interface adds the order-related functionality. Notably, it extends the previous interface, because while MenuItems without Orders make sense, Orders without MenuItems do not quite make much sense.

4.2. *The View package*

The View package, understandably, manages views and UI. Except for the occasional very simply functionality, it does not implement anything itself, instead “outsourcing” its buttons and actions to controllers via listeners (again).

Most of the view classes are pretty straightforward, so we will present only one, show images of the UI, then focus on the more interesting dialogue classes.

4.2.1. *The views*

```
private JPanel mainPane;  
    private JScrollPane tablePane;  
        private JTable menuTable;  
            private MenuTableModel tableModel;  
private JPanel buttonPane;  
    private JButton newButton = new JButton("New");  
    private JButton editButton = new JButton("Edit");  
    private JButton deleteButton = new JButton("Delete");
```

Above is the structure of the AdminView class. We are particularly fond of this indented style of writing components for views, because it quickly identifies subcomponents while remaining compact. We are not certain yet, though, whether other people find it legible or to the contrary.

```

private class MenuTableModel extends AbstractTableModel {
    private static final long serialVersionUID = 3381349039513231196L;

    private final String[] columnNames = {
        "Name",
        "Components",
        "Price"
    };
    @Getter @Setter
    private List<MenuItem> data;
}

```

For better flexibility and quality of display, as well as to allow in-line editing where it would make sense, all table-based views have a private subclass of AbstractTableModel tailored specifically for them.

4.2.2. The dialogs

The role of dialogs is to act as forms for editing or creating new table entries. A dialog is created when the user presses a button to create or edit a field. Then, the user inputs data and can opt to confirm or cancel the operation. If the input is good and the user continues with it, then the new entry is added to the table and the data by the controllers. Else, the change is simply discarded.

```

public OrderDTO getOrder() {
    int option = JOptionPane.showConfirmDialog(null, mainPane, "Cutomise menu item",
        JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE);
    if(option == JOptionPane.OK_OPTION) {
        int id, tableNumber;

        try {
            id = Integer.parseInt(this.id.getText());
        } catch(NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Error - id must be an integer!", "Error",
                JOptionPane.ERROR_MESSAGE);
            return null;
        } // TODO: check the id for uniqueness, too!

        try {
            tableNumber = Integer.parseInt(this.tableNumber.getText());
        } catch(NumberFormatException e) {
            JOptionPane.showMessageDialog(null, "Error - table number must be an integer!", "Error",
                JOptionPane.ERROR_MESSAGE);
            return null;
        }

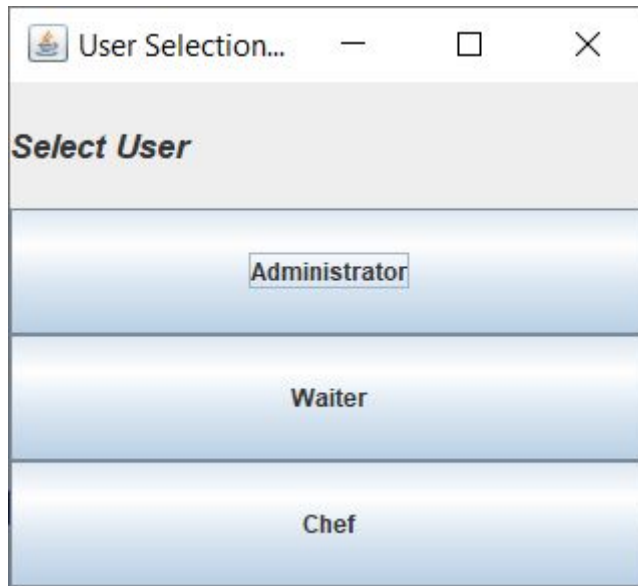
        int size = currentList.getModel().getSize();
        if(0 == size) {
            JOptionPane.showMessageDialog(null, "Error - order cannot be empty!", "Error",
                JOptionPane.ERROR_MESSAGE);
            return null;
        }
    }
}

```

A snippet of code from the method that actually displays the dialog and waits for user response is shown above. Notice how the data is checked for correctness before being accepted.

4.2.3. *The UI*

Next, we will present a few images of the UI and mention any specific features if needed.



The initial window, from which the user can be selected. It is not strikingly beautiful, but was never optimised to be, given that this project focused on the programming aspect.

Administrator view		
Name	Components	Price
Chips	-	4.12
Fish	-	5.77
Fish 'n Chips	Fish, Chips	9.89
Brit Meal	Soup, Fish 'n Chips	13.03
Swimming Fish	Fish, Soup	8.91
Fish 'n Crepes	Fish, Crepes	10.76
Soup	-	3.14
Crepes	-	4.99

New	Edit	Delete
-----	------	--------

The administrator page, from which the menu entries can be changed.

Cutomise menu item

Name

Swimming Fish

Composite item

☒

(Use Ctrl-click to select multiple items)

Chips

Fish

Fish 'n Chips

Brit Meal

Fish 'n Crepes

Soup

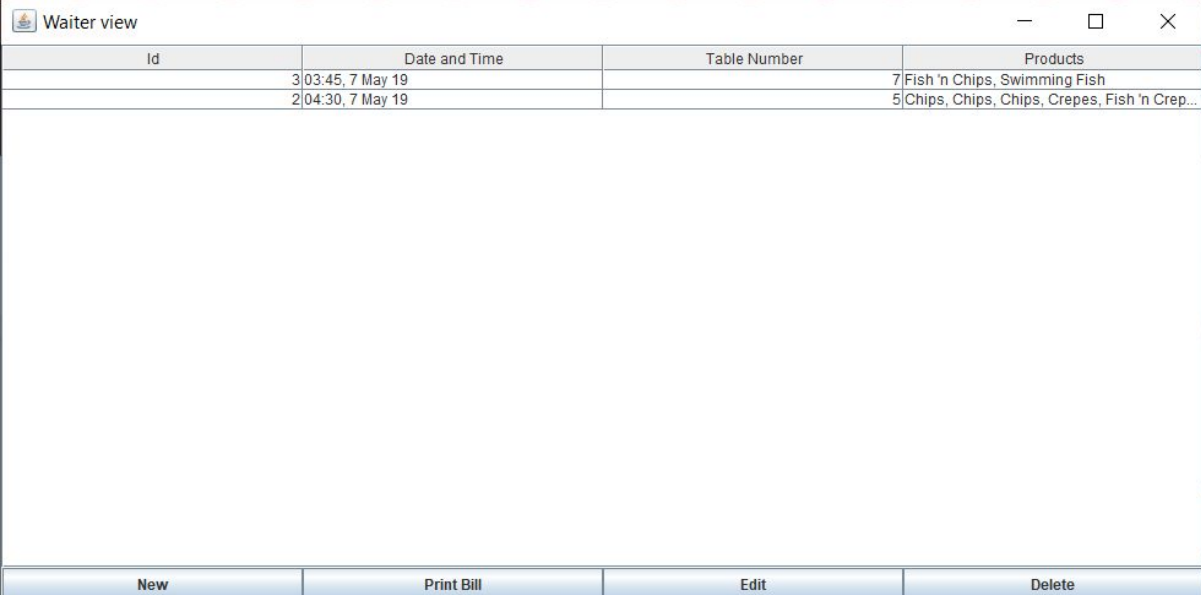
Crepes

OK

Cancel

The editing screen that we get if we select “Swimming Fish” and then click the “Edit” button. Notice how all the fields are pre-filled so that if a user clicks “Edit”, then “OK”, they will get the same item they started with.

Also, note how there is a check box for whether the item is composite or not. Depending on whether the box is ticked or not, the dialog uses a card layout to present two different appearances.



Id	Date and Time	Table Number	Products
3	03:45, 7 May 19		7 Fish 'n Chips, Swimming Fish
2	04:30, 7 May 19		5 Chips, Chips, Chips, Crepes, Fish 'n Crep...

New Print Bill Edit Delete

The waiter view. The waiter has an additional button for printing a bill of the selected Order. This will be discussed in more detail when talking about the BillWriter class.

Cutomise menu item

×

Id

Table Number

Current order:

Swimming Fish

Swimming Fish

Remove from order

Select an item to add

Chips

Fish

Fish 'n Chips

Brit Meal

Swimming Fish

Fish 'n Crepes

Soup


Crepes

Add to order

OK

Cancel

The new-order dialog, into which two portions of “Swimming Fish” have already been added. Notice the simple, intuitive interface.

 Chef view

— □ ×

Current tasks

Fish 'n Chips

Swimming Fish

Chips

Chips

Chips

Crepes

Fish 'n Crepes

Mark Completed

The Chef view is rather plain, but has all the needed functionality, given that the chef only needs to know what to prepare. It would, perhaps, have been useful to include a list of ingredients for each cooking task, but we believe that its lack will not greatly affect the chef.

4.3. The Data package

This package has only two classes: one that serializes Restaurant objects for storing and reloading data, and one that prints receipts. We will treat them in the following passages.

4.3.1. RestaurantSerializer

```
public void serialize(Restaurant subject) {
    try(FileOutputStream fout = new FileOutputStream(path + fileName);
        ObjectOutputStream out = new ObjectOutputStream(fout)) {

        out.writeObject(subject);
    } catch (IOException e) {
        System.out.println("An unexpected error occurred while serializing " + subject.toString() + ".");
        System.out.println("Please consult the stack trace for details.");
        e.printStackTrace();
    }
}
```

Serialization and deserialization are very simple, to the point where error-handling takes more code than the task itself. This is because, once properly set-up in the classes, native Java classes can handle the rest.

Notice, also, the try-with-resources structure that guarantees that the output streams will be closed, regardless of whether there are any errors or not.

4.3.2. BillWriter

```
Map<MenuItem, Long> itemAndAmount = products.stream()
    .collect(Collectors.groupingBy(
        item -> item,
        Collectors.counting()));

BigDecimal total = ( products.stream()
    .map(MenuItem::computePrice)
    .reduce((a, b) -> a.add(b))
    ).get();
```

The BillWriter is very, very similar to the receipt writer of the previous assignment. Thereby, we only present the new and improved parts, such as the code above that makes use of streams to achieve more concise code.

4.4. The Controller package

Finally, the Controller package handles a significant part of the logic, especially in terms of setting up listeners and handling user input.

4.4.1. AdminController and WaiterController

We choose to treat these two classes together because of their high similarity. Their task is mostly to handle the effects of button presses from the user.

```
view.attachEditListener(e -> {
    MenuItem itemToEdit = view.getSelectedRowData();
    Collection<MenuItem> possibleComponents = menuRestaurant.getMenu().stream()
                                                .filter(i -> (!isOrContains(i, itemToEdit)))
                                                .collect(Collectors.toList());

    MenuItem newValue;
    if(itemToEdit instanceof BaseProduct) {
        MenuItemCreationDialog dialog = new MenuItemCreationDialog(possibleComponents,
                                                                    (BaseProduct)itemToEdit);

        newValue = dialog.getItem();
    } else {
        MenuItemCreationDialog dialog = new MenuItemCreationDialog(possibleComponents,
                                                                    (CompositeProduct)itemToEdit);

        newValue = dialog.getItem();
    }

    if(null != newValue) {
        menuRestaurant.editInMenu(itemToEdit, newValue);

        view.setData(menuRestaurant.getMenu());
    }
});
```

Above is the listener for when the user clicks the “Edit” button on the AdminView. Its functionality can be boiled down to creating the appropriate dialog, then checking whether it has returned any useful results and storing them, if needed. There are a few interesting tricks such as the possibleComponents line (which avoids situations in which editing an item would allow the user to have an item have itself as a subcomponent).

4.4.2. UserSelectionController

```
public UserSelectionController(IRestaurantProcessing restaurant) {
    this.restaurant = restaurant;

    this.chef = new ChefController(restaurant); // Chef is actually always "awake".

    view = new LoginView();
    attachViewButtonListeners();
    view.setVisible(true);
}
```

This class does not have much functionality, as its main role is to get the other, more specialized views running. We have attached the above code only to highlight how the ChefController is running for the whole duration of the application, since it cannot receive notifications if it is not instantiated (this applies to Observer objects, too).

4.4.3. ChefController

```
private void attachListeners(IRestaurantProcessing restaurant) {
    restaurant.addChangeListener(e -> {
        if(e.getPropertyName().equals("orderProducts")) {
            if(null == e.getOldValue()) {
                // Something new was added

                @SuppressWarnings("unchecked") // If we got here, it's definitely correct to cast
                Collection<MenuItem> tasks = (Collection<MenuItem>) e.getNewValue();

                for(MenuItem i : tasks) view.addTask(i);
            } else if(null == e.getNewValue()) {
                // Something was removed; technically this is optional!

                @SuppressWarnings("unchecked") // If we got here, it's definitely correct to cast
                Collection<MenuItem> tasks = (Collection<MenuItem>) e.getOldValue();

                for(MenuItem i : tasks) view.removeTask(i);
            } else {
                // Something was edited; technically here we could just add
                // what's new and ignore what was removed!

                @SuppressWarnings("unchecked")
                Collection<MenuItem> oldTasks = (Collection<MenuItem>) e.getOldValue();
                @SuppressWarnings("unchecked")
                Collection<MenuItem> newTasks = (Collection<MenuItem>) e.getNewValue();
            }
        }
    });
}
```

Above is an example of how the Observer replacement works. In the constructor, we receive an IRestaurantProcessing object that we *store no references to afterwards*. We merely add a ChangeListener that does something when the Restaurant class fires change events. In this case, it is easy to use, since we have quick access to both the new and the old values, unlike the usual update(Object) method.

5. Results

Based on extensive test runs of the application, its performance has proven acceptable, its interface manageable enough and no errors have been noticed. The program has not been unit-tested in any way, but since the main issues that could arise in a such program are related to the UI, unit-testing would not be the best way to go about it anyway.

6. Conclusions

I believe this topic has been a good exercise in working with user interfaces. The dialogues especially have been a good source of practice.

There is only one topic on which I would choose to complain: the map-based structure recommended in the lab is, I feel, quite inefficient. I feel serialization is very useful for storing setting, user preferences, game saves and so on, but for an application such as this, a database structure would be more fitting (for example, because it's not impossible for the administrator to edit the menu while there are still things stored over, which might cause trouble for everyone later).

7. Bibliography

1. <https://stackoverflow.com/questions/46380073/observer-is-deprecated-in-java-9-what-should-we-use-instead-of-it> for proposing the listener-based alternative to the Observer pattern
2. <https://stackoverflow.com/questions/6234893/jlist-select-multiple-items> for helping me understand how to select multiple list entries via code
3. <https://stackoverflow.com/questions/23925315/count-int-occurrences-with-java8> for inspiration in dealing with streams and collectors
4. <https://stackoverflow.com/questions/16214480/adding-elements-to-a-jlist> for suggesting explicitly declaring a DefaultListModel to permit adding elements to a JList
5. <https://stackoverflow.com/questions/9955595/how-to-display-multiple-lines-in-a-jtable-cell> for showing exactly how much code having genuine multi-line table cells would take