Group Report Stage 1

Date: 31/1/2019

Group 2

Authors:

Ioannis Athanasiou ia50@hw.ac.uk
Nanduri Ram rnn1@hw.ac.uk
Schatton Bartosz bps2@hw.ac.uk
Thomas Triffterer tt63@hw.ac.uk

Program: Café Register

State: Based on our Assumptions, this program meets the specification fully

Link to the Repository: https://github.com/trz-maier/hwu-ase-group3

Task:

- John: OrderController class and OrderController test class.
- Thomas: FileReader class and FileReader test class.
- Ram: Bill, Item, Order, OrderItem classes and Bill test class.
- Bartosz: OrderFrame class.

Releases

- V0.1: First prototype release able to load and display items in Frame
- V0.2: Functional order creation as well as adding/removing items from order
- V0.3: Fully functional version in accordance with specification
- V1.0: Final version tested and approved for submission

Changed classes

The following changes were made compared to the initial design:

- Additional OrderFrame interaction methods
    - setOrderItems()
    - setOrderTotals()
    - setBillString()
- Additional OrderController methods
    - validateCustomerId() for public usage across the app
- Model Changes
    - Bill.showBill() --rename--> Bill.getBillString()
    - Added missing Order.removeOrderItem()
    - OrderItem now implements Comparable interface to allow for sorting of OrderItems which is needed for discounts.

Data Structures

- Inventory - TreeMap (key: itemId, value: Item object)
    - keeps items sorted using a key of type:= `name+id`
    - relatively quick to search
    - adding and removing inventory items is expected to be rare
- Orders - ArrayList of Order objects
    - keeps orders sorted in the order they are added
    - easy to add new orders
    - iterating through orders only required at application exit
- OrderItems - ArrayList of Item objects
    - keeps order items in the order they were added
    - easy to add a new item to the end of the list
    - easy to access and remove items by index
- FileReader – Hashmap (key: UUID, value: Item object)
    - While parsing the Items and Orders files it is necessary to check if an Item with the specific UUID already exists.
    - In case the Item exists, we need a reference to this Object.

Limitations

The overall design, while mostly good for representing our model and associations, ended up being restrictive and problematic for doing Unit Testing. For example, when appropriate encapsulation, limited access scopes and separation of concerns are practised, it can be challenging or impossible to verify the inner states of certain classes. A good example would be the `OrderController` not needing a getter for its pending order to restrict access via certain add/remove operations. This ended making tests which should be verifying the state of this pending order quite difficult.

Known Issues & Concerns

- Displayed bill is properly aligned for item prices and totals not larger than £99.99, for a value greater than this the digits extend to the right and therefore not perfectly aligned.

Design considerations

`OrderItem` class could be avoided for the time being, but it provided a comfortable level of abstraction, allowing us to have a very restricted set of Item instances. Furthermore, this allows for possible future changes to be incorporated more easily, such as supporting products with different (base) prices per order (e.g. due to VAT increase).

Assumptions

- Prices will not be changed, since that would cause inconsistencies between generated reports of different time periods.
- For the `TreeMap` data structure, items will have an id with a name prefix along the lines of <item_name><unique_id> so as to provide both an alphabetically (pre)sorted list of items in the inventory, as well as efficient retrieval through their id being used as a key for the TreeMap. This further means that we do not expect item name's to be changed.
- Getter/Setter methods & data structures for method-local variables are omitted from the UML diagram as their usage will be dependent on the implementation and each will be used as-needed.
- The main running method of our application, as well as its corresponding class, are not shown in the UML since they have no ontological value.
- Customer ID is expected to contain exactly 8 alphanumeric characters and is validated for that.
- FileReader input files are expected to be found inside the designated resources folder (`/res`)
- The **Orders file** is expected to be a comma-separated csv file with each line representing a single item for an order in the format:

  `<CustomerID>,<Date>,<Item>`

- The **Items file** is expected to be also a comma-separated csv file with each line representing a specific kind of product (“same barcode”, not “same category”). Each line is expected to have a format of

  `<ItemID>,<itemName>,<CategoryName>,<Price>`

- **Discounts**
  - If you buy a coffee and a newspaper, the newspaper is free.
  - You get 20% discount on any combination of Food+Drink (applicable to the cheapest combinations).
  - If you buy 3 Jam Doughnuts you get one of these for free.

Testing

The unit tests were structured into suites based on the class whose functionality they are validating, along with having corresponding names (Eg. `FileReaderTest`)

- **Bill**: Testing three dummy Orders with a number of OrderItems for the calculation of the subtotal, discount and total methods to match the expected values. The orders are created in such a way as to ensure that all types of discounts and all flow of control in calculating discount are covered.
- **FileReader**: We test if the FileReader class behaves correct if an invalid path is parsed to the file reader (null pointer, not existing file or empty file name). We have two test cases which test if a valid Items and Order is parsed correctly. We also added test cases for an empty file, A files with invalid Separators, Categories, UUIDs and prices.
- **OrderControler**: Multiple scenarios were run for the various methods
    - Different cases and precise Exception type/message for Customer ID validation, since this static method is providing this logic across all packages.
    - Success/Fail cases with Exception type matching for creation of orders and addition of OrderItems.
    - Removal of Items and the Report Generation processes could not be easily checked without severely changing the access scope and/or design approach for multiple elements of the project. In retrospect, if we had initially followed a more strict MVC pattern, it might have been easier to perform more checks.

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	100% (14/ 14)	74.4% (61/ 82)	65.5% (410/ 626)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
IO	100% (1/ 1)	90.9% (10/ 11)	96.2% (102/ 106)
control	100% (1/ 1)	52.9% (9/ 17)	49.5% (49/ 99)
exceptions	100% (3/ 3)	100% (3/ 3)	100% (6/ 6)
gui	100% (4/ 4)	62.5% (10/ 16)	46.6% (125/ 268)
model	100% (5/ 5)	82.9% (29/ 35)	87.1% (128/ 147)

Exceptions

We introduced three exceptions:

- `EmptyOrderException`: Is thrown when the user tries to submit a Order, without any Items.
- `InvalidCustomerIdException`: Is thrown when a new Order with an invalid CustomerId is created.
- `NoOrderException`: For cases where there is no pending order at all.

Furthermore, we used these pre-defined exceptions:

In the `FileReader` Class:

- `IOexception`: Is thrown when an invalid filename is parsed to the function parseOrders or parseItems
- IllegalState`Exception`: for cases where a method invocation is made with valid parameters, but the overall state of the application is not appropriate for it (eg creating a new order without cancelling the previous one).
- Inside the private methods of the class `ParseException`, `IllegalArgumentException`, `InvalidCustomerIdException` and `NoSuchElementException` is thrown if the file format is corrupted.

Class Diagram