F21AS Group 3 Stage One Report

Katie Baker	Ryan Farish	Anthony Gabini	Hamish MacKinnon	Michael Pidgeon
H00327433	H00264812	H00356910	H00257227	H00360036

4th March 2021

1 Link to Project Repository and Project Development Responsibilities

Repository via GitLab: https://gitlab-student.macs.hw.ac.uk/mp2012/f21as-group-3

Each member of the team was designated a particular section of the project to focus on, with a general delineation by back-end (data classes) and front-end (GUI classes) work. The team would meet on a weekly basis and provide update reports on their designated task for that week. When called for, team members collaborated in different areas when requested. Our individual contributions are detailed as following:

Anthony Front-end development of GUI; designed GUI; tying in back-end implementation of the basket functionality to the GUI; created UML Class Diagrams (both Development Stage and Final).

Hamish Project setup; project planning in Trello; Maven package and version management; reading and constructing menu items; automated reporting on program exit, trigger and procedure; bug reporting and fixing, code review, testing.

Katie Project management; designed GUI; developed front-end classes; contributed to data classes; developed front-end functionality for rendering menu items, adding items to basket, viewing basket and checking basket out; code review, software testing, bug reporting and fixing; report writing: Graphical User Interface; ArrayList; HashMap.

Michael Back-end development, including basket checkout and pricing (discounts); reading and initialising orders and customers; unit tests for back end classes.

Ryan Front-end development of GUI headers and button/action functionality; contribution of implementation to basket classes, ensuring back-end tie in; ensuring planned UML contains necessary methods; bug testing general program; responsibilities to ensure layout and formatting of the project is consistent across the code and report write ups.

---

2 Conformance with Specification

This program meets the full specification laid out in *F21AS Coursework 2021.pdf* for Stage 1 with no exceptions.

3 UML Class Diagram

During development we realised that we had to make changes from the planned data structures in our initial UML class diagram (see Use of Data Structures for more details). We have therefore included the updated class diagrams here (see Figure 1 and Figure 2). Due to the size and complexity of the program, we split the class diagrams into back-end and front-end specific diagrams. We include a copy of the overall class diagram in our repository: https://gitlab-student.macs.hw.ac.uk/mp2012/f21as-group-3/-/blob/master/docs/out/ASEFinalClassDiagram/ASEFinalClassDiagram.png.

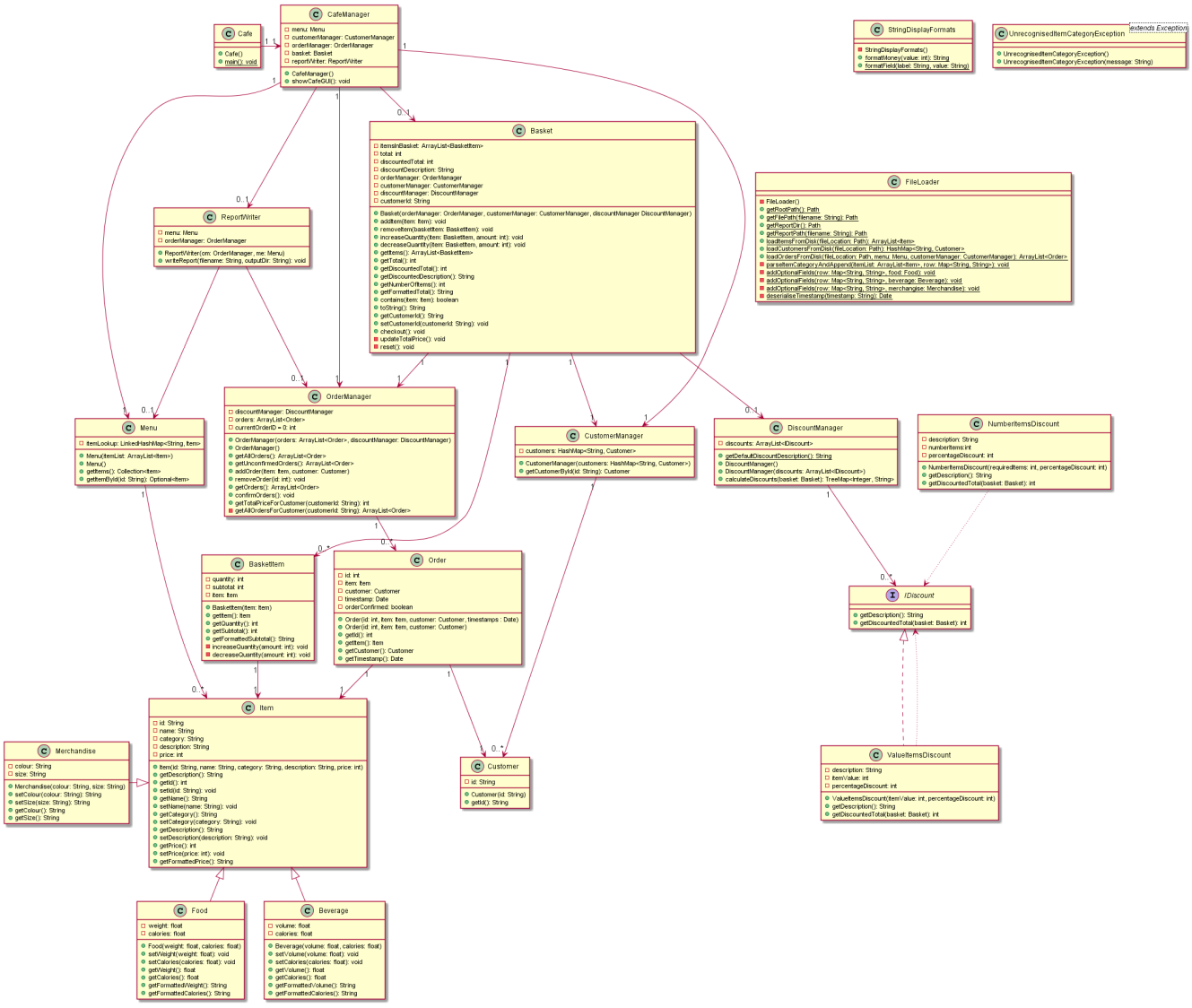

Figure 1: Back-end UML Class Diagram

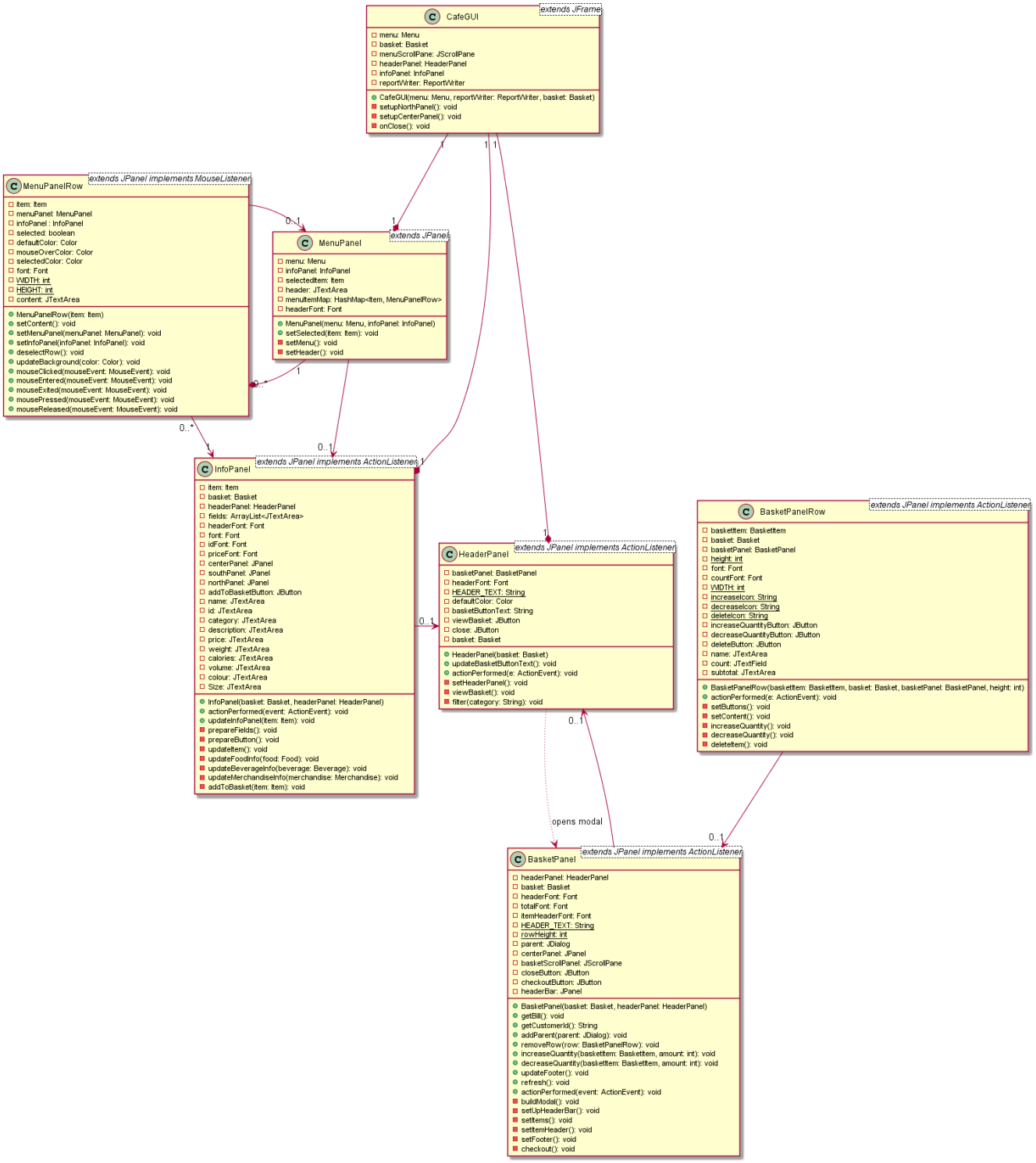

Figure 2: Front-end UML Class Diagram

4 Decisions about Functionality

Decisions about functionality were agreed upon during team meetings, and during sub-team discussions for front-end and back-end. The justification for decisions were based on the belief that they would provide the most functionality while maintaining an efficient outcome that would see benefit to the programmers and/or users.

4.1 ID Format

IDs for menu items are 3 alphabetical characters signifying the category followed by a non-negative integer. This is by convention, as the program does not check this; enforcement of types relies on the Category field of the csv input, and the IDs are used as a unique identifier.

Customer IDs are integers that begin and count up from 1001. Again, by convention.

Order IDs are integers that begin and count up from zero. This is enfored more formally in the `OrderManager` class.

While **reports** are not required to be stored or retrieved by an ID, reports from different runs are recorded in the output directory as "cafe" with a time and date stamp appended to distinguish these reports in an intuitive way.

4.2 Static Helper Classes

When opportunities to extract reusable code arose across the codebase, instead of duplicating the lines of code in these places they instead call the static methods on classes that have no public constructor or other way of being instantiated and storing state. This meant that the same code for formatting currency can be used in the GUI as well as in the report writing. Similarly, the file reading code in `FileLoader` can be used for loading past orders as well as the `Menu`.

4.3 Graphical User Interface

4.3.1 Overall design

The GUI design went through a number of iterations whilst focusing on providing a clear, simple user interface. As our understanding of the requirements became clearer the design crystallised into a single window with multiple panels for displaying data, multiple buttons for triggering functions and a basket modal for viewing the bill. The evolution of our designs can be seen in Figure 3. Figma was used to make the final design.

The final GUI design can be seen in the screenshot of the final software in Figure 4. A number of changes were made from the final design in the actual implementation:

1. Removed the planned functionality to add any number of items to the basket - this was not deemed necessary as the quantity can easily be changed in the basket view.
2. Removed planned buttons for sorting and filtering menu items - due to time constraints we could not develop the menu sorting and filtering we had envisioned. We chose to focus on developing and improving the core functionality of the software instead of implementing new features.

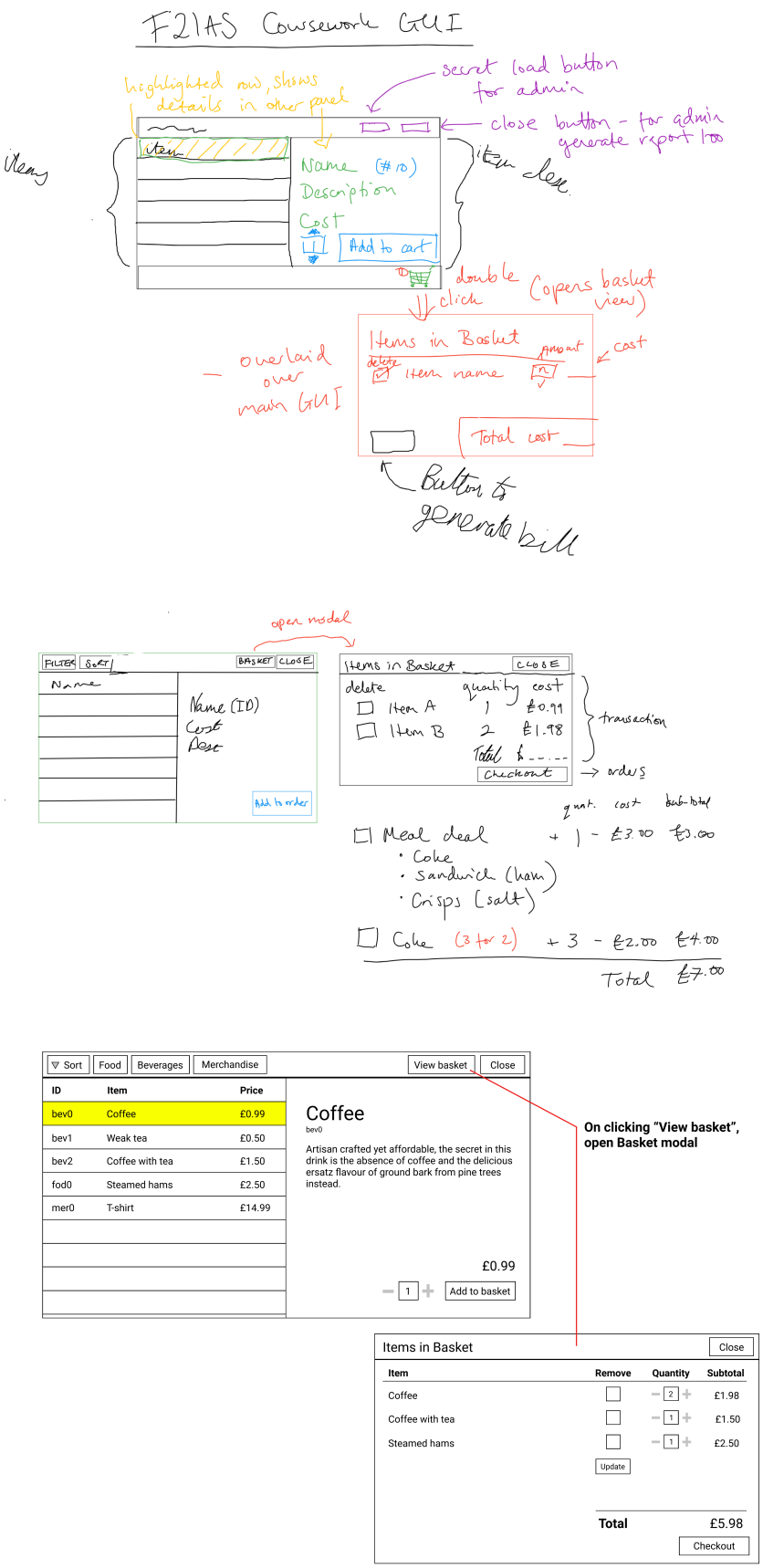

Figure 3: GUI designs, in order of creation

3. Replaced inline checkboxes for inline buttons to remove items from the basket - the inline buttons make the delete functionality clearer and take up less space by not requiring an additional "Update" button.

4. Added a count on the "View Basket" button to show how many items are in the basket - this was added to make it easier for the user to keep track of how many items are in the basket and to provide real-time feedback after the "Add to Basket" button was pressed.

5. Added text indicating the current discount applied - this was added to provide the user with information about why a discount was applied and the resulting new total price.

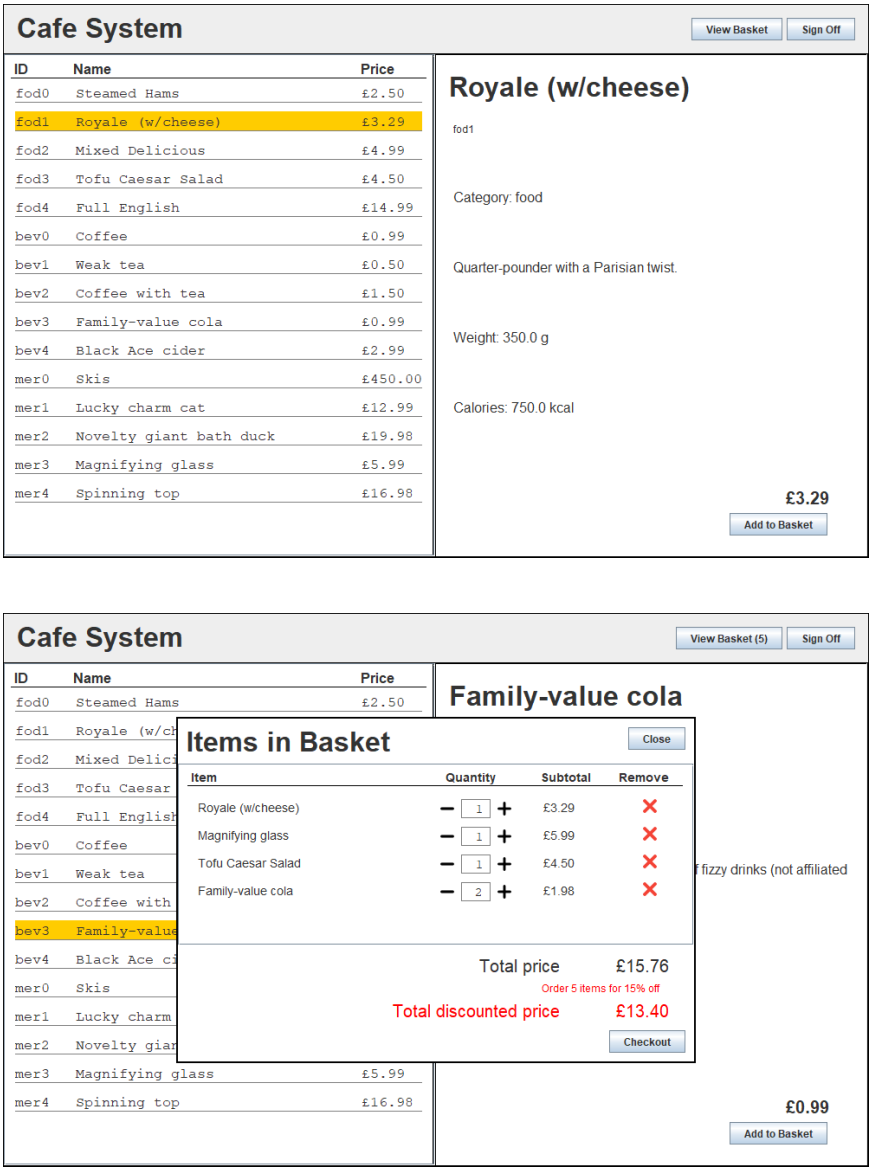

Figure 4: Cafe System GUI

4.3.2 Functionality

The main Cafe System window consists of three panels which show a header bar at the top, the menu on the left and the selected item details on the right. The left panel renders the menu items in rows. These rows are instances of

the `MenuPanelRow` class, which implements the `MouseListener` class to enable a mouseover effect. This turns the background blue and changes the mouse cursor into a hand pointer to indicate they are clickable, which is good for the user experience. Upon clicking, the selected row turns yellow and the item information is rendered in the right-hand side panel. We designed this in such a way that the most pertinent information about all the items (ID, name and price) was visible at the same time whilst allowing more in-depth item detail to still be accessible by the user.

The right panel offers the "Add to Basket" functionality which adds the selected item to the basket. The item is added to the `itemsInBasket ArrayList` in the `Basket` class as a `BasketItem` object. By passing the `Basket` instance to multiple GUI classes, the basket can be updated in multiple places in the software: the aforementioned "Add to Basket" button and the "Remove", "Add", "Minus" and "Checkout" buttons in the basket panel. The changes to the basket cascade throughout the GUI by triggering a repaint of the various `JPanel` objects.

The header panel has a "View Basket" button, which upon clicking triggers the basket modal to render. The basket panel contains rows of basket items and has a number of fields: item name, quantity, subtotal and removal. The basket panel affords a simple, clear view for visualising the bill which includes individual subtotals, final total and discount total.

Finally, the program is exited by clicking the "Sign Off" button in the header bar. This writes a report with the data required by the specification: a list of all the menu items, the number of times each item was ordered, and the total revenue for all orders. A dialog box indicating where the report is written to also opens when exiting the program. This works by hooking into the JFrame close event using a `WindowAdapter` with a function call written into the inherited textttwindowClosing method.

4.4 Report Format

The report writing code uses `String.format` to insert relevant information into a natural language format. An alternative approach would have been to write out tabular data structures (like CSV). However, as the specification asks for a report, more human readable formats seem preferable.

4.5 Item Initialisation

In our iterative development plan, the addition of the derived types of `Item` (`Food`, `Beverage` and `Merchandise`) and the unique fields for each came at a late stage when the various systems already worked without the extra features. To split the work, one team member extended the test data CSV files, another extended the CSV reading code and a final team member modified the GUI to display these extra fields when the relevant item is selected. As we used branch based development, each branch should in theory allow the program to run even with dependant work not implemented. On this basis, and the fact that the reading code arrived first, GUI second, and data third, the Item initialisation code is very permissive and constructs successfully in the absence of the "new" fields. An alternative approach would have been to throw exceptions much more aggressively in the presence of missing data, but this would have been counterproductive to speedy development of an effective working solution.

4.6 Order Management and Discounts

4.6.1 Basket and Checkout

We interpreted the specification as requiring a 1:1 mapping between orders and items, so each order can contain only a single item. We wanted the user to be able to purchase multiple items at the same time, for two reasons:

- Better user experience.
- Allow 'multi-buy' discounts to be offered, where a certain number of items purchased at the same time triggers a discount.

This problem was solved by introducing a `Basket` class, which contains multiple `BasketItem` objects. These are essentially the same as the basic `Item` class, except they allow the quantity to be adjusted.

The `Basket` class offers a *Checkout* method which converts the basket to individual orders, which are stored using the `OrderManager` class.

The overall flow of the `Basket` is summarised in Figure 5.

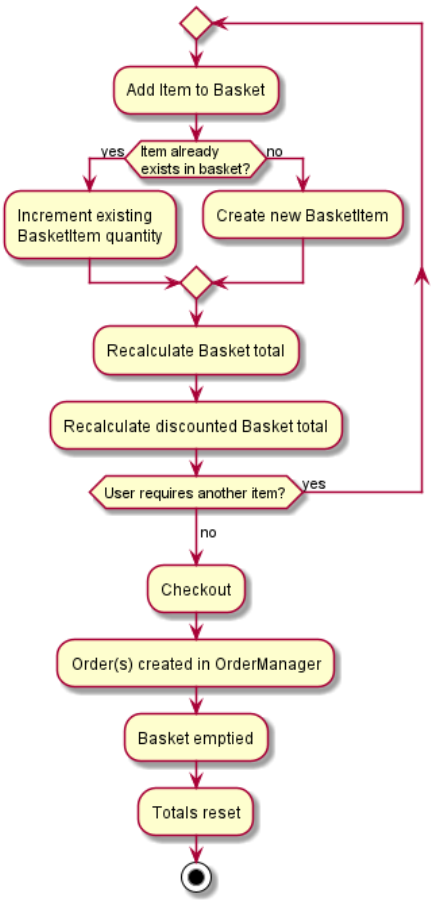

Figure 5: UML Activity Diagram showing simplified Basket flow

4.6.2 Discount Management

Our implementation of discounts is designed for flexibility and extensibility. Currently three discounts are offered, as defined in `CafeManager`:

- Buy 2 items get 5% off
- Buy 5 items and get 15% off
- Buy items valued £20 or more and get 25% off

These discounts are linked to a `DiscountManager` class. Upon any change to the `Basket`, this class calculates the relevant discount amount for each possible discount, and returns the optimal discount. This allows the GUI to update the discount amount and description dynamically when items are added or removed, as shown in Figure 4.

An `IDiscount` interface is defined. This requires any discount class to implement two simple methods as shown below:

Listing 1: IDiscount Interface

```
public interface IDiscount {
        public String getDescription();
        public int getDiscountedTotal(Basket basket);
}
```

This interface means additional discounts can be easily added in future. It also means Java generics can be used to hold a variety of discount classes in the `DiscountManager` class:

Listing 2: DiscountManager use of IDiscount

```
public class DiscountManager {
        private ArrayList<IDiscount> discounts;
        // ...
```

Two concrete implementations of `IDiscount` are provided: `NumberItemsDiscount` (order certain number of items) and `ValueItemsDiscount` (order above a certain value of items). Both can be instantiated multiple times; as shown above, `NumberItemsDiscount` is used twice, to offer a tiered system of discounts.

---

5 Use of Data Structures

The following data structures were chosen based from the requirements of the program. Chosen structures were deemed appropriate to adopt and implement from the advantages they would bring as compared to other data structures, following the consistent structure of the methods within the code.

5.1 ArrayList

An `ArrayList` was used in the `Basket` class to store the items in the basket. The main reason was to retain the order that they were added to the basket, something that we thought would be preferable to the users. The disadvantage to this approach became apparent when developing the basket panel, when items quantities had to be updated. Without a key to lookup the item, the `ArrayList` must be iterated through until there is a match. This makes the code quite inelegant, but does not impact performance as there are usually very few items in a basket. Performance would become more an issue if there were lots of basket items.

An `ArrayList` is also the core data structure in `OrderManager`. Unlike `Basket`, while orders have IDs, there is no need in the program look up orders by their ID. The main requirement is a data structure that can sensibly grow as more and more elements are added to the end, which an `ArrayList` handles cleanly.

In the GUI, an `ArrayList` was used for the `InfoPanel` class to store the `JTextArea` objects. As there are ten fields with very similar field formatting requirements, it was much more efficient to process the fields in a loop than applying the formatting individually. An `ArrayList` was chosen for this as it retains the order of the items and no lookup by key was required.

5.2 HashMap

A `HashMap` was used in the `CustomerManager` class to store the customers, indexed by customer ID. We chose this data structure as we wanted to be able to lookup customers by ID and were not concerned with maintaining or enforcing any ordering.

In the GUI, a `HashMap` was chosen to link the `Item` objects with the `MenuPanelRow` objects, thereby directly associating the data object with the view object. The order the values were stored in this map was not important as the menu item order was maintained in an `LinkedHashMap`. Moreover, this `HashMap` was used to find the `MenuPanelRow` object for a given `Item` in order to change the formatting (removing the yellow row highlighting) so fast lookup by key was required to ensure the GUI remained responsive.

5.3 LinkedHashMap

A `LinkedHashMap` was used in the `Menu` class to store the menu items. An efficient data structure for iteration was required as the items needed to be loaded from lines in a file, stored in the order they were added and then looped through for rendering in the GUI. `ArrayList` is ideal for these requirements, however whilst developing the `FileLoader` and `ReportWriter` classes we needed to lookup items in the menu and came across the same issue as described for the `Basket` class. Items are by the specification defined with an ID and a simple change to `LinkedHashMap` inside the `Menu` class allows for fast lookup by ID using a hash based data structure while maintaining the low complexity iteration from the linked list maintained inside the `LinkedHashMap`.

The report writer extracts information about orders from the `OrderManager` and about the items sold from the `Menu`. The number of objects sold for each item is kept in a key-value store matching the `String` id from the menu to an `int` counting the number of sales. This pairing is useful as it allows simple count incrementation by lookup for each item without requiring sorting or filtering of the orders in advance. A similar map is built for matching IDs to revenues stored with each order. The map lookup and update operation works in constant time, so the summation operation remains

linear relative to the number of orders in the order manager. A `LinkedHashMap` was chosen to preserve the order of the menu items such that it is the same as is displayed in the GUI.

5.4 Collection

The collection interface is used by functions that request the items in the `Menu`. This is a polymorphic facade, not a true data structure itself. In these cases the real data structure is the `LinkedHashMap`, where the linked list aspect is used for iterating through the elements efficiently.

---

6 Testing and Exception Handling

6.1 Test Strategy

Unit tests were written with JUnit for back-end classes. Test classes have the same name as the class they test, with `test` prepended to the name.

As our GUI classes do not contain business logic/calculations, they are more suited to reliability testing, UI testing and automated UI test frameworks like Selenium, instead of JUnit. Due to time constraints we did not implement Selenium testing but did perform thorough UI and reliability testing.

We used GitLab's Issue Tracker to log any bugs we found as part of this testing and performed bug fixing as part of our development iterations. Example: https://gitlab-student.macs.hw.ac.uk/mp2012/f21as-group-3/-/issues/1

6.2 Back-end Unit Tests

Tests written are summarised in Table 1. They are divided into tests for the base case functionality ('happy path'), which assumes no errors or unusual input; and edge case tests, where the application's robustness to unusual conditions is assessed. Not all classes were complex enough to warrant tests for edge cases.

We did not test classes with only trivial logic. For example, `Order` contains only simple getters, therefore does not have any unit tests associated with it.

Some of these tests are arguably less of a unit test, and more of an integration test. For example, the `Basket` tests involve instantiating order, customer and discount managers, passing them appropriate data, and only then testing the Basket functionality itself. We viewed this as a positive: it improves the robustness of our test suite, and the likelihood of errors spanning different classes being caught.

Separate test data CSV files are provided, in order to separate test and production data. In addition, individual classes are mocked out with test data in test helper methods where required.

Table 1: Summary of unit tests for back-end

Class	Base case tests	Edge case tests
`Basket`	Adding and removing items, including multiple items of different types. Calculating discounted and non-discounted price. Checkout process.	Large order results in multiple possible discounts being available; does the app select the correct one. Evaluate different scenarios on checkout to ensure correct rounding, and reconciliation of the basket total to the individual item totals.
`BasketItem`	Altering item quantity and calculating total price of the item(s).	N/A
`Cafe` and `CafeManager`	Instantiation without throwing exception.	N/A
`DiscountManager`	Correct discount calculation.	'No discount' state returns the price without adjustments.
`FileLoader`	Imports for menu, customers and orders.	Items not recognised in order import - check handles in a non-fatal way, skipping over the unknown item.
`Menu`	Getting item by ID.	Attempting to get an unknown item - check returns an empty `Optional`.
`NumberItemsDiscount`	Whether to apply discount; correctly calculated application if required.	Handling rounding up and down appropriately, where required.
`OrderManager`	Bulk import of orders.	Imported order IDs are non-sequential: ensure ID generator handles appropriately, and starts from (highest ID + 1).
`ReportWriter`	A short menu with few orders produces a simple report	Empty menus and order lists summarise correctly
`StringDisplayFormats`	A typical amount (£10.50) converted to string correctly.	Very low and high amounts are displayed correctly, including commas to separate thousands where required.
`ValueItemsDiscount`	Whether to apply discount; correctly calculated application if required.	Rounding, and boundary conditions of whether discount should be applied (e.g. discount starts at £5; make sure applied for £5.00 spend, but not £4.99).

6.3 Exception Handling

We found that, in general, the in-built Java exception classes met our needs. For example, an `IllegalStateException` is thrown in the constructor for `BasketItem` if the price of the item is negative. This class of exception is caught in the basket and handled.

Checked exceptions were also appropriately handed in the `FileLoader` class. `IOException` is caught, and results in a human-readable message being logged, followed by the full stack trace.

In two instances, we felt introducing custom exception types improved the user experience, by allowing us to handle specific exceptions in a non-fatal manner. They also make our code more self-documenting, and improve its readability.

- **FileImportRowException**: used to signify there is an issue with a specific row of the CSV import. Our code separately handles this exception, skipping over the relevant row and notifying the user via the console.
- **UnrecognisedItemCategoryException**: used to handle the case where an item category is unknown, and log a message to the console.