

F21AS Group 3 Stage Two Report

Katie Baker
H00327433

Ryan Farish
H00264812

Anthony Gabini
H00356910

Hamish MacKinnon
H00257227

Michael Pidgeon
H00360036

1st April 2021

1 System Functionality

Our Cafe Manager application meets and extends the Stage Two specification. Before development started, a GUI mock-up was produced, see [Figure 1](#). We wanted to keep access to the electronic point of sale (EPoS) GUI we developed in Stage One of the coursework. This was achieved by adding a button to the Cafe Manager GUI that would open the EPoS window when pressed.

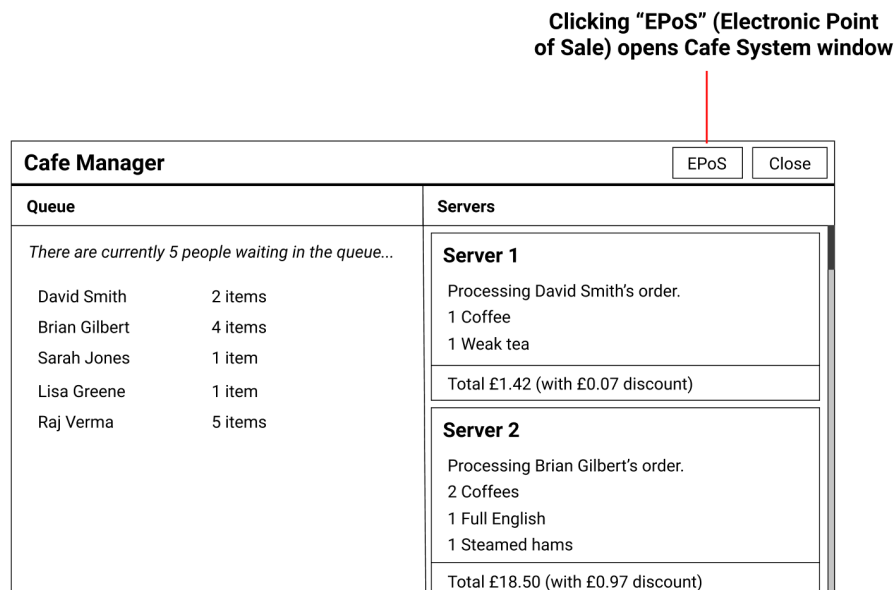


Figure 1: Design mock-up of new GUI, showing queue and server panels with button to open the electronic point of sale GUI from Stage One. We later added a button to open simulation controls that can be used for changing the speed of the simulation.

1.1 Core Functional Requirements

We successfully implemented all core functional requirements:

1. The order queue is displayed in the GUI. Multiple serving staff process orders - this represented in the GUI using a card for each server, see [Figure 2](#).
2. Customers and their orders are loaded in from .csv files in the same manner as for Stage One. Serving staff can only process a single order at a time, which takes a specified amount of time. Orders are taken from the front of the queue when a staff member is free.
3. The program exits and a report is generated when the queue is empty, see [Figure 3](#) and [Figure 4](#).
4. GUI shows customers, orders in the queue, and details of each staff member's current task, see [Figure 2](#). Order processing is slowed down.
5. A logger records timestamped events. A complete log is written to a file upon application exit, see [Figure 5](#) (this is a separate, more technical record than the report in Requirement 3).

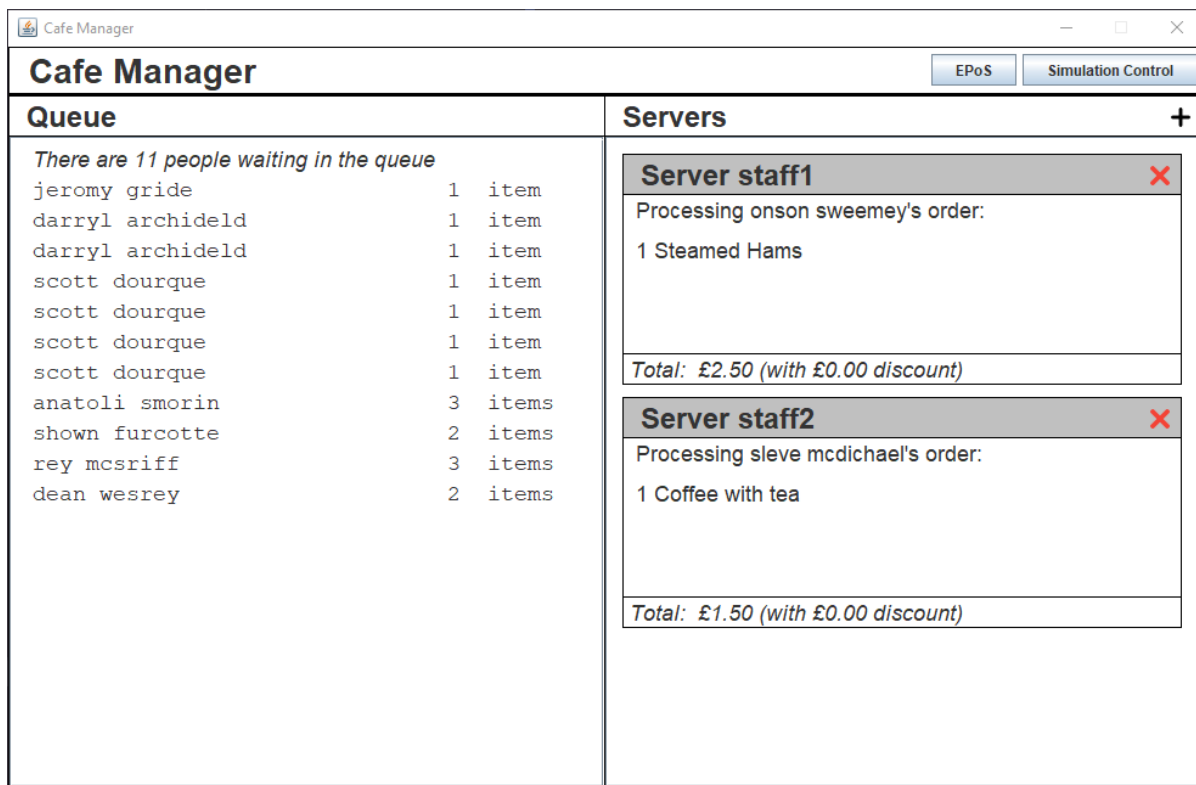


Figure 2: Implemented GUI, showing the order queue and information on which order each server is processing. Note the means of accessing extended functionality: the '+' icon to add additional servers, the 'x' icons to delete servers and the *EPoS* and *Simulation Control* buttons.

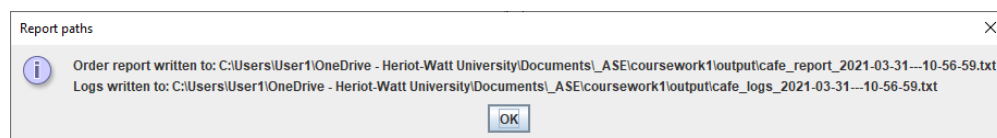
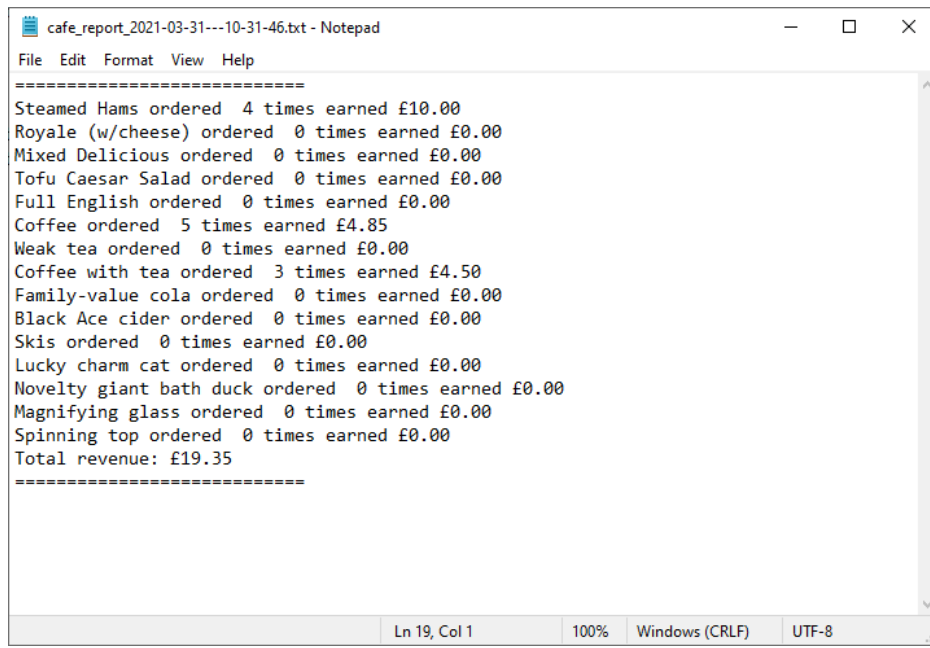
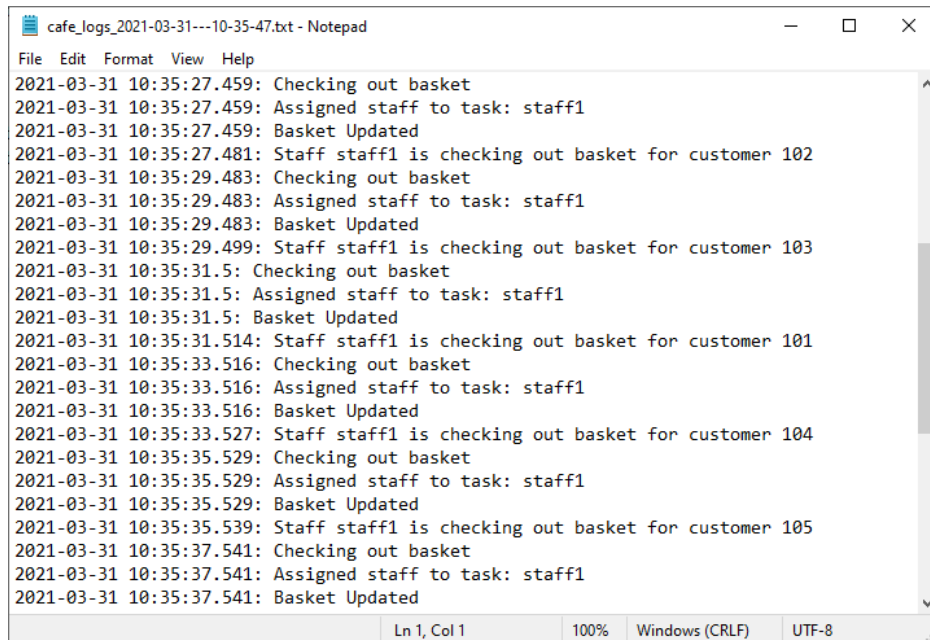


Figure 3: Alert showing paths for the order report and application log files - shown on exit.



```
cafe_report_2021-03-31---10-31-46.txt - Notepad
File Edit Format View Help
=====
Steamed Hams ordered  4 times earned £10.00
Royale (w/cheese) ordered  0 times earned £0.00
Mixed Delicious ordered  0 times earned £0.00
Tofu Caesar Salad ordered  0 times earned £0.00
Full English ordered  0 times earned £0.00
Coffee ordered  5 times earned £4.85
Weak tea ordered  0 times earned £0.00
Coffee with tea ordered  3 times earned £4.50
Family-value cola ordered  0 times earned £0.00
Black Ace cider ordered  0 times earned £0.00
Skis ordered  0 times earned £0.00
Lucky charm cat ordered  0 times earned £0.00
Novelty giant bath duck ordered  0 times earned £0.00
Magnifying glass ordered  0 times earned £0.00
Spinning top ordered  0 times earned £0.00
Total revenue: £19.35
=====
Ln 19, Col 1 100% Windows (CRLF) UTF-8
```

Figure 4: Example order report written on exit.



```
cafe_logs_2021-03-31---10-35-47.txt - Notepad
File Edit Format View Help
2021-03-31 10:35:27.459: Checking out basket
2021-03-31 10:35:27.459: Assigned staff to task: staff1
2021-03-31 10:35:27.459: Basket Updated
2021-03-31 10:35:27.481: Staff staff1 is checking out basket for customer 102
2021-03-31 10:35:29.483: Checking out basket
2021-03-31 10:35:29.483: Assigned staff to task: staff1
2021-03-31 10:35:29.483: Basket Updated
2021-03-31 10:35:29.499: Staff staff1 is checking out basket for customer 103
2021-03-31 10:35:31.5: Checking out basket
2021-03-31 10:35:31.5: Assigned staff to task: staff1
2021-03-31 10:35:31.5: Basket Updated
2021-03-31 10:35:31.514: Staff staff1 is checking out basket for customer 101
2021-03-31 10:35:33.516: Checking out basket
2021-03-31 10:35:33.516: Assigned staff to task: staff1
2021-03-31 10:35:33.516: Basket Updated
2021-03-31 10:35:33.527: Staff staff1 is checking out basket for customer 104
2021-03-31 10:35:35.529: Checking out basket
2021-03-31 10:35:35.529: Assigned staff to task: staff1
2021-03-31 10:35:35.529: Basket Updated
2021-03-31 10:35:35.539: Staff staff1 is checking out basket for customer 105
2021-03-31 10:35:37.541: Checking out basket
2021-03-31 10:35:37.541: Assigned staff to task: staff1
2021-03-31 10:35:37.541: Basket Updated
Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

Figure 5: Example log file written on exit.

1.2 Extended Functional Requirements

1. Added controls to alter speed of simulation (accessible via a GUI modal), see [Figure 6](#).
2. Added controls to add and remove staff during the simulation (accessible via buttons in the main GUI window, see [Figure 2](#)).
3. Kept the EPoS GUI from Stage One that can be used to create new orders in real-time. Any orders added are

appended to the back of the order queue, see [Figure 7](#).

Our repository is hosted on GitLab: <https://gitlab-student.macs.hw.ac.uk/mp2012/f21as-group-3>. This includes the complete application exported as a JAR file.



Figure 6: Simulation Control panel for adjusting speed of the simulation

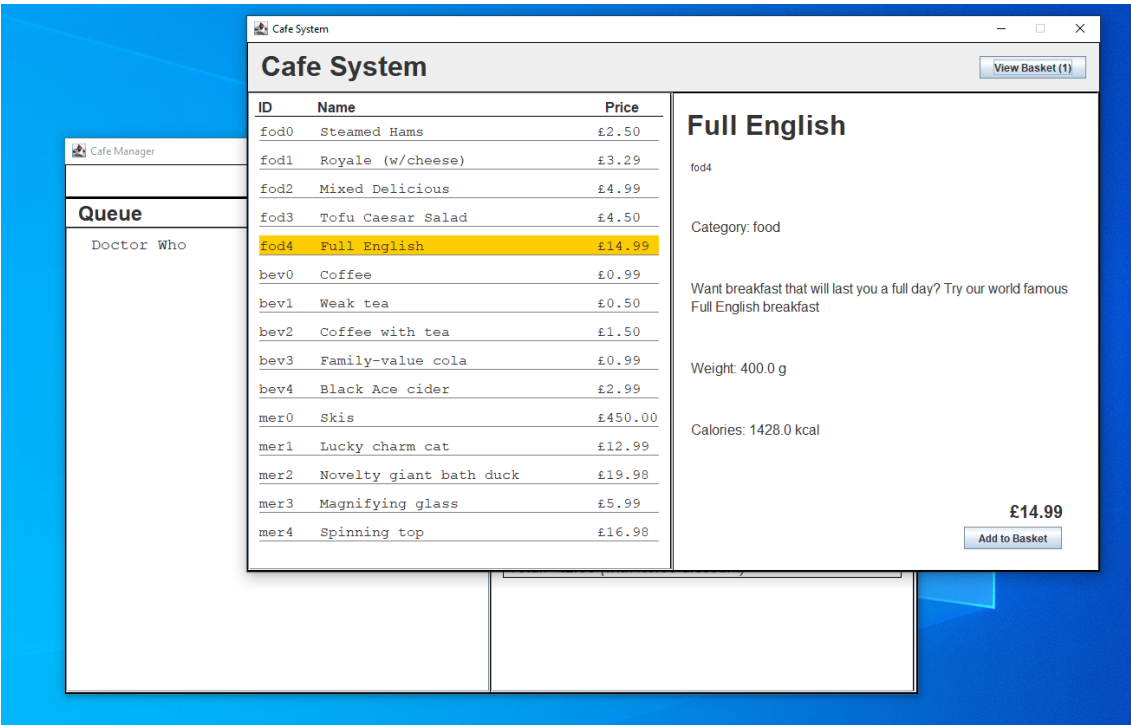


Figure 7: Example of using the EPoS application (from Stage One) to add additional orders to the back of the queue.

2 UML Class Diagrams

See [Figure 8](#), [Figure 9](#) and [Figure 10](#) for UML class diagrams. The full resolution diagrams are available in the repository: <https://gitlab-student.macs.hw.ac.uk/mp2012/f21as-group-3/-/tree/master/docs/Stage%20Two%20Class%20Diagrams>.

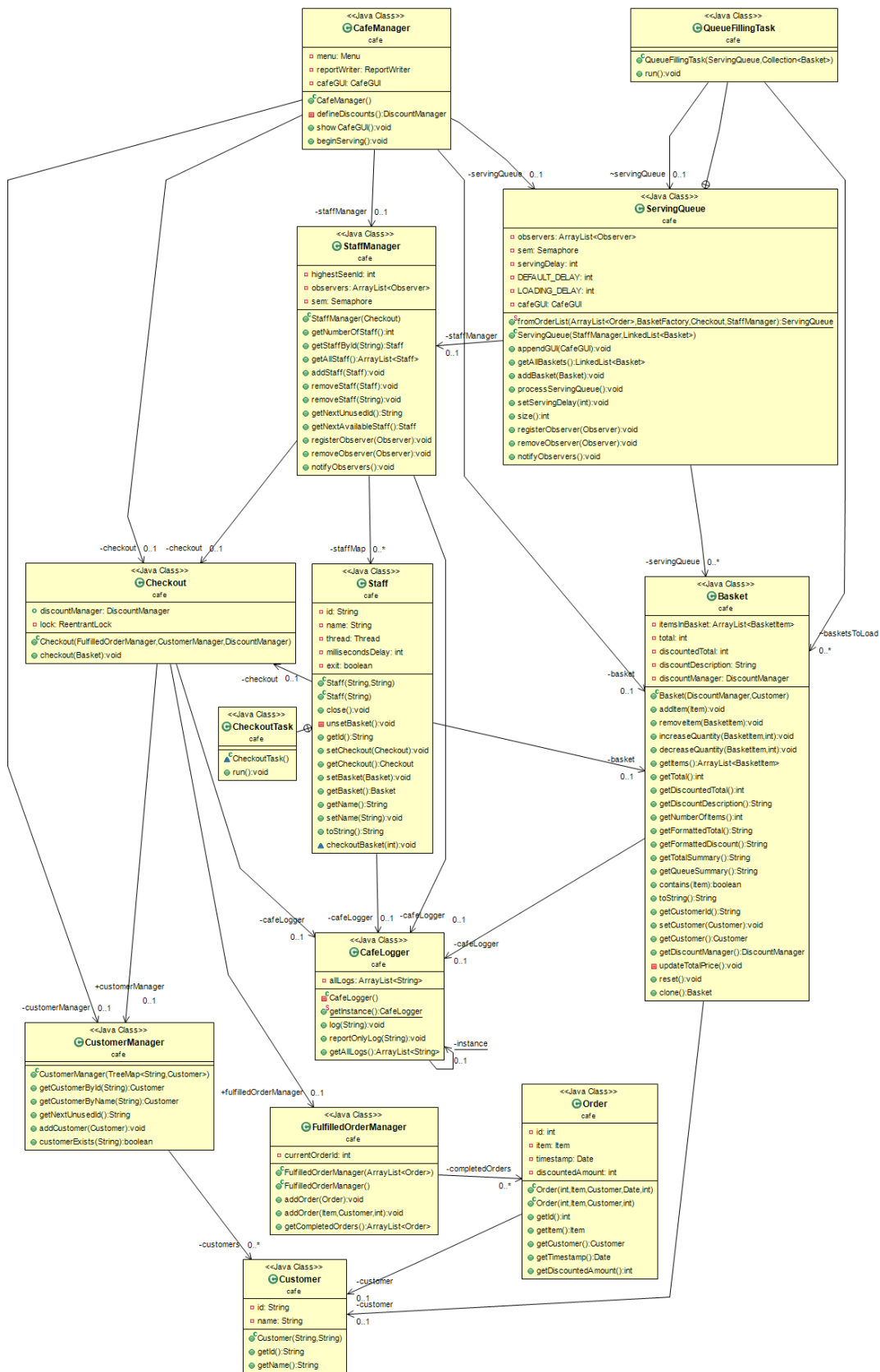


Figure 8: UML Class Diagram showing the new ServingQueue, CafeLogger, StaffManager, QueueFillingTask, CheckoutTask and FulfilledOrderManager classes

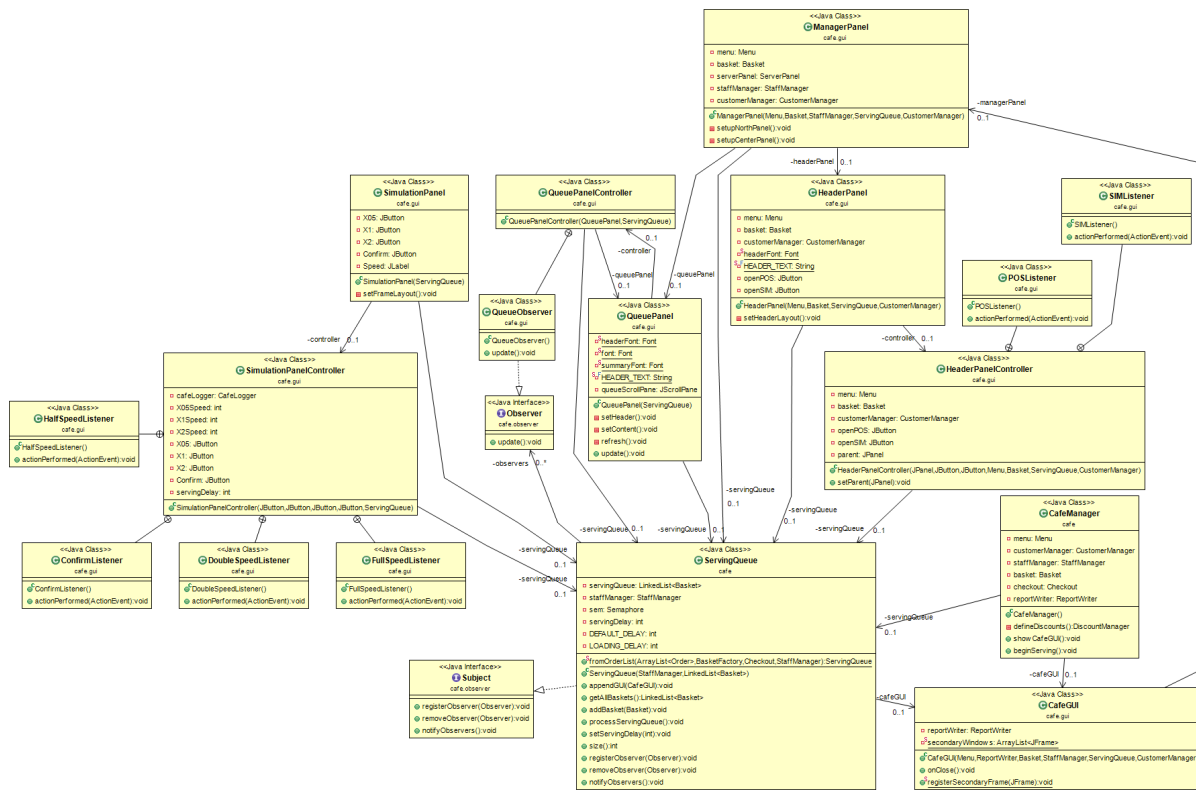


Figure 9: UML Class Diagram showing the new GUI classes responsible for the Queue panel, the Simulation Controls and the Header Panel (with the button to open the old EPoS view)

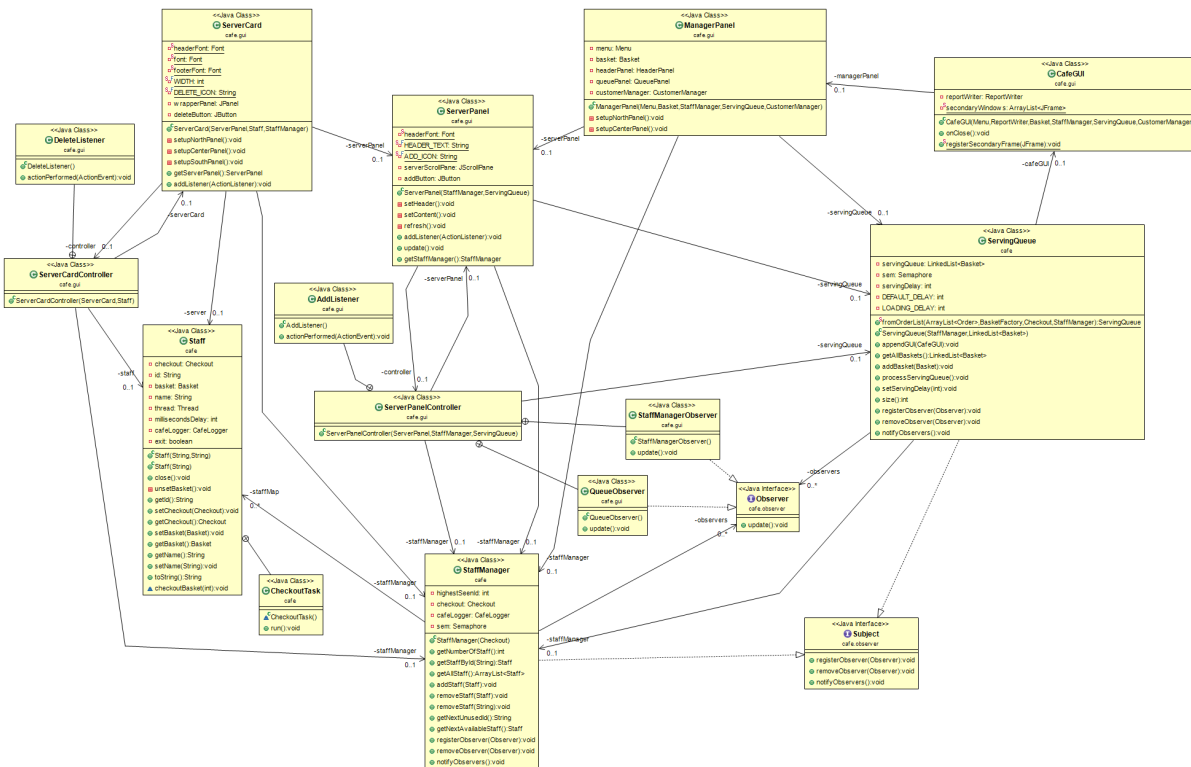


Figure 10: UML Class Diagram showing the new GUI classes responsible for the Server panel and Server cards

3 Agile Software Development

We used the following Agile processes:

- **Sprints:** We divided the work into four one-week sprints. We used the online application Trello to create and manage the sprint board (see [Figure 11](#)).
- **Weekly sprint planning meeting:** At the beginning of each sprint, we held a planning meeting where we would create new tasks or take existing tasks out of the backlog and assign them to group members.
- **Backlog:** We used a backlog to prioritise project deliverables. We prioritised the five core requirements as per the specification. We did not estimate each story individually, but rather attempted to shape the stories so they were all roughly the same size.
- **Scrum:** We used aspects of this methodology where appropriate. One technique was nominating a rotating 'product owner' for each sprint planning meeting, who made the final decision on product functionality and prioritisation. We also started each planning meeting with a short review and retrospective of the previous sprint, and demos of new functionality.
- **User stories:** We framed our work from the point of view of customer requests, using the "As a... I want... So that..." syntax. Cards were created for the sprint board which contained the user story, and more detailed acceptance criteria as a checklist of tasks (see [Figure 12](#)).
- **Refactoring:** Regular incremental changes were made to improve code quality and the internal structure of the program. For an example, see: https://gitlab-student.macs.hw.ac.uk/mp2012/f21as-group-3/-/merge_requests/55. This change improved the robustness of our concurrency model, without altering any application features. The test-driven development approach we adopted resulted in an extensive test suite, which gave us confidence no breaking changes were introduced by refactors.
- **Sprint burnup chart:** We reviewed this chart at every sprint meeting to assess how we were progressing against the backlog of user stories (see [Figure 13](#)). A burnup chart offers one key advantage over a traditional burndown chart - it allows variable scope to be represented, which is helpful when working in an agile manner, with new stories being added (or removed) at each sprint planning meeting.
- **Pair programming:** For more challenging stories such as the threading implementation, we found screen sharing tools helpful in facilitating pair programming. We regularly switched 'driver' and 'navigator' roles, and found this way of working helpful both in terms of completing tasks, and also sharing knowledge among the team.

Although there were no 'releases' as such for this project, we successfully produced working software at the end of each sprint. If this was a real client engagement, we would have been able to release this software (at least to a UAT environment) on a regular basis.

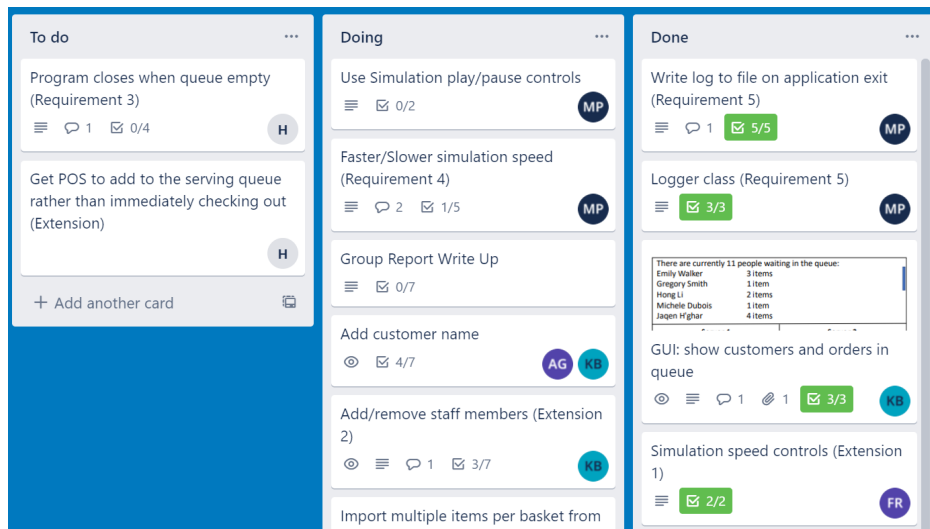


Figure 11: Sprint planning and tracking with Trello.

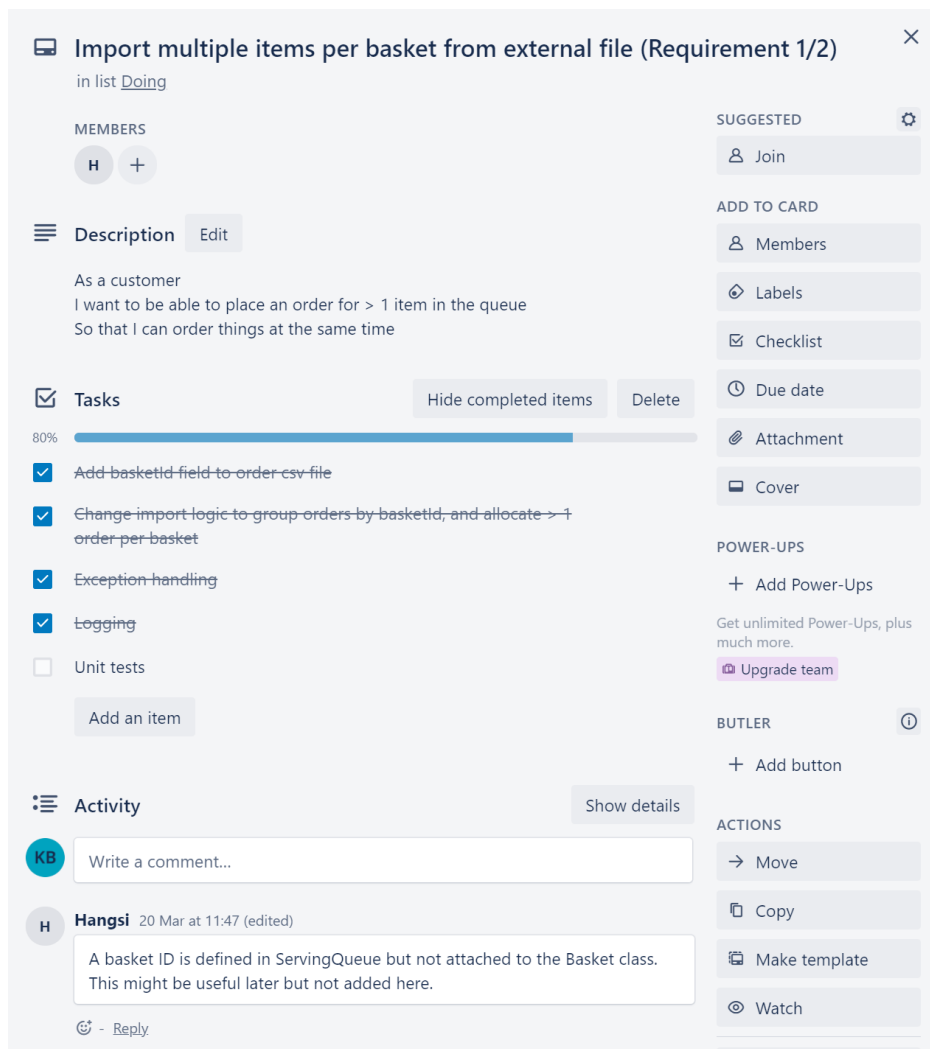


Figure 12: Example sprint story with a customer requirement, task list and assigned developer.

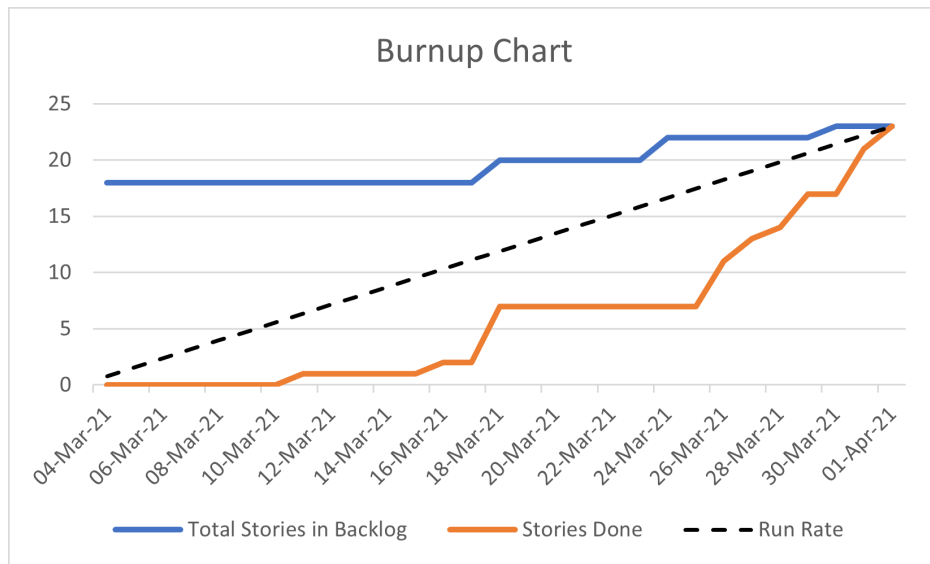


Figure 13: Final burnup chart as at 1 April 2021. The blue line shows the number of stories increasing as we discovered new information about the task.

4 Threading Implementation

Threading was used in the application in the two required locations: first to progressively load the customers with items into a queue, and second to simulate the process of customer orders being processed and "checked out" by staff members.

Java has a number of similar but subtly different options for implementing thread safety. We ensured thread-safety via two approaches:

1. Semaphores
2. Extrinsic Monitors

4.1 Progressive loading of Baskets into a ServingQueue

The application distinguishes between an order, required by the part 1 specification to encapsulate a single item, and a basket, which contains potentially multiple items and has an associated customer.

A list of orders and known customers is defined in csv files, and at application startup, these are loaded in and combined into baskets appropriately. This loaded collection of baskets is incrementally added to the serving queue that is displayed in the GUI and that Staff instances take baskets from by using a thread that is started at the end of this loading-at-startup method.

Semaphore The methods that read and write from the serving queue are made thread safe by the use of a semaphore with a single permit that will deactivate threads attempting to access it while another thread already has the permit.

4.2 Checking out Baskets using Staff

To explain this, we differentiate the two ways in which we express customers and staff.

The application by default starts with two staff members present, and the user can add or remove servers at their discretion by clicking the add button on the staff panel. These additional staff members will process orders automatically. These will process the orders until none remain.

Each `Staff` instance starts a thread. This thread runs a basket processing when the serving queue chooses the staff to process the `Basket` at the front of the queue. This marks the staff as busy and unable to accept new baskets until it has finished.

Semaphore Semaphore permits are acquired and released in the runnable class `CheckoutTask`, nested within the `Staff` class. `CheckoutTask` implements a `run` method which instructs the thread what to do, this being to “checkout” the basket it is working on using the member `Checkout` instance, and then to unset the basket. Before this, the thread acquires a permit from a semaphore which holds only one permit at most. This semaphore controls how many threads can run at any given time, and the use of a single permit helpfully causes the staff instance to be fully occupied by processing the single basket and unable to acquire more than that.

Extrinsic Lock An extrinsic lock is introduced in the `Checkout` class, found within the only method defined in the class. The method performs the checkout functionality which the `Staff` use on their threads. This method is locked using a `Reentrant` lock in the beginning, and is only unlocked once the basket reset is executed by the thread. Extrinsic allows the cafe application to have more explicit control over the mechanism of allowing threads to access the method, reducing the chances of deadlocks and thread starvation that would otherwise cause the application to experience an uninterruptible pause in functionality once the process of checking-out customers has begun. Given the try-wait structure of the checkout method which is extrinsically locked, another thread attempting to use the method will be simply put on hold until the previous thread has unlocked the method. This lock means that the fulfilled order processing call stack beneath this checkout stage does not need to be modified to become thread safe, as it is all run during this lock.

5 Design Patterns

5.1 Singleton

The `CafeLogger` class is implemented as a singleton, as required by the specification. The single instance is constructed or returned as appropriate by calls to the static `CafeLogger.getInstance()` method in classes that require logs to be written.

5.2 MVC

The Model-View-Controller (MVC) pattern was used for separating the responsibilities of the GUI into separate classes. The GUI components which use the MVC pattern are given in [Table 1](#). In each case, a view class is associated with one controller class (which handles button presses and updates to the view when data is updated) and at least one model class (which contains the data required for the view or button functionality).

Table 1: GUI components using the MVC pattern

GUI element	Function	Model classes	View class	Controller class
Header panel	Open EPoS and simulation control modals	Menu, Basket, ServingQueue, CustomerManager	HeaderPanel	HeaderPanelController
Queue panel	Render the customer queue, update when servers process orders and new orders are added to the queue	ServingQueue	QueuePanel	QueuePanelController
Server panel cards	Render a card for each server with details of the order being processed by the server	StaffManager, Staff	ServerCard	ServerCardController
Server panel	Render the server cards and add new servers	ServingQueue, StaffManager	ServerPanel	ServerPanelController
Simulation modal	Change the simulation speed	ServingQueue	SimulationPanel	SimulationPanelController

The MVC model was used to handle button clicks. The Swing JButton objects are contained in the view and the action listeners are contained in the controller. When the buttons are pressed the action listener in the controller is triggered. The functionality of adding and deleting a serving staff is shown in the UML sequence diagrams in [Figure 14](#). This functionality is achieved using the MVC and Observer patterns.

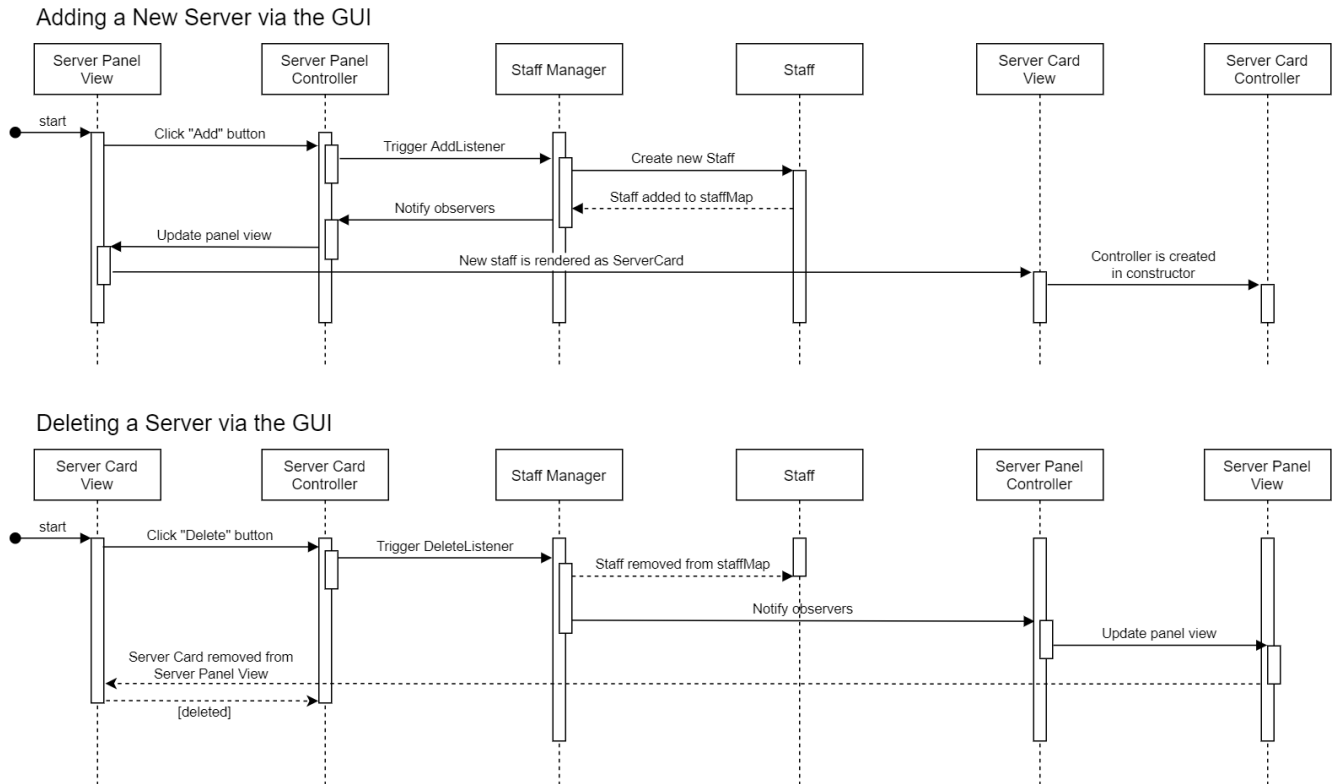


Figure 14: UML sequence diagrams showing adding and deleting serving staff via the GUI.

Adding a server When the user clicks the "Add" button in the ServerPanel view, the AddListener class is triggered in the ServerPanelController object (see Figure 15). The AddListener creates a new Staff object which is passed to an instance of the model class, StaffManager. StaffManager implements the Subject interface and when a new Staff is added to the staffMap, notifyObservers() is called. This in turn triggers the ServerPanelController (which implements the Observer interface) to call a refresh of the ServerPanel view. As the ServerCard objects are added to the window in a loop of the staffMap values, the new staff is rendered as a ServerCard.

```

1 public class AddListener implements ActionListener {
2     @Override
3     public void actionPerformed(ActionEvent e) {
4         String id = staffManager.getNextUnusedId(); // get a suitable staff id from the staffManager
5         Staff staff = new Staff(id); // create a new Staff object
6         staffManager.addStaff(staff); // pass the new Staff to staffManager, which
7     } // calls notifyObservers() to trigger GUI update
8 }
9

```

Figure 15: AddListener class

Deleting a server When the user clicks the "Delete" button in the ServerCard view, the DeleteListener class is triggered in the ServerCardController object (see Figure 16). The DeleteListener calls the removeStaff(staff) method of the StaffManager model class, passing the Staff object to be deleted. When removeStaff(Staff staff) is called, the staff thread is gracefully stopped and the object is removed from the staffMap (see Figure 17). Finally, notifyObservers() is called, which triggers the ServerPanelController and updates the view as described in the

previous paragraph.

```
1 public class DeleteListener implements ActionListener {
2     @Override
3     public void actionPerformed(ActionEvent e) {
4         staffManager.removeStaff(staff); // call the removeStaff method in the staffManager object
5     }
6 }
7
```

Figure 16: DeleteListener class

```
1 public void removeStaff(Staff staff) {
2     cafeLogger.log("Removed staff: " + staff.getId()); // log the event
3     staff.close(); // gracefully stop the staff thread
4     staffMap.remove(staff.getId()); // remove the staff from the staffMap
5     notifyObservers(); // notify observers to trigger GUI update
6 }
7
```

Figure 17: removeStaff method in the StaffManager class

5.3 Observer

Our aim was for the view layer of our MVC pattern to dynamically update based on model changes, without having to poll the model or own a reference. We used the Observer pattern to facilitate this. We defined two interfaces, following the conventional pattern as shown in [Figure 18](#).

```
1 public interface Observer {
2     public void update();
3 }
4
5 public interface Subject {
6     public void registerObserver(Observer obs);
7     public void removeObserver(Observer obs);
8     public void notifyObservers();
9 }
10
```

Figure 18: Observer and Subject interfaces

Two GUI classes required dynamic updates: `QueuePanel` (shows the state of the order queue) and `ServerPanel` (shows the state of each server). Therefore, both classes' controllers implement the `Observer` interface. The data for the two classes come from `ServingQueue` and `StaffManager` classes - therefore these two classes implement the `Subject` interface. When an update is required, these classes call `update()` as defined by the `Subject` interface; this in turn calls a separate `update()` method in the controller, which updates the GUI class itself. This pattern is summarised in [Figure 19](#).

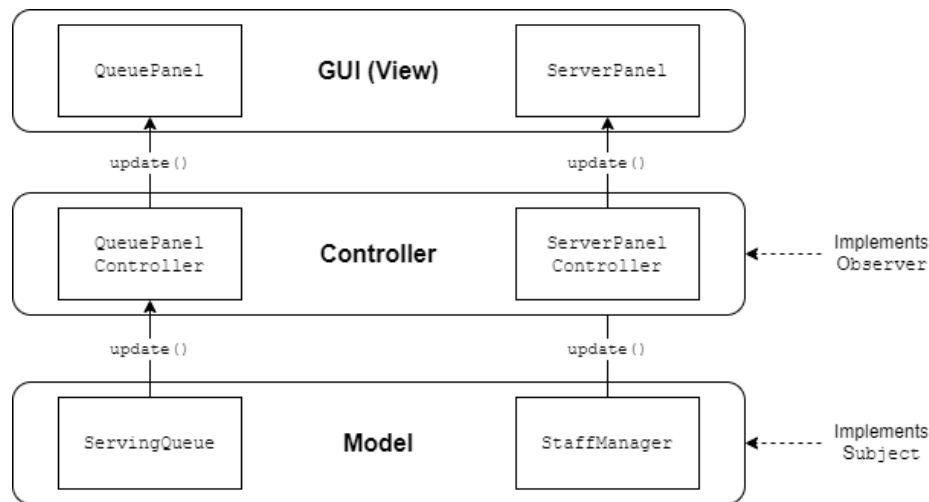


Figure 19: Summary of how the observer pattern was used to dynamically update the GUI.

This pattern achieves our aim of swiftly and efficiently updating data in the GUI, improving the user experience. Use of a standard, idiomatic pattern also makes the codebase easier to reason with and understand.

5.4 Factory

One challenge we experienced moving from Stage 1 to Stage 2 was handling multiple baskets as the customers load from file. Previously, we had assumed only a single basket existed that was built up from selections from the menu.

We created a `BasketFactory` class to easily instantiate these multiple baskets. The alternative approach would be to call the `Basket` constructor directly each time, however this factory-based approach offered two advantages:

- The `Basket` constructor needs to be passed information on discounts. `BasketFactory` only needs to be told about discounts once, then it can use this information to create baskets (as opposed to multiple other classes needing knowledge of discounts).
- In Stage 2, we require 'filled' baskets – pre-populated with one or more items. The factory class exposes a `fromOrder` method that neatly encapsulates converting from an order to a basket.

6 Comparison Between Stage One and Stage Two

The software development methodology was different between the stages with Stage One using planned weekly iterations and Stage Two using agile software development with sprints lasting one week.

We found both methods offered different strengths as summarised below:

Advantages of Planned Iteration (Stage 1)

- Up-front design work ensured all team members had a shared understanding of the basic architecture, with UML diagrams to refer to when programming.
- Work being assigned to iterations in advance gave us a clear plan to monitor progress against.

Advantages of Agile (Stage 2)

- More flexibility to adapt to changing requirements (we were able to incorporate ideas for product extensions such as keeping the old EPoS system to add orders to the queue).
- More flexibility to alter the scope. We added some stories and removed others, using our burn-up chart as a reference to ensure our target was realistic.
- Agile techniques proved highly useful - particularly the sprint review and planning meeting, and pair programming for more challenging areas such as threading.
- User stories were an intuitive and user-focused way of documenting requirements, with a summary followed by more detailed acceptance criteria.

Overall, we found the agile method more flexible - it allowed us to react to changing requirements quickly and accurately. The planned iteration method did not work as well in reality due to new information arising during the course of development - although the initial plan was helpful at first, it was quickly superseded by tacit knowledge among team members.

A hybrid approach may be worth investigating, where limited 'time-boxed' up-front design work gives a shared initial understanding, with an acceptance the design is likely to change as the team completes sprints.