Advanced Software Engineering

Group Report 2

[Calum Henning, Duncan Austin, Katarina Alexander, Kate-Lynne Thomson & Stuart Paterson]

Link to GitHub: https://github.com/chenning17/Advanced_Software_Engineering

Summary of work completed

The work was split evenly according to ability and previous experience. Each person in the group had their own parts to work on, as seen below:

Calum: main gui and helper methods, start up gui functionality and simulation settings, threading bugs, pair programming (discounts and log file)

Duncan: implementing observer pattern, implementing producer-consumer for threads, online priority queue.

Katarina: Adding process time functionality, start up GUI, Sales Assistant class, worked on discounts, pair programming

Kate-Lynne: Cafe Simulation class, start up GUI, write to report class, singleton LogFile class, adding process time to order.csv, pair programming

Stuart: Refactor order, random generated customers, pair programming for producer-consumer, simulated time creation and implementation, work on report generation,

Both the written code and this Group Report were created by everybody collaboratively and equally.

The Programme Specification

This program meets the specification fully.

General functionality:

The program created simulates a cafe, with separate queues for customers and online orders, that are processed by multiple sales assistants simultaneously. Each order contains one or more items that each have an associated time it will take a sales assistant to prepare. Therefore each order will take a varying amount of time to prepare depending on the items it contains. When an order is processed the best discount available is applied to the order.

Information about the currently running simulation is displayed in various sections of the main GUI, such as the current actions of each sales assistant and the state of both the standard and online queues. During the running of the simulation, key events are written to a log file that can be viewed to see when individual things happened.

When the simulation ends, the program outputs a summary of the orders taken to a report file.

Extra Features:

On startup of the program, the user is presented with a GUI prompting them to input various choices for the settings of the simulation. Users can set the simulation speed, number of assistants, input menu file and input orders file.

Orders stored in the CSV file have a date and time associated with them. When the program is started the simulation has its own time to replicate the cafe running for a day over the course of a few minutes. When the simulated time reaches the time of an order from the CSV file that order is added to the queue to be processed. Each type of item has a different time factor associated with it in the menu CSV, that represents how quickly a sales assistant can prepare it.

Online orders are randomly generated while the cafe is open and are added to a separate online priority queue. These orders are prepared in advance then wait for the customer to arrive. When the customer arrives they can immediately collect their order. If customers arrive before their order is ready they will wait for it to be prepared. The GUI displays how many online orders are awaiting processing, how many are ready to collect and whether any customers are awaiting their order. Online orders will contain between 1 and 4 items randomly selected from the menu. If this order qualifies for a discount this will be applied to the total price.

Bugs:

At the end of the program, the clock will stop running at the time the last customer arrives, however servers can still be processing an order for a customer. While not a bug with the code itself, we have no had enough time to fine tune how long it takes to prepare an order when compared to the current time in the simulation.

Class Diagrams:

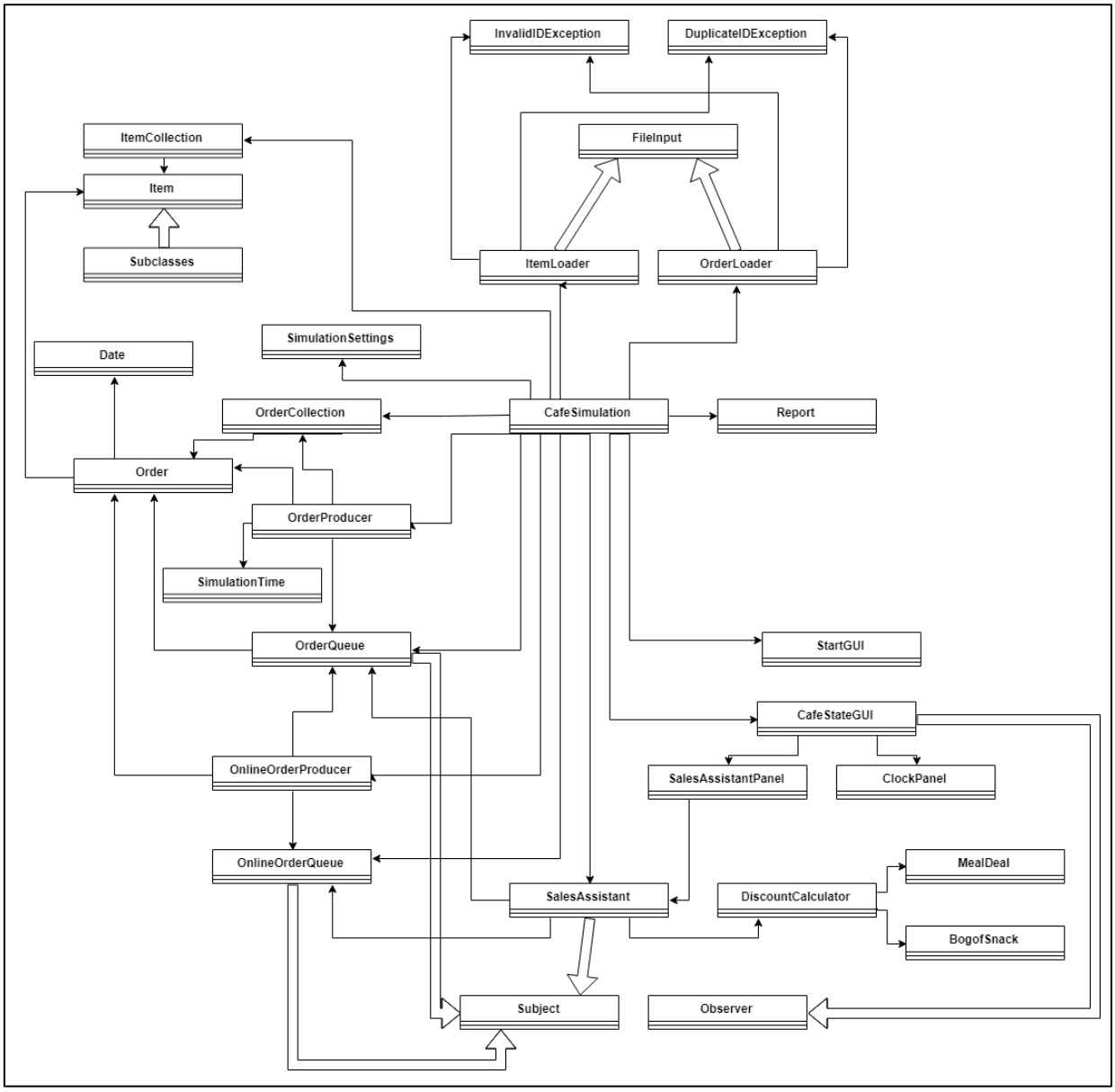


Figure 1: Class Association Diagram

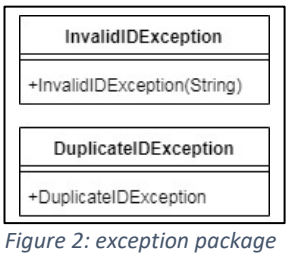


Figure 2: exception package

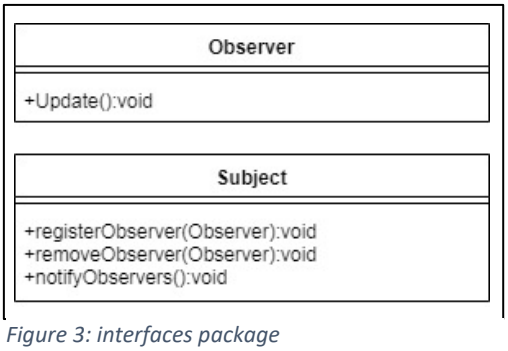


Figure 3: interfaces package

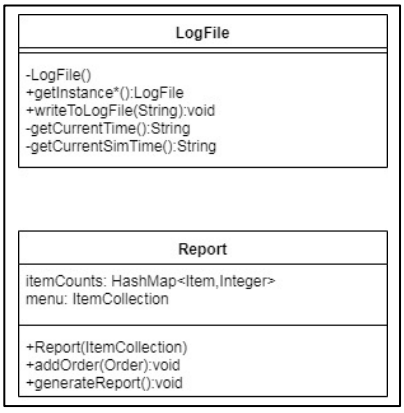


Figure 4: fileWriting package

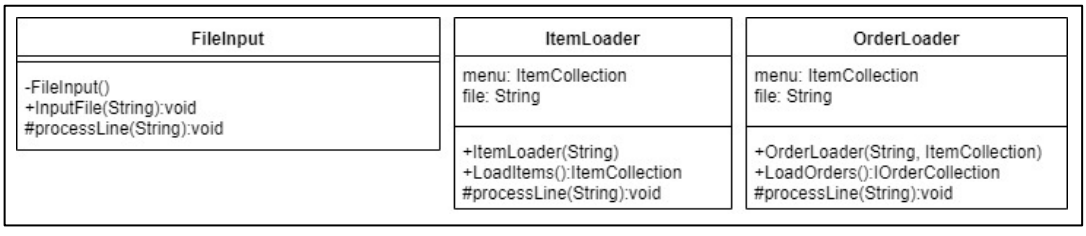


Figure 5: fileReading package

Figure 6: model package

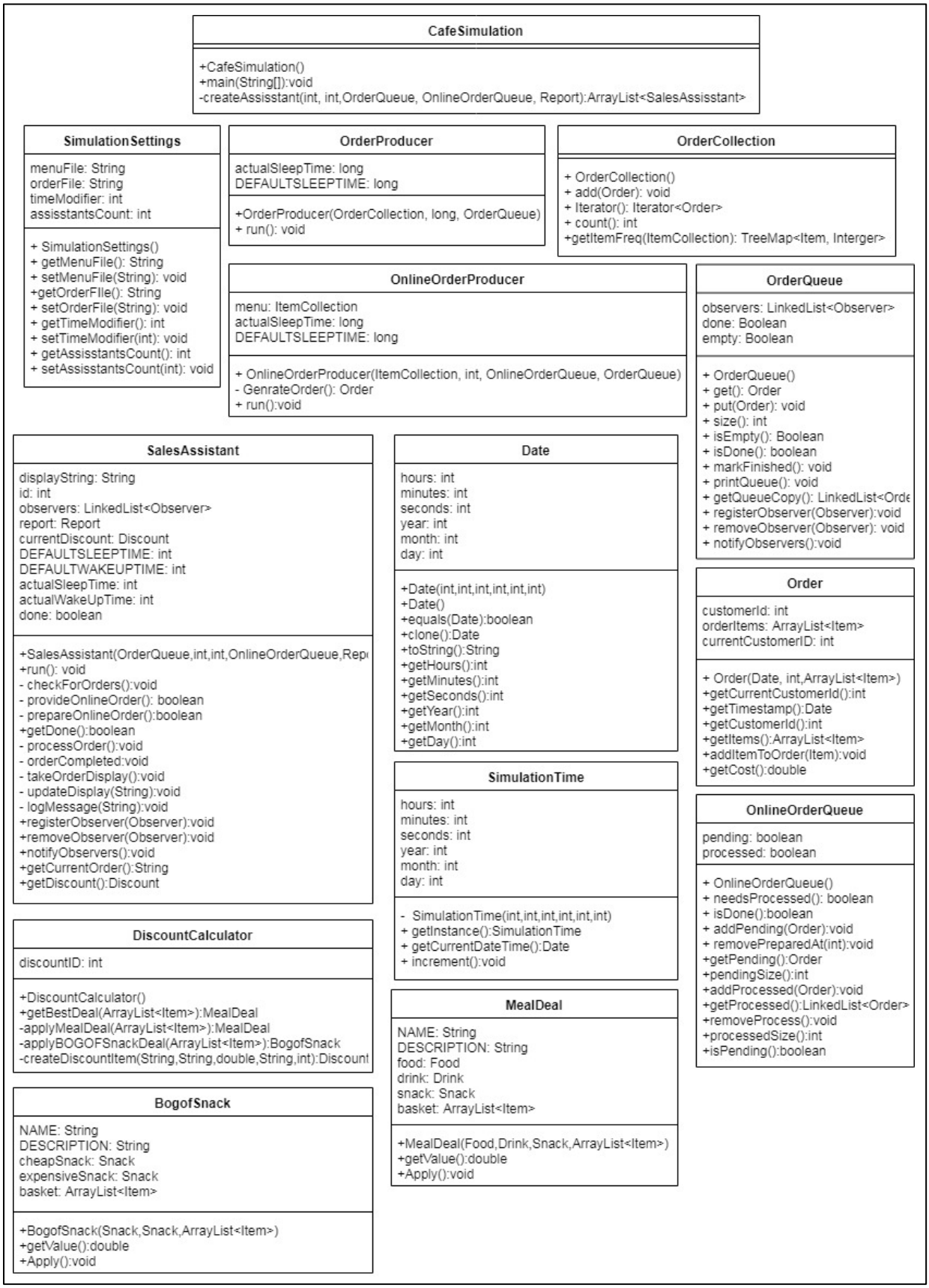


Figure 7: view package

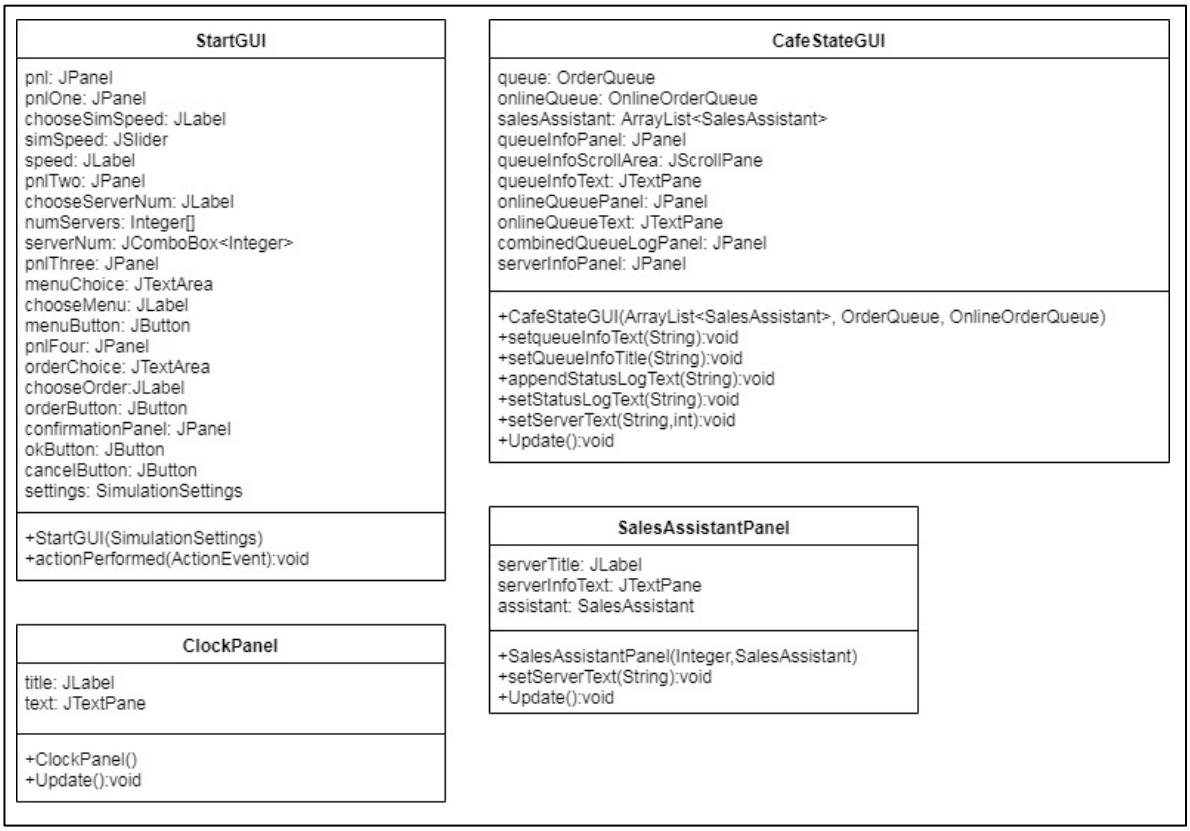
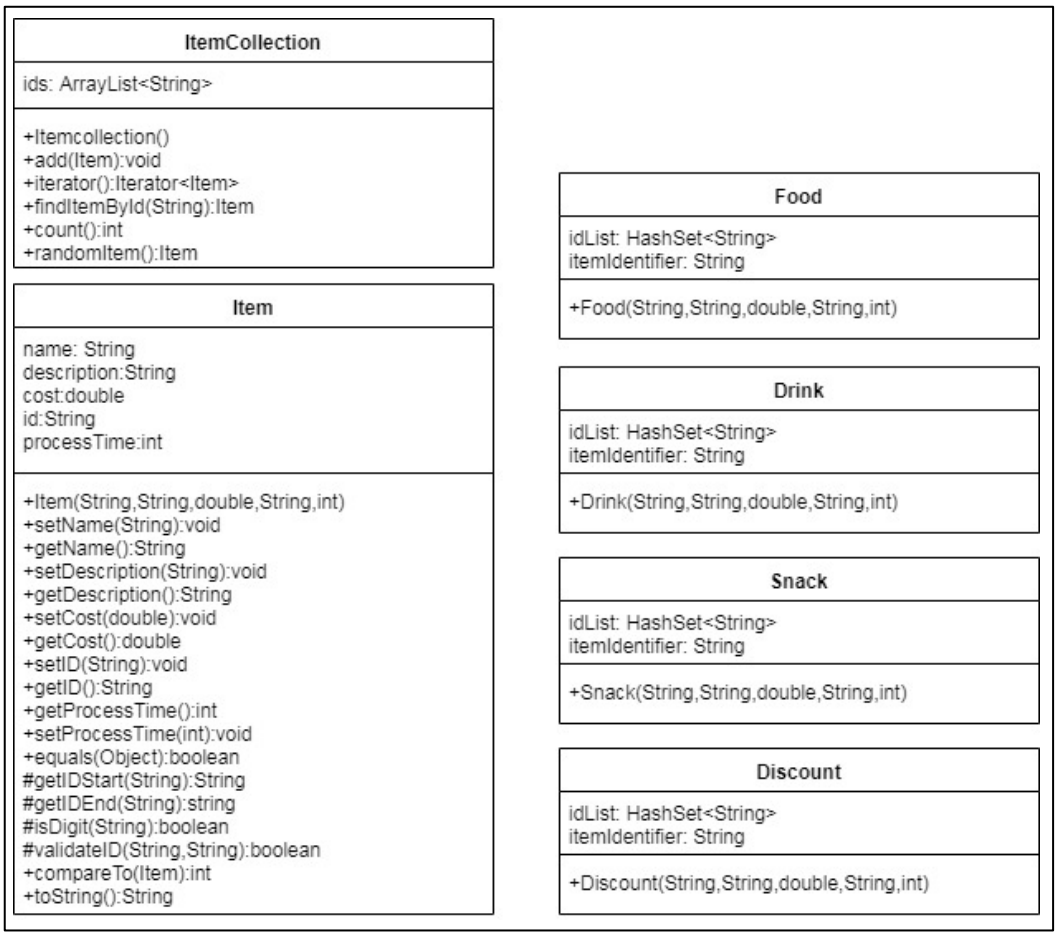


Figure 8: model.items package



Agile Development:

To allow us as a group to keep track of our project we chose to undertake weekly meetings (Tuesday mornings), with some weeks meetings being held more frequently. The reason we chose to do this weekly was due to conflicting class schedules and other commitments by each member, therefore it wasn't possible to meet everyday, as the agile process is normally implemented. In order to keep in communication with each other we used a Slack channel. To allow us to keep track of our user stories and objectives for each sprint we used Google sheets, which allowed us as a group to update individually which tasks we had done per sprint.

Sprint Planning

The points below show how the user stories were divided into sprints

Sprint 1 (26/2-5/3)

- Refactoring of Order class to include multiple items. Update other classes and CSV to support this change.
- Creation of class to write to log file.
- Add a delay between adding orders to the queue to slow processing to real time.
- Create sales assistant class to process orders.
- New GUI to show order queue and sales assistants.

Sprint 2 (5/3-12/3)

- Design of startup GUI
- Increase stability of threaded portions of code.
- Functionality to randomly generate customers
- Different items take different times to process.
- Change order production so the don’t appear uniformly.

Sprint 3 (12/3-19/3)

- Additional complexity to sales assistant. Display different stages of order processing.
- Report generation on program close.
- Create an online priority queue.
- Addition of simulated date and time.
- Addition of startup gui functionality.

Sprint 4 (19/3-21/3)

- Tidying code. Updating comments, variable names and package structure.
- Creating UML diagrams.
- Finalising submission report.

Refactoring:

The order class from submission 1 used a single item for each order. This was refactored so that orders can store a list of items that were ordered simultaneously. The date stored in an order was also refactored. Rather than using real time, a simulation time was created and times in this format were stored in the orders. This allowed the cafe to simulate a full day in only a few minutes.

A processing time was added to the Item class from submission 1. This time it takes for a sales assistant to process an order in the new GUI is based on the total of the processing time for each item.

Retrospectives and Sprint planning meetings were carried out every Tuesday during the first hour of lab time to discuss previous sprints, and plan what was going to be carried out for the next sprint. Scrum masters were rotated for each meeting. As there was no customer or management to deal with, their main role was to keep everyone focused and on task, and ensuring that everyone was able to get their ideas across

Stand up meetings were used, however we were not able to carry these out every day due to differing schedules and commitments. They tended to reliably occur on tuesdays and thursdays around the lectures for the course. These would sometimes occur on other days if everyone was around, but these were impromptu and not scheduled in advance. Meetings usually involved quickly going around the group to check how everyone was getting on with their user story for the sprint, and if they needed any help working it out. If people did need help, typically doing using pair programming at a time convenient for both members of the pair.

Time-boxing was used to ensure tasks had a fixed time periods. Tuesdays acted as deadlines for sprints. Retrospectives and sprint planning meetings were capped at half an half an hour. Stand up meetings were limited to 15 minutes.

When pair programming one person would write the code while offered their insight so that decisions could be made collaboratively. Every 20 minutes or so the pair would swap roles so that the other person was coding.

Thread Usage In Program:

Threads are used for 2 main purposes, producing customer orders and consuming them. The code for this is based on the producer-consumer pattern provided in the lectures, however some changes have been made (detailed below). There are 2 producers, 2 shared objects, and 1 consumer.

Producers

OrderProducer iterates over all of the orders provided in the CSV when the program starts. Increments the simulation time until it reaches the time the next order is to be added to the simulation. Once it reaches that time, the Order is added to the Order queue (one of the shared objects mentioned below).

OnlineOrderProducer randomly generates an order of between 1 and 4 Items, and assigns it a new customer id number. The generated orders are added to the OnlineOrderQueue (the second shared object) in 2 stages: firstly added to a “Pending Orders” section, to represent

the online order being received by the cafe, and then after a delay the order will be added to a Queue, to represent the customer actually arriving at the cafe to collect their order. It will create and add these random orders until either the regular orders have stopped, or the simulation time reaches 1500 hours.

Shared Objects

OrderQueue uses a LinkedList to store the Queue, as LinkedList implements the add() and remove() methods defined in the Queue interface. A modification was carried out to the method used by the consumer to retrieve the the first item from the queue. In the method provided in the lectures, threads wait inside the get() method until they are notified:

```
while(this.empty) {
    try {
        wait();
    }
    catch (InterruptedException e) {
        //do nothing
    }
}
notifyAll();
```

However, as there are 2 shared objects being checked by the consumers, having them wait on one queue isn't ideal, as there could be orders building up in the other queue that might need processing. To deal with this, the while loop was replaced with an if statement that throws a "NoSuchElementException":

```
if(this.empty) {
    throw new NoSuchElementException("No orders to get");
}
```

This is caught in the checkForOrders() method in the SalesAssistant (described below).

OnlineOrderQueue extends OrderQueue by introducing two new ArrayList member variables: pendingOrders (mentioned above) represents orders which have been placed but not yet prepared by staff, and processedOrders, which represents orders that have been prepared and are awaiting collection. Both of these ArrayLists have an associated boolean flag to indicate whether the SalesAssistants should take any action.

Consumer

The SalesAssistant class acts as a consumer for both queues. These threads run while both queues have items in them, and there are still orders to be added from the CSV. This allows the simulation to keep running even if the SalesAssistant threads have managed to process all orders and empty both queues.

The main run() method alternates between checking for orders to process, and waiting for a small amount of time between checking. Priority is given to online orders. If an online order is available, SalesAssistants will move the order from pendingOrders to ProcessedOrders in the OnlineOrderQueue, and take time to sleep to simulate "preparing" the order. The prepared order is then left until the customer actually arrives in the queue, at which point the order is added to the report and the order is removed from the processedOrders ArrayList.

Whenever an order is processed (either prepared in advance for an online order or processed for a normal order), threads sleep for varying times depending on the Items they are preparing. Each Item has been given a time it takes to prepare in the Menu CSV file.

Whenever a thread sleeps, the number is multiplied by a modifier that is provided when the program starts, to allow for running the simulation at different speeds.

Each SalesAssistant is observed by a separate panel of the GUI, so that only the area relevant to the thread is updated when the notifyObserver method is called. The Queues are observed by the same section of the GUI.

An overview of the different kinds threads is shown below:

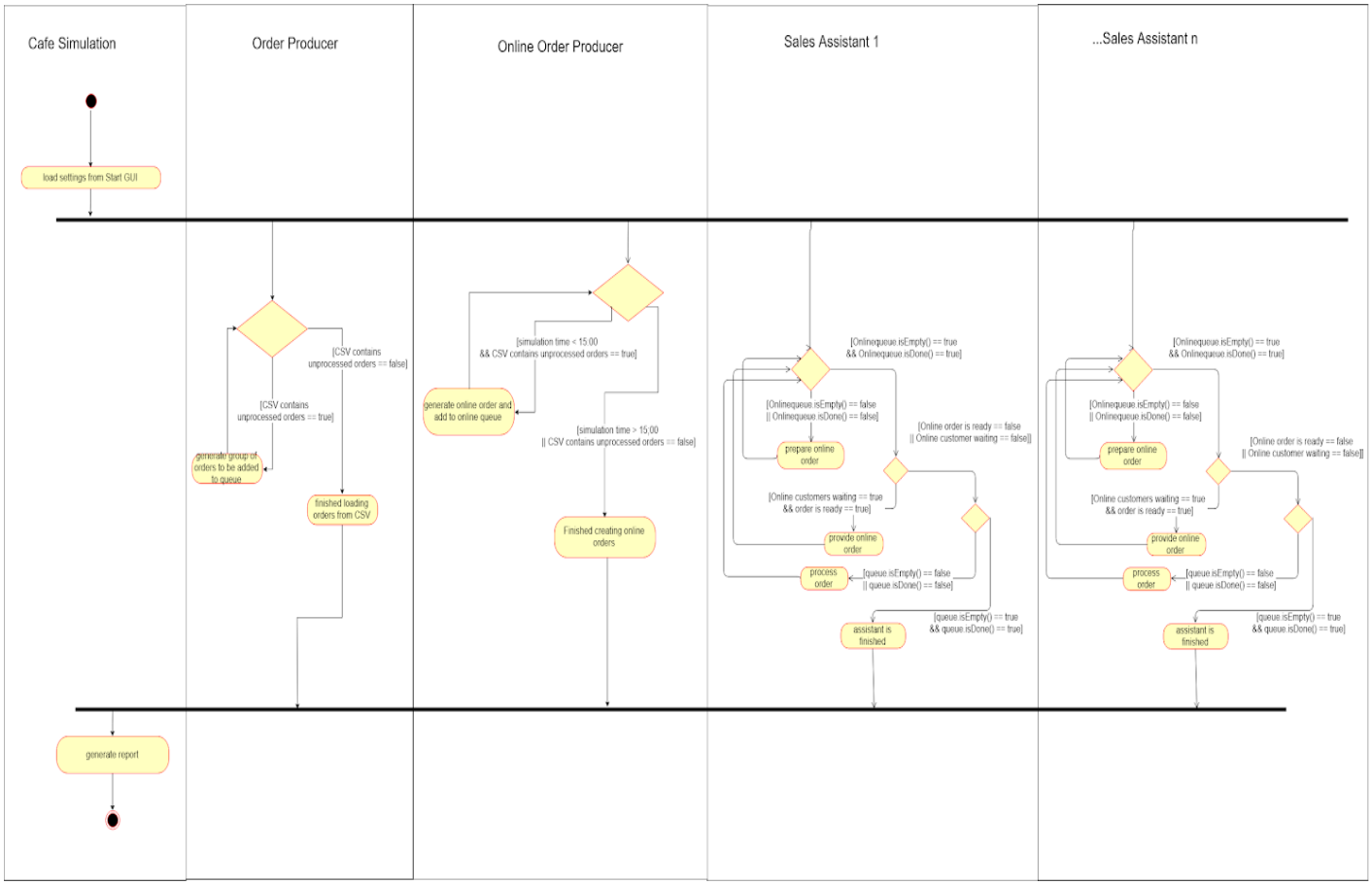


Figure 9: Threads Activity Diagram

Design Patterns:

Two Singleton patterns are used. The first controls writing to a log file. Log files are named with the date they are created. All additions to the log file are done using the writeToLogFile method, which appends the the supplied string to the current simulation time writes it as a line in the log file.

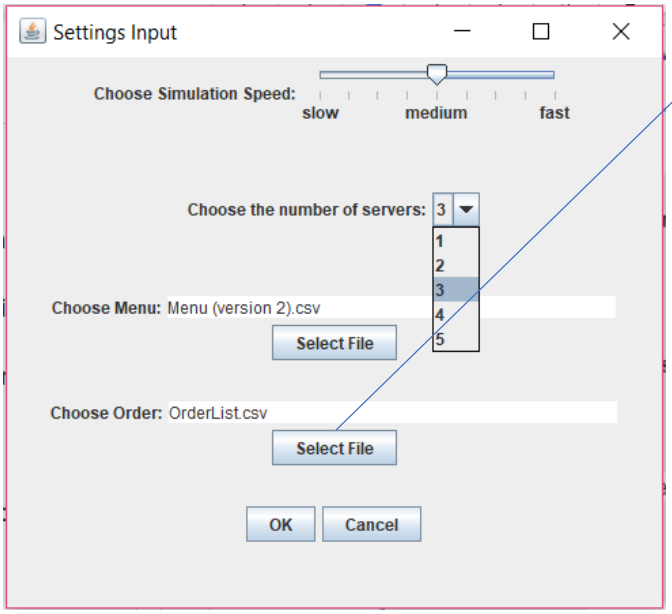
A second singleton controls the simulated date and time. This must be a singleton so that all objects within the program are using exactly the same time.

The Model-View-Controller pattern was used as a basis for the main GUI. All the data is stored in the model and the main GUI is the view. There is no controller as the main GUI has no interactive elements so any changes to the data happen within the model.

To reflect changes to the model data in the view the observer pattern was used in multiple places. The sales assistant panel observes it's sales assistant in the model, the queues are observed by the queue panel and the simulation time panel observes the simulation time in the model. To ensure uniform implementation of this pattern across the program Subject and Observer interfaces were created and these had to be implemented by any class using the pattern.

Producer/consumer was used as a basis for implementing multiple Sales Assistants "consuming" customers from the same shared Queue, however this has been modified from the lecture material to accommodate multiple shared objects. This is discussed in more detail in the section above.

Sample screenshots



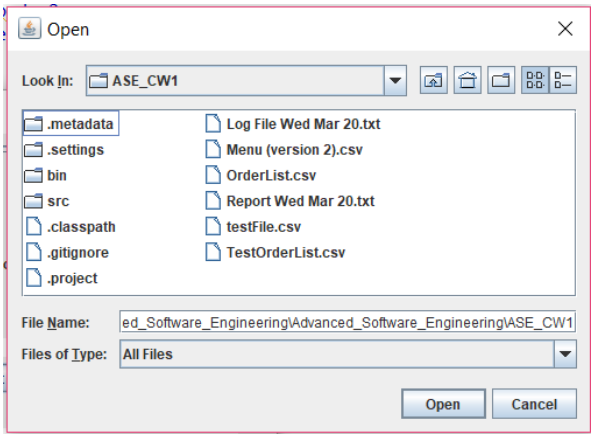


Figure 10: StartGUI

Figure 11: Choose Menu/ Choose Order Pop-Up

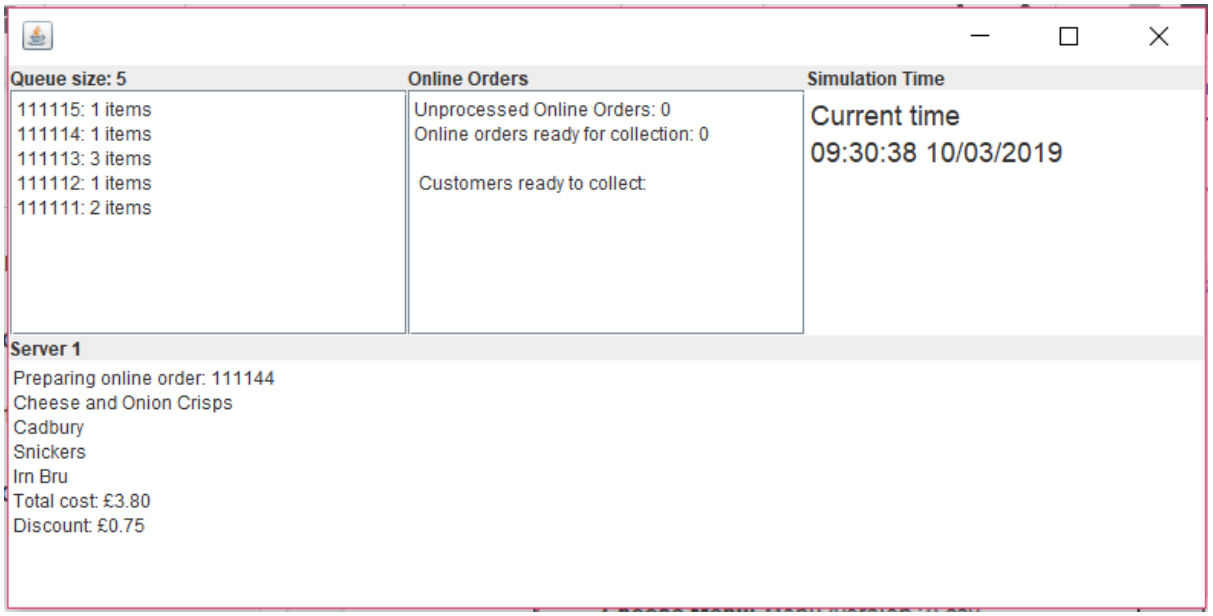


Figure 12: CafeStateGUI (One Server)

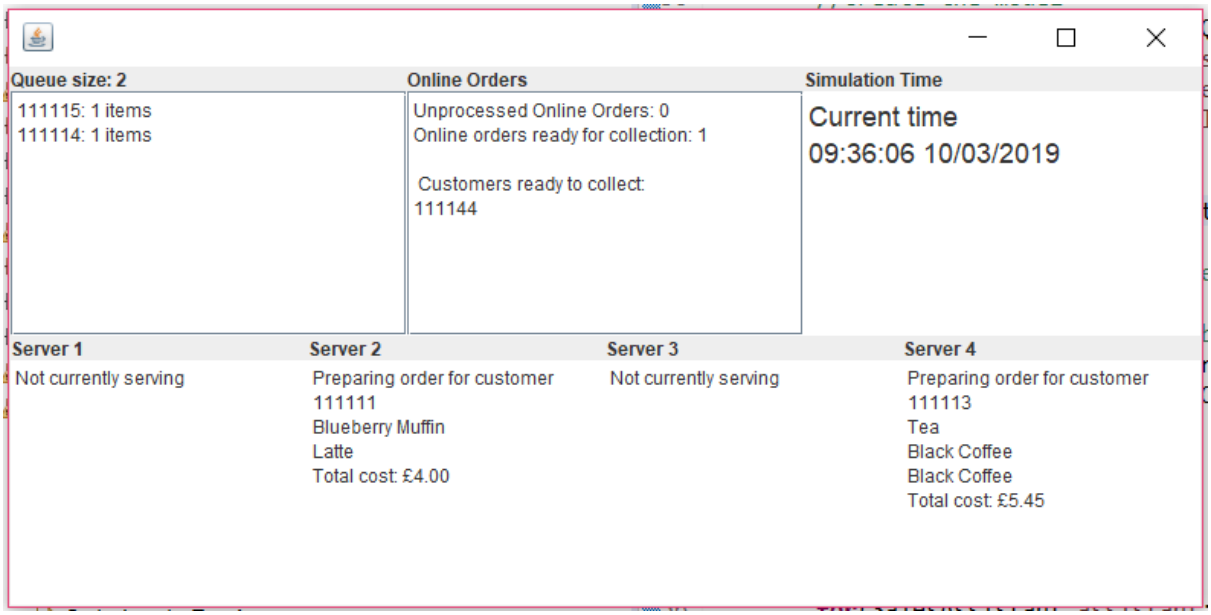


Figure13: CafeStateGUI (Four Servers)

Comparison of Stage 1 and Stage 2:

Stage 1:

During the plan-driven stage, the planning of everything first was difficult due to not knowing how things would go and what the best way would be. In places the plan wasn't detailed enough or we made misjudgements in the class structure. Sometimes we missed the fact that helper classes might be required in order to achieve certain functionalities. It was difficult to estimate the time it would take to complete tasks as we hadn't tried to complete any of it yet due to all being done at the start. It was also harder to communicate changes to the plan if we made any because of the less actively communications as a group and not having a set times to do so, such as our planning sessions or stand up meetings.

Overall, it was frustrating sticking to the requirement of doing all the planning before you do coding. Problems that would normally be handled by rethinking the approach, such as the poor modelling of orders, were not changed to comply with the requirements. Team was aware there was better way to handle it, but had to make do with the decisions we'd made.

Stage 2:

The sprint planning sessions were helpful and gave us a little more time to plan in more detail each feature we wanted to implement in that sprint, each plan was usually focused on a particular topic allowing planning sessions to be more concise and to the point. It was difficult to do 'true' agile since we all have different schedules and times when we can work, therefore it was hard to do proper stand-ups. We did try a few of these though. Due to this, communication and keeping each other up to date wasn't always easy. Methods used (slack, google drive) worked well enough to get the job done, but definitely weren't as good as proper agile would have been (ie working on it full time daily meetings etc?). Also due to having individual sprints it was easier to see when work was due for each other and when tasks were being completed. We found pair programming was helpful and made sure we could keep up with changes in the code that each of us had made.

To conclude, we found that the flexibility and the less restrictive planning used in the Agile technique of stage 2 was a much more productive way to achieve our plans, than that of the plan-driven development in stage 1. Due to not being as restricted, we could have a fresh discussion about new ideas during every sprint. The use of pair programming helped to involve each other more in all tasks. Lastly, due to the weekly stand-up meetings, sprint planning and retrospective sessions, it forced better communication between all group members.

We had some difficulty applying strictly agile practices to the project. Instead of having a daily standup meeting we had to meet less frequently due to people's differing workloads and schedules. It would have been beneficial to meet more frequently and discuss progress more often to keep everybody in the loop. Tracking issues on github rather than using a google doc may have been a more eloquent solution. This would increase the ease of; assigning people to issues, marking them as fixed and viewing issues retrospectively.

Although we established a coding convention early and tried to adhere to it as much as possible our commenting could've been improved. We didn't establish the same level of convention for comments as we did for coding style so there was some discrepancy in comments across the program which often had to be fixed at a later date. We should have created a standard on how could should be commented and what style the comments should use rather than having to go back and alter comments.