Heriot Watt University

ADVANCED SOFTWARE ENGINEERING

F21AS

Stage 2

Group Report

Jowita Knap	H00301565
Lorenzo James	H00309663
Lucía Parga Basanta	H00313273
Sabrina Chiesurin	H00314757

Table of Contents

Overview

In this second stage, the application is extended to simulate customers queuing at the counter and having their orders processed by coffee shop staff. We use threads and three design patterns that help us to develop the application. We use Singleton pattern to implement our log class, Observer and MVC patterns in our GUI. We use Scrum as agile framework. The application fully meets the specification.

Core functions

- The simulation consist of multiple serving staff and a single queue of customers that are waiting to be served. When new costumers arrive at the coffee shop, they join the back of the queue. When they reach the front of the queue, they will be processed by the next available member of serving staff;
- When it starts up, the application read in details of existing orders. These orders are gradually added to a queue;
- When a member of serving staff becomes free, they will begin to process the order at the front of the queue. Once the order has been processed, they move on to the next order in the queue;
- Once the queue is empty, the coffee shop closes, a report is generated and the program exits;
- In the GUI is it possible to see the customers and their orders waiting in the queue, and details of what each member of serving staff is currently doing throughout the simulation. The timing is slow enough to watch the displays changing;
- There is a log in which are kept records of the events as they happen. The log is written to a file when the application exits.

Limitations and Bugs

- The user can add an unlimited number of Staff members.
- The program scales off screen when too many staff members are added.
-

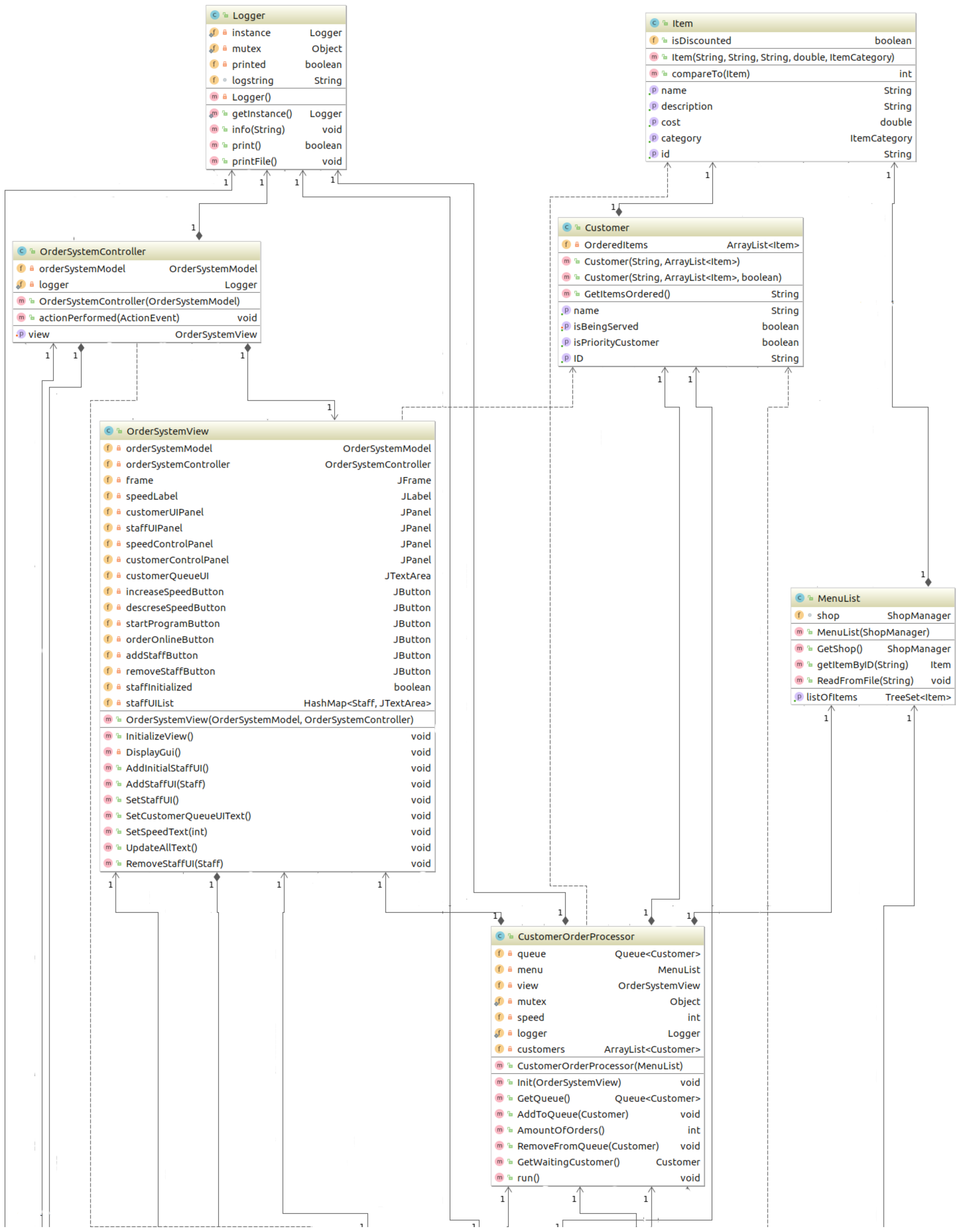
Extended functions

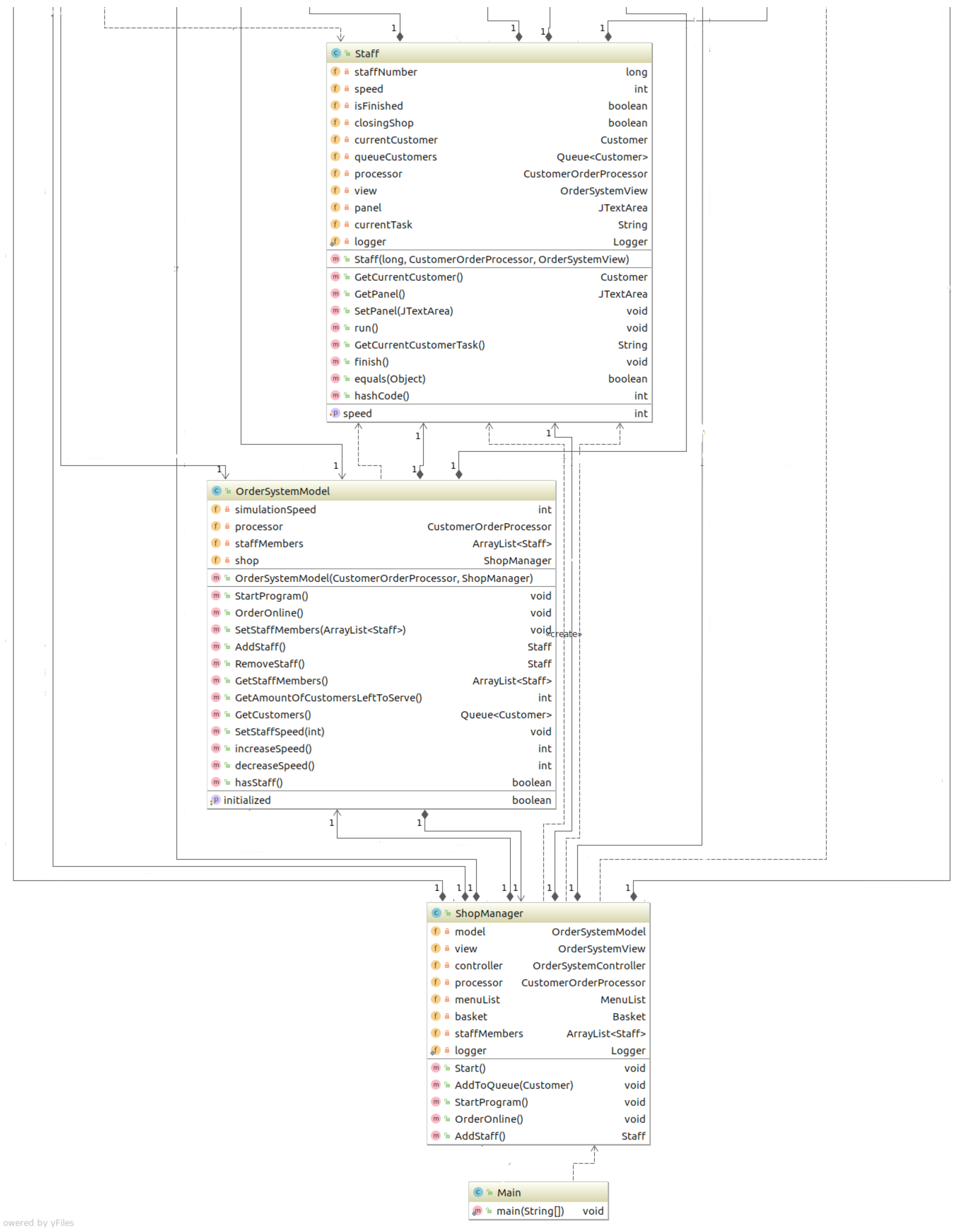
- Allow the user to press buttons to alter the speed of simulation using runtime controls;
- Allow customers to order online in advance, we used our stage one GUI to simulate this. The Customers that ordered online have the priority over the regular customers in the queue.
- Allow users to add and remove staff members by clicking buttons.

In this report are shown the details of the implementation and the methodologies used to develop and complete the work.

UML Class Diagram

The class diagram is shown below, in two pages. We use IntelliJ IDEA and GNU manipulation program.


Logger
instance Logger
mutex Object
printed boolean
logstring String
Logger()
getInstance() Logger
info(String) void
print() boolean
printFile() void
Item
isDiscounted boolean
Item(String, String, String, double, ItemCategory)
compareTo(Item) int
name String
description String
cost double
category ItemCategory
id String
OrderSystemController
orderSystemModel OrderSystemModel
logger Logger
OrderSystemController(OrderSystemModel)
actionPerformed(ActionEvent) void
view OrderSystemView
Customer
OrderedItems ArrayList<Item>
Customer(String, ArrayList<Item>)
Customer(String, ArrayList<Item>, boolean)
GetItemsOrdered() String
name String
isBeingServed boolean
isPriorityCustomer boolean
ID String
OrderSystemView
orderSystemModel OrderSystemModel
orderSystemController OrderSystemController
frame JFrame
speedLabel JLabel
customerUIPanel JPanel
staffUIPanel JPanel
speedControlPanel JPanel
customerControlPanel JPanel
customerQueueUI JTextArea
increaseSpeedButton JButton
descreseSpeedButton JButton
startProgramButton JButton
orderOnlineButton JButton
addStaffButton JButton
removeStaffButton JButton
staffInitialized boolean
staffUIList HashMap<Staff, JTextArea>
OrderSystemView(OrderSystemModel, OrderSystemController)
InitializeView() void
DisplayGui() void
AddInitialStaffUI() void
AddStaffUI(Staff) void
SetStaffUI() void
SetCustomerQueueUIText() void
SetSpeedText(int) void
UpdateAllText() void
RemoveStaffUI(Staff) void
MenuList
shop ShopManager
MenuList(ShopManager)
GetShop() ShopManager
getItemByID(String) Item
ReadFromFile(String) void
listOfItems TreeSet<Item>
CustomerOrderProcessor
queue Queue<Customer>
menu MenuList
view OrderSystemView
mutex Object
speed int
logger Logger
customers ArrayList<Customer>
CustomerOrderProcessor(MenuList)
Init(OrderSystemView) void
GetQueue() Queue<Customer>
AddToQueue(Customer) void
AmountOfOrders() int
RemoveFromQueue(Customer) void
GetWaitingCustomer() Customer
run() void


Staff
staffNumber long
speed int
isFinished boolean
closingShop boolean
currentCustomer Customer
queueCustomers Queue<Customer>
processor CustomerOrderProcessor
view OrderSystemView
panel JTextArea
currentTask String
logger Logger
Staff(long, CustomerOrderProcessor, OrderSystemView)
GetCurrentCustomer() Customer
GetPanel() JTextArea
SetPanel(JTextArea) void
run() void
GetCurrentCustomerTask() String
finish() void
equals(Object) boolean
hashCode() int
speed int
OrderSystemModel
simulationSpeed int
processor CustomerOrderProcessor
staffMembers ArrayList<Staff>
shop ShopManager
OrderSystemModel(CustomerOrderProcessor, ShopManager)
StartProgram() void
OrderOnline() void
SetStaffMembers(ArrayList<Staff>) void
AddStaff() Staff
RemoveStaff() Staff
GetStaffMembers() ArrayList<Staff>
GetAmountOfCustomersLeftToServe() int
GetCustomers() Queue<Customer>
SetStaffSpeed(int) void
increaseSpeed() int
decreaseSpeed() int
hasStaff() boolean
initialized boolean
«create»
ShopManager
model OrderSystemModel
view OrderSystemView
controller OrderSystemController
processor CustomerOrderProcessor
menuList MenuList
basket Basket
staffMembers ArrayList<Staff>
logger Logger
Start() void
AddToQueue(Customer) void
StartProgram() void
OrderOnline() void
AddStaff() Staff
Main
main(String[]) void
1
owered by yFiles

Communication and Development using Agile Processes

Scrum

In this second stage of the work we use Scrum, a methodology for effective team collaboration that help us to coordinate and split the works among the team members.

Stand-ups and sprints

We had three weeks and we divided this time available into three Sprints of one week each. It was not possible to meet every day to do stand-up meetings due to the timetables and the availability of the members. We decided to meet every Monday and Thursday to have “three-day scrum sessions”, which made it possible to synchronize the activities and create a plan for the next few days. During the stand-ups each person mentioned what they did during the sprint, what problems they came across and what they will be doing for the next few days.

Planned poker

During the first meeting, we divided the work in smaller tasks and we estimate the time needed to achieve them. We did this using the Planning Poker estimation technique. This technique allows the members of the group to estimate the time needed for each task by playing numbered cards face-down to the table, instead of speaking them aloud. Once that the cards are revealed, we did some discussions about them. We decided to use this method to avoid being influenced by the opinion of the other members of the team. We struggled with estimating the time to complete tasks due to lack of experience.

Sprint reviews

At the end of each Sprint, we had a Sprint review. During this Sprint review, the work done was summarized and discussed. We then talked about the improvement to do in the next sprint to be more efficient. The sprint reviews were not done the official way. They were more of a longer version of a stand-up meeting.

Iterations

During every sprint review we iterated on the program.

1. We wanted to have basic GUI working and also have a Staff class, MVC classes and a class that handled queues.
2. We included a logger, connecting the staff class with the class which implemented Queues and connected MVC to the GUI.
3. We added customers, changed how the queue worked and we updated the GUI
4. We decided to use pair programming to effectively use our time to add the last features and to merge each other’s work.

Pair programming

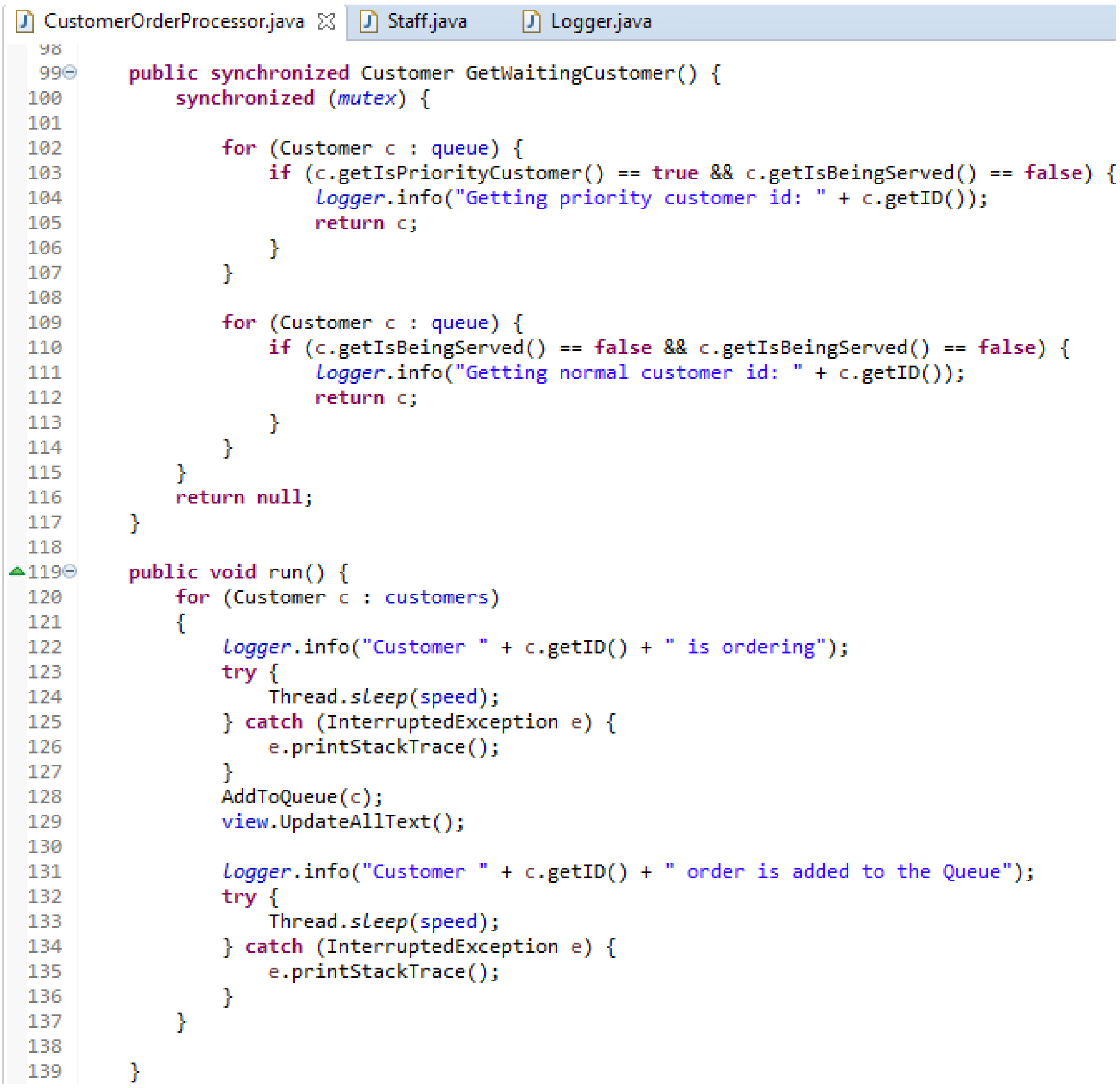
We used pair programming on tasks which were complex like threading in the staff members class. It was a success and we decided that it was an efficient way to work. During the last week of the project we were not able to use scrum effectively, because of time limitations. There were many tasks left to do so we decided that pair programming was the most effective way for all the members of the group to contribute the same amount which was a success. The heavy use of pair programming reflects in the commit history of our group which explains any imbalances.

GitHub

We use GitHub to store the code, available at:
https://github.com/lorenzo456/AdvancedSoftwareEngineeringCoursework

Threads

In this section, we present the use of threads in the application.



CustomerOrderProcessor.java | Staff.java | Logger.java

```
    public synchronized Customer GetWaitingCustomer() {
        synchronized (mutex) {

            for (Customer c : queue) {
                if (c.getIsPriorityCustomer() == true && c.getIsBeingServed() == false) {
                    logger.info("Getting priority customer id: " + c.getID());
                    return c;
                }
            }

            for (Customer c : queue) {
                if (c.getIsBeingServed() == false && c.getIsBeingServed() == false) {
                    logger.info("Getting normal customer id: " + c.getID());
                    return c;
                }
            }
        }
        return null;
    }

    public void run() {
        for (Customer c : customers)
        {
            logger.info("Customer " + c.getID() + " is ordering");
            try {
                Thread.sleep(speed);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            AddToQueue(c);
            view.UpdateAllText();

            logger.info("Customer " + c.getID() + " order is added to the Queue");
            try {
                Thread.sleep(speed);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
```

Fig.2 Here you can see the use of Threads in the CustomerOrderProcessor. This class runs as a Thread adding customers into the Queue. It also has a synchronized method which allows Staff members to get a customer from the queue. It uses a mutex lock to prevent multiple thread from accessing it at the same time.

Threads

Each thread we have in our program is called in the ShopManager Class. The way we call the Threads is by calling the Start() method in each of them. The following classes make use of threads:

Staff class:

- An instance of the staff class is always ran as a thread.
- While running the staff instance is waiting for the its turn to use the GetWaitingCustomer() method, which has a mutex lock on it. That method returns a customer the staff instance can then serve.
- Once the staff instance has a customer it's serving, the thread sleeps to simulate the processing of orders. Once that is complete it will remove the customer from the queue and look for the next customer.
- If there are no customer it will then look to close the store if no other instance of Staff is doing it already.

CustomerOrderProcessor class [Fig.2]:

- CustomerOrderProcess is always ran as a thread.
- It creates virtual customers to a list.
- In the thread this class will loop through the list of customers and add them to the queue. With a small delay to simulate customers ordering from a queue. This delay is done by making the thread sleep.

Application threads safe

The application is thread safe, with the possible exception of Staff instances closing the shop. Despite there being Booleans to prevent other threads from accessing the printFile() method in the logger (Which prints the file and closes the program) there is no mutex lock protecting it.

The other methods all either use mutex locks preventing other threads from accessing it while it's being used or make use of synchronised methods.

Design Patterns

In this section, we present the use of design patterns in the application.

Singleton Pattern

Singleton Pattern is a design pattern that is used to restrict the instantiation of a class to one single instance. This instance can be accessed by all classes in the program therefore it is important for there to only be one instance of the singleton class.

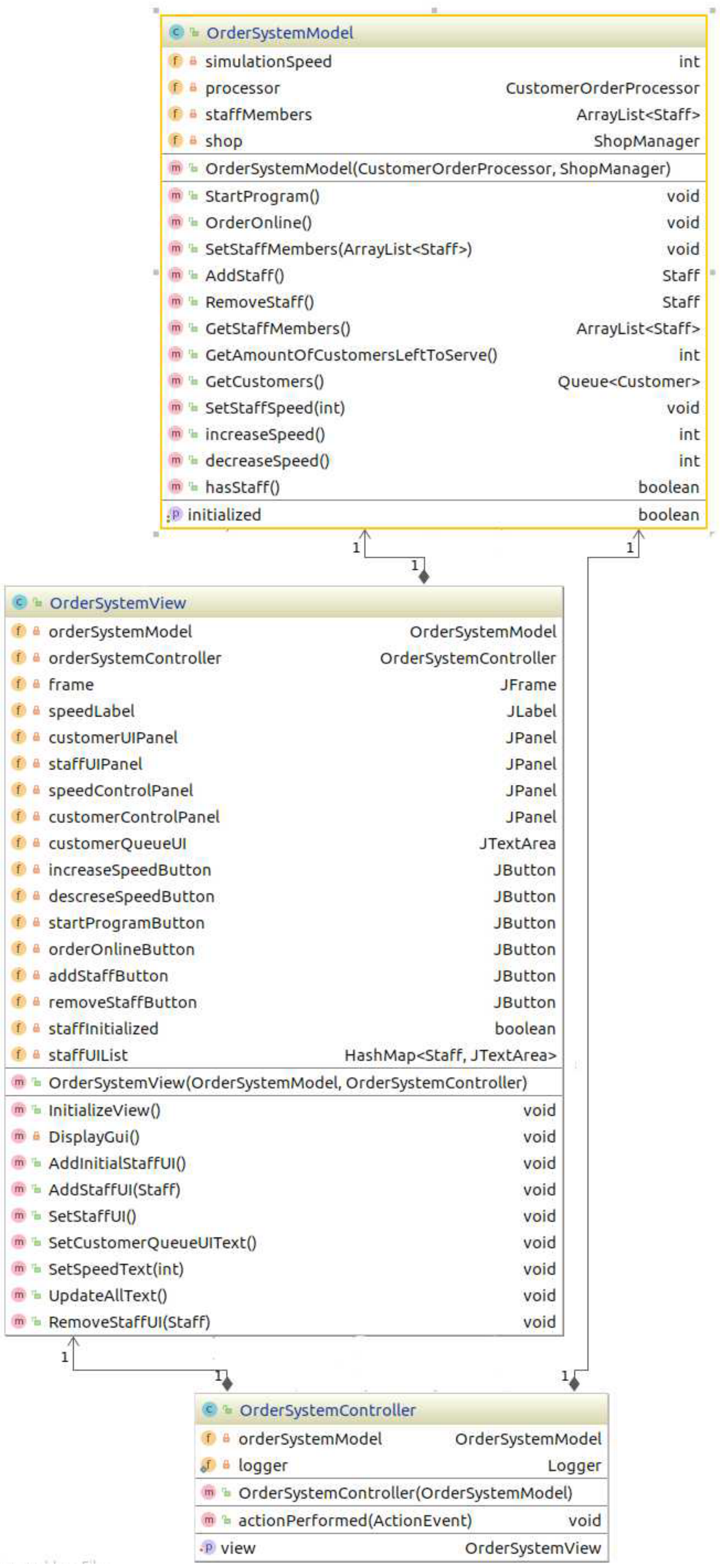
The logger class uses the singleton pattern, because we only want one instance of the logger. The logger needs to be a singleton so it is easy for each class in our program to log their current status, without having to pass a reference of the logger to each and every class. Every time an action occurs in our program it is logged in the logger class through the info(String message) method. This method adds each message passed into the logger to a string which a staff member can later print to a file, when they close the shop.

MVC Pattern

We implemented MVC Pattern for our GUI. This design Pattern is stored in the MVC package and it is implemented in the following classes:

- **OrderSystemModel**
    - Is the Model. It stores all the methods that have an effect on the GUI;
- **OrderSystemView**
    - Is the View. This class alters the look of the GUI, with the values passed on from the OrderSystemController;
- **OrderSystemController**
    - Is the Controller. This class control the methods that are created for the GUI and it pass them to the OrderSystemView;

We created a diagaram of the MVC Pattern with IntelliJ IDEA. The diagram is seen below.


OrderSystemModel
simulationSpeed int
processor CustomerOrderProcessor
staffMembers ArrayList<Staff>
shop ShopManager
OrderSystemModel(CustomerOrderProcessor, ShopManager)
StartProgram() void
OrderOnline() void
SetStaffMembers(ArrayList<Staff>) void
AddStaff() Staff
RemoveStaff() Staff
GetStaffMembers() ArrayList<Staff>
GetAmountOfCustomersLeftToServe() int
GetCustomers() Queue<Customer>
SetStaffSpeed(int) void
increaseSpeed() int
decreaseSpeed() int
hasStaff() boolean
initialized boolean
1
1
1
OrderSystemView
orderSystemModel OrderSystemModel
orderSystemController OrderSystemController
frame JFrame
speedLabel JLabel
customerUIPanel JPanel
staffUIPanel JPanel
speedControlPanel JPanel
customerControlPanel JPanel
customerQueueUI JTextArea
increaseSpeedButton JButton
descreseSpeedButton JButton
startProgramButton JButton
orderOnlineButton JButton
addStaffButton JButton
removeStaffButton JButton
staffInitialized boolean
staffUIList HashMap<Staff, JTextArea>
OrderSystemView(OrderSystemModel, OrderSystemController)
InitializeView() void
DisplayGui() void
AddInitialStaffUI() void
AddStaffUI(Staff) void
SetStaffUI() void
SetCustomerQueueUIText() void
SetSpeedText(int) void
UpdateAllText() void
RemoveStaffUI(Staff) void
1
1
1
OrderSystemController
orderSystemModel OrderSystemModel
logger Logger
OrderSystemController(OrderSystemModel)
actionPerformed(ActionEvent) void
view OrderSystemView

Observer Pattern

We use it in the MVC pattern. Our OrderSystemController class is using the Observer pattern, it observes what is happening in the OrderSystemView and passes this on to the OrderSystemModel which contain the majority of the Methods that change the GUI.

Stage 1 and stage 2

A comparison