# COS80007 Advanced Java – Assignment Two
## Formal Written Report
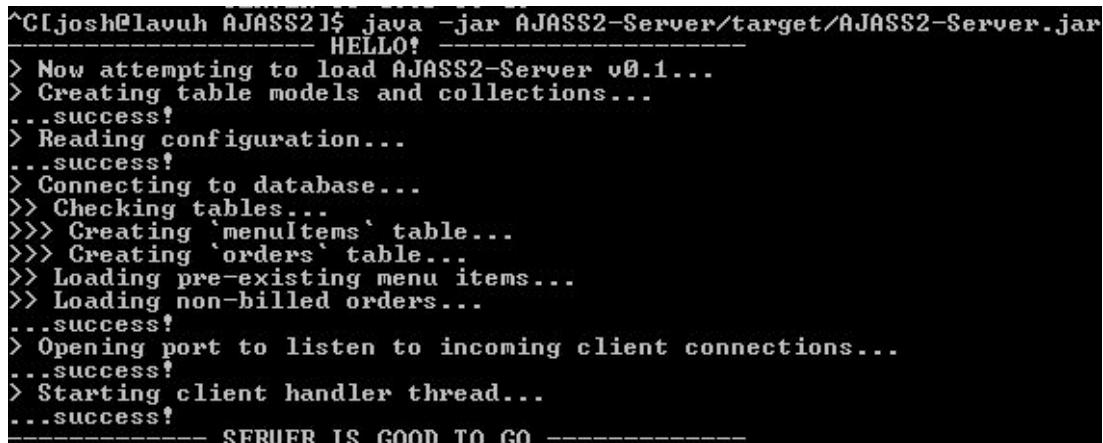
## Contents of Report

## Introduction

This report goes over the behaviour and implementation (in detail) of the program our group created for assignment two. Sequential screenshots will be provided for demonstration, and additional screenshots will be provided for explanation later on in the report. Justifications for marks in certain areas will also be included in this report.

## Demonstration of Clients & Server

This section focuses on a simulated, practical use of the client and server software. In this demonstration, there will be three actors: the waiter, the chef, and the cashier. The server will also run in the background but does not require intervention from a user.
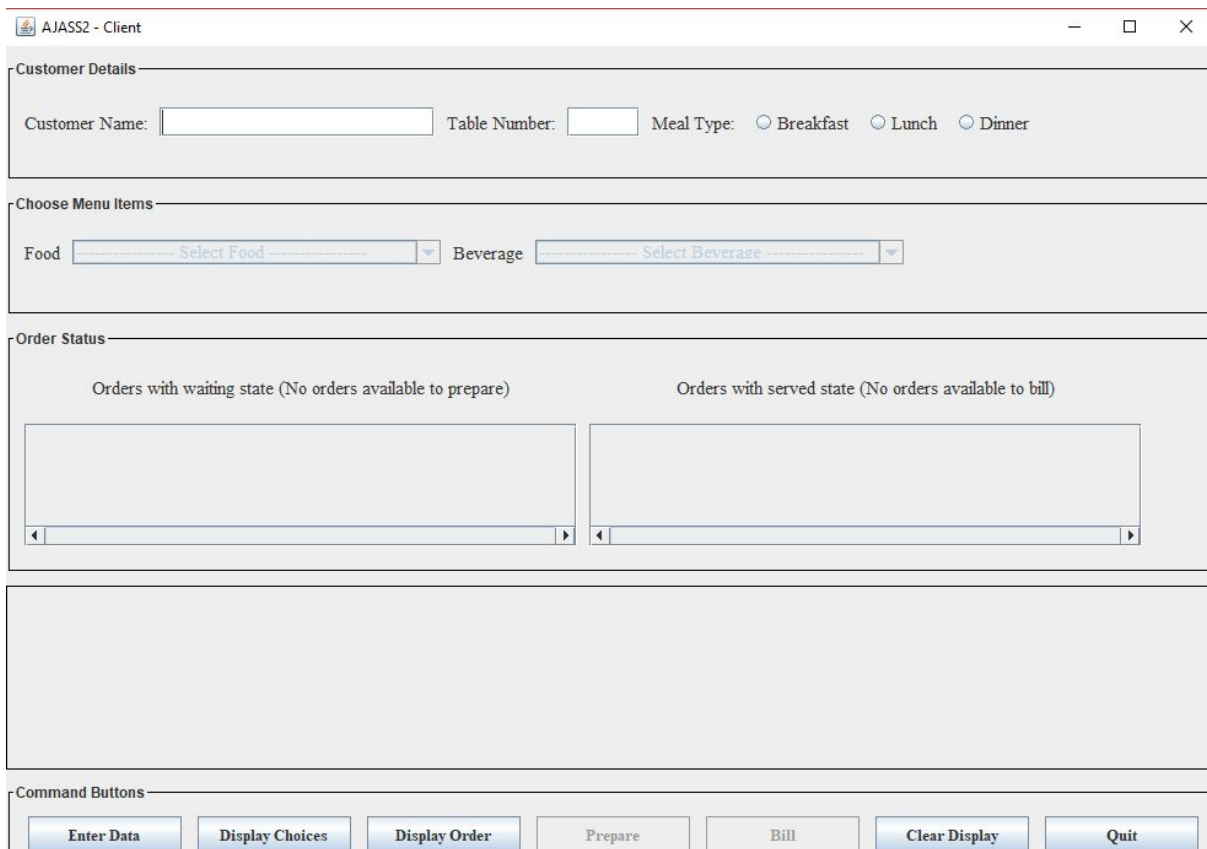
### Step 1. Opening the Server and Connecting the Clients

*Screenshot 1.1: Loading up a fresh instance of the server after compilation.*



As shown in screenshot 1.1, the server goes through a number of processes. They are all explained later in the report. The server is hosted on 0.0.0.0, meaning it can be accessed from any IP address that points to it. Now, we must connect the three clients to the server.

*Screenshot 1.2: Screen shown on all clients after successfully connecting to a server.*



*Screenshot 1.3: Output from the server after connecting 3 clients (note the different ports).*



*Screenshot 1.4: Output from the client after connecting to the server (if run from cmd prompt).*



## Step 2. Taking Orders from a Customer

There are now 3 simultaneous connections to the same server from 3 different client instances. As explained in screenshot 1.2, all client instances will always see the same thing in their user interfaces. However, each different user has a different operating purpose for the program. They are listed as follows:

- Waiter: Creates orders for a customer at a table, which can be of a food item or a beverage item, or both, but not neither.

- Chef: Takes orders that are waiting to be prepared, and prepares them. Multiple waiting orders can be prepared at the same time if necessary.
- Cashier: Takes orders that are waiting to be billed, and bills them. Multiple prepared orders can be billed at the same time if necessary.

In the first step of the order process, the waiter will take orders from customers. The customer's name, their table number, and preference of food/beverage should be recorded. The waiter can enter as many orders as necessary.

*Screenshot 2.1: Output from the server after the Waiter's client adds an order.*

> Command from Client: CLIENT_CREATE_ORDER

*Screenshot 2.2: Output from each client after the Waiter's client adds an order.*

Command from Server: SERVER_ADD_ORDER

*Screenshot 2.3: Screen shown on all clients after the Waiter adds an order.*



*Screenshot 2.4: Table shown when chef enters a specific table number and clicks the 'Display Order' button.*

| Customer Name | Ordered Items |
|---|---|
| Joshua | Egg scrambled, Chai latte regular fat cows milk |
| Bradley | Muesli commercial toasted added nuts, Coffee made up with regular fat cows milk |
| Keagan | rolled oats prepared with regular fat cows milk, Coffee cappuccino with skim cows milk |

The waiter has now added orders that the chef can see (screenshot 2.3) and must prepare them to continue the process. The waiter can continue adding orders if necessary.

**Step 3. Preparing a Customer's Order**

The specifics on how a chef may prepare an order are irrelevant in the scope of the program. We are assuming that the chef 'prepares' the orders in the program after they have been physically prepared. As shown in screenshot 2.4, the chef can view all orders on a specific table and do them all first before moving onto another table if so wanted. The chef can prepare multiple orders at a time, as stated earlier.

*Screenshot 3.1: Output from the server after the Chef's client prepares an order.*

> Command from Client: CLIENT_UPDATE_ORDER

*Screenshot 3.2: Output from each client after the Chef's client prepares an order.*

Command from Server: SERVER_UPDATE_ORDER

*Screenshot 3.3: Screen shown on all clients after the Chef prepares an order.*



The chef has prepared Joshua's order on table 1. It is now served, as seen in screenshot 3.3. The chef's job relies more heavily on preparing food than operating this software, so their role as a chef is minimal. The waiter and cashier can now also see that Joshua's order on table 1 has moved to the 'served' state.

**Step 4. Billing a Customer's Order**

The cashier is responsible for making sure that served orders are paid for. The customer will usually go up to the cashier and tell the cashier what they ordered. If this is the case, a single order can be billed for that customer. If the customer does not remember what they ordered or is wanting to pay for the whole table, the cashier can query a list of orders on a table as shown in screenshot 2.4. If a whole table of orders or a specific set of orders is being paid for at the same time, they can be billed in bulk.

*Screenshot 4.1: Receipt dialog shown on Cashier client after billing an order.*
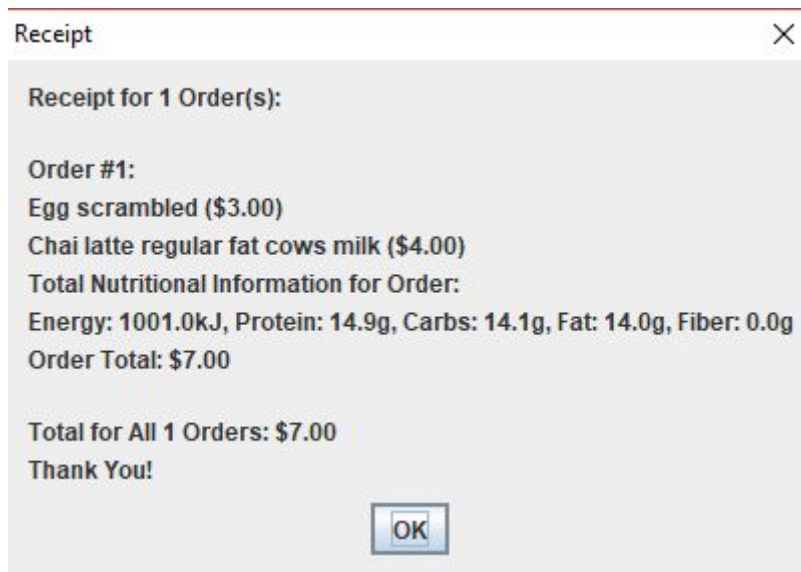


*Screenshot 4.2: Screen shown on all clients after the Cashier bills an order.*
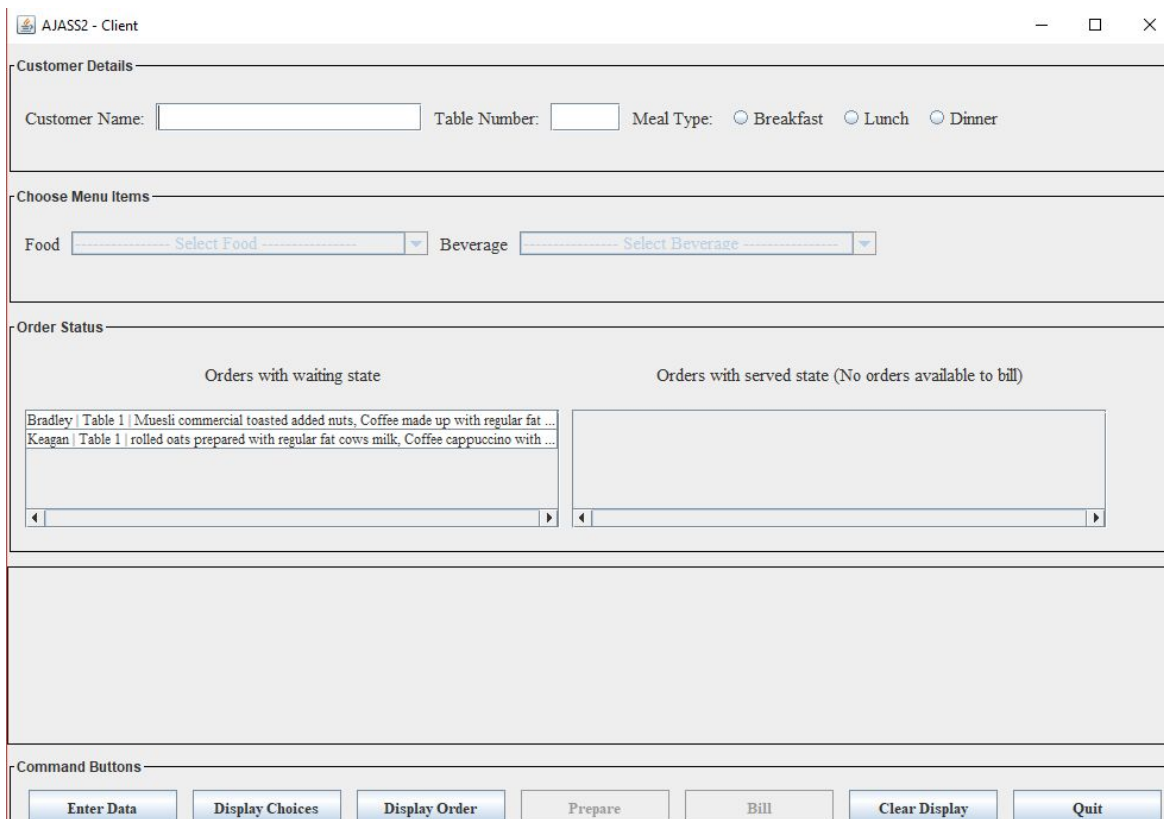
Shown in screenshot 4.2, Joshua's order on table 1 has been billed and has disappeared from the system. Command prompt outputs are not shown as they are the same from step 3, as an order's state is being updated from served to billed. A generated receipt is also shown in screenshot 4.1

### Step 5. Consuming the Process

This process is to be repeated to make full use of the software in a restaurant/food service scenario. As shown, the three use cases can make full use of the software by utilising different parts of it.

Note: Each use case has full access to other parts of the software (i.e. a waiter can bill an order, a cashier can prepare an order, etc.) because in the unlikely scenario that two use-cases are standing together in front of the same client and need to use a function that isn't available, they will need to physically go to a client that has the required feature. It's not really a security feature- it will be a burden more times than not.

## Threaded TCP Connections to Server/Clients

### Part 1. Establishing Connections

TCP Sockets were chosen as the method of choice for connection between the server and client software. RMI was considered at first, but we soon learned that RMI was not the right kind of choice for this software's requirements. RMI is good because clients can invoke methods on a remote server, but the server cannot invoke remote methods on clients. This lack of two-way communication is an intentional design choice of RMI, so we switched to TCP.

Both the client and the server software share the same way of establishing connections, sending messages, and receiving messages. The server however has to keep track of multiple, incoming connections whereas the client has one single, outbound connection.

### Part 2. Maintaining Connections

**Clients** send out a TCP Socket request to connect to the specified server, and if accepted, is stored in a ServerConnection model. The ServerConnection model has a special heartbeat thread that sends out a little "Hello" message every now and then to prevent the connection from timing out for no reason. **The server** listens for incoming TCP connections from clients, and once found, accepts them and stores the Socket in a ClientConnection model. The ClientConnection model is taken and stored in a List, where the server can keep track of all the connected clients.

### Part 3. Writing/Receiving Messages via TCP

When it comes to **writing** messages, the server can send a message back to an individual client (in special cases) or write a message in bulk to all connected clients. The client can only send messages to the server, and not to other clients directly. When it comes to **receiving** messages, both the client and server have a **read thread** that is started when the

ClientConnection/ServerConnection is created. This thread runs infinitely until the connection is lost, and continually listens for incoming 'messages'.

Messages that are received in the read thread are placed in a Linked Blocking Queue in both the client and server software, which are handled sequentially by a **handler thread**.

### Part 4. Handling Received Messages

Thanks to the existing connection software, we are guaranteed to have our message arrive in order. Both the client and server make use of a linked blocking queue to guarantee two things: 1, the messages received are kept in the same order and are processed in the same order. 2, the software waits for the full message to come through. The beauty of using a Blocking Queue is that if the software attempts to get the next element from the queue but it is empty, it will wait until another element is put into it and then get that one. This makes perfect sense for TCP as it can be latent, and messages may arrive at varying speeds. It is good to wait.

The **handler thread** continually takes messages from the blocking queue, processing them based on what data is sent through. What is sent through will be discussed in more detail later on, but for now all that needs to be understood is that messages are processed through a blocking queue as they come through. This applies to both the server and client software.

### Part 5. Summary

This is in essence how the server and client communicate with each other. Both parts of the software have threads to receive and handle messages, and sometimes write messages. Using Threads to handle the different workloads of the software is important to not hold up other important operations. The next step is to look into not *what* the software talk with, but rather *how* they talk with it. Communicating in a structured way is just as important!

## Issues with Firewall, using VPS

It quickly came to recognition that the Swinburne Firewall blocks unrecognised TCP/UDP traffic. For our group, this would have been detrimental to our demonstration as the entire demonstration would have to be done on one computer, which really isn't representative of the three use-cases the software has. Fortunately, Joshua has supplied a personal VPS (Virtual Private Server) located in Sydney to run the server instance on for testing and demonstration. Using our own mobile data, we were able to access the VPS.

Not only does improve the program's performance as it is now in a proper command-line server operating environment, but it also allows the group members to connect with different clients on different internet connections in very different locations! (We all live in different towns!) This allowed us to demonstrate without having to be physically together and creating a LAN. It also shows how the software works, even over a slower TCP connection.

## Software Communication Protocol

The client and server have a shared, specialized way of communicating with one another through the use of an enumerated type, called 'Communication'. The rule is that before any data is sent, a Communication must be sent first to let the adversary know how to treat the incoming data. There are many different types of Communications, some of which are server-specific and some of which are client-specific. This section will go over them all in detail.

### Part 1. The SENTINEL

Sentinel is a value in Communication which effectively means stop. There are some kinds of communication where there is a definitive amount of data sent (i.e. when creating an order, you have one customer name, one table number, one food item, etc.) where a Sentinel is not required. In the kinds of communications where there is an non-definitive amount of data being sent through (i.e. the list of current orders, which is never the same size always in normal operation), a Sentinel is required to terminate the communication since a while loop is used. Using sentinel breaks the wild loop and stops the communication so the next one can be handled correctly.

### Part 2. Client-to-Server Communications

All client communications are a 'wanting' or 'sending' type of communication. The client is always letting the server know something when it writes a message to it. Here are the different kinds of communications that originate from the client:

### a. CLIENT_WANT_ITEMS

When a client starts a communication to the server with CLIENT_WANT_ITEMS, it is letting the server know that the client does not have the list of menu items and requires them. This is because the Menu Items from the .csv file are read and stored server-side instead of individually on the client. No other data is sent *to the server*, as it is just a request. This is beneficial, because only the server has to have its menu items updated instead of each individual client.

### b. CLIENT_WANT_ORDERS

This communication is similar to CLIENT_WANT_ITEMS. It is letting the server know that the client does not have the list of currently waiting and served orders and required them. Like the menu items, the original and central copy of the Orders are kept server-side too. When the client loads up, it needs a list of the existing orders from the server, so it sends this request. No other data is sent to the server, it is also a request.

### c. CLIENT_HEARTBEAT

This communication is not intended to do anything. An issue our group had while developing the client/server prototype was that the connection would time out when nothing was sent for a long period of time. CLIENT_HEARTBEAT was created as a way to

resolve this, as it is sending something to the server every now and then to avoid connection timeouts.

### d. CLIENT_CREATE_ORDER

When a client starts a communication to the server with CLIENT_CREATE_ORDER, it is letting the server know that this client has created an Order through the UI and needs to be processed centrally on the server. The actual Order object is not created on the client; the data to create the Order is sent through to the server, created there, and then distributed back to all the clients. Following this communication, the client also sends a bunch of information to the server. This includes the table number, customer name, food (if selected), and beverage (if selected). No further action takes place on the client's part. The Order has technically not been created yet.

### e. CLIENT_UPDATE_ORDER

When a client starts a communication to the server with CLIENT_UPDATE_ORDER, it is letting the server know that this client has updated an existing Order's status. This can be due to preparing an order (WAITING -> SERVED) or due to billing an order (SERVED -> BILLED). Following this communication, the client also sends a bunch of information to the server. This includes the Order's old state, the table number of the order, the position of the Order in its linked list, and the Order's new state. The actual state of the Order is not updated client-side when this happens. No further action takes place on the client's parts.

### Part 3. Server-to-Client Communications

All server communications are in acknowledgement to communications from a client. The server is *lazy* and will never go out of its way to communicate with its clients unless they communicate with the server first.

### a. SERVER_SEND_ITEMS

The server only responds with SERVER_SEND_ITEMS in response to a *CLIENT_WANT_ITEMS* communication. This message is never sent to more than one-client at a time. When the server sends this, it is acknowledging a client's request to receive the list of current menu items, as the server keeps the original, central copy of them. After sending the communication, the server will send the client the list of all menu items in a *strict order*. Since the lists of menu items are mapped to a specific food type and meal type, they must be sent and re-constructed in the same way. The process is simple:

*1. For each type of food (food or beverage), send the next food type we are about to send.*

*2. Then for each meal type (breakfast, lunch, dinner), send the next meal type we are about to send.*

*3. For each item of this specific food and meal type, send each individual menu item.*

*4. If this is the last menu item for this meal type, go to step 2.*

*5. If this is the last food type, send a SENTINEL to stop communication, otherwise go to step 1.*

The client then receives the server's SERVER_SEND_ITEMS communication, and can handle the incoming data by reconstructing it in the same order that it is sent. The process is simple, too:

*1. Get the first food type the server is about to send (food, beverage).*

*2. Get the first meal type the server is about to send (breakfast, lunch, dinner).*

*3. Receive a Menu Item, store it in the mapped list of the current food and meal type. (unless 3x/xx)*

*3x. If the Object received is a food type, change the food type to the food type sent.*

*3xx. If the Object received is a meal type, change the meal type to the meal type sent.*

*4. Repeat 3, 3x, and 3xx while the Object received is not a SENTINEL communication.*

Cross-referencing the steps shows that the menu items end up in the exact same state on the client as they do on the server. Magic! Not really, just well-structured communication.

## b. SERVER_SEND_ORDERS

The server only responds with SERVER_SEND_ORDERS in response to a CLIENT_WANT_ORDERS communication. This message is never sent to more than one client at a time. When the server sends this, it is acknowledging a client's request to receive the list of current orders. The most current state of all the orders is central to the server, always. After sending the communication, the server will send the client the list of all orders in a *strict order*. Since the lists of orders are mapped to tables which are then mapped to order states, they must be sent and re-constructed in the same way. The process is simple:

*1. For each table, send the table number of the next table (0-8).*

*2. Then for each state (waiting, served, billed), send the next state we are going to send.*

*3. For each Order in this table of this specific order state, send each individual Order.*

*4. If this is the last order of this order state for this table, go to step 2.*

*5. If this is the last table, send a SENTINEL to stop communication, otherwise go to step 1.*

The client then receives the server's SERVER_SEND_ORDERS communication, and can handle the incoming data by reconstructing it in the same order that it is sent. The process is simple, too:

*1. Get the first table number the server is about to send (1-8).*

*2. Get the first order state the server is about to send (waiting, served, billed).*

*3. Receive an Order, store it in the current table in the list of the current order state. (unless 3x/xx)*

*3x. If the Object received is an Integer, change the table to that of the new table number.*

*3xx. If the Object received is an order state, change the order state to the order state sent.*

*4. Repeat 3, 3x, and 3xx while the Object received is not a SENTINEL communication.*

**c. SERVER_ADD_ORDER**

The server responds with SERVER_ADD_ORDER in response to a CLIENT_CREATE_ORDER communication. A client has created an order and has sent the data through, and now the server must take that data, create the Order itself, and distribute it back to the clients. After the CLIENT_CREATE_ORDER has been successfully handled and the Order has been created, a SERVER_ADD_ORDER is sent to each connected client.

A created order is always in the WAITING state, so all the server needs to do is send the communication first, followed by the table number the order is from, as well as the Order object itself. Yup, that's right. The Order implements *java.io.Serializable*, meaning that it can be sent as a whole Object through a TCP Socket without any fancy code. This also means that the Order object is only created on the server and processing power is saved (to a degree).

Each client receives the table number and Order, and adds it to its local Table models and updates the UI. Every client now has the same new Order on their running instance. As explained above, every state change that is mediated by a client is not actually effective until the server does the heavy lifting itself and then distributes the changes to all the clients with a communication.

**d. SERVER_UPDATE_ORDER**

This communication is different from SERVER_UPDATE_ORDER because an Object is not sent over a TCP connection. SERVER_UPDATE_ORDER is sent in response to a CLIENT_UPDATE_ORDER communication. A client has changed the state of an order and send the appropriate data through to the server, and now the server must change the central state of the order and distribute the change back to the clients. No heavy lifting is done here by the server, it is simply acknowledging a client's request to change an order's state to something else.

When sending this communication to all the clients, the server proceeds it with the order's old state, followed by the order's table number, followed by the position of the order in its linked list. This is all the information necessary to update an order's state on each client instance. Once the existing order's state has been updated on the client, the UI's tables are updated to reflect the change visually.

## Database for Persistent Data

The client-server software makes use of a MySQL database connection to make menu items and orders persistent when it is not running. It is able to independently create and maintain its own database schema, provided a connection to a MySQL server is supplied. Running the MySQL server is the responsibility of the end-user and the software will not attempt to start

a MySQL server. The process for creating the schema and manipulating data is as explained in different sections below.

**Part 1. Creating Database Schema**

The server will attempt to authenticate itself with the MySQL server specified in the 'server.settings' file which is generated after the server is run for the first time. These settings can be changed after the file is generated. If the connection to the MySQL server is unsuccessful, the program will terminate itself and will need to have its default settings adjusted. After connecting to the MySQL server, the server will create the schema if it doesn't exist, use the schema, and then create tables.

**Part 2. Creating Tables**

*Screenshot 6.1: Schema table view (after creation and population) in phpMyAdmin*

| Table ▲ | Action | | | | | | | Rows ⓘ | Type | Collation | Size | Overhead |
|---------|--------|--|--|--|--|--|--|--------|------|-----------|------|----------|
| ☐ menuItems | ☆ | Browse | Structure | Search | Insert | Empty | Drop | 39 | InnoDB | *latin1_swedish_ci* | 16 KiB | - |
| ☐ orders | ☆ | Browse | Structure | Search | Insert | Empty | Drop | 1 | InnoDB | *latin1_swedish_ci* | 48 KiB | - |
| 2 tables | Sum | | | | | | | 40 | *InnoDB* | *latin1_swedish_ci* | 64 KiB | 0 B |

The software will attempt to create two tables:

- menuItems
  - This table stores menu items loaded in from the .csv file, but also any custom user-added menu items that may not be in the .csv file.
  - During normal server startup, the server will go over the .csv file, adding any entries that exist in the .csv file, but not in the table.
- orders
  - This table stores customer orders of any state. If an order's state is updated on the server, it is also updated on here. New orders created are also put here.
  - During normal server startup, only orders that do not have a state of 'BILLED' will be loaded onto the server to conserve memory.

**Part 3. Loading Menu Items from Database**

The standalone version of the program will attempt to load in menu items from the included .csv resource file. The server program also attempts to load from the .csv resource file, but only after loading from the database. The server will read each row from the `menuItems` table and create a menu item object which is stored in the appropriate list.

After loading in all available menu items from the database, the server will also scan over the .csv file to load in any menu items that might not exist in the database yet. Any menu items from the .csv file that are found are written into the `menuItems` table on startup. This also means that in a fresh database with an empty or newly-created table, all menu items from the .csv are written into the table. The end-user does not have to manually populate the `menuItems` table.

**Part 4. Loading Orders from Database**

The standalone version of the program does not have persistence for data once it is closed. If the standalone program is closed, all orders are lost. However, the server program writes order and order state changes to a database whenever they happen, along with the programmatic handling of new orders or order state changes. It should be noted that clients do not ever create orders, nor do they attempt to update the states of orders unless the server has specifically sent a request.

When loading orders, the server will read each row from the `orders` table, specifically checking the order state first. If a row's order state is not set to 'BILLED', it will be loaded in programmatically and stored in the table models. When creating the Order models programmatically, the appropriate MenuItems will need to be included in their constructor arguments. As will be discussed in the next section, the `order` table's structure contains two foreign key columns: one for a customer's ordered food, and one for a customer's ordered beverage.

Since MenuItems have a uniquely identified price lookup (PLU) number assigned to them, they are used to reference back to the Menu Items that were programmatically loaded in before the orders were. The server searches for the existing menu items by PLU (instead of by food type and meal type) to accomplish this. Once an order is loaded in, it is placed into the correct table by its number.

**Part 5. Table Structures**

*Screenshot 6.2: phpMyAdmin table structure view for `menuItems` table.*

| | # | Name | Type | Collation | Attributes | Null | Default | Comments | Extra | Action | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ☐ | 1 | PLU 🔑 | smallint(5) | | UNSIGNED | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 2 | food_type | enum('FOOD', 'BEVERAGE') | latin1_swedish_ci | | Yes | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 3 | meal_type | enum('BREAKFAST', 'LUNCH', 'DINNER') | latin1_swedish_ci | | Yes | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 4 | item_name | varchar(255) | latin1_swedish_ci | | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 5 | price | double | | | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 6 | energy | double | | | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 7 | protein | double | | | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 8 | carbs | double | | | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 9 | fat | double | | | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 10 | fibre | double | | | No | *None* | | | ✎ Change | ⊖ Drop | ▼ More |

*Screenshot 6.3: phpMyAdmin table structure view for `orders` table.*

| # | Name | Type | Collation | Attributes | Null | Default | Comments | Extra | Action | | |
|---|------|------|-----------|------------|------|---------|----------|-------|--------|---|---|
| ☐ | 1 | order_id 🔑 | int(11) | | | No | None | | AUTO_INCREMENT | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 2 | table_number | tinyint(3) | | UNSIGNED | No | None | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 3 | order_status | enum('WAITING', 'SERVED', 'BILLED') | latin1_swedish_ci | | No | None | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 4 | customer_name | varchar(255) | latin1_swedish_ci | | No | None | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 5 | food | smallint(5) | | UNSIGNED | No | None | | | ✎ Change | ⊖ Drop | ▼ More |
| ☐ | 6 | beverage | smallint(5) | | UNSIGNED | No | None | | | ✎ Change | ⊖ Drop | ▼ More |

These tables are programmatically created, as per requirements. It is just nicer to see the structures instead of the actual SQL statements that create them.

**Part 6. Creating/Updating Orders in Database**

Creating orders in the database is straightforward, and happens as soon as the order is programmatically created. Since all of the data that is needed to store in a row already exists in the method , it is only a matter of extracting it and putting it in the SQL INSERT statement.

Updating orders in the database is relatively easy too, since each order is uniquely identified by a number, much like the menu items. This originally was not our design choice, as orders in the standalone version are separated out into different lists which are sorted by table number and order state. The order models themselves do not know which table number they belong to or which state they are in.

However to uniquely identify an order in the database an ID is required because multiple orders can originate from the same table number, be ordered by a customer with the same name, and contain the same food and/or beverage. This is only used for the SQL UPDATE statement so the correct row will always be updated. It is simply a matter of setting the new order status for the row with the specified order ID. To programmatically update an order status requires knowing the order's table number, current order state, and its position in that specific list. This is why it is easier.

## Loading Internal Resources

### Part 1. Configuration Settings

Both the client and server programs make use of a common Configuration system to load required values that cannot be stored within the program. You may have seen the '.settings' file that is generated when the client or server are run for the first time, and will likely not work without being correctly configured. Configuration is very straightforward: key = value.

The programs will first check if the specified .settings file (client.settings for client, server.settings for server) exists. If it doesn't exist, a default settings file is generated beside the program JAR that will be loaded instead. This default settings file exists as a resource inside the compressed JAR file, and is converted into an InputStream which can then be written to an external file.

After obtaining the .settings file, it is read line-by-line using a simple BufferedReader. Lines starting with # are ignored for comment purposes. Each line is expected to follow a simple

key=value format. For example, the server's port would be written as server_post=6969. Lines that cannot be split by an '=' character delimiter are rejected and are not read. Acceptable lines are passed as a key -> value pair into a HashMap<String, String>. The key can then be read programmatically to retrieve a value.

In the client, settings include the server's hostname as well as its port. In the server, settings include the outbound port that Sockets listen on, as well as MySQL server information. Configurations can also throw errors if required keys are not included in the .settings files. For example, if a client.settings file does not contain the server_port it will not know what port to connect to and will not be able to function correctly. We can tell Configuration that the 'server_port' key is required and if not found, will not allow the client to operate.

### Part 2. Reading .CSV Resource File

The process of reading a .CSV resource file is similar to the way that a default .settings resource file is written to an external file. Instead of converting it to a file, the server and standalone prototypes pass the InputStream of the resource file to a BufferedReader. This allows it to be read line-by-line. Reading a .CSV file is exceedingly easy, as individual rows are individual lines in the text file, with each column being separated by a ',' delimiter.

The content of each row is programmatically separated and passed into a MenuItem constructor, and the resulting MenuItem is kept in a specific Linked List. As explained in readme.txt, the combo boxes can retrieve linked lists of a specific food type (food, beverage) and specific meal type (breakfast, lunch, dinner). The lists can easily be iterated over to display each individual menu item in the combo boxes.

The .CSV file does not need to be taken out of the .JAR and stored separately to be read. This is only done with the .settings files because they need to be modified by the end user. The menu items do not need to be modified; the database can easily have new rows inserted in `menuItems` to accomplish the same task. The .CSV file is used to populate an empty `menuItems` table.

## Extension Features

### Extension 1. Group Order

The first extension is the functionality of adding a group order instead of just a singular order. A group order is an order for two to eight customers whose items are all added, served and billed together. As seen below in screenshot 7.1.1, the full functionality of this extension integrates with the individual order layout, letting users have full functionality to both order types.

This extension is for the standalone version only.

*Screenshot 7.1.1: Full display of group order functionality*



**Order type selection**

At a base state, with no options selected, the GUI has been altered to give the option to select inputting either a single or a group order as seen in screenshot 7.1.2. Clicking on either of these options will load the corresponding details panel for the user to fill in. When a order is successfully entered or the display is cleared, this options will once again be presented to the user to allow for a new group/single order to be created at any time.

*Screenshot 7.1.2: Order type selection buttons*
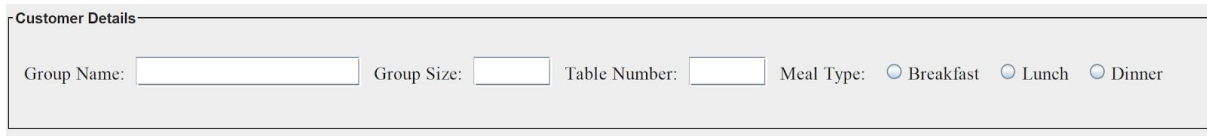


**Group order details**

Once the user clicks the 'group' button in screenshot 7.1.2, the details panel for a group order is loaded which can be seen below in screenshot 7.1.3. This panel is slightly altered from the single order detail panel, as it now includes an input areas for 'group size' and instead asks for the group

name. All customers in the group will have the same selection depending upon which meal type is selected.

*Screenshot 7.1.3: Details input areas for a group order*



**Ordering multiple items**

After a meal type has been selected, the dropdown lists in screenshot 7.1.4 will be enabled and filled with the corresponding meal items, working the same as with the single order. However, as seen below, three new options have been added for the group order functionality.

*Screenshot 7.1.4: Added buttons to allow for multiple item ordering*



The 'add' button adds the users selected menu items to a temporary collection if the chooses were valid. If the user attempts to add more items than the group size, an error will be displayed warning the user which can be seen in screenshot 7.1.5.

*Screenshot 7.1.5: Error received when ordering more items than the group size*
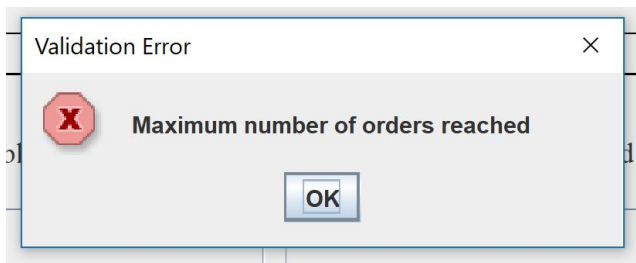


The 'view' button creates a popup tab which displays the current group order that has yet to been prepared. This can be seen below in screenshot 7.1.6.

*Screenshot 7.1.6: View display of current group order*
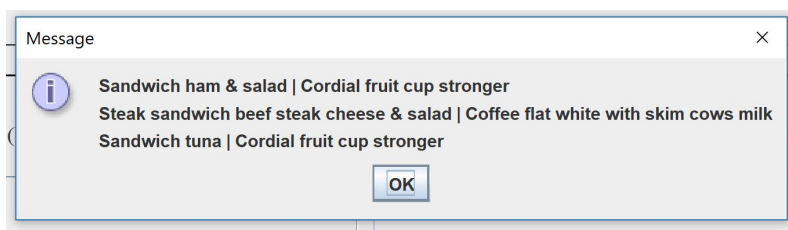


The 'reset' button removes all items currently stored in the temporary collection for the group order.

**Managing group orders**

Once the user is happy with their order details and item selection and have entered the order, it is displayed together in the order status panel which can be seen in screenshot 7.1.7. Instead of the individual row that singular order holds in this table, a group row has been widened to display all items with the group's details.

*Screenshot 7.1.7: Group order held together in a single row*



This has the same functionality as a singular order, meaning it can be manipulated and selected in the same manner. This can be seen in screenshot 7.1.8, where the user selected the group order and then clicked the 'prepare' button which moves the order row to the served state table.

*Screenshot 7.1.8: Group order being processed together*



**Viewing group orders**

A group order also also the same functionality of displaying in the ordered items table when the 'display order' button is clicked. The format has been altered, which can be seen in screenshot 7.1.9, so that all the group order's items are displayed together.

*Screenshot 7.1.9: Display of group order in a specified table*



**Extension 2. Automatic Reconnection**

When the server process is killed or drops out, the clients (by default) are left stranded without any possibility of input and more importantly, output. This is because changes/creations are sent to the server, where they are distributed back to the clients. This next extension feature aims to make the process of reconnection to the server seamless for the clients, as the only option is to close the client and re-open it after connection is lost.

The process/features are explained in the following paragraphs on the next page.

## Part 1. Detecting Connection Loss

The first feature of the automatic reconnection is to take UI control away from the client while there is no connection to a server. This is done by disabling all text boxes, combo boxes, and buttons. The user is also notified by means of a non-blocking dialog.

To programmatically detect connection loss, the Client Controller has a method that checks a ClientConnection model to the current one stored. The connection is considered "valid" if the connection stored is equal to the one given. To invalidate a connection when it is lost, SocketExceptions are caught and the writeToServer() returns false, meaning that the connection has been lost.

When writeToServer() returns false, the stored connection is set to 'null' and the reconnection process begins. This starts with disabling the UI and showing a dialog, as explained above. Once a connection is re-established, the UI is automatically re-enabled.

### Part 2. Killing Worker Threads

Threads will continue to operate even if connection is lost. To combat this, instead of looping infinitely, the threads will only continue to loop while the ClientConnection model they reference is still valid. This ensures that these threads end when they are no longer needed.

### Part 2. Reconnection

The client is completely useless while not connected to a server as it cannot manipulate any data on its own. The process to reconnect the client to the server as soon as possible is started immediately. The client will first attempt to immediately reconnect to the server in the same manner as when the client starts up. If it is successful, the client writes a CLIENT_WANT_ORDERS communication to the server. This is because when the client loses connection, all stored orders are cleared. Writing this communication makes the server send back all the orders as they are persistent because of the database.

If the client is unable to re-establish a connection, the SocketException/IOException is caught. This is because the only exceptions thrown are when the client is unable to open a socket to the server. When this is the case, the same method is re-called inside a thread which then sleeps for 0.5 seconds before trying again. This loop continues until the connection has been re-established.

When the connection is re-established, all the UI elements are re-enabled for input automatically, and there is no dialog. This feature is exceedingly useful for unexpected

disconnections as the clients no longer have to do anything. This also allows the server to be updated with minimal disruption. A list of screenshots will be provided illustrating the process.
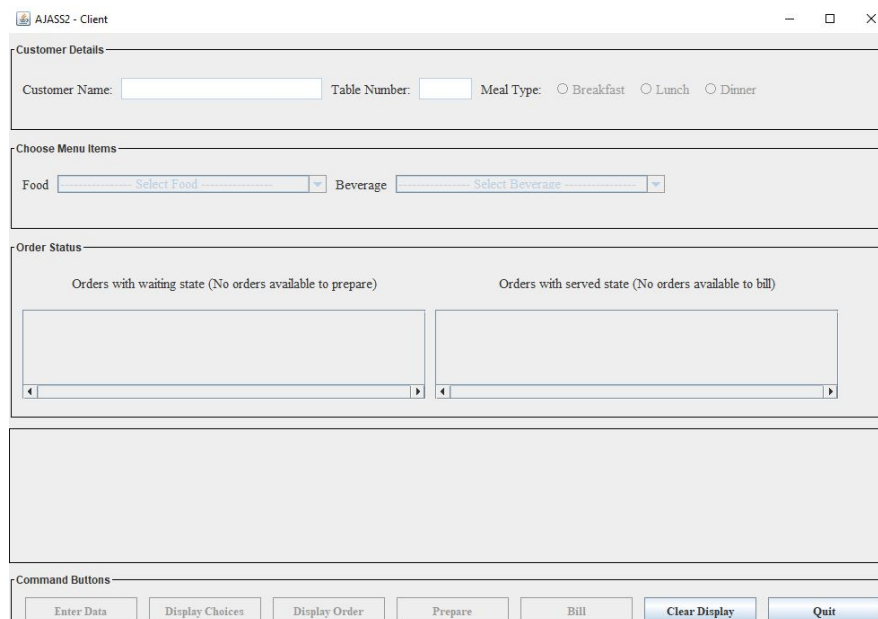
*Screenshot 7.2.1: Stopping the server. Ctrl+C works too.*



*Screenshot 7.2.2: Dialog that is shown when the connection is lost to the server on a client.*
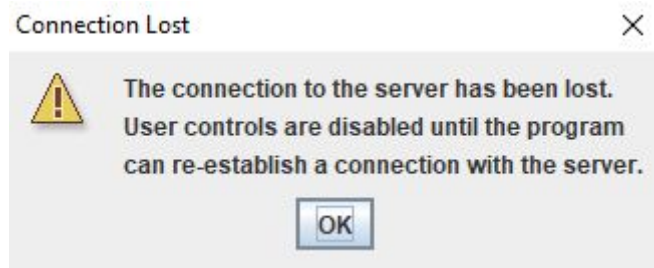


*Screenshot 7.2.3: Screenshot of the client showing all elements disabled and orders missing*



*Screenshot 7.2.4: Screenshot of the client after connection to the server is re-established*

## Model-View-Controller

### Project Structure

To ensure that this project accurately follows the principles of MVC, we separated our files out in their respective MVC folder to ensure that they did not overlap. 'Main' and 'Utilities' are outside these folders as they do not fit any MVC category. Main only initialises the application and utilities only defines functions which other classes can call upon, meaning they are both outside the MVC. The folder 'enums' is part of the model, however it was seperated to improve clarity.

*Screenshot 8.1: Project file layout*
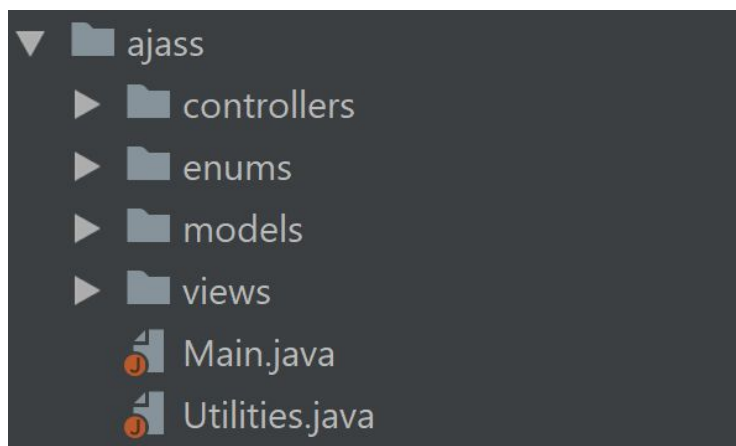


### Model

Model holds all our collection classes and enumerations. These classes are within the model classification as their only purpose is to the data given to them from construction, as well as providing the functionality of getters and setters to alter and retrieve the data. The enumeration files are also within the model as they publicly define an enumerated type which can be accessed by any other class.

**View**

View holds all our classes which are used to display elements to the user. All view classes implement the 'IView' interface which defines generic methods that must all define. Our only JFrame class is 'MainView' which defines its controller as well as initialising the GUI. The other classes in the view are all JPanels, which display their specific content along with their functions needed to alter the display and the users interactions.

**Controller**

Controller holds the one controller necessary for the entire application, which alters the various model classes data depending upon the users interaction with the application. Our controller is responsible for creating orders once the user has filled in the details, altering the order's states, retrieving information from tables and generating receipts.

## Addressing Multiple Beverages

Our group neglected to recognise the "multiple beverages can be selected" requirement until it was too late to change it. There is simply not enough time left at the time of writing this report to add it. This section just serves the quick purpose of showing how it can be implemented. Obviously we cannot store multiple foreign key values in one column in the `orders` table, so another table would have to be created.

This table would have the properties of a weak entity, as it is a table with columns comprised of only foreign keys (which also make up the primary key). This is needed to link zero to many beverages to one order. The Order model would also need to be changed to accept an array of MenuItem for beverages, instead of one or no MenuItem for a beverage. The controllers in all 3 separate programs would need to be updated to accommodate the change.

This will unfortunately take too much time, and will also unfortunately eat up some of our marks for not fully implementing it in the standalone. This also affects our marks in the client-server prototype since it is based off the requirements in the standalone. :(