Software Technologies for Data Science
Lab - 8 Report

Matthew Clarke-Williams
Id : 189469874

December 14, 2018

Abstract

This report is supposed to be a supplement to the back end of the tills Python code. It contains reasoning and notes along with screenshots of the till server to assist in the ease of understanding and adaptation of the code in the future if desired. The code itself is fairly well documented so this report serves as more of a way to get into the mindset of what the server does and perhaps what could be adapted in the future.

1 The Database

The database is what lies at the heart of the backend-server. It enables the storage and manipulation of all the requests sent by the front end till system. It needed to be well thought out so the till runs smoothly and efficiently with scalability present wherever possible.

First lets talk about the products table. The text file `products.txt` contains a variety of SQL statements including the creation of the products table. I made a few modifications to the file. This was to change the `classid` for the snacks/cookies etc to be unique since they shared the same one. I also added two *products* to the product table which were deals you could get (the meal deal and the half price cookie deal).
The `product` table has :

- `productid` : A unique ID for each product
- `pos` : The position in the till (front end)
- `name` : The name of the item
- `classid` : The class that the product is in (hot drinks will be different from cold drinks)
- `shift` : an indication of small medium or large size
- `price` : The item's price
- `plu` : A unique code for each item (like a barcode that a cashier could type in)

I needed to make a `transactions` table where each transaction would be stored with the total for that transaction, the amount paid and the change given would be recorded. It's structure is as follows:

- `transactionId` : A unique number for each transaction
- `totCost` : The total for that transaction
- `amountGiven` : The amount paid by the customer
- `change` : The change required to give the customer

The `transactionId` was a primary key which is auto incremented. This ensured that no two transactions would be assigned the same transaction ID.

Next I needed a `sale` table. Here would be where I recorded every individual sale in each transaction. The format of this table is:

- `saleId` : A unique integer number for each sale item

•`transactionId` : The transaction ID of the current transaction
•`prodId` : The product ID of the product selected
•`price` : The price of the product ordered
•`quantity` : The quantity of the product ordered
•`refund` : An indicator to say whether this sale was a refund or not (refund if = 4)
•`mealDeal` : An indicator to say whether this sale item was a meal deal (=1) or not
•`halfCookie` : An indicator to say whether this sale item was a half price cookie deal (=1) or not
Where `prodId` is a foreign key from the `products` table and `transactionId` is a foreign key from the `transactions` table.

Creating an additional table called `cur_transaction` made operations easier when dealing with refunds since it was a grouped by version of the `sale` table. Here, the products were grouped by their product ID and the quantities and totals were summations of the respective fields for that product. Therefore, an item that was bought and refunded would appear twice in the `sale` table but only once in the `cur_transaction` table but with quantity and price equal to 0. It had the following columns:
•`prodId` : The ID of the product ordered
•`price` : The price of the product ordered
•`quantity` : The quantity of the product ordered
Where `prodId` is a foreign key from the `products` table.

The creation of all of these tables along with the creation of the database can be found in the file `DatabaseV1.py`. This only needs to be ran once (or every time you wish to permanently delete and remake the tables).

2 The Backend

2.1 Sections 1, 2 & 3 - The Set Up

The backend is where the adaptation, manipulation and send requests happen. The first thing that happens is the connection to the database and creation of a cursor. Next in section 2, there are four html actions which allow the modification of what can be seen on the till interface. Some encoding/decoding needed to be done to convert this from the Python 2 file to Python 3. We needed a way to identify if we were in a transaction or not and what our transaction number is. So section 3 addresses this issue by creating a variable called `in_transaction` that is set to False initially and the transaction number set to infinity just so it would throw an error if it didn't get changed. We also needed a variable called `title` that would change the display text of the section at the top of the till. This would change when you click on sandwiches or drinks.

2.2 Section 4 - The functions

In section 4, we make a variety of functions that will be called when the backend receives a get request from the front end. The first function, `check_make_transaction` takes in two arguments, `in_transaction` and `tran_numb`. The purpose of the function is to check whether the system is currently in a transaction (*i.e.* `in_transaction` *will be true*) and if not, start a new transaction. The transaction number (`tran_numb`) will then be returned along with the updated version of `in_transaction`.

`act_plu` is also another function defined in this section. will take in two arguments - `dic` and `additSubtext`. Before progressing, I will explain what `dic` is. Later on in the code, I break down the requests sent by the server and sort them into a dictionary. Each request looks something like `action=program&refund=0&num=1&pos=58&shift=0&value=&randn=41358`. This tells us what kind of request is given. In the backend, I make a dictionary called `dictionary` which

contains the keys of everything before the '=' with values being the matching value after '='. So in the case above we would have:

dictionary = {action : program
refund : 0
num : 1
pos : 58
shift : 0
value :
randn : 41358}

So coming back to the function `act_plu`; the argument `dic` is this dictionary described above. `additSubtext` is short for *additional subtext* and is an extra string displayed in the target display with the words *"refund"* if the item was a refund. So when the `action` in the request is equal to *'plu'*, the function `act_plu` is called. It matches the plu number typed in by the cashier if the plu code is valid, otherwise, displays an error - `This is not a valid PLU code`. It uses an SQL query to match the PLU codes to those known in the `products` table and if there are any results (the length isn't 0), it will add the item to the sale table.

Next is `act_cash`. This is called when the cashier wishes to close the transaction (accept payment). It takes in the variables `dic` and `subtext` (what will be written in to display on the screen). It will calculate the total of the sale (with an SQL select query with a group by), and work out any change that needs to be given if the payment is over. There is no option included for when less cash is given than the total as this was not mentioned in the spec. Also this would be where humans would only accept a cash payment that is at least equal to the total. If payment splitting was a possibility, this is where the code would need to be adapted to account for this. Upon receiving payment, calculating the total and stating the change, the transaction table is updated with these values and the transaction is closed, setting `in_transaction` is to False again.

The next function made is `act_prog` which is called when the action is program. This is when the cashier presses one of the buttons for an item on the till interface. e.g. pressing *Apple Juice* will send an `action=program` request to the server. The function is very similar to the `act_plu` function and takes in the same arguments. The only difficulty is that the *shift* and *pos* arguments in the request both need to be matched according to the spec. By default, any button pressed will have a shift of 0. This is changed by pressing one of the size buttons (e.g. a large latte). So the function starts by breaking down the dictionary (request) to find the quantity, shift, pos and whether the item is a refund or not. Then it attempts an SQL select query. If the item with these shift and pos values exists, the item and quantity will be inserted into the `sale` table. If the item does not exist, it will set the additional subtext to be `'This item wasn't in the database'`. And this will appear at the top of the target display.

Next up is the `act_clr` function. This request happens when the cashier wants to void the transaction. It takes in the same two parameters as before and first checks to see if the system is currently even in a transaction. If it determines that there is no current transaction (`in_transaction == False`), then it returns the additional subtext saying *There is no transaction active*. However, if the system is in a transaction, the function deletes all entries from both the `sale` and `transactions` tables and returns the additional subtext *The transaction has been void*. These additional subtexts will be displayed in the target display.

The last function I made was a function that dealt with requests where the action was *status*. This occurred when the page changes to either sandwiches or drinks. It takes in three arguments, the dictionary, a title and the subtext. Depending on which page was clicked changes

the title to "Drinks and Snacks" or "Sandwiches". Originally, the variable `target_display` was used to show this but now isn't needed so can be commented out.

2.3 Section 5, 6, 7 - Do Get

These next sections are where the the requests get dealt with. We now have all the tools needed to deal with the requests, just a few database queries will need to be done. Section 6 fills in the current transaction table. As it stands, the `sale` table can be quite messy when it comes to refunds. It's hard to keep track of how many items are actually being purchased so to get around this problem, all of the entries from sale with the current transaction number were inserted into the `cur_transaction` table grouped by the product ID. The quantities and prices were summations and so items that had been added and refunded would have a 0 in the quantity and prices fields.

Section 7 is where all the deals are calculated and applied. It starts by checking whether the last request had an `action=program` or `action=plu` request since we only need to recalculate a deal when there has been a change to the items in the transaction. Since this could change which deals we need to apply, all previously calculated deals will be deleted if there are any. This is done with simple SQL select and delete queries.

Next we needed to think about which deals should be applied first. I thought that this order needed to be from largest discount to smallest since it usually works that way in most shops. The meal deals always gave a larger discount in general so they were the first priority. I needed to calculate an upper bound on the number of meal deals that could be given with the transaction. This was easy to calculate as the number of meal deals only depended on three things - the number of drinks, the number of sandwiches/paninis and the number of snacks. The upper bound had to be the minimum of these three. Each product had an assigned *class ID*. This was a certain number for hot drinks, a different one for cold, another for sandwiches and so on. I changed the original `products.txt` file to have more classes when it came to snacks as previously, these weren't distinguishable through class ID alone. Then some SQL select queries were used to find the number of non-refunded items in all these categories and these values were used in the calculation of the upper bound.

Now we needed to calculate the discounts given in the meal deals. I made an effort to make the deals section as scalable as possible and so class ID's were used meaning that if another product got added to the database, it can be given the right class ID so that it can be included in the deals. The prices for the free snack can be different and my code accounts for this. The meal deals should give the maximum discounts where possible so I made a few variables - `snack_prices` and `pricesMD`. The first was a list of tuples of prices and quantities like so

```
snack_prices = [(65, 4), (85, 5)]
```

And the latter is an ordered list of these prices like so

```
pricesMD = [85, 85, 85, 85, 85, 65, 65, 65, 65]
```

`pricesMD` was then capped to be the length of the number of meal deals we can have. Then each discount was added to the `sale` table.

Next the cookie deals had to be calculated. This needed a similar approach as before with an upper bound being calculated. Similarly to before, this part is scalable so if the cookie prices were different, the largest discounts would be given first. The amount of cookie deals couldn't be more than these three things:

- The number of hot drinks in the order

•The number of cookies in the order

•The number of REMAINING hot drinks after we MAY have used some in the meal deals

Point number 3 needs a little more explaining. If we had an order of:

10 hot drinks
4 cold drinks
6 paninis
5 packets of crisps
12 cookies

We would use all of the cold drinks and 1 hot drink to get 5 meal deals. This would leave us with 9 hot drinks and 12 cookies so we would get 9 cookie deals.

This 9 is obtained by:

$$\begin{aligned}\text{Amount of cookie deals} &= (\text{number of drinks ordered}) - (\text{the number of meal deals given}) \\ &= (10+4) - (5) \\ &= 9\end{aligned}$$

So the upper bound for the number of cookie deals is simply:

$$min(\textit{number of hot drinks},\quad \textit{number of cookies},\quad (\textit{number of drinks} - \textit{number of meal deals}))$$

This upper bound is then calculated and exactly like before, the prices of all the cookies are ordered so we give the largest discount possible. The discounts are then inserted into the `sale` table and we move onto the very last bit of our code.

2.4 Section 8 - The Display

This last section runs a couple of SQL queries that select all the items from our current transaction in the sale table. It then lists these items in a string which is passed onto the function `build_action_refill` which displays this in the target display on the till interface. The total is also caculated and it is also displayed here. I had some issues with the XML and the total which for some reason meant that the total wasn't showing the most updated version. If I added items to the transaction, the total wouldn't update unless I pressed the *clr* button on the till. I don't know why this was and if it was due to the javascript coding so I displayed the total in the target display as well. It can be seen that the total is updating correctly every time so I'm unsure why the XML version isn't as smooth. Maybe you're meant to press clr each time?

3 Screenshots

Here are a few scenarios that put the till server to the test...

3.1 Transaction 1

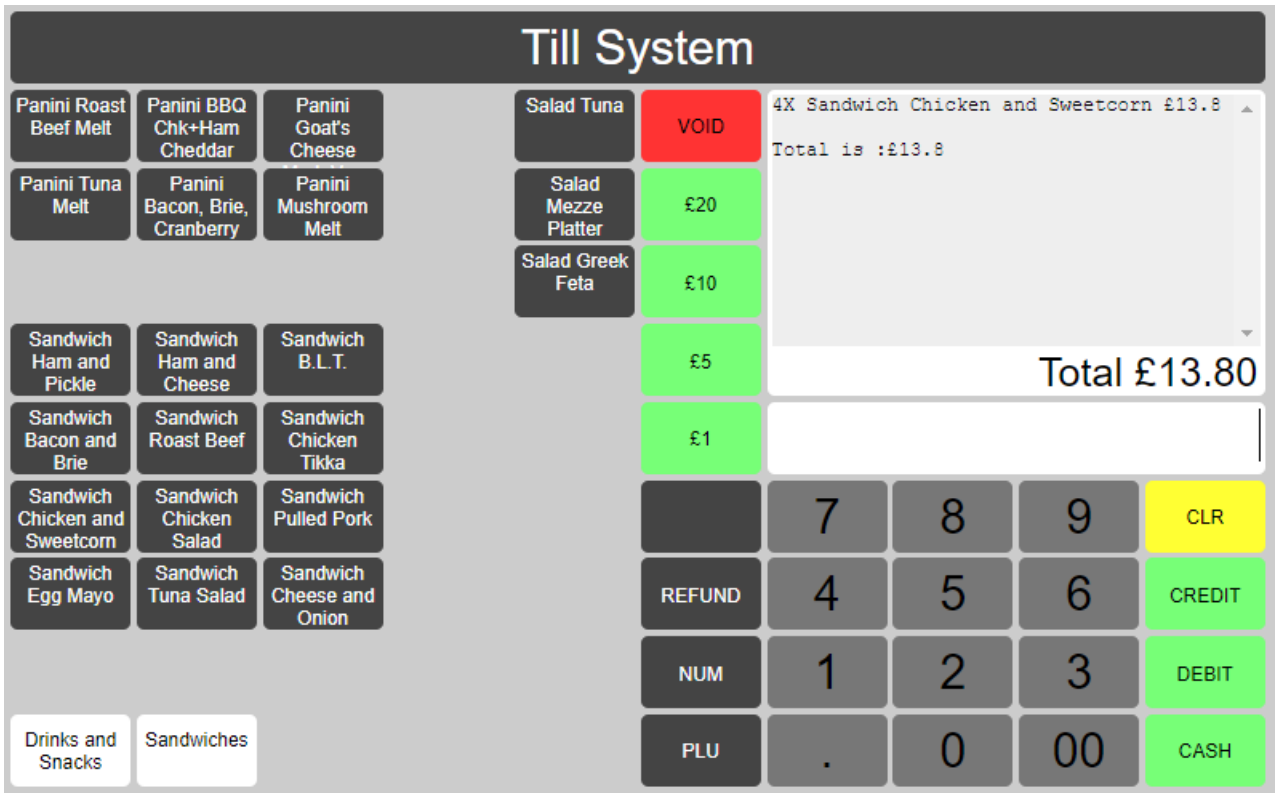

Figure 1: 4 sandwiches are added

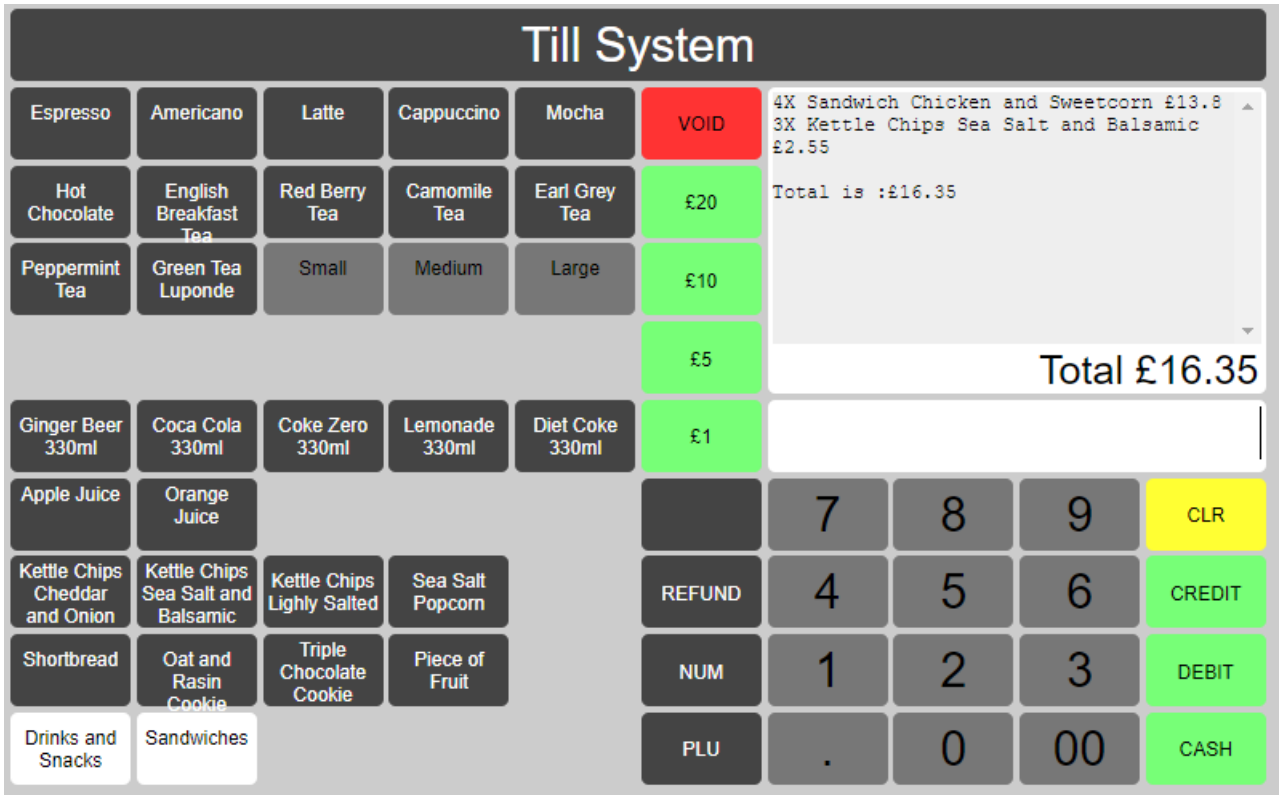

Figure 2: 3 packets of crisps are added

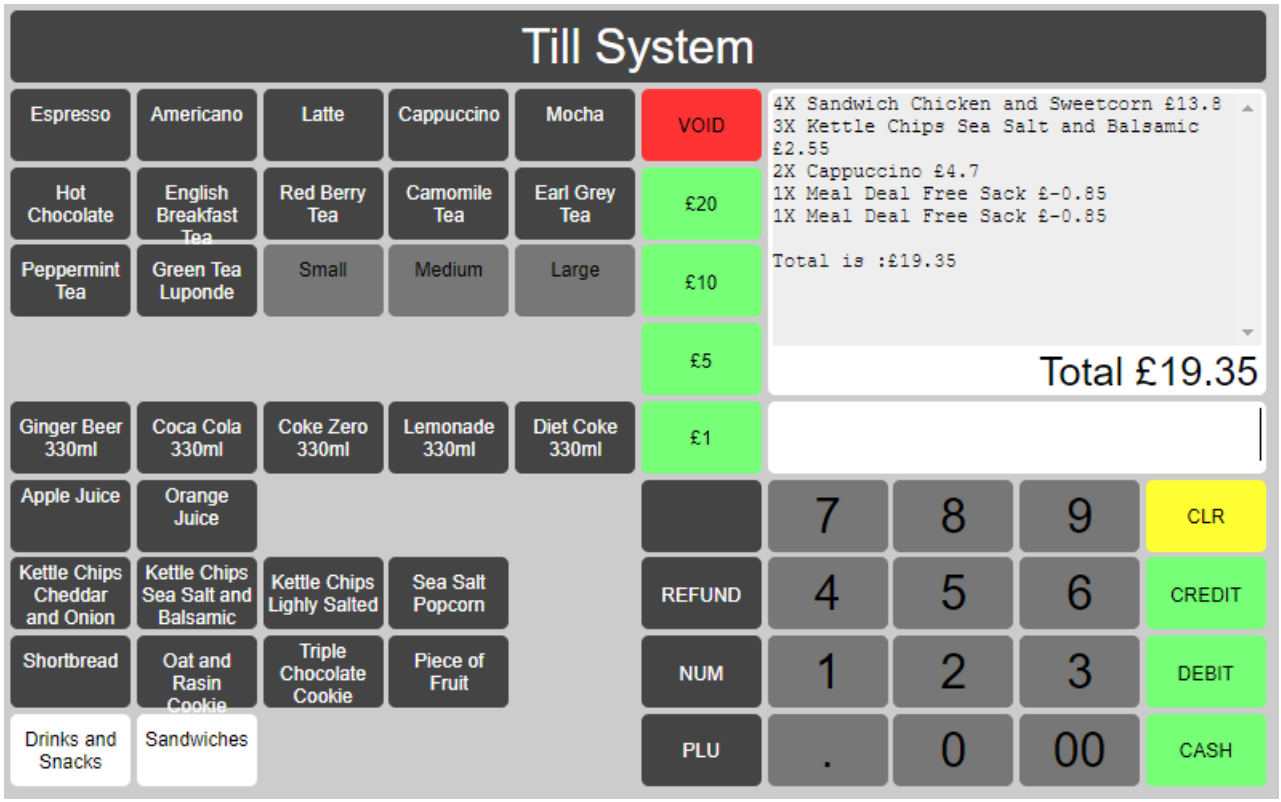

Figure 3: 2 Large cappuccinos are added and we now qualify for 2 meal deals

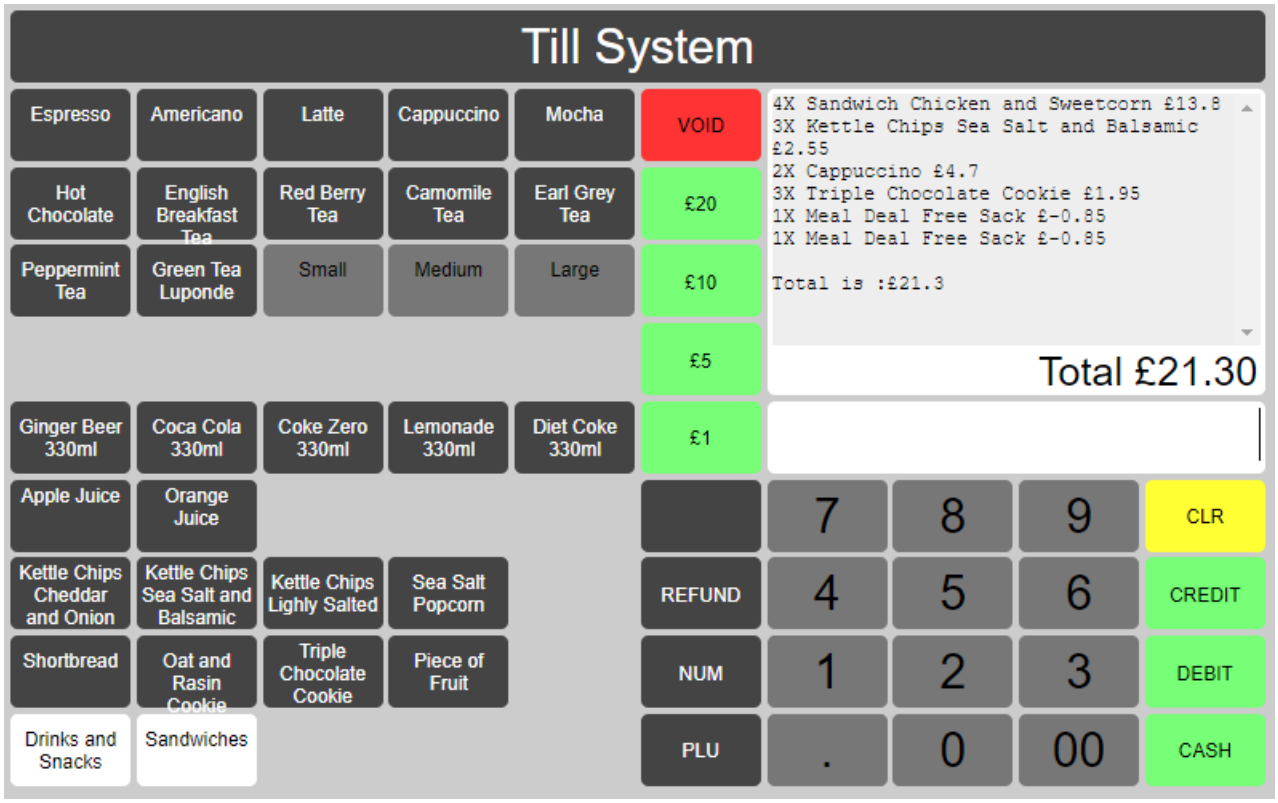

Figure 4: 3 cookies are added but no cookie deals since the meal deals are giving a better discount

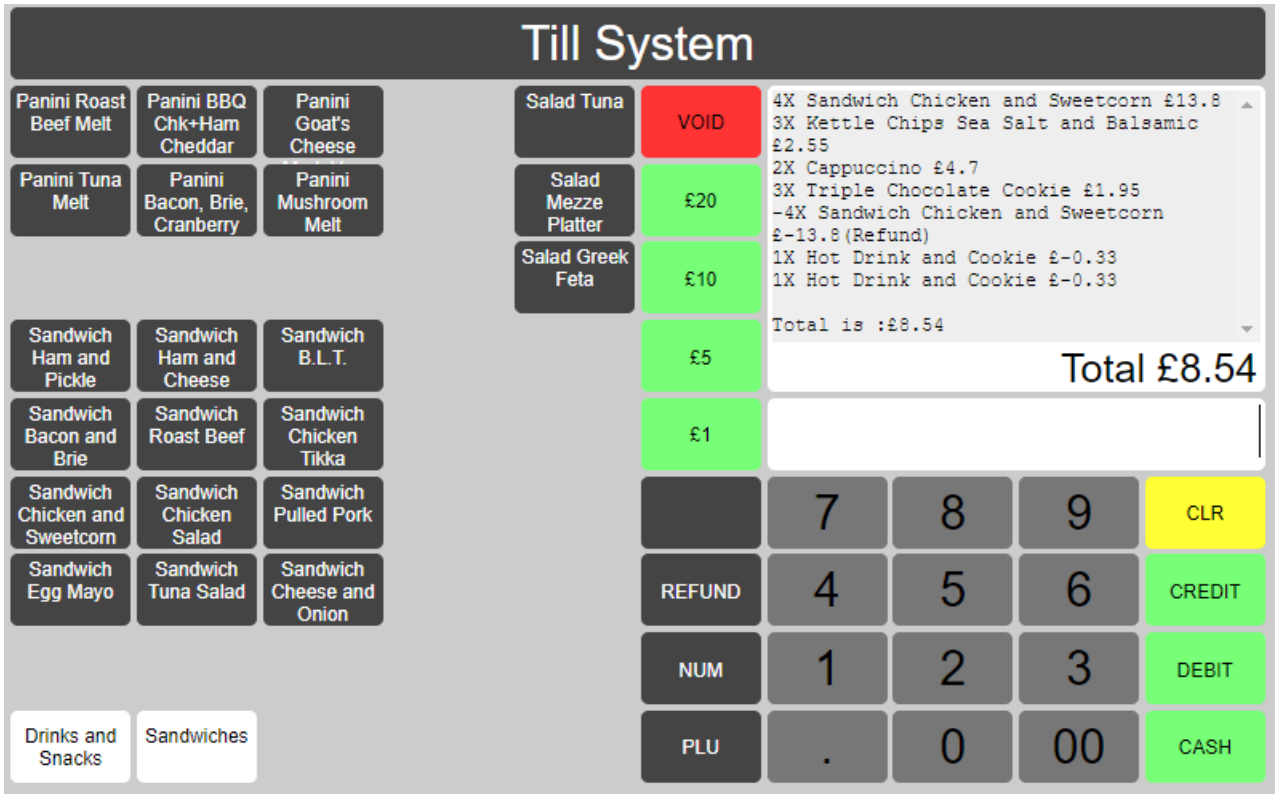

Figure 5: We refund the 4 sandwiches and we no longer qualify for any meal deals. We still can get the cookie deals though so these are added instead.

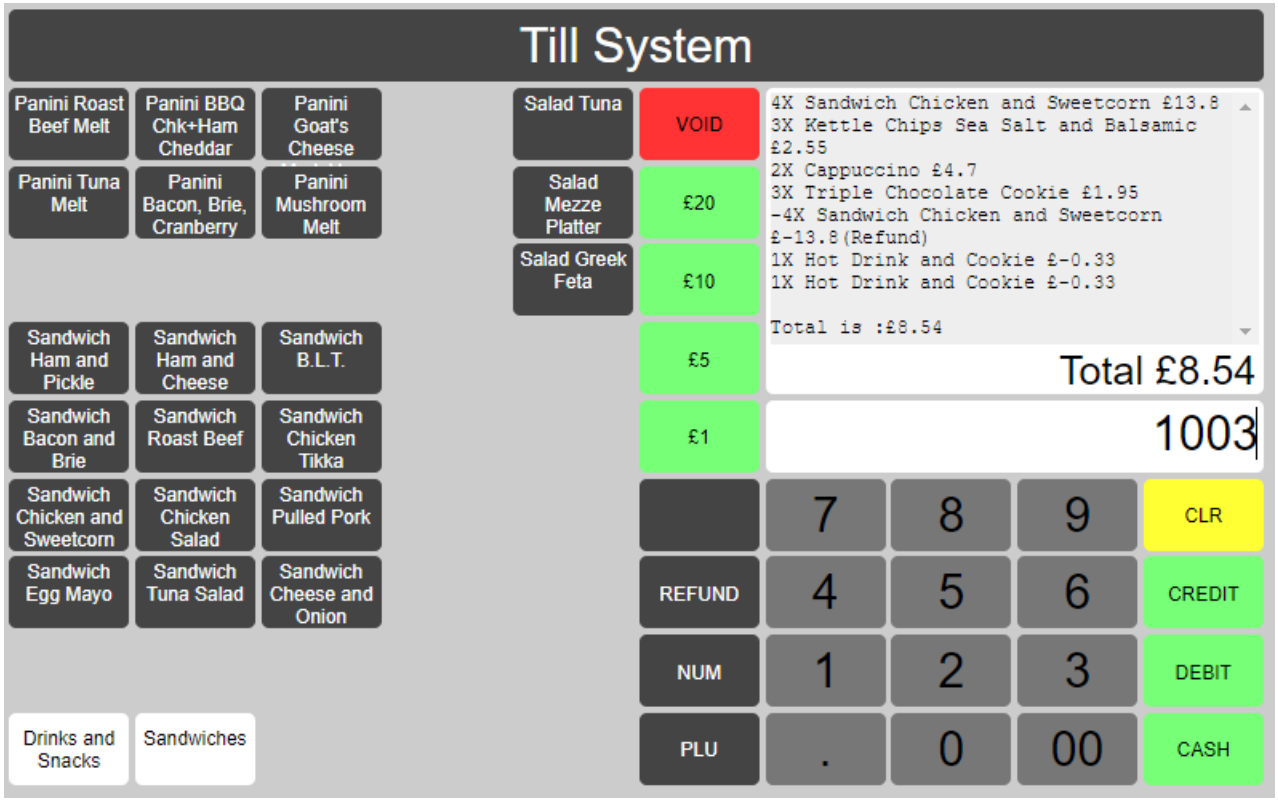

Figure 6: About to add in the item with PLU code = 1003

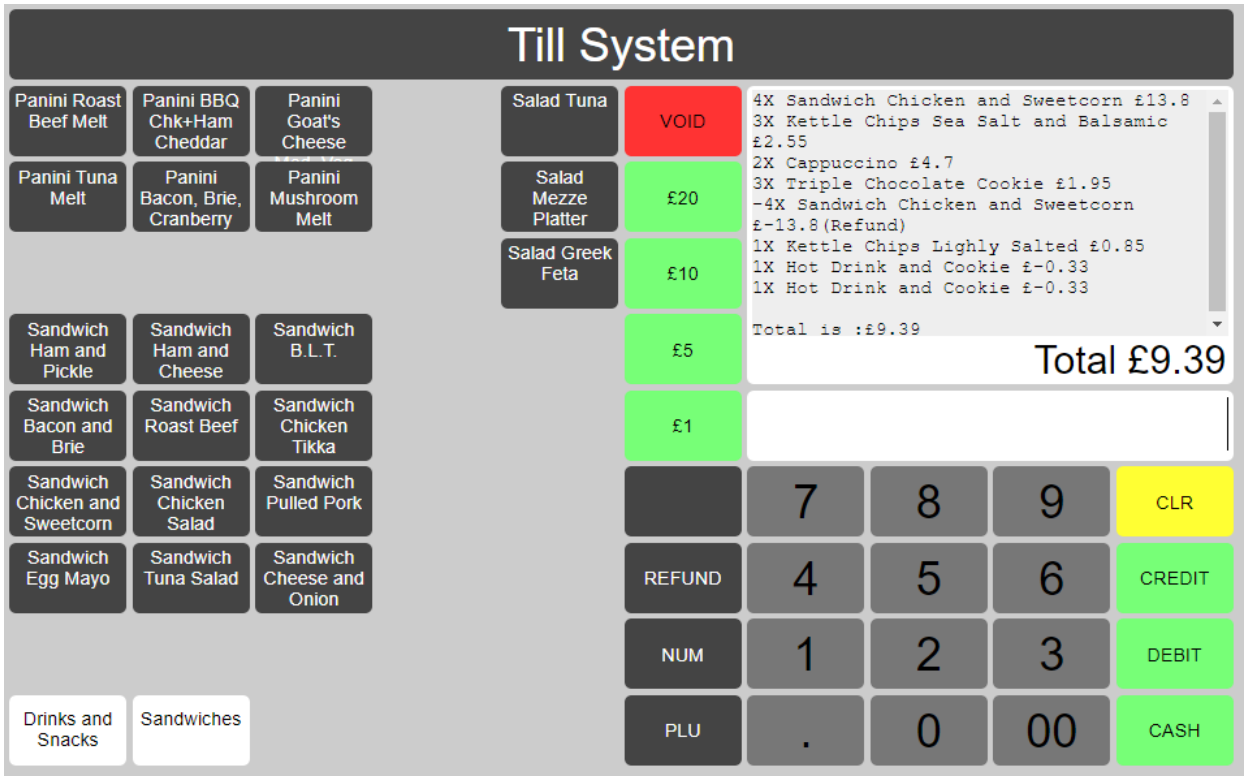

Figure 7: The item is added with the PLU code (kettle chips lightly salted)

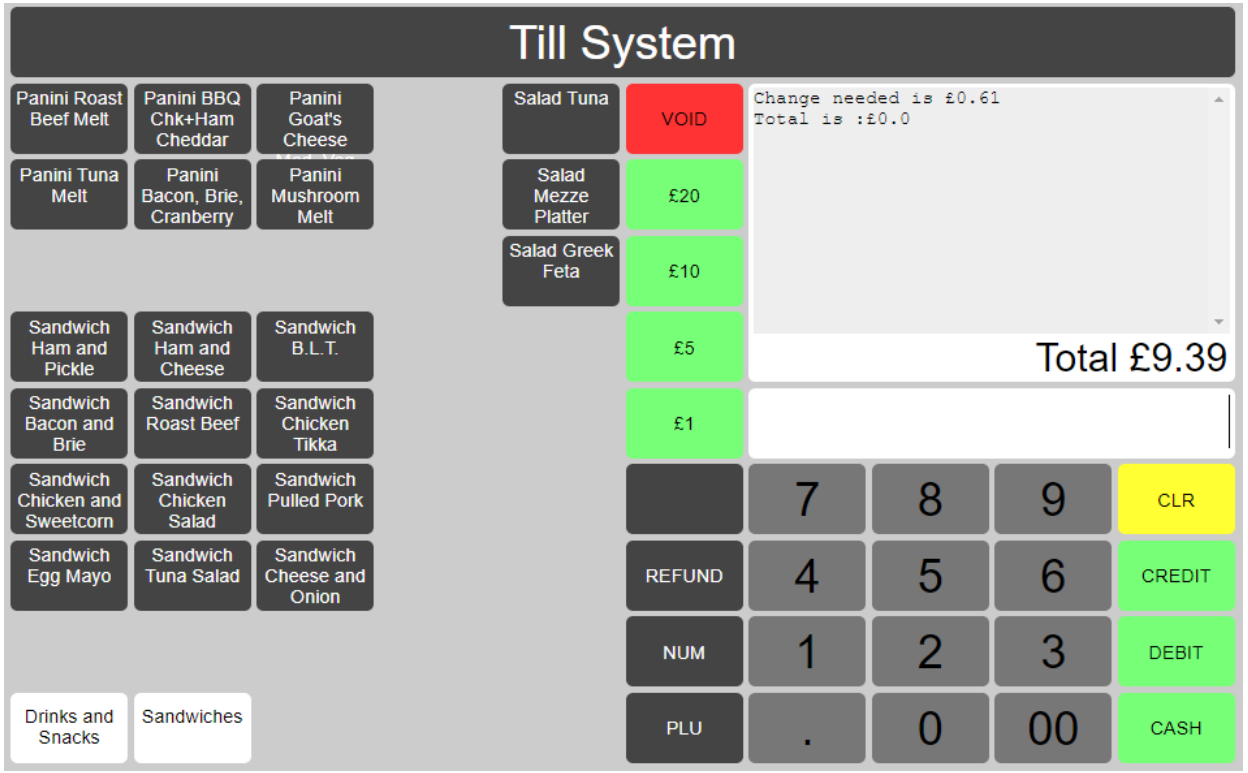

Figure 8: £10 was paid and the amount of change needed is stated

3.2 Transaction 2

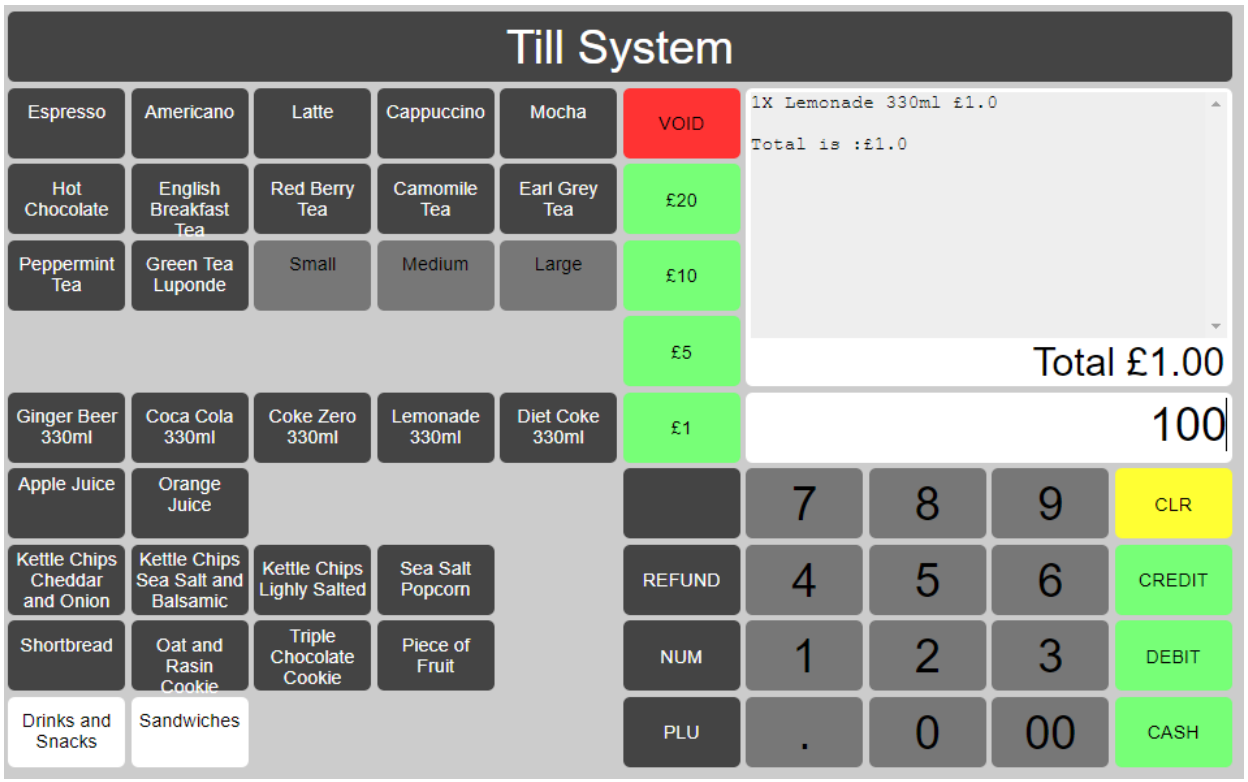

Figure 9: A lemonade gets added to the order

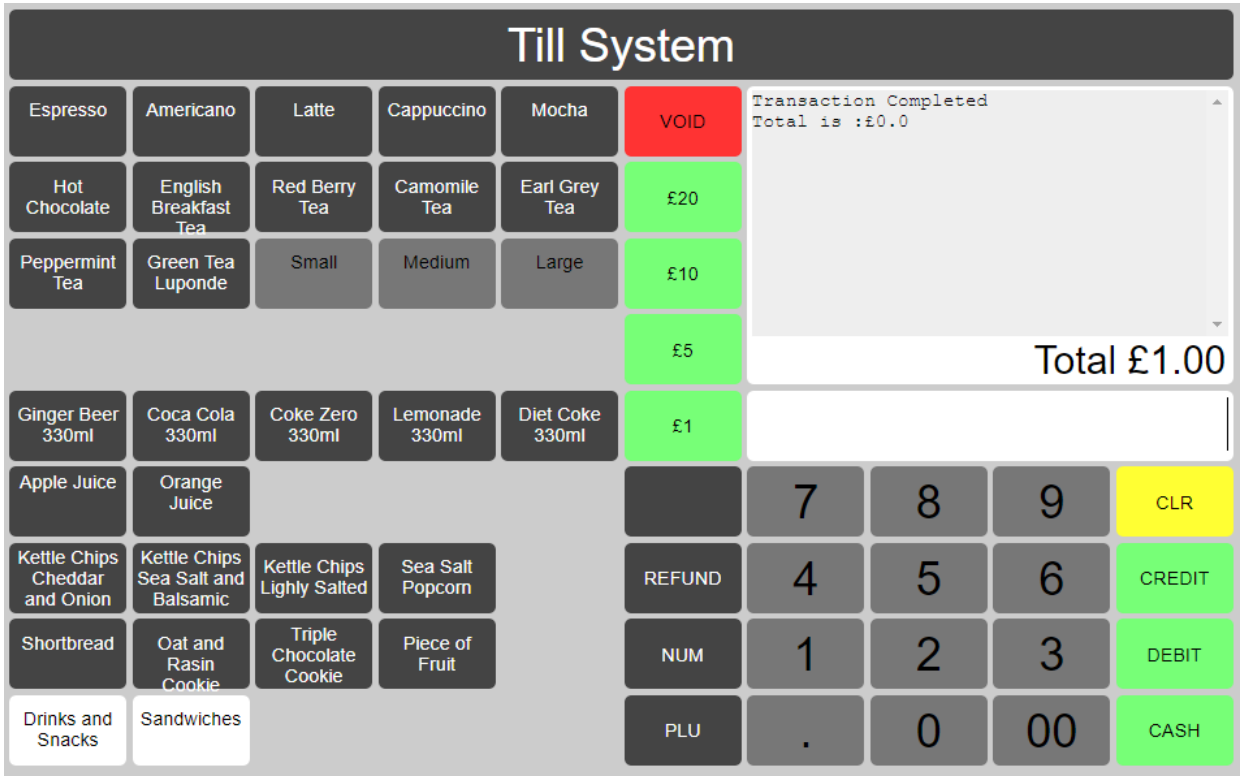

Figure 10: The exact amount of money is given and the transaction is closed

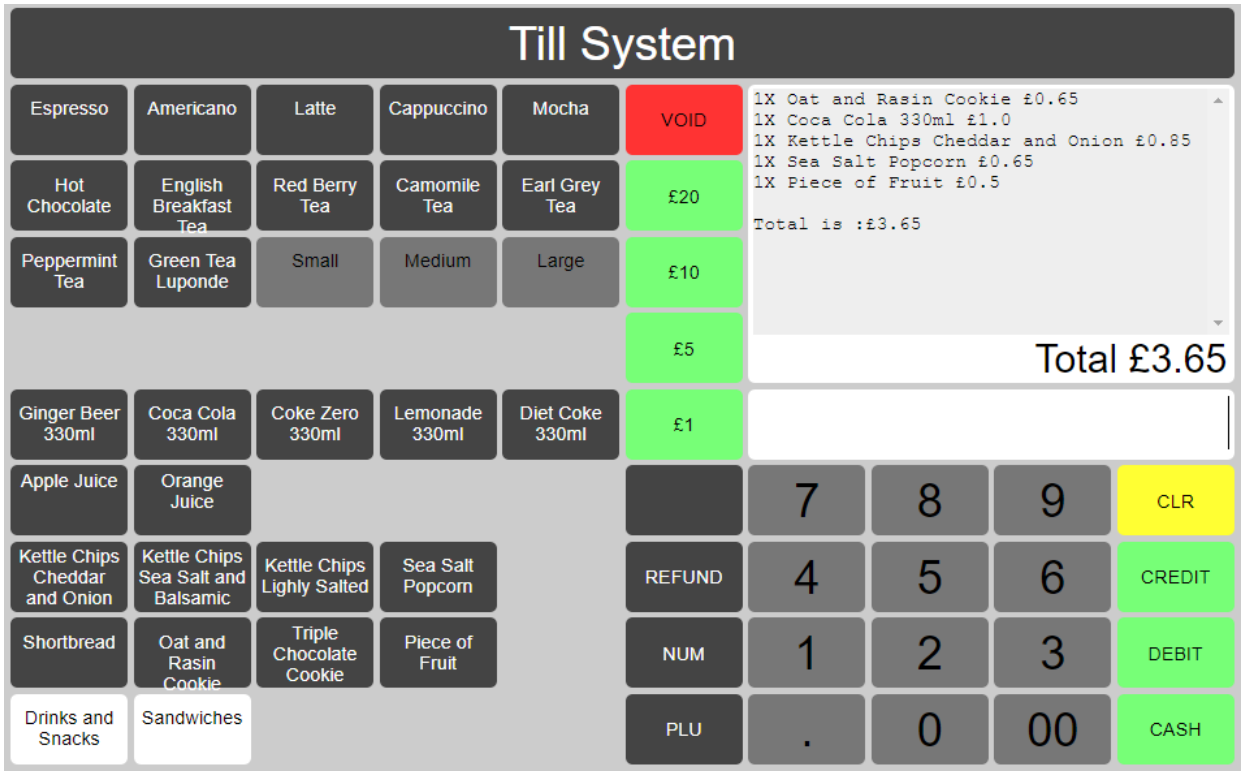

Figure 11: A variety of items are added to the sale

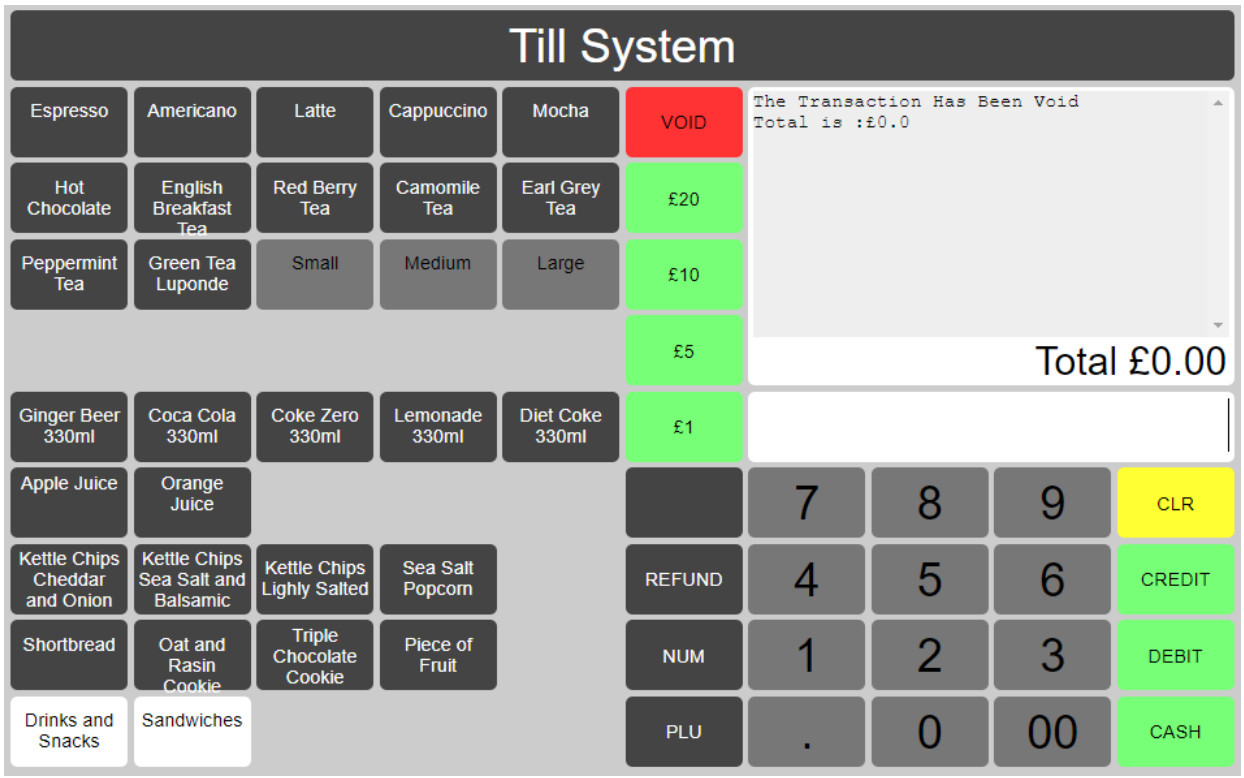

Figure 12: The void button is pressed and the transaction is void

3.3 The Database

	transactionId	totCost	amountGiven	change
	Filter	Filter	Filter	Filter
1	1	939.0	1000	61
2	2	100.0	100	0

Figure 13: The transaction table showing the two transactions (the third one was void)

	saleId	transactionId	prodId	price	quantity	refund	mealDeal	halfCookie
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	1	59	345	4	0	NULL	NULL
2	2	1	71	85	3	0	NULL	NULL
3	3	1	28	235	2	0	NULL	NULL
4	6	1	68	65	3	0	NULL	NULL
5	9	1	59	-345	-4	4	NULL	NULL
6	12	1	72	85	1	0	NULL	NULL
7	13	1	5002	-33	1	0	NULL	1
8	14	1	5002	-33	1	0	NULL	1
9	15	2	40	100	1	0	NULL	NULL

Figure 14: The sale table showing all the items sold

4 Task 2.2

4.1 Part 1

For task 2.2, I first wrote an SQL query that returned the *item name*, *product id* and *number sold* and converted this into a pandas dataframe. I then exported this dataframe as a csv file named `"QuantitiesSold.csv"`. I wrote another SQL query which returned me the total value of all sales and like before, exported the result to a csv. The file name is `"TotalValueOfSales.csv"`. Similarly, I created a csv file called `"TotalNumberOfDeals.csv"` which has column names `"Type of Deal"`, `"Total Number of Deals"`, and `"Total Deals Cost (pence)"`. It contains two entries, one with the relevant values for the meal deals and the other for the cookie deals.

Unfortunately, I didn't include a timestamp in my database or keep a record of the type of payment used so there are no reports for these variables.

4.2 Part 2

The next part wasn't doable due to the database not supporting separate days however, I did include in my code how I would obtain the results if my database did support this feature.

4.3 Part 3

For part 3, I made a csv file that was similar to the dataset file provided. It excludes the timestamp but the other columns are provided. This can be found as the csv file named `"Task2_2Table.csv"`. I also exported both my sale table and my transactions table to csv files named `"saleTable.csv"` and `"transactionTable.csv"` respectively.