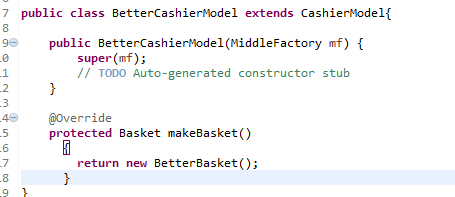
Catshop - 2.1-2.2 Report

# 2.1 – BetterBasket

The initialisation of BetterBasket was altered in main.java and BetterCashierModel.java.



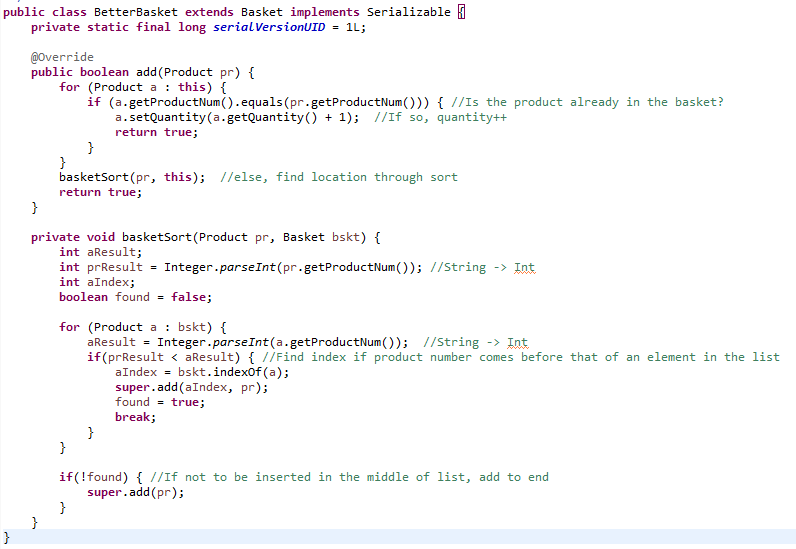
< - main.java



< - BetterCashierModel.java

This class inherits from Basket, and overrides it's add method, which first cycles through the items in the basket to check if the item is already in the basket. If it is, then we simply increase the quantity, and end the method.

If the item is not in the basket, I call a method called basketSort, which compares the product numbers in the basket to find the correct place for the product to be positioned to be in ascending order. This includes either placing it at the end of the basket list, or inserting it at the correct index in the list using ArrayList function indexOf(product) and add(index, product).



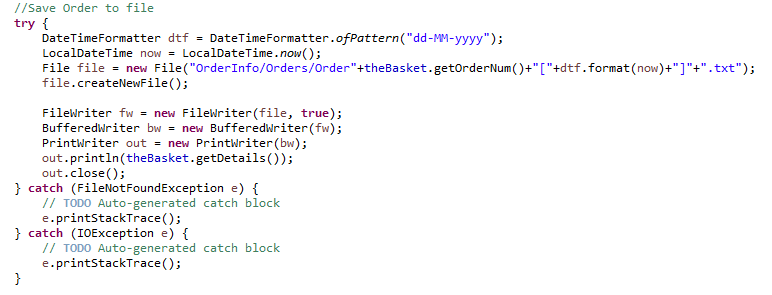
Testing was done manually for this bit, by adding items to the basket in the cashier menu, and seeing the quantity increase as more of the same product were added, whilst also making sure the quantity in stock reduces by 1 for each item added just as before. The other part was adding product numbers in varying orders to make sure they were being added in the correct place and in ascending order.

In terms of critical reflection, I think that the code works perfectly the way that I implemented it. There was another way of sorting the basket using a collection sort, however I found it easier to have a quick for loop running through the list to find the right location. In my opinion this is more optimal than using Collections.sort() method as it has less code to run through to have the same effect, and everything is handled in the same class.

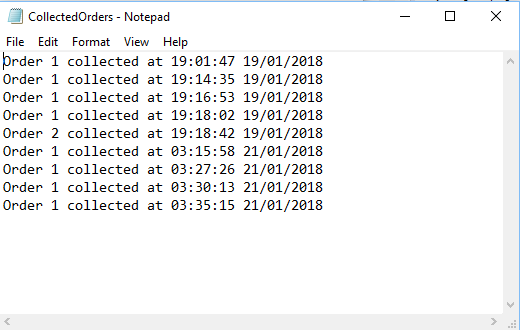
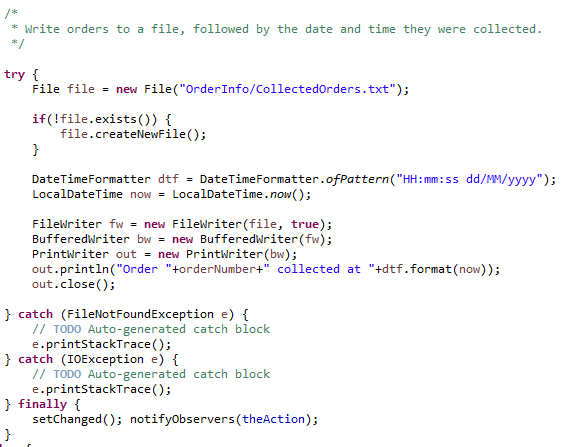
# 2.2 – Catshop extensions

For most of the extensions, I made good use of the JOptionPanes to help add functionality to the program.

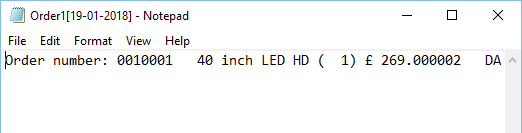
1. The first thing I added to Catshop was recording orders to a file, both when purchased and when collected. Firstly, I worked on saving the receipts of orders to a text file. To do this I used DateTimeFormatter and LocalDateTime to help create a unique file name to avoid clashes, then created a new text file in an appropriate folder titled with the order number and the date purchased. The actual writing was done using 3 kinds of writers, FileWriter, BufferedWriter and PrintWriter, and the information of the order was collected using theBasket.getDetails(). The screenshot below in from CashierModel.java which saves the receipt on purchase.



I also employ a similar piece of code to this in CollectModel.java which adds a line of code to a collection file, which prints the order number and date and time it was collected. The images on the next page show the CollectModel.java code, as well as the output in the .txt file from both.



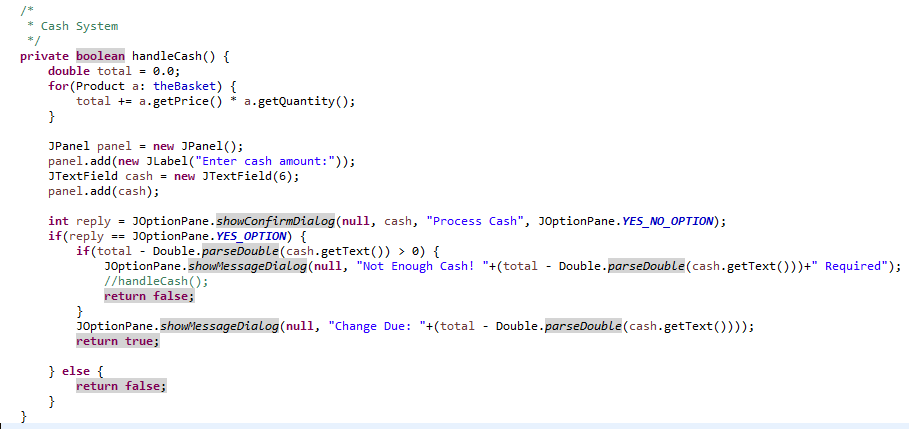
The Orders shown are from multiple restarts of the program during testing, however in normal practice these will increment alongside the orders throughout the day.

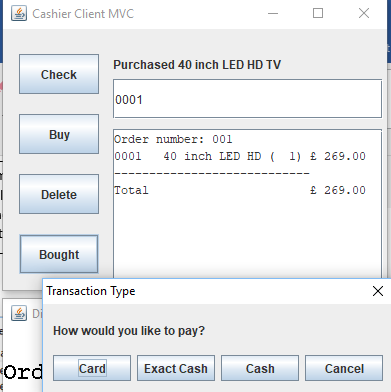
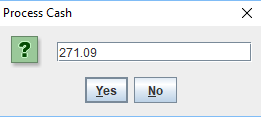
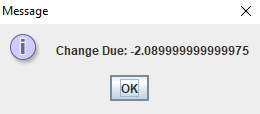
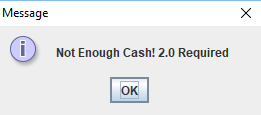


The order information here does not format the same way as in Java, so the receipt is saved lengthways, however is saved nonetheless. The only downside is if the program is reset, the order number also resets, so any new receipts with the same order number are appended to the existing file.

1. I wanted to add extra functionality to CashierModel.java to make it more similar to a real checkout, so I added a Boolean function called handlePayment(), which pops-up with a JOptionPane giving the cashier the ability to choose the type of payment for the customer. The buttons Exact Cash and Card continue straight to the picker window, however there is also a Cash button and a Cancel button. Cancel, as is named, cancels the transaction in case the customer wants to add something or changes their mind, however the Cash button invokes another function called handleCash(). This allows the cashier to input how much cash they have received from the customer. If the customer provides more or exactly the same amount of cash required, a JOptionPane pops-up with the amount of change to give the customer. If the customer does not give enough cash, a different window will pop-up alerting the cashier how much money is needed to complete the transaction, and the payment will be cancelled. I did wish to recal handleCash() recursively until completed or cancelled, however if the input the second time round was less than the total, the transaction would still complete, so I commented out this feature.

The only downside of this is that the windows that have popped up before saying the remainder required must be closed after the transaction is completed or cancelled, so must be closed manually. A version of this which closes the pane automatically on recursion would be better.





4.

Change due calculation. Could be improved if fixed to 2 d.p.

Prompt for cash input

3.

2.

1.

Insufficient Cash prompt

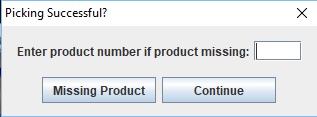
1. I wanted to add functionality to the warehouse collector also, so that they can confirm all items were found, or if some items are missing from the order. I first started with adding a JOptionPane for the picker to select whether the order was picked successfully, or whether an item(s) was not found when they start press the Picked button. If there is an item missing in the warehouse, the picker inputs the product number, and the quantity missing form the order, for instance if the customer orders 4 TV’s, but only 2 can be found, the refund amount (product price \* qty) will be multiplied by that number.

Once the picker has finished adding the missing items, the order number and the refund amount is added to a globally initialised TreeMap, which stores the order number and refund pair. This is later used in the collection window. After this, I also set the stock of that item to 0 so that no more customers can purchase that product via StockReadWriter theStock.buyStock().

Globally Initialised TreeMap to be read by other packages



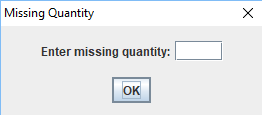
This code is added above the normal code in doPick().



Initial prompt when completing pick. Press continue if there is nothing missing or once missing items have been added.

1.

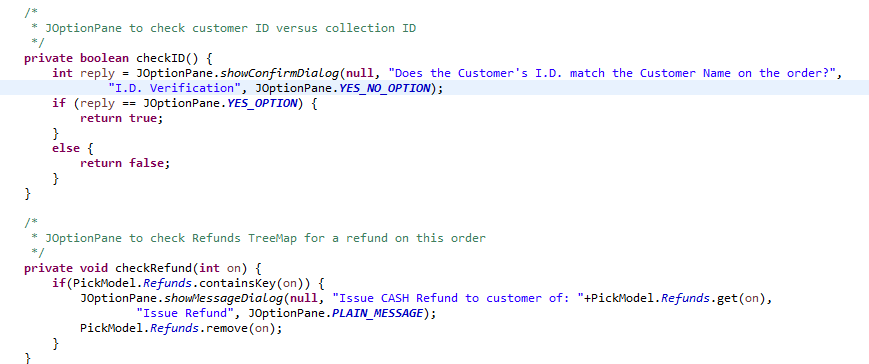
2.

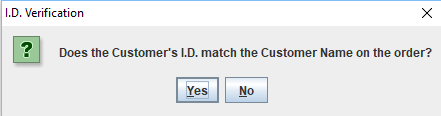


If product is in the pick basket, enter quantity for refund amount. In hindsight a better prompt would check that entered quantity is not greater that amount ordered.

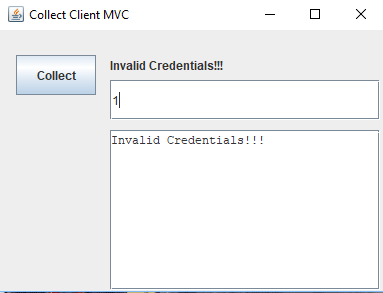
1. I included an addition to the CollectModel.java class, whereby before a customer can collect their order, the clerk must check the I.D. of the customer and makes sure that it matches that of the name on the order (Obviously this is handled in real life rather than on the system). This check is done by running a boolean method called checkID().

If the I.D. does not match, the collection is cancelled. Otherwise if the I.D. does match, another function called checkRefund() is run, which checks the TreeMap ‘Refunds’ (mentioned in part 3) to see if there is a refund to be issued to the customer if an item was not found in the warehouse. If a value is found in the TreeMap, a prompt will appear telling the clerk to issue a cash refund of the amount.

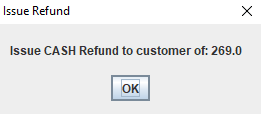




I.D. prompt on collection attempt

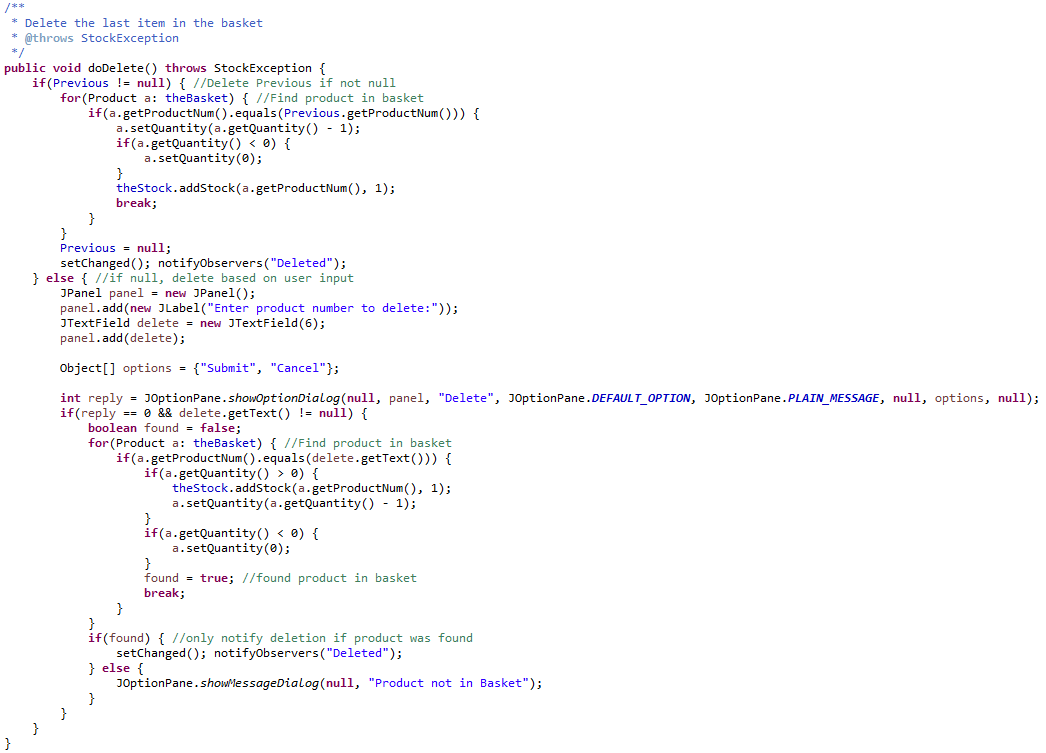


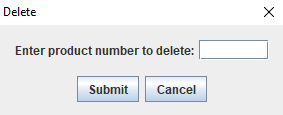
If I.D. does not match, collection fails



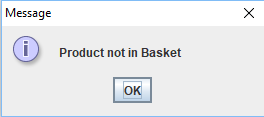
Refund prompt if there is a refund amount on the order

1. As can be seen in part 2, I added a delete function to the cashier window. On the first click, the button deletes the last item added to the basket, by reducing the quantity by 1. On any subsequent clicks however if no new items have been added, a JOptionPane will pop-up asking which product the cashier would like to delete. Once a product number is entered, a for loop is run to check if the product is in the basket and notify the cashier if the product is not in the basket.

Then if the quantity is greater than 0, we reduce it by 1. For both types of deletion, I then use the addStock() function from StockRW.java to add the stock back to the system.



Prompt on second+ deletion



Prompt if product not in basket

All the additions listed above were tested using console prints, and manual testing. For instance, when I was testing to see whether declaring an item could not be found in the system, I used the check function in the cashier window to make sure the stock becomes 0. There were a few bugs along the way which did require fixing, for instance the recursion of the cash handler, or re-adding stock during the deletion function, however through persistence and examining the code provided to me initially, I managed to make each function work as intended.

I have listed a few possible improvements that could be made in the text boxes along the images, however in general I have tried to make the code as optimised as possible to avoid too much strain on the system. The one problem associated in a system such as this with so many JOptionPanes is that there can be clashes with many JOptionPanes as they overlap, however if each window were to be run on a separate machine, this would not be an issue.

I tried to make the system as practical as possible also, and make it reflect somewhat the systems used in retail. Whilst those have many more features and functions, I feel I have managed to implement some important ones and make the system nicer to use. Other improvements that could be made would include making a new GUI that allows multiple inputs text fields instead of series’ of JOptinPanes. This could also help add more buttons to the current windows, and thus add further functionality.