COMPUTING COURSEWORK DOCUMENTATION

Pizza Takeaway Ordering Program

Josh Barrows
Candidate Number: 2318

Contents

Analysis:

Description of the problem:

Project Idea:

To create a program that allows the customer to input their order of pizza, drinks and other extras, and then also enter their personal details so that their food can be delivered to their address. Also, the customer's details and orders will be stored in a database in SQL, which will be designed to be normalised to make it efficient and to remove data redundancy. The customer can then login with their account details when they next wish to order pizza. The project will involve building and maintaining a database that will interact with the Python program to store the inputs the customer enters in a usable format.

What is required by end user?

The end user will require a program that they can use to order pizza from. They will want options such as; whether the pizza is stuffed crust, its size and its toppings. They will also be able to input their customer details so they are saved onto the system so next time they can just use their login details to access their account. The system has to be relatively easy to use so that the customer, who will probably lack much computing knowledge, can successfully use the system.

Identification of the stakeholders:

Who is the end user (stakeholders)?

I aim to make my program as accessible as possible, so to include as many people as possible. The main end user group is of course customers who wish to order pizza from the takeaway service, however I will create the program to make it as easy to use as possible so that the end user is not a small group of people, but a wide range of ages and technical ability, etc. – the software will be usable by those with no computing experience, as many people who order pizza are not well accustomed to computer programming!

Other stakeholders in the program will be the chef and delivery man, who can log in to the system to view orders and update status on home delivery. Another account will also be required for a system manager, who will be able to see an overview of the system and the current orders.

Specific Users:

- Typical customer: Ben Smith

    - Orders pizza takeaways usually each weekend, so has lots of experience with multiple other pizza order systems.
- Prospective owner/administrator: Adam Jones
    - This person is planning on starting a takeaway business and requires custom made software that will suit his requirements
    - Adam Jones will approve the final program, based on whether it meets the requirements

Justify why the problem can be solved by computational methods:

The problem is suited for a computer program because the inputs are all pre-determined so the program simply has to enter the details of the order into the database. Overall, using a computer is far more efficient and timesaving than doing the job manually. Other features of my program that make the software ideal for a computer to carry out is the ability to find a user's login details in the database and compare them with what has been inputted. This can be justified because a computer could be able to do this search and check within milliseconds whereas a human could take several moments determining whether the inputted details are actually correct.

Furthermore, the problem is suitable for a computational approach because the whole principal of a pizza delivery ordering system is that the customer can buy their food without having to physically go to a store. Therefore, a computational approach is required as using a computer is the only way pizza can be ordered without going to visit the local shop.

Also, the overall calculation of the cost of the order means that it is suitable for computational solution because this arithmetic can be quickly and accurately carried out by the computer.

The outputs produced by the problem are valuable to the stakeholders involved, because it sends the order details to the chef, a major stakeholder, and then to the delivery person when ready. Without this, no pizza would be cooked and nothing delivered.

Relating to my objectives (detailed further down in the Analysis), I have identified which aspects require computational properties:

- To allow the end user to order food of their choice
- To allow the end user to input their customer details
    - These requires computational solution because to order a pizza, the user can only choose from certain options. And to input customer details this requires an input form and then needs to be saved in the database
- To show the user a summary of their order when completed
- To present the user with a status update of their order once it has been ordered
    - These require computational solution because the program will have to store and then locate the order within the database so that a summary can be

created and the order status updated by the staff and fetched for the customer

- To store their details in a database, and allow them to login to the system with a username and password which will bring up their details to save them re-entering them when they make a second order
- To allow the user to order more than one pizza per order
    - This requires computational solution because it connects to a database so the user can remain logged in throughout using the program, and the user can order multiple pizzas at once
- To calculate the total cost of the customer's order
    - This requires computational solution to do the sums on the total price of the order

In order for my solution to work, assumptions have to be made. Firstly, the program will only work if the database is stored locally, so currently the program would have to be stored within the same network as the orders database. This means that the customer would have to order instore on a computer. Therefore, there is an assumed remote link between the program and database which does not exist in the solution's current form. Hence, my solution is only a prototype solution because it does not have the required networking capabilities to allow it to communicate between the database and remote clients.

Research:

Background Research of the end user (Interview):

I carried out some background research with a typical customer to find out what it is that is required, so I could be sure to include these in the design for my program.

I carried out an interview with a typical pizza-ordering individual, Ben Smith:

"Ben, how often do you use pizza takeaway services a month?"

Usually, I tend to order at least once a fortnight. Occasionally, I may even order a pizza each weekend. It depends on my schedule and whether I've got friends around.

"What sort of takeaway products would you expect from a pizza place?"

Most services I've used offer pizzas, with the whole array of toppings and sizes, and the options of a drink and extras, such as garlic bread. The best takeaways offer cheese stuffed crust too.

“Would you be happy to use a program to order pizza rather than visiting the actual branch?”

Yes, I have no problems with this. The pizza-ordering systems available today are far more useful and timesaving than driving down to the local store, when I could have it delivered to my house at no cost to me.

“What sort of payment options would be acceptable?”

Just the option to pay by card is adequate for everyone.

“Do the aesthetics or the usability of the program matter to you as a customer? Would it affect your experience?”

In my opinion, as long as the program is functional, easy to use and has no bugs, I personally am not entirely bothered by the aesthetics of the program, as long as the software remains simple to use and isn’t a complete eyesore.

“What sort of price are most pizzas and how much are you prepared to pay for delivery?”

Usually, the cost of pizzas is between 6 and 10 pounds, depending on the quality of the pizza. The best takeaway places offer delivery for free.

To ensure that I included all the required functions, I also interviewed a prospective pizza takeaway owner on their views on my system.

I carried out an interview with a prospective buyer of my software, Adam Jones:

“What types of accounts would you like there to be on the system?”

On my system there must be an account for me, the system administrator, and for the chef and delivery man to use. Obviously, the customer needs to be able to make an account also.

“How many orders should the customer be able to make?”
The customer can order as often as they like, and to order as many items in one order as they want. There will be no restriction on this.

“What food should be available to buy?”

On my system, the consumer should be able to purchase a pizza, of a size and topping of their choice, and also order it with extra sides such as garlic bread and also drinks.

“Would you recommend prioritising style or substance in the development of this software?”
The main focus should be to develop the software so that it performs all the core

functionality requirements of the program. It is paramount that the program can correctly and efficiently process the customers' orders. This is more important than the styling of the program. Style and design is still crucial for delivering a good quality service to consumers but is not as critical as developing the main program.

"Will your staff, such as the chef and delivery man require a login to the system, and if so will they require special privileges?"
Yes, there will be a need for my other staff to have a login. The chef will need to be able to see the current orders so he knows what to cook and the delivery man will require access to the system so he can see where he needs to deliver. Both will also have to be able to update the status of the orders.

Analysis of interviews:

My interviewing process has been essential to the planning of my pizza takeaway program. It has offered me invaluable insight into the thoughts and requirements of consumers and also has given me guidance as to the path the software development takes.

Some key information learnt:

- There must be a wide range of pizza available, by demand of both the consumer and owner
- The takeaway app must be efficient and easy to use, by demand of both consumer and owner
- By general consensus, the usability of the program is more important than the aesthetics, although the styling of the software cannot be completely ignored

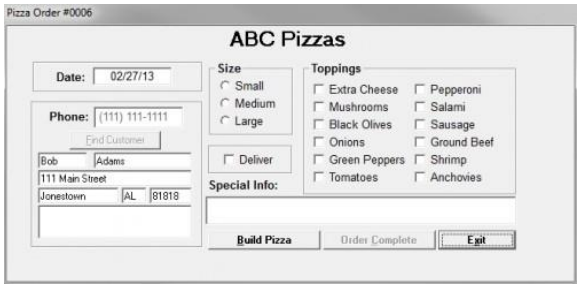
What I will take away from these interviews:

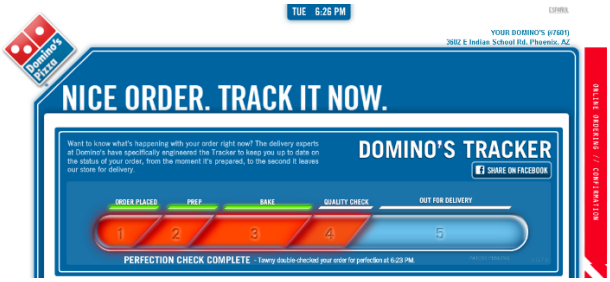
- To ensure my pizza takeaway offers a big range of pizzas, drinks and sides
- To create an effective and simple program that can be accessed by the consumer, owner, and other pizza staff involved, such as a chef and delivery man
- I will prioritise the usability of the program over the design

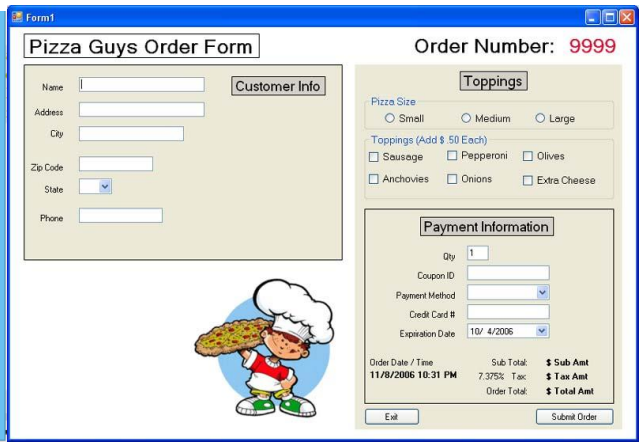
Other sources of research information/evidence:

I have used existing pizza takeaway services to analyse the types of products and services they usually supply.

Below are some screenshots of current systems on the market:







These systems above range from local businesses to pizza delivery services in other parts of the world to the systems used by huge multinational companies like Dominos.

A list of the features of pizza takeaway ordering systems that I have identified from my research:

- The ability to choose any pizza topping type from a big list
- The option to add extras such as drinks, leading to increased choice for the customer and more revenue for the takeaway supplier
- The takeaway services offer real-time updates to the customer as to the whereabouts of their order
- The service may offer loyalty discounts to repeating customers
- These pizza takeaways often insert a picture of each pizza, with it looking very appetising, in order to sell it

Areas of this research I can adapt and apply to my project:

- I will use several of the ideas found in pre-existing solutions to improve my program and to add additional features to make it stand out
- The idea of ‘customer loyalty’ discounts shouldn’t be too difficult to implement and would greatly improve the user experience
- Real-time updates are always favoured by the consumer so is something I have already factored into my program

Features of the proposed solution:

Further meetings with stakeholders:

Following on from my first meeting, I have decided to host another interview to allow me to inform stakeholders of the limitations and my new ideas.

Here is the second interview with a prospective owner/administrator, Adam Jones:

"Following on from our previous meeting, I have identified a few limitations of the program. These are not necessarily essential to the core workings of the program however you may find them useful. The program has no current way of interrogating the data collected from users, and cannot create financial or marketing reports. These features are in a roadmap for future features but will not be included in the final release."

Thank you for informing me of these limitations, however they are not essential to my program and will not affect the usability of the software. I understand that these features could be included in a future release further along down the road.

"Since our last meeting several new ideas have surfaced from my other research that I would like to consider including in the final product. These are the 'loyalty rewards' for repeating customers, which is guaranteed to bring back old customers if they will get money off their bill. Another idea I found was having real-time status updates as to how the pizza is doing that the customer can see on their order page. This adds that extra level of customer interaction with the business."

These are great suggestions and I would fully support their inclusion in the program.

Objectives (Features of the proposed solution):

1) To allow the end user to order a certain size pizza of their choice
2) To allow the end user to order a certain crust type of pizza of their choice
3) To allow the end user to order a certain pizza topping of their choice
4) The user can order additional drinks to go with their order
5) The user can purchase extras with their order, including garlic bread, etc
6) To allow the end user to select a payment method of their choice when making an account
7) To allow the end user to input their name when making an account
8) To allow the end user to input their house number and postcode when making an account
9) To allow the end user to input their username when making an account
10) To allow the end user to input their password when making an account

11) To allow the end user to input their security question and answer when making an account
12) To allow the end user to input their mobile number when making an account
13) The program will plot the user's postcode on top of Google Maps using the API to show the users delivery point in relation to the pizza shop (RGSHW)
14) The program will use the Haversine Formula to work out the straight line distance between 2 coordinates on the map to work out the distance between them
15) The program will only allow an account to be made if the user is within a specified distance from the shop, set at a 10km radius
16) To show the user a summary of their order when completed
17) To present the user with a status update of their order once it has been ordered
18) To store their details in a database, and allow them to login to the system with a username and password which will bring up their details to save them re-entering them when they make a second order
19) Admins can view all the data in the Customer Information table
20) Allow admins to add user accounts
21) Allow admins to update user accounts
22) Allow admins to suspend user accounts
23) Block customers from logging in if their account is suspended
24) Allow admins to re-activate user accounts
25) To create other types of accounts besides regular customer logons, such as specialised logins for pizza chefs and delivery staff

26) To allow the customer to order more than one pizza per order
27) To allow the customer to order more than one drink per order
28) To allow the customer to order more than one extra per order
29) To calculate the total cost of the customer's order
30) To offer discounts to recurring customers
31) The customer can change their personal details once they have an active account already, should they need to do so
32) The admin can delete orders
33) The admin can view data analytics which provides information about how the system is performing
34) Data analytics will total up and output the total amount of orders
35) Data analytics will total up and output the total cost of orders
36) Data analytics will total up and output the total amount of user accounts
37) Data analytics will total up and output the total amount of active accounts
38) Data analytics will total up and output the average order value
39) Data analytics will create a HTML pie chart showing the most popular pizza sizes
40) Data analytics will create a HTML pie chart showing the most popular pizza crust types
41) Data analytics will create a HTML pie chart showing the most popular pizza topping
42) Data analytics will create a HTML pie chart showing the most popular extras

43) Data analytics will create a HTML pie chart showing the most popular drink
44) The staff account can see all the outstanding orders that require attention
45) The staff account can view what has been ordered in each order
46) The staff account can edit the status of each order
47) To create a Graphical User Interface that is simple to use but also has all the required functions
48) To implement the database SQL to store the information output by the running program
49) To ensure that the PyQt windows interact properly with the Python coding, which in turn interacts with the SQL database
50) To make the coding as efficient as possible (e.g.: passing parameters, reusable procedures), with no duplicate coding or other inefficient techniques being used
51) To make the SQL database in 3NF to ensure that the data is properly normalised, which would maintain data integrity and eradicate data redundancy
52) Customers can recover their password if they forget it
53) To hash the stored passwords in the database for added security
54) Passwords entered when creating an account must be checked for strength
55) All necessary inputs are validated using Regular Expressions, so that the program cannot be crashed (e.g.: postcode validation, mobile number validation, etc)
56) The program will use the system call to open the Print Window, allowing the user to print to a printer
57) The program will use the system call to open the Print Window, allowing the user to save their document in a variety of file types, including PDF and XPS
58) The customer has an option to print the order confirmation at the time of placing the order
59) The customer can revisit a previous order confirmation and print the order if necessary
60) To create multiple account types with tiered access levels and permissions, including an administrator account with access to the whole system, a secondary staff account that can see the outstanding orders and a basic account for customers
61) Customers will be greeted with a personalised welcome message once logged in

Areas for development (Limitations):

- In the future, I could add modules onto the basic operation of the program that offer additional features and extras that are not included in the core software. These modules include:
- A ‘Marketing’ Module:
    - This could be used to create reports on customer loyalty to identify the highest recurring customers, the highest spending customers and to break down the business income into days of the week and times of the month. This would allow marketing material to be sent out in the form of mailshots that could be personalised to individual customers and to promote pizza buying in periods of low sales.

- A ‘Multi-User’ Module:
    - This would break down the company stats by each employee. Rather than a generic overview, this would specify how many pizzas each chef cooked, or which delivery man delivered the most amount of pizzas.
- Finally, the program in its current state will only connect to the Pizza-Order database if both the program and database are stored locally. A pizza takeaway would have to run the program on the consumer’s local computer but be able connect to the main database across the internet. However, I have decided this level of complexity is not required for my program but would have to be taken into account if this software was to be deployed.

Software and hardware requirements:

Hardware and Software Requirements:

To run my program, the user will be required to have Python installed to run the program. The Graphical User Interface forms that are essential to the program will be stored with the main programming as well as the SQL database. For simplicity these will all be stored in the same ZIP folder so the program works correctly.

My program should be capable of running on basic hardware. All that is required is a computer with an input device such as a mouse and keyboard. It is also compulsory to have an output device such as a screen so the consumer can see what they are doing. The only storage device required is a hard disk. This need not be particularly large or fast, as the program and related database are only tiny file sizes and there is no rush for the program to run quickly, as this is not a real time processing application.

Specific Specifications of a computer system:

- A computer monitor/screen with a resolution of at least 1500 x 900 pixels, as this is the largest window in my program and in order for it to be displayed properly the screen must have at least this resolution
- The CPU of the program requires a 1GHz processor or faster. The program has been tested on this speed processor and it works as required
- The program needs 10MB of storage to store the Python coding and related UI files

Success Criteria:

My success criteria are based upon my stakeholder requirements and is measurable criteria that can be demonstrated through testing. It will be used with evidence of testing to evaluate the final product.

- The customer can effectively use the program to its fullest extent. They can:
    - Login with their details, which will bring up their details previously entered into the system
    - New users can sign up to the system and enter their details, such as their name and address
    - Can then order from the options provided, they can choose: pizza size, the option of stuffed crust, topping, extras and drinks
    - Order more than one pizza or drink
    - Receive a full output as a confirmation of their order, including all items ordered and a total cost
    - See an up to date order status, which tells the user the progress of their pizza
    - Recover their password if they forget it
    - Update their personal information

- The program will be regulated by an administrator account:
  They can:
    - Add special accounts such as chefs and delivery men logins, which would have access to the full orders table
    - View all the outstanding orders
    - Delete Orders
    - Edit Customer details
    - Suspend users
    - View Data Analytics

- Another special account is the staff's logon:
  They can:
    - Login to the system with their credentials and be recognised as a higher-profile user, so are presented with a different screen
    - See orders that need to be cooked and delivered
    - Change the status of orders so the customer can get an update

My success criteria are justifiable because they take into account the needs of all the stakeholders and are measurable in a test plan. This is essential given that these criteria have been drawn up to determine later on in the programming stages whether or not the aims of the program have been accomplished.

Design:

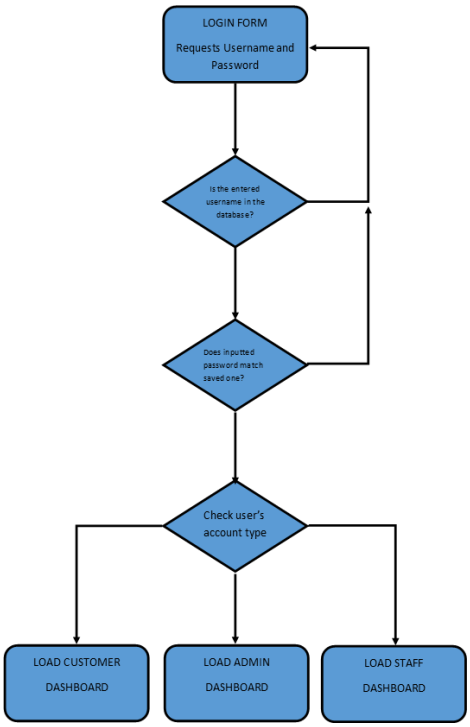
Decompose the problem:

Due to the complexity of my program, it will not be possible to code it linearly. Therefore, I have broken down the program into its component parts, into sets of small problems that are suitable for computational solution.

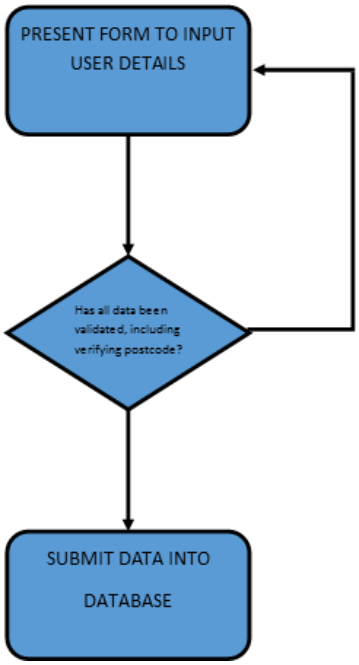
I decomposed the coding into groups based on each individual process that occurs in my program. This ranges from logging in, to ordering pizza or recovering a lost password.

Each sub-problem that I have defined is a thread of GUIs, calculations and inputs/outputs, and are defined below.

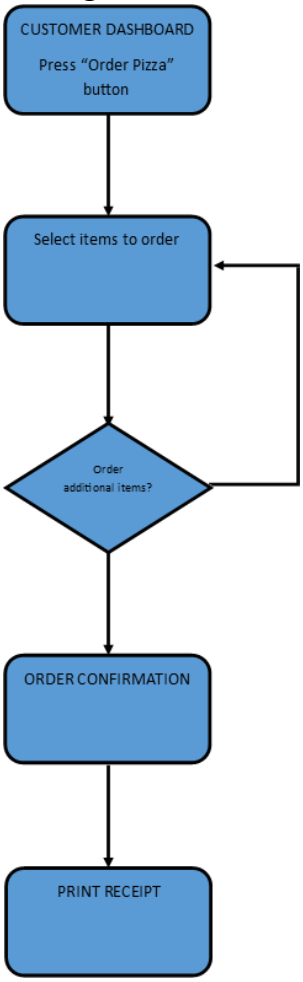
1) Logging in to the system



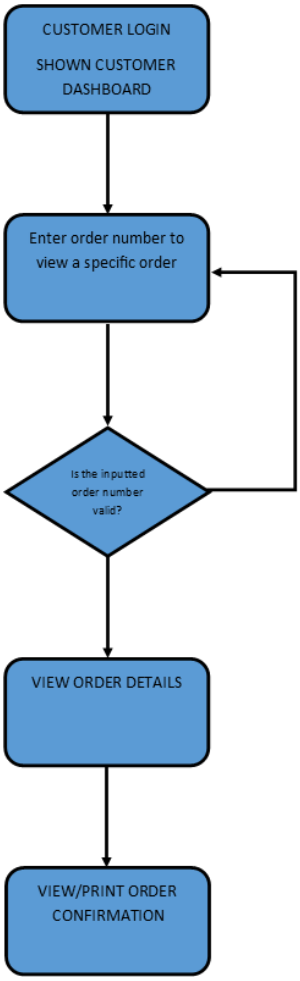
2) Creating a user account



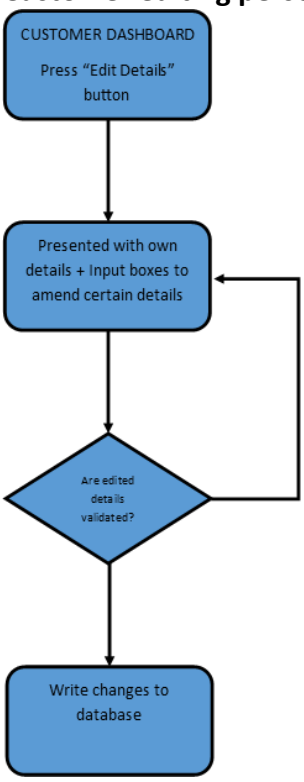
3) Placing an order



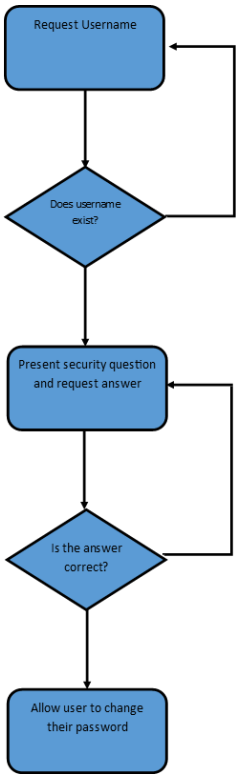
4) Customer reviewing previous order



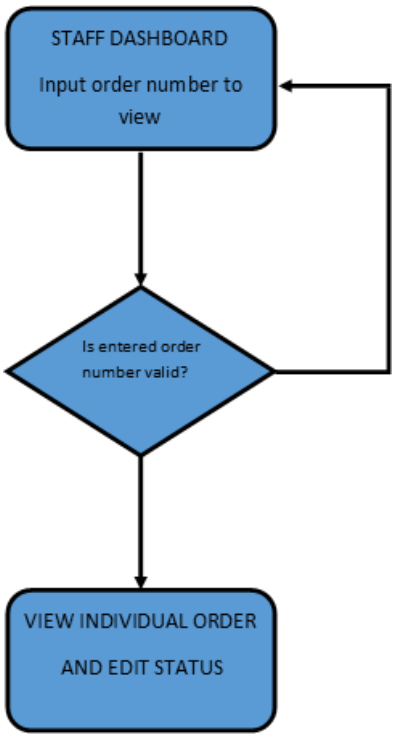
5) Customer editing personal details



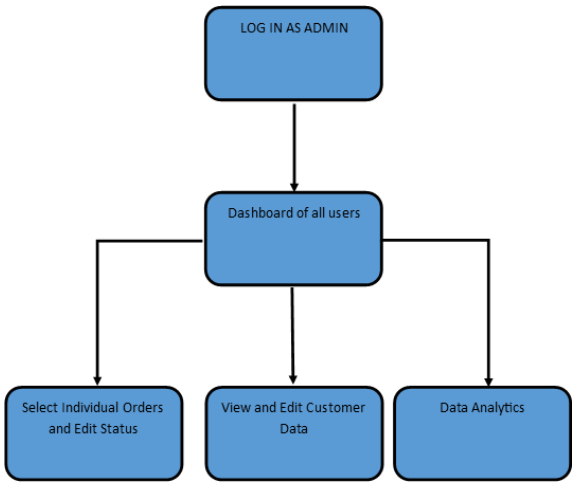
6) Password Recovery



7) Staff View Orders



8) Admin Options



Structure of the solution:

The structure of the solution will be to initially get the 'bare bones' of the program up and running, by creating the GUIs and getting them to load properly. Then the focus will be on linking each window to the next by using the action buttons, and after this the database must be set up and linked to. Then, the program will be developed with the main stakeholders in mind. The customers' side of the program will be focused on, followed by the staff and admin logins. Finally, the last stage of my solution will be to 'add complexity'. This means to implement the Google Maps API, make HTML pie charts and create the password recovery tool.

Algorithms:

1) Logging in to the system

```
START
        INPUT USERNAME
        INPUT PASSWORD
        IF USERNAME IN CUSTOMER TABLE THEN
                IF PASSWORD = SAVEDPASSWORD THEN
                        IF ACCOUNTTYPE = "Customer" THEN
                                CALL LOGIN CUSTOMER
                        ELSE IF ACCOUNTTYPE = "Admin" THEN
                                CALL LOGIN ADMIN
                        ELSE IF ACCOUNTTYPE = "Staff" THEN
                                CALL LOGIN STAFF
                        END IF

                ELSE
                        OUTPUT "Wrong Password!"
                END IF

        ELSE
                OUTPUT "Wrong Username!"
        END IF
END
```

2) Creating a user account

```
START
        INPUT firstname
        INPUT surname
        INPUT housenumber
        INPUT postcode
        INPUT mobilenumber
        INPUT paymenttype
        INPUT username
        INPUT password
        INPUT securityquestion
        INPUT answer
        IF firstname = " " THEN
                OUTPUT "Answer all questions"
        ELSE IF surname = " " THEN
```

```
            OUTPUT “Answer all questions”
ELSE IF housenumber= “ ” THEN
            OUTPUT “Answer all questions”
ELSE IF postcode = “ ” THEN
            OUTPUT “Answer all questions”
ELSE IF mobilenumber = “ ” THEN
            OUTPUT “Answer all questions”
ELSE IF paymenttype = “ ” THEN
            OUTPUT “Answer all questions”
ELSE IF username = “ ” THEN
            OUTPUT “Answer all questions”
ELSE IF password = “ ” THEN
            OUTPUT “Answer all questions”
ELSE IF securityquestion = “ ” THEN
            OUTPUT “Answer all questions”
ELSE IF answer = “ ” THEN
            OUTPUT “Answer all questions”
END IF
Valid = TRUE
FOR i=0 TO LENGTH(firstname)
            IF (ASCII_CODE(firstname[i]) > 64 AND ASCII_CODE(firstname[i]) < 91) OR
            (ASCII_CODE(firstname[i]) > 96 AND ASCII_CODE(firstname[i]) < 123) THEN
                        Valid = FALSE
                        EXIT FOR
            END IF
NEXT i
IF Valid = FALSE THEN
            OUTPUT “Invalid name format”
ELSE IF POSTCODE INVALID THEN
            OUTPUT “Invalid Postcode”
ELSE IF MOBILENUMBER < 11 AND FIRST 2 LETTERS ARE NOT ‘07’ OR CONTAINS LETTERS THEN
            OUTPUT “Invalid Mobile Number”
END IF
FOR EACHUSERNAME IN CUSTOMER TABLE:
            IF EACHUSERNAME==USERNAME THEN
                        OUTPUT “This username is already being used. Please try another.”
            END IF
IF LENGTH(PASSWORD) < 6 THEN
            OUTPUT “Password is too short”
END IF
FUNCTION HAVERSINE(LON1, LAT1, LON2, LAT2):
            LON1, LAT1, LON2, LAT2 = MAP(RADIANS, [LON1, LAT1, LON2, LAT2])
            DLON = LON2 - LON1
            DLAT = LAT2 - LAT1
            A = SIN(DLAT/2)**2 + COS(LAT1) * COS(LAT2) * SIN(DLON/2)**2
            C = 2 * ASIN(SQRT(A))
            R = 6371
             TRAVELDIST = C * R
             RETURN TRAVELDIST
END FUNCTION
IF TRAVELDIST > 10 THEN
            OUTPUT “This location is too far away to deliver to.”
ELSE:
            OUTPUT “Delivery Possible”

            OPEN FILE “Customer Table”
            CUSTOMERS:= FILE.READ(“Customer Table”)
            NumRows = LENGTH(Customers)
            Customers RESIZE NumRows +1
            LastRowNum = LENGTH(Customers)
            Customers[LastRowNum,0] = Firstname
```

```
                Customers[LastRowNum,1] = Surname
                Customers[LastRowNum,2] = housenumber
                Customers[LastRowNum,3] = postcode
                Customers[LastRowNum,4] = mobilenumber
                Customers[LastRowNum,5] = paymenttype
                Customers[LastRowNum,6] = username
                Customers[LastRowNum,7] = password
                Customers[LastRowNum,8] = securityquestion
                Customers[LastRowNum,9] = answer

        END IF
END
```

3) Placing an order

```
START
        REPEAT UNTIL ANOTHER= “False”:
                INPUT pizzasize
                INPUT pizzacrust
                INPUT pizzatopping
                INPUT drink
                INPUT extra
                OPEN FILE “Orders Table”
                ORDERS:= FILE.READ(“Orders Table”)
                NumRows = LENGTH(Orders)
                Orders RESIZE NumRows +1
                LastRowNum = LENGTH(Orders)
                Orders[LastRowNum,0] =pizzasize
                Orders[LastRowNum,1] = pizzacrust
                Orders[LastRowNum,2] = pizzatopping
                Orders[LastRowNum,3] = drink
                Orders[LastRowNum,4] = extra

                SizeList = INTEGER Array()
                SIZELIST=SIZELIST+PIZZASIZE
                CrustList = INTEGER Array()

                CRUSTLIST=CRUSTLIST+PIZZACRUST
                ToppingList = INTEGER Array()
                TOPPINGLIST=TOPPINGLIST+PIZZATOPPING
                DrinkList = INTEGER Array()
                DRINKLIST=DRINKLIST+DRINK
                ExtraList = INTEGER Array()
                EXTRALIST=EXTRALIST+EXTRA
                IF “Add Another” BUTTON PRESSED THEN
                        ANOTHER = “True”
                ELIF “Order Done” BUTTON PRESSED THEN
                        ANOTHER = “False”
                END IF
                FOR EACHPIZZA IN TOPPINGLIST
                        COST = COST + 7.99
                FOR EACHDRINK IN DRINKLIST
                        COST = COST +2.99
                FOR EACHEXTRA IN EXTRALIST
                        COST = COST + 3.99
        OPEN FILE “Orders Table”
        ORDERS:= FILE.READ(“Orders Table”)
        NumRows = LENGTH(Orders)
        Orders RESIZE NumRows +1
        LastRowNum = LENGTH(Orders)
        Orders[LastRowNum,0] =sizelist
        Orders[LastRowNum,1] = crustlist
```

```
        Orders[LastRowNum,2] = toppinglist
        Orders[LastRowNum,3] = drinklist
        Orders[LastRowNum,4] = extralist
        OUTPUT sizelist, crustlist, toppinglist, drinklist, extralist
END
```

4) Customer reviewing previous order

```
START
        INPUT ORDERNUMBER
        FOR EACHORDERNUMBER IN ORDERS TABLE THEN
                IF EACHORDERNUMBER == ORDERNUMBER THEN
                        CALL ORDER VIEW
                        FETCH ORDERDETAILS FROM ORDERS TABLE WHERE ORDERID = ORDERNUMBER
                        OUTPUT ORDERDETAILS
                ELSE
                        OUTPUT "Order Doesn't Exist"
                END IF
END
```

5) Customer editing personal details

```
START
        INPUT btnPersonalDetails PRESSED:
                CALL PERSONALDETAILS VIEW
                FETCH PERSONALDETAILS FROM CUSTOMER TABLE
        INPUT housenumber
        INPUT postcode
        INPUT mobilenumber
        INPUT paymentmethod
        INPUT password
        IF POSTCODE INVALID THEN
                OUTPUT "Invalid Postcode"
        ELSE IF MOBILENUMBER < 11 AND FIRST 2 LETTERS ARE NOT '07' OR CONTAINS LETTERS THEN
                OUTPUT "Invalid Mobile Number"
        ELSE IF LENGTH(PASSWORD) < 6 THEN
                OUTPUT "Password is too short"

        END IF

        INSERT HOUSENUMBER, POSTCODE, MOBILENUMBER, PAYMENTMETHOD, PASSWORD INTO
        CUSTOMER TABLE

        OPEN FILE "Customer Table"
        CUSTOMERS:= FILE.READ("Customer Table")
        NumRows = LENGTH(Customer)
        Customers RESIZE NumRows +1
        LastRowNum = LENGTH(Customers)
        Customers[LastRowNum,0] =housenumber
        Customers[LastRowNum,1] =postcode
        Customers[LastRowNum,2] =mobilenumber
        Customers[LastRowNum,3] =paymentmethod
        Customers[LastRowNum,4] =password
END
```

6) Password Recovery

```
START
        INPUT username
        FOR EACHUSERNAME IN CUSTOMER TABLE
```

```
                IF EACHUSERNAME == username THEN
                        FETCH SECURITYQUESTION FROM CUSTOMER TABLE
                        OUTPUT SECURITYQUESTION
                        INPUT SECURITYANSWER
                        IF SECURITYANSWER == CORRECTANSWER THEN
                                INPUT NEWPASSWORD
                                IF LENGTH(NEWPASSWORD) <6 THEN
                                        OUTPUT "Password is too short"
                                ELSE

                                        OPEN FILE "Customer Table"
                                        CUSTOMERS:= FILE.READ("Customer Table")
                                        NumRows = LENGTH(Customer)
                                        Customers RESIZE NumRows +1
                                        LastRowNum = LENGTH(Customers)
                                        Customers[LastRowNum,0] =newpassword

                                END IF
                        ELSE
                                OUTPUT "Incorrect Security Answer"
                        END IF
                ELSE
                        OUTPUT "This username does not exist"
                END IF

        NEXT

END
```

7) Staff View Orders

```
START
        INPUT ordernumber
        IF ORDERNUMBER IN ORDERS TABLE THEN
                CALL ORDER VIEW
                FETCH ORDERDETAILS FROM ORDERS TABLE WHERE ORDERID = ORDERNUMBER
                OUTPUT ORDERDETAILS
                IF ORDERSTATUS CHANGED THEN

                        OPEN FILE "Orders Table"
                        ORDERS:= FILE.READ("Orders Table")
                        NumRows = LENGTH(Orders)
                        Orders RESIZE NumRows +1
                        LastRowNum = LENGTH(Orders)
                        Orders[LastRowNum,0] =orderstatus
                END IF
        ELSE
                OUTPUT "Order doesn't exist"
        END IF

END
```

8) Admin Options

To view a specific order and edit status/delete:

```
START
        INPUT ORDERNUMBER
        FOR EACHORDENUMBER IN ORDERS TABLE
                IF EACHORDERNUMBER== ordernumber THEN
                        CALL ORDER VIEW
                        FETCH ORDERDETAILS FROM ORDERS TABLE WHERE ORDERID = ORDERNUMBER
                        OUTPUT ORDERDETAILS
                        IF ORDERSTATUS CHANGED THEN
                                OPEN FILE “Orders Table”
                                ORDERS:= FILE.READ(“Orders Table”)
                                NumRows = LENGTH(Orders)
                                Orders RESIZE NumRows +1
                                LastRowNum = LENGTH(Orders)
                                Orders[LastRowNum,0] =orderstatus

                        END IF
                        INPUT “btnDeleteOrder” PRESSED
                                DELETE ORDER FROM ORDERS TABLE WHERE ORDERID = ORDERNUMBER

                ELSE
                        OUTPUT “Order doesn’t exist”
                END IF
END
```

To view/edit customer data:

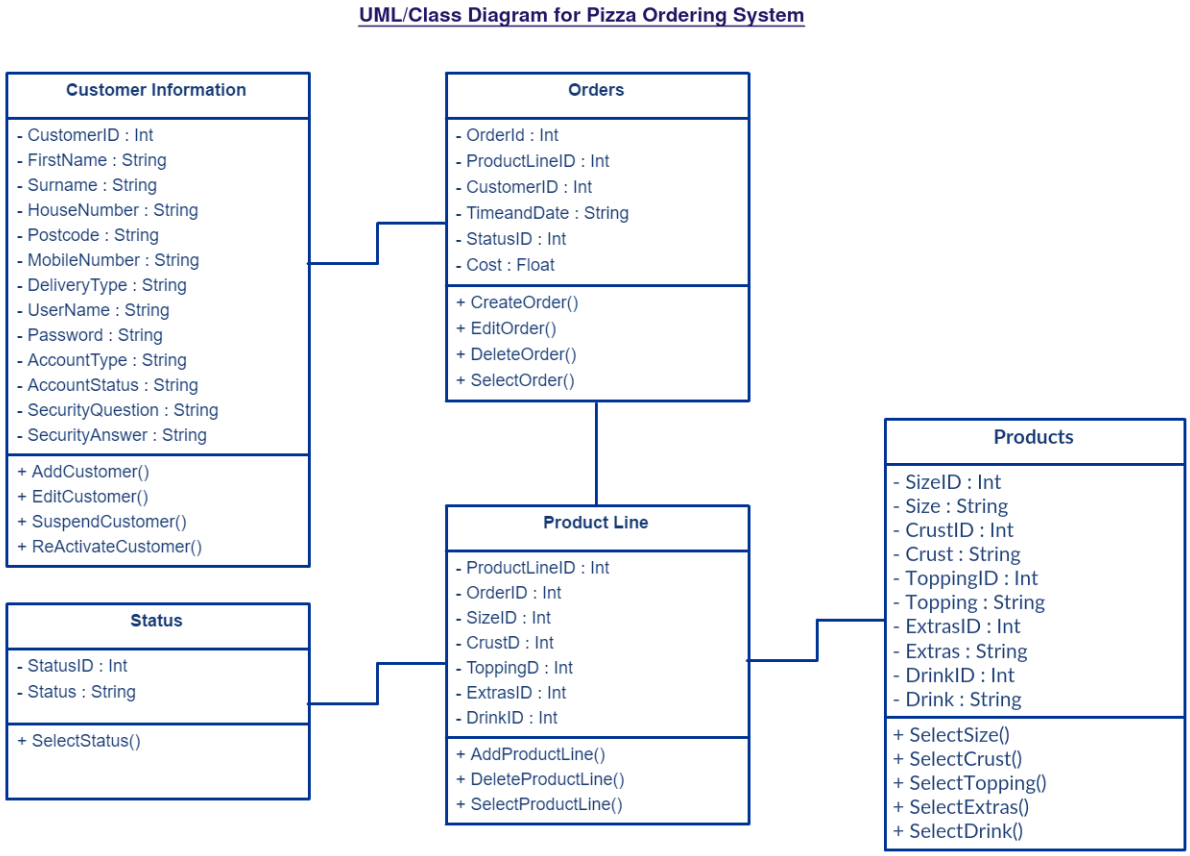
```
START
        FETCH CUSTOMER TABLE
        OUTPUT CUSTOMER TABLE
        INPUT “btnUpdateTable” PRESSED
                FOR EACH IN EACHCOLUMN
                        FOR EACH IN EACHROW
                                OPEN FILE “Customer Table”

                                CUSTOMERS:= FILE.READ(“Customer Table”)
                                NumRows = LENGTH(Customer)
                                Customers RESIZE NumRows +1
                                LastRowNum = LENGTH(Customers)
                                Customers[LastRowNum,0] =each

        INPUT “btnSuspendUser” PRESSED
                SET CUSTOMER_TABLE .“AccountStatus” TO “Suspended”
END
```

To view data analytics:

```
START
        FETCH ALL FROM ORDERS TABLE WHERE ORDERID IS UNIQUE
        TOTALORDERS = COUNT(ORDERID)
        TOTALREVENUE = COUNT(COST)
        AVERAGEORDER = TOTALREVENUE / TOTALORDERS
        FETCH ALL FROM CUSTOMER TABLE WHERE ACCOUNTTYPE = “Customer”
        TOTALACCOUNTS = COUNT(CUSTOMERID)
        ACTIVEACCOUNTS = COUNT (CUSTOMERID WHERE LASTORDER < 1 MONTH)
```

```
        OUTPUT TOTALORDERS, TOTALREVENUE, AVERAGEORDER, TOTALACCOUNTS, ACTIVEACCOUNTS
        FOR EACH OF PIZZASIZE, PIZZACRUST, PIZZATOPPING, DRINK, EXTRA IN PRODUCTLINE TABLE:
                COUNT(EACH)
                CREATE PIE CHART
    END
```

UML Diagram:



Usability features:

Initially, I have designed some mock-up illustrations of my program forms. These give a basic sense to how the program will operate and look, and their functionality can be built upon at a later stage in my design. I have included the primary form design and the final design next to it, as well as with annotations that demonstrate how the forms have been designed to be straightforward and simple to use, yet retain all the key information required of them.

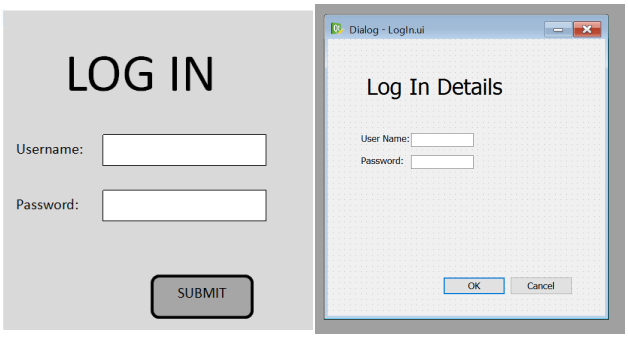
In my program I have used PyQt4 to design and develop my input forms that the program will load for the end user to use.

1) Logging in to the system

When first considering the design for my program, I created a home screen that the user was first presented with:

However, since then, I have removed this screen and hence there is no updated version. The form contained limited options and served little purpose. All the functionality of this stage of my program has since been transferred elsewhere.

A multitude of this functionality has been transferred to the first screen the user now sees. This is the Log In window. My initial mock-up design looked like the design on the left, and I have also included the basic first-draft of my GUI.

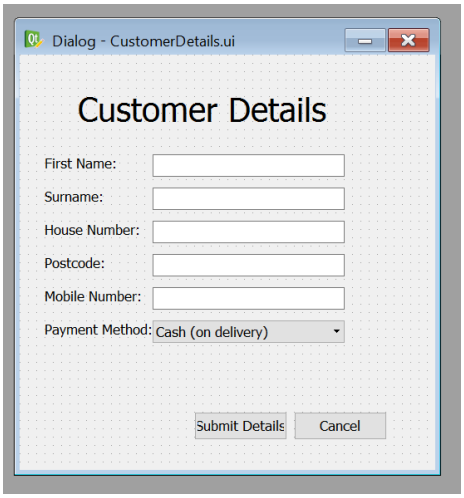


My final form design for the login screen includes the option to recover a forgotten password which was not previously seen, and now contains the button to sign up.

2) Creating a user account

I created a mock-of what the window could look like where the user enters all their details:

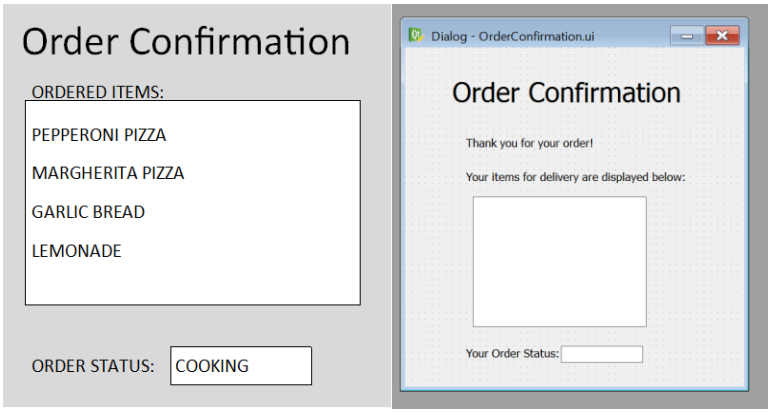
Similarly, the customer signup screen has been overhauled since its initial design. It now contains far more information to be filled in, each added as the program has grown in complexity. Below is the first version of the GUI.



3) Placing an order

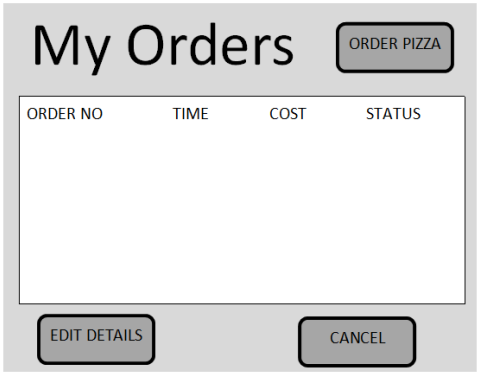
The pizza ordering screen has not evolved much since its original conception. The design remains simple to use, but still consists of all the required functionality. I first created a mock-up of how the GUI will be designed:

The order confirmation has seen one of the biggest changes since it was first designed. My mock-up of this window and the first draft of the actual GUI shows how the customer can see what they have ordered, and also see the order status of their order which will be updated when the staff progress the order forwards.

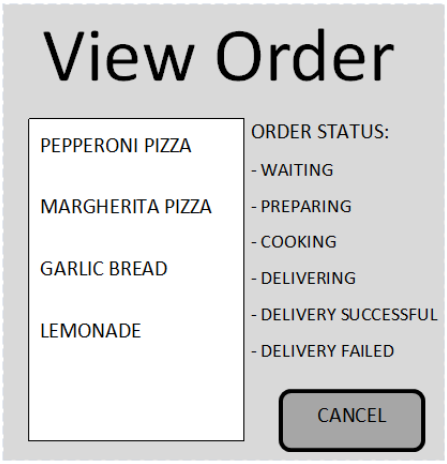


4) Customer reviewing previous order

A new form is the users home screen. This is the default screen the user sees when they login. It contains details of previous orders and the options to order pizza, edit details or view particular orders. There are similar screens for the admin and staff accounts, except they can view and edit all orders from all accounts.

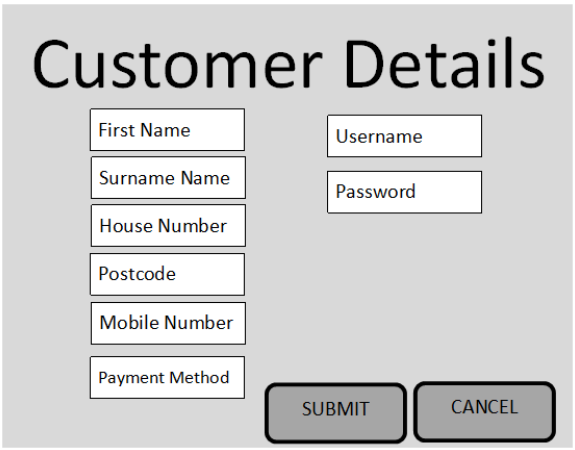


In order to review a previous order, the user has to type in then order number and hit enter or press “Select”. This then shows the order in more detail by loading a new window which shows what has been ordered and its status, as well as a link to the order confirmation screen, which contains details including price and payment method.



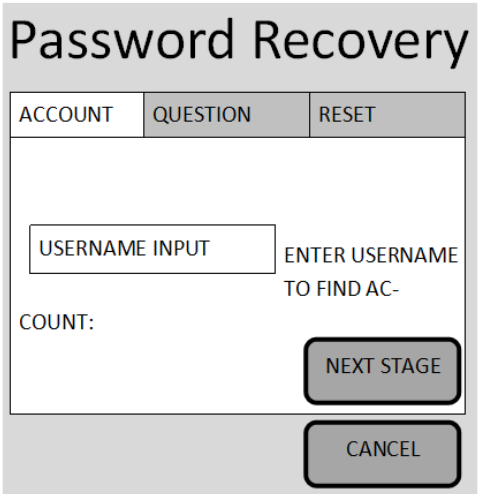
5) Customer editing personal details

Customers also have the ability to view their current personal details that are held in the system and then edit selected parts. This means contact details and delivery addresses etc. can be updated.



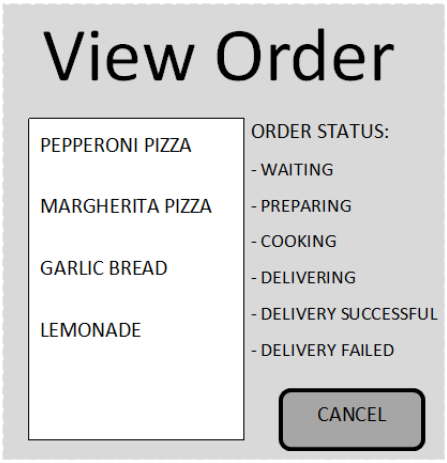
6) Password Recovery

The password recovery form uses tabs to take the user through each step of the process, with the tabs acting as markers as to the progress of the recovery. The tabs are faded out when not in use so the user cannot skip ahead or backwards.



7) Staff View Orders

The staff account can also view each order individually, as the chef needs to be able to see what's been ordered and then update the order status when necessary. This window looks similar to the version used by the customer, except it has the ability to edit the status rather than just view it.

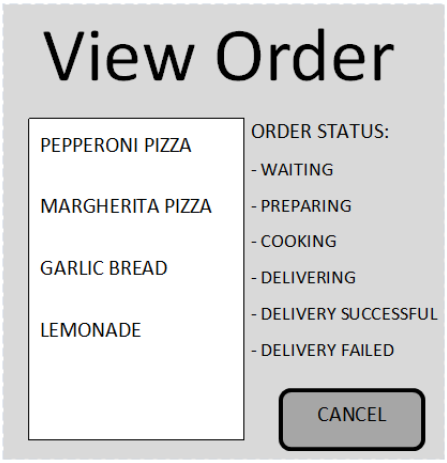


The interface is clean and very simple to use. The actual items ordered are easy to read and changing the status is done by selecting a radio button.

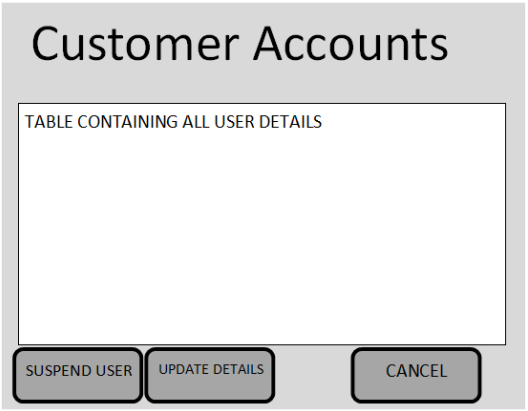
8) Admin Options

The screens below represent the viewing and editing of a customer order, viewing and editing customer details and loading the data analytics page.

Likewise, to the staff account, the admin can also view and edit orders, although the account can view every order in the database, not just outstanding orders. They also have the option of deleting the order entirely.



To view and edit the customer details when in admin mode, a window with a table that links to the SQL database is used. Simply changing the required details and pressing update will change the database data to the new information.



Finally, the data analytics window will consist of user friendly HTML pie charts that present data in an easy to read format. The window will also provide other stats that are both informative and useful.

Justification of fonts, colours and themes:

I have designed my program graphical interfaces to be simple to use, and have understated colours and easy to read fonts that ensure the program does not become too complex or confusing for the user.

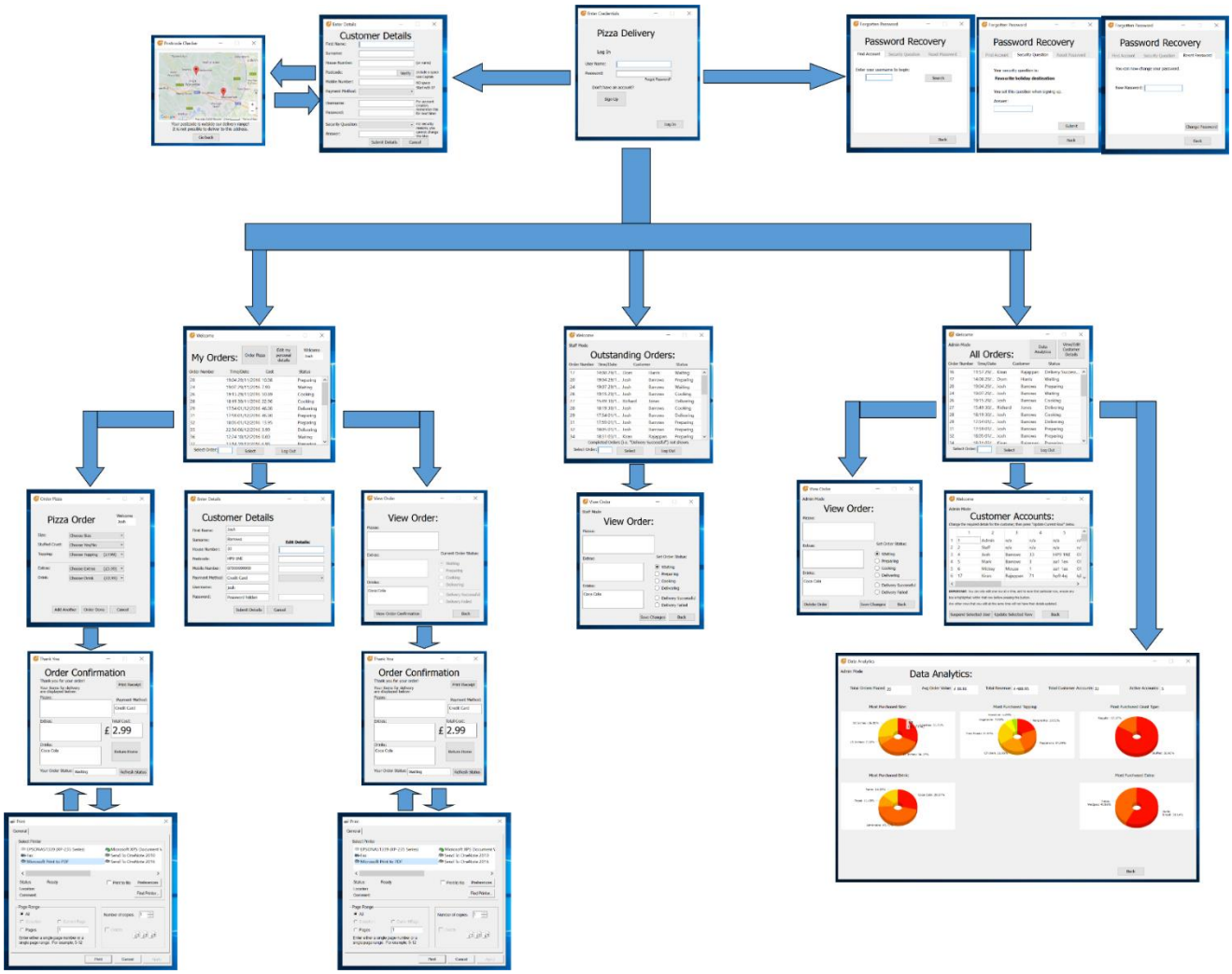
The font I chose was 'MS Shell Dlg 2' because it is easy to read. I purposefully stayed away from overly-stylised fonts that I thought would detract from the usability of the program.

Furthermore, all text in my program is black. I chose this colour because it looks smart and maintains the continuity between different windows.

The background colour is a beige colour that takes a "back seat" and allows the content on the GUI to take centre stage, meaning the user is not put off by any over-zealous backgrounds or other unnecessary colours that detract from the usability experience.

Furthermore, the size of the windows follow a standard that I decided upon, with most windows being 500px by 500px, with some being wider than this, and others such as the data analytics window being much larger if necessary. Other elements that have justifiable sizes are the buttons, which have mostly been created in the default size available, with only certain action buttons being larger. Regular sized buttons create an easier experience for the user who will already be accustomed to such sized buttons.

The overall layout and links between windows:
All windows have their own class, as is the nature of PyQt. Hence, the relationships and links between each GUI window/class is shown below:



Key variables and structures:

Tables and keys used:

- Orders table
    - Used to hold all the current orders that have been sent in. It links multiple product lines to one order, which is all linked to one customer
    - Contains the primary key ‘OrderID’ which is unique for each order
    - Foreign keys:
        - ‘ProductLineID’ from the Product Line table
        - ‘CustomerID’ from the Customer Information table
        - ‘StatusID’ from the Status table
    - Finally, the Orders table contains the field ‘Time/Date’.

- Sample data:

	FullOrderID	ProductLineID	CustomerID	TimeandDate	StatusID	Cost
	Filter	Filter	Filter	Filter	Filter	Filter
1	16	16	17	11:57 29/11/2016	6	17.96
2	17	17	18	14:08 29/11/2016	1	17.96
3	20	20	4	19:04 29/11/2016	2	10.98

- Customer Information table
  - This table contains the details of every account in the database so is used a lot - for example when logging in, ordering, changing details, creating an account, etc.
  - Primary key is 'CustomerID', which is linked to the Orders table and is unique for each user
  - The table also has fields for holding the customer details asked for in the customer details input form, such as name and address, and their account login and password.
  - Sample data:

CustomerID	FirstName	Surname	HouseNumber	Postcode	MobileNumber	DeliveryType	UserName	Password	AccountType	AccountStatus	SecurityQuestion	SecurityAnswer
Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
4	Josh	Barrows	99	HP13 6QT	07948274613	Credit Card	josh	2c3ffdda0...	Customer	Active	Favourite holida...	spain
5	Mark	Barrows	3	aa1 1en	07492011472	Debit Card	mark	b8bdba22...	Customer	Active	Favourite holida...	spain
6	Mickey	Mouse	1	aa1 1aa	07943047214	Credit Card	mickey	2a24f5109...	Customer	Active	Favourite holida...	spain

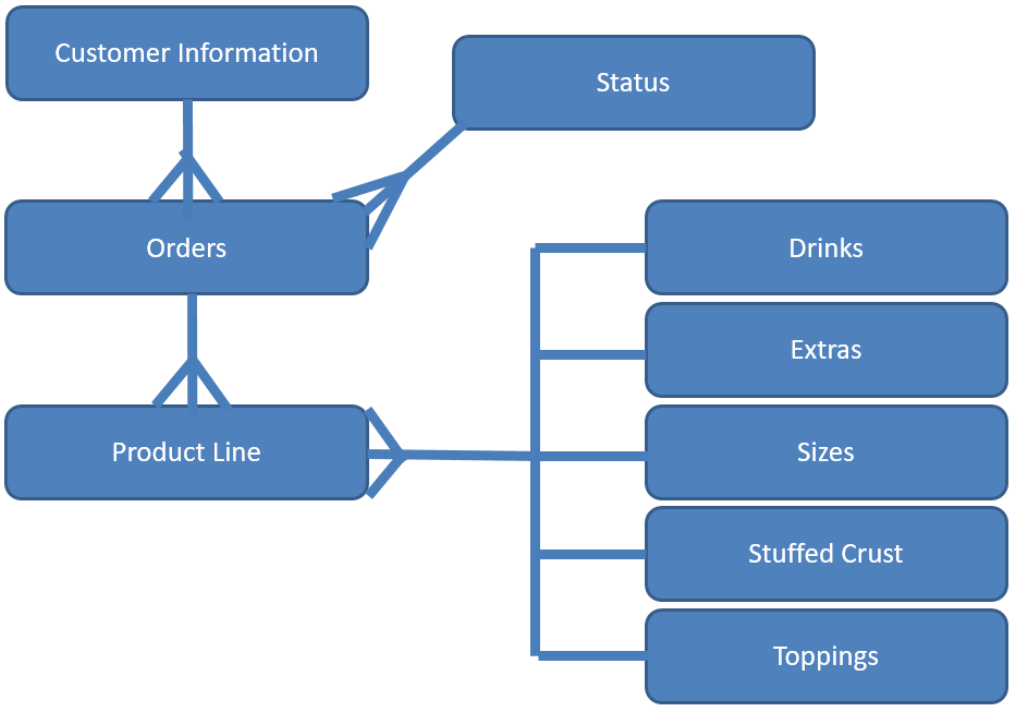
- Product Line table
  - Contains its primary key, 'ProductLineID', and then consists of all the other foreign keys from the other tables which store the products.
  - Foreign Keys:
    - SizeID
    - CrustID
    - ToppingID
    - ExtraID
    - DrinkID
  - This table contains all the information about each order – what the user has ordered in each order
  - Sample data

ProductLineID	OrderID	SizeID	CrustID	ToppingID	ExtrasID	DrinkID
Filter	Filter	Filter	Filter	Filter	Filter	Filter
3	3	2	1	5	1	3
4	4	0	0	0	0	2
5	5	0	0	0	2	0

- Product tables
    - Size
    - Stuffed Crust
    - Toppings
    - Drinks
    - Extras
        - These are all tables consisting of a primary key that links to product line table and they also contain the actual products available, such as the types of toppings or drinks.
- Status table
    - Contains the 'StatusID' primary key which links to the Orders table, which is used for keeping track of each order, by reporting the current process of each pizza preparation.
    - Sample Data:

StatusID	Status
Filter	Filter
1	Waiting
2	Preparing
3	Cooking

Entity-Relationship Diagram:



Key Variables Used:

Key Variable Required	Data Type	Which Window /Procedure	What it stores	Why it's needed
usernameoptionsid	Integer	First defined in the Log In GUI. Is found throughout the program to display the current users first name on the window.	Stores the customerID from the database of the currently logged in user **Sample Data:** "1" "6" "4"	This is a global variable that is necessary so that the program can keep the user logged on and show their correct orders and details, as well as allowing orders to be made in the correct name.
overallpizza, drinkslist, extraslist	String	Found in the Pizza Ordering procedure, and also in the Order Confirmation windows	Stores the pizzas, drinks and extras currently being ordered before they are saved into the database, with a space in between each item **Sample Data:** "Lemonade fanta pepsi"	These are used during the ordering process and are used to keep a tally of all the items ordered, so they can then be displayed all at once in the order confirmation. Contrary to their names, they are not lists but in fact variables.
pizzaprice, drinksprice, extrasprice	Float	Found in the Pizza Ordering procedure, and also in the Order Confirmation windows	The running cost of the order **Sample Data:** "4.99" "7.99"	Similar to the variables above as they are used in the same section of code, however, they are used to keep track of the total cost of the order
usernameattempt	String	Used in the Log In procedure	The username just entered into the program **Sample Data:** "Josh" "randomusername"	Used during the login process to temporarily store the inputted username. It is

				matched up against each username in the database till the equivalent is found
passwordattempt	String	Used in the Log In procedure	The password just entered into the program **Sample Data:** “Password” “randompassword”	Temporarily stores the data from the inputted password when logging in. It is then hashed and compared to the correct password to see if they match.
secondorder	Boolean	Used in the Order Pizza GUI	It is initially False, when more items are submitted, the product line in the database gains a new orderID, but on the second time around it is True, so the next product line keeps the same orderID as the previous record, which keeps them linked together as one order in the database table **Sample Data:** “True” “False”	This is another global variable that is used so that the program is aware of whether it has added more than one pizza or another item to the database or not.

Inputs and Outputs:

Below are the possible inputs and outputs of my program.

- Inputs:
    - Logging in with username and password
    - Signing up by entering customer details
    - Entering details required for password recovery
    - Ordering pizza
    - Selecting a previous order to view its details
    - Updating customer details
    - Deleting orders
    - Suspending users
- Outputs:
    - Displaying the pizzas ordered to the user
    - Displaying the cost of the order
    - Transferring the meal choices into the Product Line and Orders tables in the SQL database, and pairing it up with the relevant customer's details.
    - Creating user accounts
    - Updating details into the correct SQL rows
    - Retrieving saved previous orders

Test data for development:

When I begin iteratively developing my program, I will use the following data to create different account types so that each section of the program can be accessed. These logins below can be used to access all the account types specified.

Data Used:	**Explanation:**
Username: josh	This is the username for my account, where I will do the majority of the testing and development
Password: 123456	This is the password for the above account. It is short and easy to input deliberately, because I require it so much
Username: admin	This account is used for accessing the administrators section of the program
Password: password	This is the password for the above account.
Username: staff	This login accesses the staff section of the program
Password: password	This is the password for the above account.

Validation Table,

and Test Data to be used during the iterative development of the solution:

These are examples of the data I have been using in my development of the program. It is useful to ensure that the program cannot be crashed by inputting invalid data and that there is sufficient protection to stop wrong inputs being accepted.

The table first describes all the validation measures that will be put in place, and then highlights some test data that would be invalid and could crash the program if not effectively defended against.

Several of the validation procedures use REGEX functions to ensure that the input is valid. An example of this is the postcode checker, as well as the mobile number and name verifier.

Where Validated:	Type of validation:	Test Data Examples:
Signing up – First Name	Text only, no digits, more than 0 characters, Less than 15 characters	Josh1234 [Blank Input] Joshjoshjoshjosh
Signing up – Surname	Text only, no digits, more than 0 characters, Less than 15 characters	Barrows9876 [Blank Input] barrowsbarrowsbarrows
Signing up – House Number	Text or digits (might be a house name), more than 0 characters, less than 15 characters	[Blank Input] 12345678901234567
Signing up – Postcode	Must contain 6 or 7 characters with a space in-between, can only consist of numbers and letters in the correct slots. i.e. HP13 6QT	HP136QT (no space) hiosfh898 H9 1DE 97G E35
Signing up – Mobile Number	Digits only, must be 11 characters, start with “07”	0790fhsiche 0790123456789 99070707077
Signing up - Username	Cannot be the same as an existing username in the database, more than 0 characters	[Enter a username already known to be stored in database]
Signing up – Security Question	Must select a predefined question from a list of options	[Blank Input]
Signing up – Security Answer	More than 0 characters	[Blank Input]
Signing up – Password	Must be more than 6 characters, to ensure that the user enters a secure password	Passw
Order Overview Screen – Select Order	Must be a valid order in the Orders table in the database	Fdijij

		[Order number that doesn't exist]
Ordering Pizza – Size, Stuffed Crust, Topping	If one drop-down menu is selected, they must all have an option selected, as all 3 are required i.e. You cannot order a pepperoni pizza without also specifying its size and crust type	[Attempt to order a pizza topping and not specify its size or crust type]
Edit personal details – House Number, Postcode, Mobile Number, Password	All these are subject to the same validation as when initially signing up	As above
Password recovery – Setting a new password	Must be more than 6 characters, to ensure that the user enters a secure password	Passw

Post-development Test Data:

Once my program is in the post-development phase, I have data that is suitable for the finished program.

Table: CustomerInformation

	CustomerID	FirstName	Surname	ouseNumb	Postcode	MobileNumber	DeliveryTyp	UserName	Password	AccountType
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Admin	n/a	n/a	n/a	n/a	n/a	admin	9ab810a9...	Admin
2	2	Staff	n/a	n/a	n/a	n/a	n/a	staff	9ab810a9...	Staff
3	4	Josh	Barrows	3	aa1 1en	07903488648	Debit Card	josh	1a3a8375...	Customer
4	5	Mark	Barrows	3	aa1 1en	07492011472	Debit Card	mark	8ccb7c21...	Customer
5	6	Mickey	Mouse	1	aa1 1aa	07943047214	Credit Card	mickey	2a24f510...	Customer
6	17	Kiran	Rajappan	71	hp9 4ej	07638571359	Credit Card	kiran	6acf42c3...	Customer
7	18	Dom	Harris	87	hp14 8fh	07938502761	Debit Card	harrisdj	5c396055...	Customer
8	20	Richard	Jones	43a	hp10 4gh	07820486716	Debit Card	jonesra	0f4f19a3...	Customer
9	21	Dylan	Smith	23	hp13 6gh	07458311294	Credit Card	dsmith	fd881594...	Customer
10	22	Annie	Jacobs	42	hp9 3if	07346912571	Debit Card	ajacobs	1bd74c7...	Customer

This is the post-development data that contains the customer information so that there are accounts in the system initially.

Table: Orders

	FullOrderID	oductLineI	CustomerID	TimeandDate	StatusID	Cost
	Filter	Filter	Filter	Filter	Filter	Filter
1	16	16	17	11:57 29/11/2016	6	17.96
2	17	17	18	14:08 29/11/2016	1	17.96
3	20	20	4	19:04 29/11/2016	1	10.98
4	24	24	4	19:07 29/11/2016	1	7.99
5	26	26	4	19:15 29/11/2016	3	50.89
6	26	27	4	19:15 29/11/2016	3	50.89
7	26	28	4	19:15 29/11/2016	3	50.89
8	27	29	20	15:49 30/11/2016	1	17.96
9	28	30	4	18:19 30/11/2016	1	22.96
10	28	31	4	18:19 30/11/2016	1	22.96

This is the post-development order data that will be used to populate the outstanding orders.

Table: ProductLine

	oductLineI	OrderID	SizeID	CrustID	ToppingID	ExtrasID	DrinkID
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	1	1	2	2	2	2
2	2	2	1	1	2	1	2
3	3	3	2	1	5	1	3
4	4	4	0	0	0	0	2
5	5	5	0	0	0	2	0
6	6	6	0	0	0	1	0
7	7	7	0	0	0	0	4
8	8	8	0	0	0	0	2
9	9	9	0	0	0	0	3
10	10	10	0	0	0	1	0
11	11	11	0	0	0	0	3
12	12	12	0	0	0	1	0
13	13	13	0	0	0	2	0
14	14	14	0	0	0	1	0
15	15	15	0	0	0	0	2
16	16	16	3	1	4	1	2
17	17	17	2	1	3	2	2
18	18	18	2	2	5	1	2
19	19	19	3	1	2	2	4
20	20	20	4	1	3	0	0

This is the product line post-development data that is utilised by the orders table.

This post-development data is justified because it is validated and operational data that will ensure the program works as intended and can therefore showcase the many aspects of my application.

Developing the coded solution

Iterative development of a coded solution:

Initial Stage of development:

Loading the basic first-design GUIs was the initial stage of development. This involved the primary pieces of coding to load the user interfaces as shown below.

```
import sys, os
from PyQt4 import QtCore, QtGui, uic,

win1 = uic.loadUiType("HomeScreen.ui") [0]

class FirstWindow(QtGui.QMainWindow, win1):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
```

This loaded the first window I created, the home screen. The now-defunct window was used as the initial loading screen, as shown below.

Furthermore, I also coded for the other windows in the first phase of my program. This included the login screen and the sign up screen, amongst others.

2nd Stage – Linking Windows:

In the second stage of my development, I coded for the buttons so that they responded when pressed. This involved essentially registering for the button to be pressed and then closing the current window and opening the new appropriate window.

```
import sys, os
from PyQt4 import QtCore, QtGui, uic

win1 = uic.loadUiType("HomeScreen.ui") [0]
win2 = uic.loadUiType("Login.ui") [0]
win3 = uic.loadUiType("Signup.ui") [0]

class FirstWindow(QtGui.QMainWindow, win1):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.loginbutton.clicked.connect(self.login)
        self.signupbutton.clicked.connect(self.signup)

    def signup(self):
        self.signupwindow = ThirdWindow()
        self.signupwindow.show()
        self.hide()

    def login(self):
        global usernameoptionsid
```

This then allowed me to utilise all the buttons in the program and move between windows.

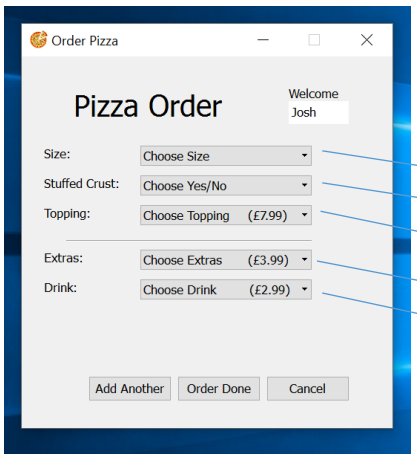
3rd Stage – Connecting to database:

Once the GUIs had been properly set up, the next stage of coding required was to connect to the database I had created. This consists of tables that contain the customer details as well as the orders. The database is essential for the program to function correctly and also to meet the objectives laid out in my analysis.

```
import sys, os
from PyQt4 import QtCore, QtGui, uic, QtSql
import sqlite3 as lite
con = lite.connect('PizzaDatabase.db')
cur = con.cursor()

win1 = uic.loadUiType("HomeScreen.ui") [0]
```

This successfully allowed me to connect with the database, stored in the same system folder, so then I could code the inserting and reading of data from the file.

An example of this is the pizza order window, in which the database is used to insert the order and store it in the correct table so it can be retrieved later on in the program.



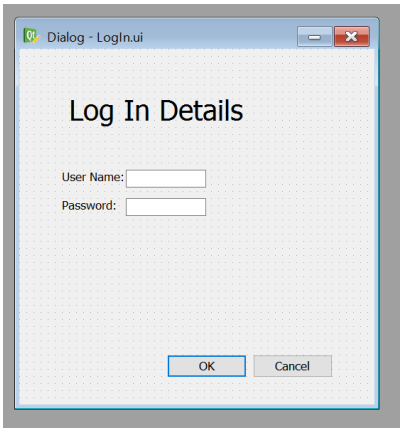

```
def orderdone(self):
    global secondorder
    sizebox = self.Size.currentIndex()
    crustbox = self.StuffedCrust.currentIndex()
    toppingbox = self.Topping.currentIndex()
    extrasbox = self.Extras.currentIndex()
    drinksbox = self.Drink.currentIndex()
```

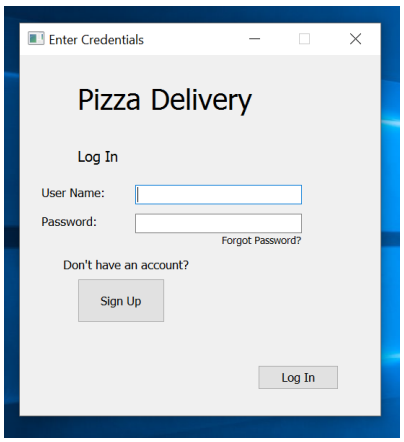
```
sql = """INSERT INTO "ProductLine" ("ProductLineID", "OrderID", "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" ) VALUES ( ?, ?, ?, ?,  ?, ?, ?)"""
cur.execute(sql, (newproductid, neworderid, sizebox, crustbox, toppingbox, extrasbox, drinksbox, ))
con.commit()
```

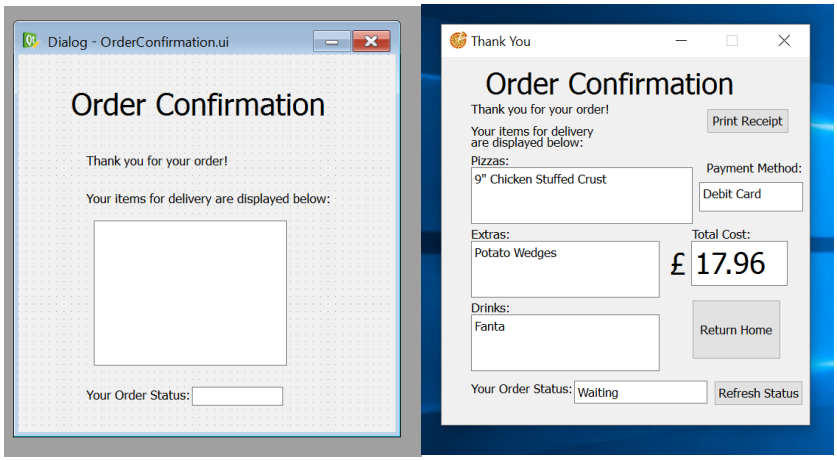
4th Stage – Focusing on Customer side of program:

A large chunk of the GUIs and coding is on the customer's side of the program, so I decided this was a sensible place to start coding. The functionality for the staff and admin accounts could wait till late in the development phase, as I chose to work on one stakeholder at a time.

I developed my GUIs as I progressed with the coding of the program. Initially, I created some windows that only contained basic functionality and had no emphasis on design. As I added more features to my program, I improved the user interfaces. The below screenshots give an idea of the iterative development that has occurred.







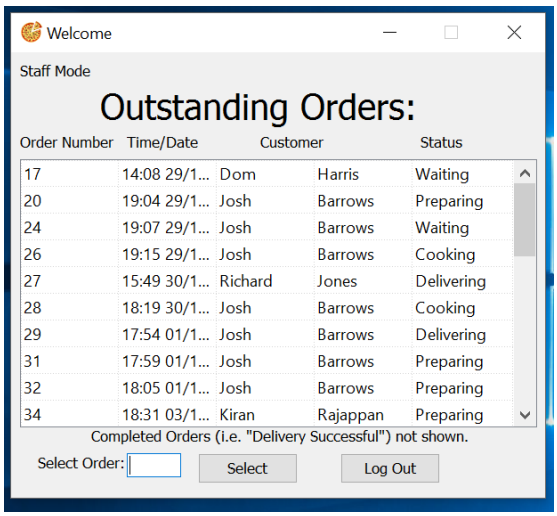
5th Stage – Developing the staff login:

After creating the customer login, the staff login was developed next. In order to login to this elevated account, the Customer Details table in the database was given an extra column that specified the account type. Regular accounts created on the signup page are given "Customer" accounts, whereas the staff login is saved as "Staff". The program checks this when the user logs in to determine which order screen should be displayed – the customer, staff or admin one. The code for this is below.

```
if self.verifyhash(passwordattempt, passwordoptions[0][0]):
    self.hide()
    accounttypesql = """SELECT "AccountType", "AccountStatus" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ % (usernameoptionsid[0])
    cur.execute(accounttypesql)
    accounttype = cur.fetchall()

    if accounttype[0][1] == "Active":
        if accounttype[0][0] == "Customer":
            self.Order = TenthWindow()
            self.Order.show()
        elif accounttype[0][0] == "Staff":
            self.Order = EighthWindow()
            self.Order.show()
        elif accounttype[0][0] == "Admin":
            self.Order = FourteenthWindow()
            self.Order.show()
    else:
        self.Order2= SecondWindow()
        self.Order2.show()
        QtGui.QMessageBox.information(self, "Account Suspended", "This account has been suspended. \n Contact the system administrator.")
else:
    self.hide()
    self.Order2= SecondWindow()
    self.Order2.show()
    QtGui.QMessageBox.information(self, "Incorrect Details", "Your username/password is incorrect. \n Please try again...")
```

The prototype for the staff home orders screen was very different to the final product. Initially, I created a window that contained all the required details, however was only basic and did not operate overly well.

After a review of the particular GUI, I decided to modify the window so that the present order details are shown in a more useful and easy to read format as shown below.



The table shown makes use of an Inner Join SQL query. This collated several tables in my database together, including the names from the Customer Information table, the statuses from the Status table, and the order number and date/time from the Orders table. This Inner Join increases the complexity of the coding and is suited for the situation as it brings information from multiple sources under one table. The coding for this is shown below.

```
s=""" SELECT DISTINCT Orders.FullOrderID, Orders.TimeandDate, CustomerInformation.FirstName, CustomerInformation.Surname, Status.Status
FROM Orders
INNER JOIN Status
ON Orders.StatusID=Status.StatusID
INNER JOIN CustomerInformation
ON Orders.CustomerID=CustomerInformation.CustomerID
WHERE Orders.StatusID < 6
"""
cur.execute(s)
self.data = cur.fetchall()
```

6th Stage – Creating the Admin account:

The admin account is also an elevated account, like the staff account. However, the admin account has a higher level of access to the inner workings of the program and consequently, additional GUIs and coding are required.

The admin login is given its own home screen in which the user can see all the orders in the system, and view each specific order and edit its status, the same as the staff account. The only difference is the administrator has the option to delete each order if they wish to do so.

One of my objectives was to be able to edit customer details, and be able to suspend customer accounts if it is deemed necessary. To implement this, I coded for the SQL Customer Information table to display to the admin, and be editable in real-time also. This means that the admin can simply change the required boxes in the table and press Save.

The code to load the window is demonstrated below.

```
def load_data(self):

    cur = con.cursor()
    j='select * from CustomerInformation'
    cur.execute(j)
    self.data = cur.fetchall()
    if len(self.data)>0:
        for jj in range(len(self.data)-1,-1):
            print("removing",jj)
            self.data.pop(jj)
            self.model.removeRow(index.row(self.data[jj]))
    self.model=QtGui.QStandardItemModel(self)
    self.tableView.setModel(self.model)
    for row in self.data:
                items = [
                    QtGui.QStandardItem(str(field))
                    for field in row
                ]
                self.model.appendRow(items)
```

To then update the actual database of any changes made, the following complex code is required.

```
    self.data=[]
    self.model = QtGui.QStandardItemModel(self)
    self.tableView.setModel(self.model)
    self.load_data()

def upd_record(self):
    index = self.tableView.currentIndex()
    cell_contents=index.data()

    idcust=self.model.data(self.model.index(index.row(), 0))
    print(idcust)
    firstname=self.model.data(self.model.index(index.row(), 1))
    surname=self.model.data(self.model.index(index.row(), 2))
    house=self.model.data(self.model.index(index.row(), 3))
    postcode=self.model.data(self.model.index(index.row(), 4))
    mobile=self.model.data(self.model.index(index.row(), 5))
    pay=self.model.data(self.model.index(index.row(), 6))
    user=self.model.data(self.model.index(index.row(), 7))
    password=self.model.data(self.model.index(index.row(), 8))
    hashedpass = hasher(password)
    typeaccount=self.model.data(self.model.index(index.row(), 9))
    statusaccount=self.model.data(self.model.index(index.row(), 10))
    question=self.model.data(self.model.index(index.row(), 11))
    answer=self.model.data(self.model.index(index.row(), 12))

    print(firstname,surname,house, postcode, mobile, pay, user, password, typeaccount)
    self.model.data(self.model.index(4,1))
    s='UPDATE CustomerInformation SET FirstName=?,Surname=?,HouseNumber=?,Postcode=?,MobileNumber=?,DeliveryType=?, Password=?, AccountType=?, AccountStatu
    cur.execute(s,(firstname,surname,house, postcode, mobile, pay, hashedpass, typeaccount, statusaccount, question, answer, idcust))
    con.commit()
    self.load_data()
```

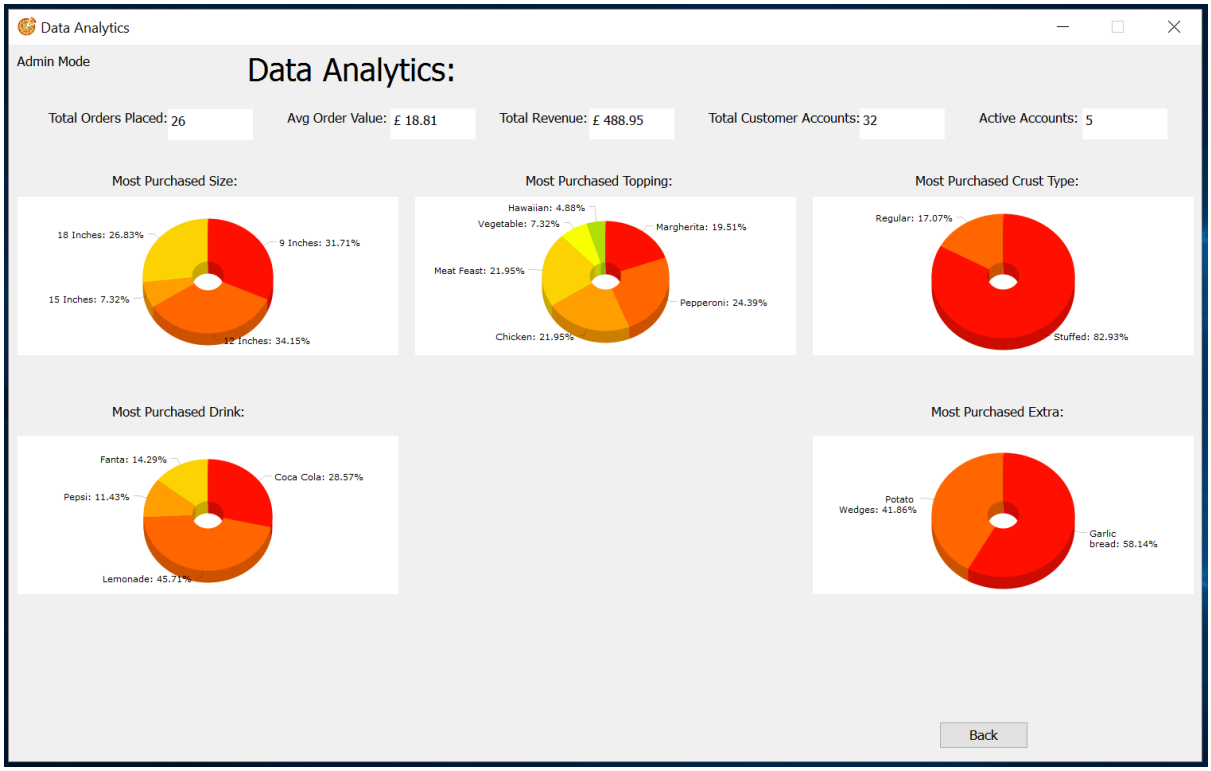
This takes the data from each row in the table view the admin has edited, and updates the database to reflect the changes made.

7th Stage – Adding complexity:

With my underlying program successfully functioning with the required functions in place, I reviewed my software and decided to add extra features that will increase the complexity of my program.

Firstly, I created the GUI for the Data Analytics section of my program. This consisted of a large, relatively empty window. I then used HTML to create multiple interactive pie charts. These load with an animation, show the percentage share of each item in the chart, respond to being hovered over with the mouse by giving exact values of the data and can be pressed to give another animation.

The final window for this is displayed. Due to the late stage of the programming that this GUI was developed it in, there is no prototype, because the final window was essentially the original.

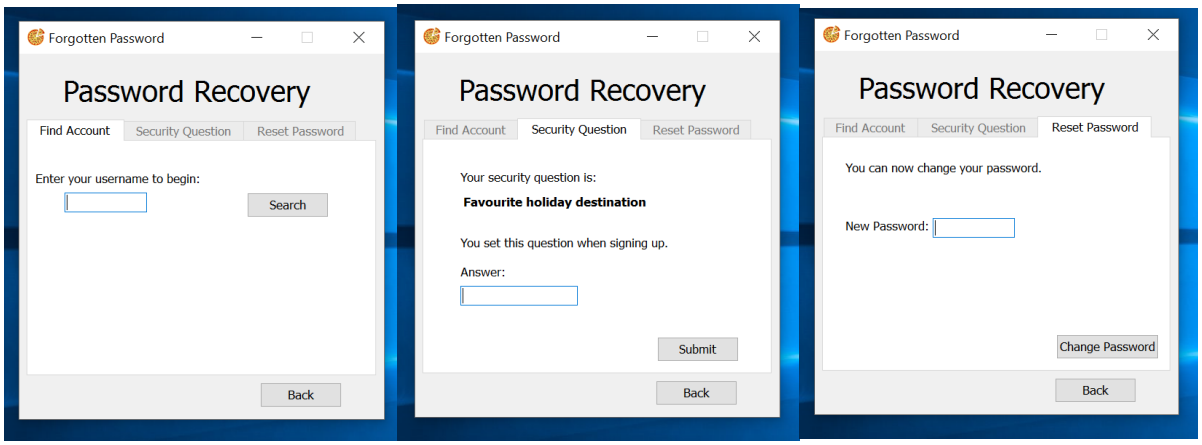


The window gives a brief overview of the business performance, including total/active customer accounts, total orders placed and the revenue that has created. There is also a more complex sum being used that then divides the total revenue by the amount of orders placed to work out the average order value – a useful statistic.

However, the main feature of the Data Analytics window is the HTML pie charts. These have been coded and then implemented through the PyQt web view function. This called the HTML coding and then displays the charts. The coding for this is below.

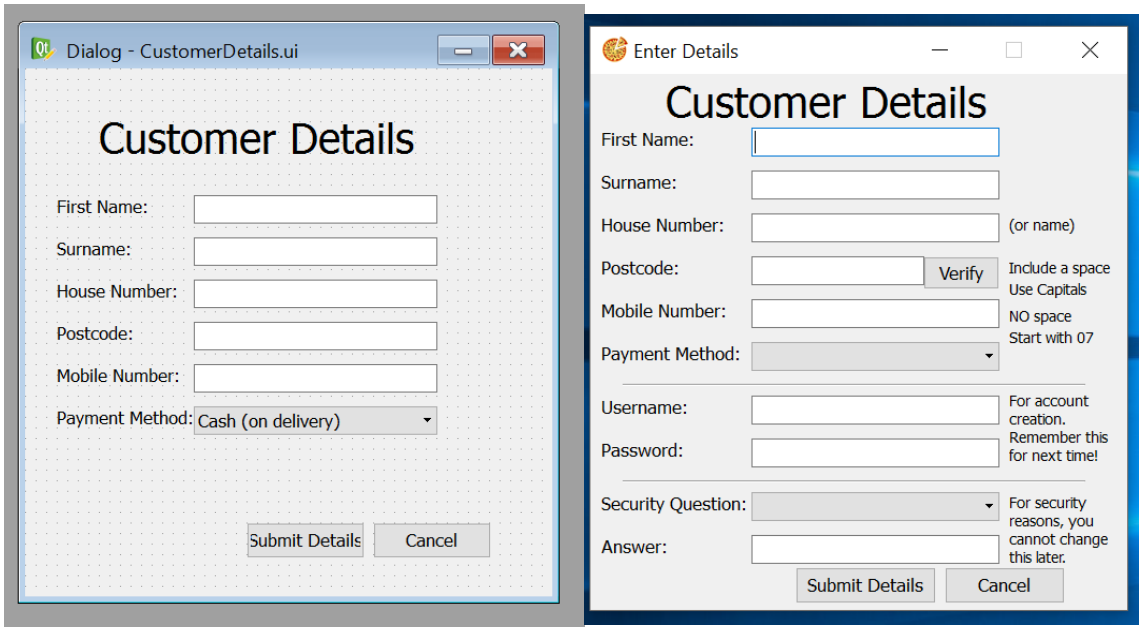
```
toppings="""
<!DOCTYPE html>
<html>
	<head>

		<!-- amCharts javascript sources -->
		<script type="text/javascript" src="https://www.amcharts.com/lib/3/amcharts.js"></script>
		<script type="text/javascript" src="https://www.amcharts.com/lib/3/pie.js"></script>


		<!-- amCharts javascript code -->
		<script type="text/javascript">
			AmCharts.makeChart("chartdiv",
				{
					"type": "pie",
					"angle": 11.7,
					"balloonText": "[[title]]<br><span style='font-size:14px'><b>[[value]]</b> ([[percents]]%)</span>",
					"depth3D": 15,
					"innerRadius": 20,
					"labelRadius": 10,
					"minRadius": 0,
					"marginBottom": 0,
					"marginTop": 0,
					"outlineThickness": 2,
					"pullOutOnlyOne": true,
					"startDuration": 4,
					"startEffect": "elastic",
					"titleField": "category",
					"valueField": "column-1",
					"theme": "default",
					"allLabels": [],
					"balloon": {
						"animationDuration": 0,
						"borderThickness": 1,
						"fadeOutDuration": 0,
						"fontSize": 0,
						"maxWidth": 0,
						"pointerWidth": 0
					},
					"legend": {
						"enabled": false,
						"accessibleLabel": "",
						"align": "center",
						"labelText": "",
						"markerType": "circle",
						"rollOverGraphAlpha": 0
					},
					"titles": [],
					"dataProvider": [
						{
							"category": "Margherita",
							"column-1": z1
						},
						{
							"category": "Pepperoni",
							"column-1": z2
						},
						{
							"category": "Chicken",
							"column-1": z3
						},
						{
							"category": "Meat Feast",
							"column-1": z4
						},
						{
							"category": "Vegetable",
							"column-1": z5
						},
						{
							"category": "Hawaiian",
							"column-1": z6
						}
					]
				});
		</script>
	</head>
	<body>
		<div id="chartdiv" style="width: 100%; height: 400px; background-color: #FFFFFF;" ></div>
	</body>
</html>
"""

	toppings=toppings.replace("z1",str(margcount))
	toppings=toppings.replace("z2",str(peppcount))
	toppings=toppings.replace("z3",str(chickcount))
	toppings=toppings.replace("z4",str(meatcount))
	toppings=toppings.replace("z5",str(vegcount))
	toppings=toppings.replace("z6",str(hawcount))
	self.webViewtopping.setHtml(toppings)
```

Another feature I added to my program was the password recovery menu. This was a window that contained multiple tabs that automatically move along as the correct data is inputted. The program now asks for a security question when signing up, which is stored in the database and is used to recover an account if necessary.



The tabs intuitively help the user along, whilst the tool is highly powerful and extremely effective at recovering lost accounts. It a real-life scenario, it would be used to save the system administrator from having to constantly be resetting passwords that people forget.

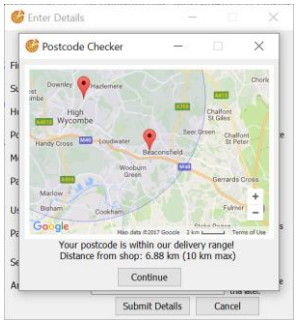
The signup window has seen one of the biggest changes throughout the development of my program. The below screenshots show the prototype and the final version of this.



This shows the vast development my program has undertaken – the inclusion of an account username and password later on in the software progression, a security question to recover lost passwords, pointers telling customers how to correctly fill in the boxes and a verify button that uses the Google Maps API to plot a customer's postcode on the map and determine whether they are eligible for home delivery based on distance from the shop.

The Google Maps API is utilised to plot the 'pizza shop' (RGSHW) and also to plot the postcode of the customer's address. This flagship feature can then tell the user whether or not they are close enough (<10km radius) to the shop to get delivery.

This feature is activated by first entering the required postcode and then pressing the Verify button. This is shown below.



The Google Maps API is called by coding in HTML. The code is implemented and is used to load the map and place the shop marker and its 10km radius, and to figure out the location of the postcode entered.

The code for the initialisation of the map is below.

```
<!doctype html>
<html>
   <head>

      <script src="http://maps.googleapis.com/maps/api/js?key=APIKEYHERE&sensor=false&libraries=geometry" type="text/javascript"></script>
                                    <script>
                                        var geocoder;
                                        var map;

                                        function initialize() {
                                          geocoder = new google.maps.Geocoder();
                                          var latlng = new google.maps.LatLng(51.6409256,-0.73800);
                                          var mapOptions = {
                                            zoom: 11,
                                            center: latlng,
                                            disableDefaultUI: true,
                                            zoomControl: true,
                                            mapTypeControl: false,
                                            scaleControl: true,
                                            streetViewControl: false,
                                            rotateControl: false,
                                            fullscreenControl: false,
                                            mapTypeId: google.maps.MapTypeId.ROADMAP
                                          }
                                          map = new google.maps.Map(document.getElementById('map-canvas'), mapOptions);
                                          var shoplocation = new google.maps.LatLng(51.6409256,-0.73800);
                                          var shopmarker = new google.maps.Marker({position:shoplocation});
                                          shopmarker.setMap(map);
                                          var radius = new google.maps.Circle({
                                          center: shoplocation,
                                          radius: 10000,
                                          strokeColor: "#0000FF",
                                          strokeOpacity: 0.4,
                                          strokeWeight: 1,
                                          fillColor: "#0000FF",
                                          fillOpacity: 0.05
                                          });
                                          radius.setMap(map);
```

The URL at the top of the coding is used to connect to the Google API, and contains my API key that is required to access Google Maps API. Other parts of the code includes “ var mapOptions = { “, which then contains all the controls on the map, such as the ability to remove the Street View function, the zoom control, the map type and where the centre of the Map is.

The next section of code is responsible for finding the coordinates of the postcode entered and plotting the marker on the map:

```
}

function codeAddress() {
  var address = 'z1';
  geocoder.geocode( { 'address': address}, function(results, status) {
    if (status == google.maps.GeocoderStatus.OK) {
      map.setCenter(results[0].geometry.location);
      var newmarker = results[0].geometry.location;

      var marker = new google.maps.Marker({
          map: map,
          position: results[0].geometry.location
      });
      var shopcoord = new google.maps.LatLng(51.6409256,-0.73800);

      var p2 = new google.maps.LatLng(51.6409256,-0.73800);
```

I have then utilised another API, from “api.postcodes.io/postcodes/”, which is responsible for outputting the coordinates of the postcode into the Python part of the program, and then the Haversine formula is used to work out the straight line distance between the shop coordinates and the customer’s coordinates:

```
try:
    from urllib.request import urlopen     #import modules to access a webpage
    response = urlopen("https://api.postcodes.io/postcodes/%s" %(postcode))  #Calls separate postcode API
except:
    verifiedpostcode=False    #HTML webpage displayed in case of failure
    self.rangelabel.setText("Failed to determine eligibility.")
    self.webView.setHtml("""
                         <center><h1>An Error Occured</h1></center>

                         <center><p>The Postcode Checker cannot find this address.</p></center>

                         <p>This could be for a number of reasons:</p>
                         <ul>
                         <li>The postcode you entered is invalid - Check your postcode and try again.</li>
                         <br>

                         <li>You have no internet connection - Check your connection and try again.</li>
                         </ul>
                         """)

else:
    full=response.read().decode("utf8")   #If response received, it is decoded
    import json   #import json module
    result1 = json.loads(full)
    lat=result1['result']['latitude']   #Pick out logitude and latitude from data
    long=result1['result']['longitude']
    from math import radians, cos, sin, asin, sqrt    #import maths functions
    def haversine(lon1, lat1, lon2, lat2):          #Function for haversine formula, used to work out distance between 2 coords
        lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
        dlon = lon2 - lon1
        dlat = lat2 - lat1
        a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
        c = 2 * asin(sqrt(a))
        r = 6371
        return c * r
    traveldist=haversine(long, lat, -0.73800, 51.6409256)        #Calls function and passes in parameters
```

The program then decides whether the postcode is suitable in the following code, and changes the GUI labels and button text to suit.

```
if verifiedpostcode==True:
    self.rangelabel.setText("Your postcode is within our delivery range!")
    self.okbutton.setText("Continue")
else:
    self.rangelabel.setText("Your postcode is outside our delivery range! \n It is not possible to deliver to this address.")
    self.okbutton.setText("Go back")
```

If “verifiedpostcode=True”, then the program will allow the user to make an account, otherwise the user cannot make an account until a valid postcode is entered.

Testing to inform development:

To assist with the iterative development of my program, I tested my program at each stage of the process. This involved testing all the functions added to my program in that stage to ensure they worked as expected and without crashing the program.

The stages below relate to the stages described in the above section.

Stage 1 – Initial Stage of Development

The initial stage involved loading a simple GUI window, which proved challenging because it was the first window so there was no prior working code to reuse.

To test this, I ran the code to check it worked.

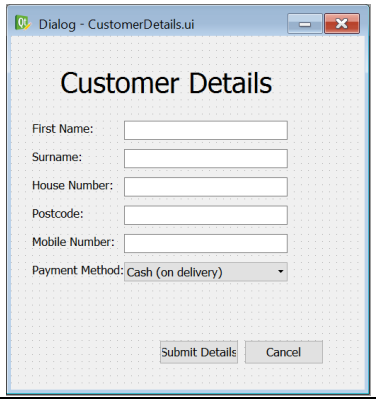
The table below lists the different tests, how they were tested and whether they were successful. I have also detailed any remedial action taken when required.

What testing:	**How tested:**	**Successful? (+Remedial Action taken if not successful)**
- Does the Home Screen load correctly **Link to Objectives:** "49) To ensure that the PyQt windows interact properly with the Python coding, which in turn interacts with the SQL database"	- Ran the program to see whether the window rendered at all/as expected	- Yes, as shown below 

Stage 2 – Linking Windows

In the second stage of development, I linked the windows so that the appropriate new window opens when a button is pressed.

What testing:	**How tested:**	**Successful? (+Remedial Action taken if not successful)**
- Does the 'Log In' button load the login window	- Pressed the Log-In button on the Home Screen window	- Yes

Link to Objectives: "49) To ensure that the PyQt windows interact properly with the Python coding, which in turn interacts with the SQL database"		
- Does the 'Sign Up' button load the signup window **Link to Objectives:** "49) To ensure that the PyQt windows interact properly with the Python coding, which in turn interacts with the SQL database"	- Pressed the Sign Up button on the Home Screen window	- No, I resolved the issue by correcting my coding. The Sign Up window could not be found because the spelling of the .ui file was incorrect. Once corrected, the program worked as expected, as shown below 

Stage 3 – Connecting to database

When testing the ability to connect to the database, I first designed the database and created some tables. I then attempted to initialise the connection, as well as read some data from a table.

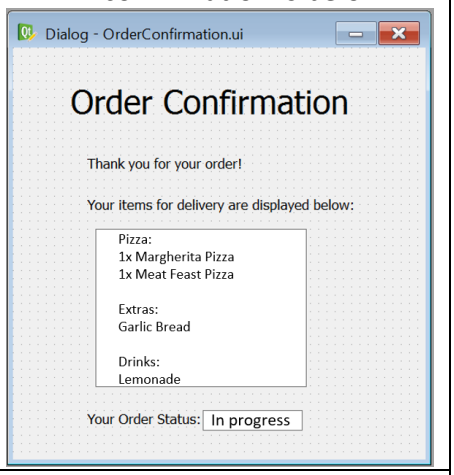
What testing:	**How tested:**	**Successful? (+Remedial Action taken if not successful)**
- Connecting to database **Link to Objectives:** "48) To implement the	- Ran the program to see whether a connection could be initialised	- Yes

database SQL to store the information output by the running program"		
- Reading from database tables **Link to Objectives:** "48) To implement the database SQL to store the information output by the running program"	- Used a SELECT SQL query to retrieve a piece of data from the database	- Yes

Stage 4 - Focusing on Customer side of program

This was a large proportion of the programming, with lots of GUIs and coding required. There is therefore a large amount of testing in this stage.

What testing:	**How tested:**	**Successful? (+Remedial Action taken if not successful)**
- That all the UIs open as expected, and the buttons properly link the windows together **Link to Objectives:** "49) To ensure that the PyQt windows interact properly with the Python coding, which in turn interacts with the SQL database"	- Ran the program. Pressed each button and opened each window	- Yes
- The customer can login **Link to Objectives:** "18) To store their details in a database, and allow them to login to the system with a username and password which will bring up their details to save them re-entering them when they make a second order"	- Used a customer account username and password to gain access	- Yes

- Ability to place an order and get a confirmation **Link to Objectives:** "16) To show the user a summary of their order when completed"	- Go through order process	- Yes, the working order confirmation is below 
- Ability to recall a previous order	- Entered a previous order number	- Yes
- Ability to print an order confirmation	- Placed an order and pressed Print button	- Yes
- Ability to view and edit personal customer details **Link to Objectives:** "31) The customer can change their personal details once they have an active account already, should they need to do so"	- Logged in on customer account, attempted to view personal details to see whether the program could retrieve the data from the database and make amendments if needed	- Partially, the data successfully loaded, however, when saving the data, the program overwrote all of the records in the database where the user had not re-entered their details. To solve this, I had to change the coding so that the database is only updated with new information if the input box has had data entered, otherwise the program was simply overwriting the database with empty slots

5th Stage – Developing the staff login

What testing:	How tested:	Successful? (+Remedial Action taken if not successful)
- The program recognises the high priority logon and loads the appropriate window **Link to Objectives:** "25) To create other types of accounts besides regular customer logons, such as specialised logins for pizza chefs and delivery staff"	- Used the staff username and password to access the system	- Yes
- That the staff account can see all orders that need completing **Link to Objectives:** "44) The staff account can see all the outstanding orders that require attention"	- Logged in and ensured that all the orders in the database table showed up	- Yes
- The ability to view each orders contents and then update its current status **Link to Objectives:** "45) The staff account can view what has been ordered in each order 46) The staff account can edit the status of each order"	- Selected an order in the list and altered its status	- Yes

6th Stage – Creating the Admin account

What testing:	How tested:	Successful? (+Remedial Action taken if not successful)
- The program recognises the high priority logon and loads the appropriate window **Link to Objectives:** "25) To create other types of accounts besides regular customer logons, such as specialised logins for pizza chefs and delivery staff"	- Used the administrator username and password to access the system	- Yes
- That the admin account can see all the orders in the system **Link to Objectives:** "25) To create other types of accounts besides regular customer logons, such as specialised logins for pizza chefs and delivery staff"	- Logged in and ensured that all the orders in the database table showed up	- Yes
- The ability to view each orders contents and then update its current status	- Selected an order in the list and altered its status	- Yes
- The ability to delete an order **Link to Objectives:** "32) The admin can delete orders"	- Viewed an order and pressed delete	- Yes
- Updating and suspending user accounts	- Attempted to change a user's details and then	- Yes

Link to Objectives: "22) Allow admins to suspend user accounts"	suspend their account	

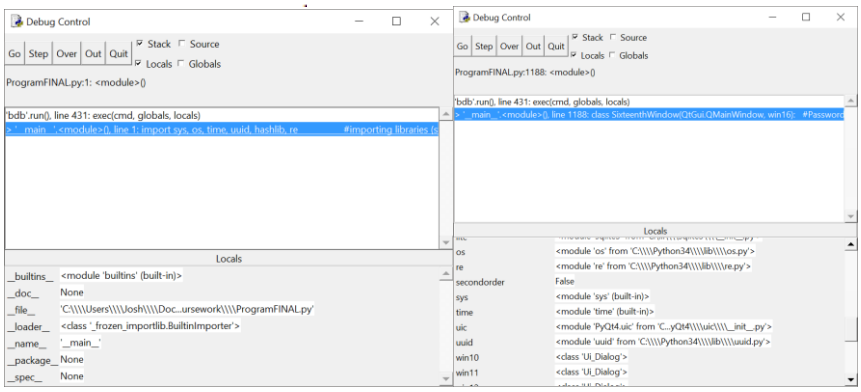
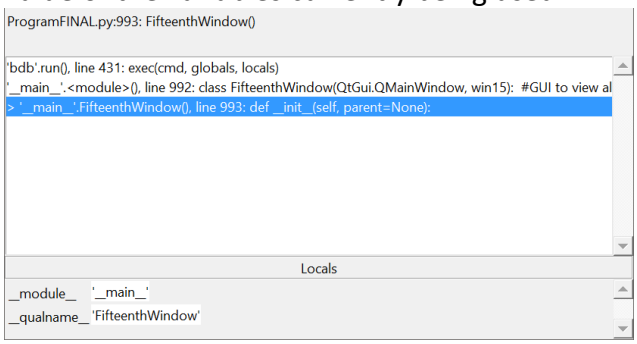
7th Stage – Adding complexity

What testing:	**How tested:**	**Successful? (+Remedial Action taken if not successful)**
- The sign up window works and requests all the required details **Link to Objectives:** "6) To allow the end user to select a payment method of their choice when making an account 7) To allow the end user to input their name when making an account 8) To allow the end user to input their house number and postcode when making an account 9) To allow the end user to input their username when making an account 10) To allow the end user to input their password when making an account 11) To allow the end user to input their security question and answer when making an account 12) To allow the end user to input their mobile number when making an account"	- Created a new account	- Yes

- The Google Maps API postcode checker works **Link to Objectives:** "13) The program will plot the user's postcode on top of Google Maps using the API to show the users delivery point in relation to the pizza shop (RGSHW) 14) The program will use the Haversine Formula to work out the straight line distance between 2 coordinates on the map to work out the distance between them 15) The program will only allow an account to be made if the user is within a specified distance from the shop, set at a 10km radius"	- Entered a UK postcode. - The program then lets you know whether it is a valid postcode based on distance from the shop.	- Yes. The API was used to show the map of the local area, and the API plots the postcode of the customer on the map, along with the marker of the "pizza shop" (RGSHW) and a 10km radius around it.
- That the data analytics works as expected, correctly rendering the pie charts **Link to Objectives:** "33) The admin can view data analytics which provides information about how the system is performing"	- Loaded the data analytics window	- Not initially. The HTML coding was new to me and required lots of prior learning and took a while to code. However, all the 5 pie charts now work successfully and are visually pleasing and highly informative
- That the password recovery window takes then user through the process of accessing their lost account **Link to Objectives:**	- Attempted to recover a password for a customer's account	- Yes

“52) Customers can recover their password if they forget it”		

Evidence of program features:

Feature:	Coding Evidence / Actual Testing:	Link to Objectives:
Using ONE of: debugger, variable watch, breakpoints	**Proof of using debugger during development:** >>> [DEBUG ON] >>>  Debugger was used to see the values of the local variables as the program progressed. The program could be ‘stepped’ so that it went through each line of code one at a time. The debugger shows which procedure is currently running, and also holds the value of the variables currently being used:  This was useful during the development of my program, as it allowed me to run through the program line by line to fix known errors far more easily than would have otherwise been possible.	

Comments for every loop, selection or procedure	Loop:
	`for each in allnames:`
	`  if usernamebox== each[0]:  #This stops a username being used twice`
	`  QtGui.QMessageBox.information(self, "Error!", "This username is alread`
	`  used = True`	

	Selection:
	`sql = """SELECT "ProductLineID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)  #Finds all productlines associated with the order`
	Procedure:
	`def back(self):  #Back function`
	`  self.hide()`
	`  self.newwindow=FourteenthWindow()`
	`  self.newwindow.show()`	

Parameter passing	`def hasher(password):  #The hash function used to hash a password`
	`  salt = uuid.uuid4().hex`
	`  return hashlib.sha256(salt.encode()+password.encode()).hexdigest()+":"+salt`
		50) To make the coding as efficient as possible (e.g.: passing parameters, reusable procedures), with no duplicate coding or other inefficient techniques being used

Presence and data type check contained in reusable procedures		

Mix of functions and procedures (or methods that return and don't return values)	Function:
	`def haversine(lon1, lat1, lon2, lat2):  #Function for haversine`
	`  lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])`
	`  dlon = lon2 - lon1`
	`  dlat = lat2 - lat1`
	`  a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2`
	`  c = 2 * asin(sqrt(a))`
	`  r = 6371`
	`  return c * r`
	Procedure: (No return of value)
	`def back(self):  #Back function`
	`  self.hide()`
	`  self.newwindow=FourteenthWindow()`
	`  self.newwindow.show()`	

Variables of different scopes: class, object, local, global	Class:
	`class EighteenthWindow(QtGui.QMainWindow, win18):`
	`  def __init__(self, parent=None):`	

<table>
<tr><td></td><td>Local:

```
submit(self):
firstnamebox = self.FirstName.text()
surnamebox = self.Surname.text()
```

Global:

```
global usernameoptionsid
usernameoptionsid=""
```

</td><td></td></tr>
<tr><td>Use of constants with capitalised names</td><td></td><td></td></tr>
<tr><td>Use of modules imported as needed.</td><td>

```
import sys, os, time, uuid, hashlib, re
from PyQt4 import QtCore, QtGui, uic, QtSql
import sqlite3 as lite
from urllib.request import urlopen
import json
from math import radians, cos, sin, asin, sqrt
```

Proof: (Of time module usage)

```
timedate=time.strftime("%H:%M")+ " " + time.strftime("%d/%m/%Y")
```

Proof: (Of REGEX module usage)

```
namerule = re.compile(r'^[a-zA-Z]+$')          #Vali
if not namerule.search(firstnamebox + surnamebox):
```

Proof: (Of PyQt4 modules)

```
class SecondWindow(QtGui.QMainWindow, win2):
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
```

Proof: (Of json module usage)

```
result1 = json.loads(full)
lat=result1['result']['latitude']
long=result1['result']['longitude']
```

Proof: (Of SQLite modules)

```
con = lite.connect('PizzaDatabase.db')
cur = con.cursor()
```

</td><td></td></tr>
</table>

<table>
<tr>
<td></td>
<td>

Proof: (Of uuid and hashlib modules)

```
def hasher(password):   #The hash function used to hash a password
    salt = uuid.uuid4().hex
    return hashlib.sha256(salt.encode()+password.encode()).hexdigest()+":"+salt
```

</td>
<td></td>
</tr>
<tr>
<td>Log in and out with hashed password data</td>
<td>

```
if self.verifyhash(passwordattempt, passwordoptions[0][0]):
```

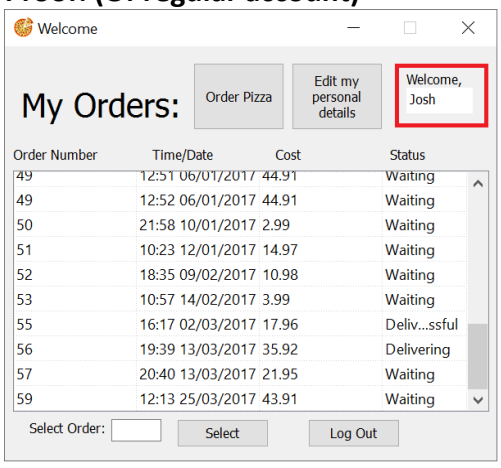
Proof: (Shows hashed passwords stored in database)

UserName	Password
Filter	Filter
admin	a0d3810260c0ff26b6ec8925dd18e093e51d7d868c42ea4847f81aa493d09673:d908c199b70447bb844ce5dbc2e2c666
staff	e537c9584690a3c93358741ae08ffdc0c52d9a10aa6925bdada69a09070ed3ac:d8dd3675ad2949fe94063001f0295489
josh	2c3ffdda0478c6239e61ff10fadcad532276ff098b1ebae8234d0eb01dfc74f0:524bfd9706644b3f92b34ad9fe188121

</td>
<td>53) To hash the stored passwords in the database for added security</td>
</tr>
<tr>
<td>Two types of user accounts: admin and regular. Admin can do things to regular accounts.</td>
<td>

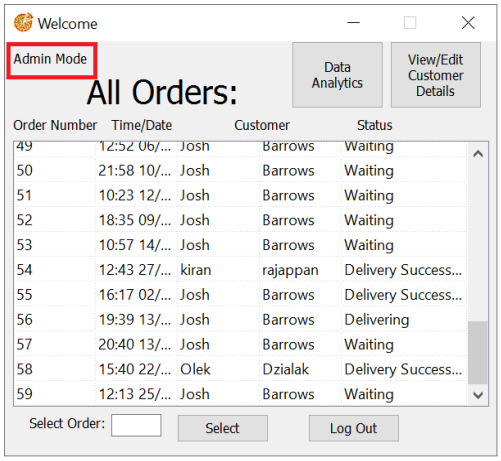
```
if accounttype[0][1] == "Active":
    if accounttype[0][0] == "Customer":
        self.Order = TenthWindow()
        self.Order.show()
    elif accounttype[0][0] == "Staff":
        self.Order = EighthWindow()
        self.Order.show()
    elif accounttype[0][0] == "Admin":
        self.Order = FourteenthWindow()
        self.Order.show()
```

Proof: (Of regular account)

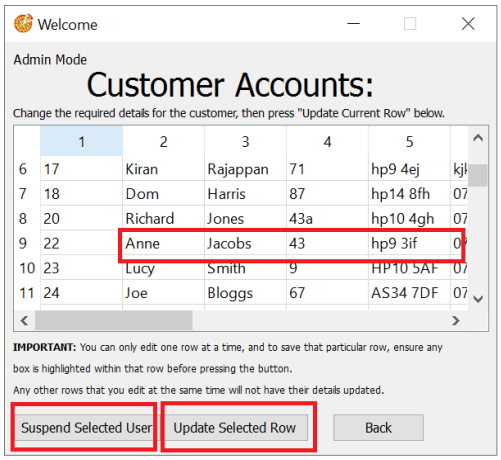


</td>
<td>25) To create other types of accounts besides regular customer logons, such as specialised logins for pizza chefs and delivery staff</td>
</tr>
</table>

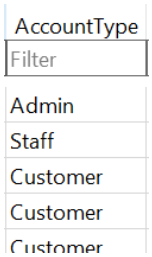
Proof: (Of admin account)



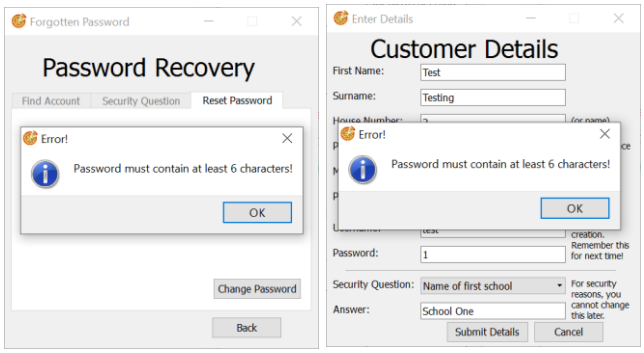
Admins can do things to regular accounts:
Admins can change each users' details, as well as suspend/reactivate customer accounts.



Proof of different account statuses saved in database:



Passwords enforced/ checked for strength.	`if len(passwordbox) <6: #Validation of password` `QtGui.QMessageBox.information(self, "Error!", "Password must contain at least 6 characters!")`	54) Passwords entered when creating

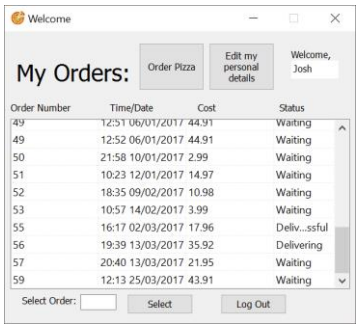
<table>
<tr><td></td><td>Proof:
 
</td><td>an account must be checked for strength</td></tr>
<tr><td>Sql parameter queries (with question marks)</td><td>

```
sqladdproductline = """INSERT INTO "Orders" ("FullOrderID", "ProductLineID", "CustomerID", "TimeandDate", "StatusID") VALUES(?, ?, ?, ?, "1")"""
```

</td><td>48) To implement the database SQL to store the information output by the running program</td></tr>
<tr><td>Inner join or nested sql queries</td><td>

```
        s=""" SELECT DISTINCT Orders.FullOrderID, Orders.TimeandDate, Orders.Cost, Status.Status
FROM Orders
INNER JOIN Status
ON Orders.StatusID=Status.StatusID
WHERE Orders.CustomerID = '%d'
        """ %(usernameoptionsid[0])   #This is another Inner Join query, bringing together releva
        cur.execute(s)
```

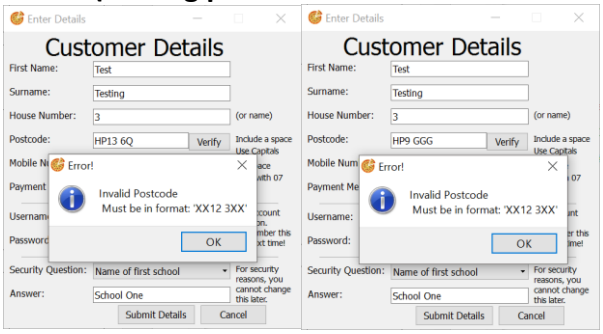
Proof: (The below table in the GUI is from the code above)

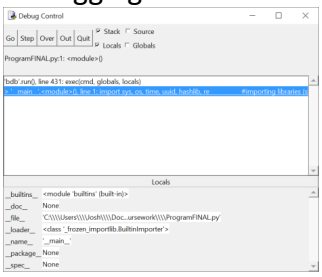

</td><td></td></tr>
<tr><td>Inheritance</td><td></td><td></td></tr>
<tr><td>Hungarian notation</td><td>

```
global currentdetails         #Makes the i
currentdetails = cur.fetchall()
global mobilenumstr
global housenumstr
housenumstr = str(currentdetails[0][2])
mobilenumstr = str(currentdetails[0][4])
```

</td><td></td></tr>
</table>

	The variables 'mobilenumstr' and 'housenumstr' are both the string versions of their integer counterparts and their data type has been included in their name for extra clarity.	

Regular expressions	`def verify(self):  #Function to verify the postcode entered`
	`  postcodebox2 = self.Postcode.text()`
	`  postcoderule = re.compile(r'^([A-PR-UWYZ0-9][A-HK-Y0-9][AEHMNPRTVXY0-9]?[ABEHMNPRVWXY0-9]? {1,2}[0-9][ABD-HJLN-UW-Z]{2}|GIR 0AA)$')`
	`  if not postcoderule.search(postcodebox2):`
	`  QtGui.QMessageBox.information(self, "Error!", "Invalid Postcode \n Must be in format: 'XX12 3XX'")  #Validation using REGEX`
	
	Proof: (Testing postcode validation in screenshots below)
	 
		55) All necessary inputs are validated using Regular Expressions, so that the program cannot be crashed (e.g.: postcode validation, mobile number validation, etc)

Multiple inheritance		
Debugging mode when program outputs what procedure is running and values of variables which get saved to disk file.	Debugging mode:  What procedure is running: ProgramFINAL.py:288: FifthWindow() 'bdb'.run(), line 431: exec(cmd, globals, locals) '__main__'.<module>(), line 287: class FifthWindow(QtGui.QMainWindow, win5): #This is the Order (> '__main__'.FifthWindow(), line 288: def __init__(self, parent=None): #Constructor method Values of variables: secondorder False sys <module 'sys' (built-in)>	
Shortest path algorithm		
Polymorphism		

Final Code:

```
import sys, os, time, uuid, hashlib, re              #importing libraries (system, operating system, hashing functions, regex)
from PyQt4 import QtCore, QtGui, uic, QtSql          #importing relevant PyQt4 modules
import sqlite3 as lite                               #importing library to connect to SQLite database
con = lite.connect('PizzaDatabase.db')               #Connecting to database
cur = con.cursor()                                   #Setting cursor of database

win2 = uic.loadUiType("LogIn.ui") [0]                #Loading all required GUIs
win3 = uic.loadUiType("CustomerDetails.ui") [0]
win4 = uic.loadUiType("PizzaOrder.ui") [0]
win5 = uic.loadUiType("OrderConfirmation.ui") [0]
win8 = uic.loadUiType("ViewOrdersScreen.ui") [0]
win9 = uic.loadUiType("OrderView.ui") [0]
win10 = uic.loadUiType("ViewUserOrders.ui") [0]
win11 = uic.loadUiType("UserOrder.ui") [0]
win12 = uic.loadUiType("DetailsAccepted.ui") [0]
win13 = uic.loadUiType("UpdateDetails.ui") [0]
win14 = uic.loadUiType("AdminScreen.ui") [0]
win15 = uic.loadUiType("ViewAllDetails.ui") [0]
win16 = uic.loadUiType("PasswordRecovery.ui") [0]
win17 = uic.loadUiType("DataAnalytics.ui") [0]
win18 = uic.loadUiType("PostcodeChecker.ui") [0]

class SecondWindow(QtGui.QMainWindow, win2):         #loads the login window
    def __init__(self, parent=None):                 #initialising the window (constructor)
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)                           #Setting up the UI
        self.loginbutton.clicked.connect(self.login)    #Defining what happens when each button is pressed
        self.PassWord.returnPressed.connect(self.login)
        self.signupbutton.clicked.connect(self.signup)
        self.forgot.clicked.connect(self.forgotpass)
    def signup(self):                                #Sign Up function - activated when "Sign Up" button pressed
        self.signupwindow = ThirdWindow()
        self.signupwindow.show()
        self.hide()
    def verifyhash(self, userpass, storedpass):      #Verifies the hash
        self.userpass = userpass
        try:                                         #Prevents crash in instance of invalid stored hash
            password,salt=storedpass.split(":")
        except:
            pass
        else:
            data = []
            data.append(password)
            data.append(hashlib.sha256(salt.encode()+self.userpass.encode()).hexdigest())
            return data[0]==data[1]
    def login(self):                                 #Login function
        global usernameoptionsid
        usernameoptionsid=""
        usernameoptions=""
        usernameattempt = self.UserName.text()
        passwordattempt = self.PassWord.text()
        if usernameattempt == "":
            self.hide()
            self.Order2= SecondWindow()
            self.Order2.show()
            QtGui.QMessageBox.information(self, "Incorrect Details", "Your username/password is incorrect. \n Please try again...")
        else:
            usernamesql = """SELECT "UserName" FROM "CustomerInformation" WHERE "UserName" = '%s' """ % (usernameattempt)
            cur.execute(usernamesql)
            usernameoptions = cur.fetchall()
            username2sql = """SELECT "CustomerID" FROM "CustomerInformation" WHERE "UserName" = '%s' """ % (usernameattempt)
            cur.execute(username2sql)
            usernameoptionjj = cur.fetchall()
            usernameoptionsid = usernameoptionjj
            try:
                passwordsql = """SELECT "Password" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ % (usernameoptionsid[0])
                cur.execute(passwordsql)
                passwordoptions = cur.fetchall()
            except:
                self.hide()
                self.Order2= SecondWindow()
                self.Order2.show()
                QtGui.QMessageBox.information(self, "Incorrect Details", "Your username/password is incorrect. \n Please try again...")
            else:
                if self.verifyhash(passwordattempt, passwordoptions[0][0]): #Compares inputted password and the stored password
                    self.hide()
```

```
            accountypesql = """SELECT "AccountType", "AccountStatus" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ %
(usernameoptionsid[0])
            cur.execute(accountypesql)
            accounttype = cur.fetchall()
            if accounttype[0][1] == "Active":
               if accounttype[0][0] == "Customer":    #Opens the appropiate window based ona the account type of the user
                  self.Order = TenthWindow()
                  self.Order.show()
               elif accounttype[0][0] == "Staff":
                  self.Order = EighthWindow()
                  self.Order.show()
               elif accounttype[0][0] == "Admin":
                  self.Order = FourteenthWindow()
                  self.Order.show()
            else:                              #If account is not active, the useris denied access
               self.Order2= SecondWindow()
               self.Order2.show()
               QtGui.QMessageBox.information(self, "Account Suspended", "This account has been suspended. \n Contact the system administrator.")
         else:
            self.hide()
            self.Order2= SecondWindow()
            self.Order2.show()
            QtGui.QMessageBox.information(self, "Incorrect Details", "Your username/password is incorrect. \n Please try again...")
   def forgotpass(self):  #Function activated when Forgot Password button clicked
      self.hide()
      self.newwindow=SixteenthWindow()
      self.newwindow.show()

class ThirdWindow(QtGui.QMainWindow, win3):                  #Loads sign up window
   def __init__(self, parent=None):                     #Initialises window
      QtGui.QMainWindow.__init__(self, parent)
      self.setupUi(self)
      self.CDOrderDone.clicked.connect(self.submit)         #Controls buttons
      self.CDCancel.clicked.connect(self.cancel)
      self.verifypostcode.clicked.connect(self.verify)
      global verifiedpostcode
      verifiedpostcode=False
   def verify(self):                                 #Function to verify the postcode entered
      postcodebox2 = self.Postcode.text()
      postcoderule = re.compile(r'^([A-PR-UWYZ0-9][A-HK-Y0-9][AEHMNPRTVXY0-9]?[ABEHMNPRVWXY0-9]? {1,2}[0-9][ABD-HJLN-UW-Z]{2}|GIR 0AA)$')
      if not postcoderule.search(postcodebox2):
         QtGui.QMessageBox.information(self, "Error!", "Invalid Postcode \n Must be in format: 'XX12 3XX'")   #Validation using REGEX
      else:
         global postcodetoverify
         postcodetoverify=postcodebox2
         self.homescreen = EighteenthWindow()
         self.homescreen.show()
   def submit(self):
      firstnamebox = self.FirstName.text()          #Store data from each input box
      surnamebox = self.Surname.text()
      housenumberbox = self.HouseNumber.text()
      postcodebox = self.Postcode.text()
      mobilenumberbox = self.MobileNumber.text()
      deliverybox = self.PaymentMethod.currentText()
      usernamebox = self.Username.text()
      passwordbox = self.Password.text()
      hashedpass = hasher(passwordbox)
      question = self.questionbox.currentText()
      answer = self.answerbox.text()
      query = QtSql.QSqlQuery()
      if firstnamebox == "" or surnamebox == "" or housenumberbox == 0 or postcodebox == "" or mobilenumberbox == "" or deliverybox == "" or usernamebox
== "" or passwordbox == "" or question == "" or answer == "":
         QtGui.QMessageBox.information(self, "Error!", "Some details have been left blank. \n Please check your information...")
      else:
         namerule = re.compile(r'^[a-zA-Z]+$')        #Validation of customer name using REGEX
         if not namerule.search(firstnamebox + surnamebox):
            QtGui.QMessageBox.information(self, "Error!", "Invalid Name \n Can only contain letters")
         else:
            postcoderule = re.compile(r'^([A-PR-UWYZ0-9][A-HK-Y0-9][AEHMNPRTVXY0-9]?[ABEHMNPRVWXY0-9]? {1,2}[0-9][ABD-HJLN-UW-Z]{2}|GIR 0AA)$')
            if not postcoderule.search(postcodebox):   #Validation of postcode using REGEX
                  QtGui.QMessageBox.information(self, "Error!", "Invalid Postcode \n Must be in format: 'XX12 3XX'")
            else:
               mobilerule = re.compile(r'^(07\d{9,9})$')   #Validation of mobile number using REGEX
               if not mobilerule.search(mobilenumberbox):
                  QtGui.QMessageBox.information(self, "Error!", "Invalid Mobile Number \n Can only contain 11 numbers, starting with '07'")
               else:
                  checkusername = """SELECT "UserName" FROM "CustomerInformation" """
                  cur.execute(checkusername)
                  allnames = cur.fetchall()
```

```
                used = False
                for each in allnames:
                    if usernamebox== each[0]:      #This stops a username being used twice
                        QtGui.QMessageBox.information(self, "Error!", "This username is already in use. \n Please choose another...")
                        used = True
                if used == False:
                    if len(passwordbox) <6:       #Validation of password
                        QtGui.QMessageBox.information(self, "Error!", "Password must contain at least 6 characters!")
                    else:
                        if verifiedpostcode==False:    #Ensures postcode has been verified
                            QtGui.QMessageBox.information(self, "Error!", "You need to verify your postcode first!")
                        else:
                            try:
                                if postcodetoverify==postcodebox:
                                    self.Order = TwelthWindow()
                                    self.Order.show()
                                    sql2 = """SELECT "CustomerID" FROM "CustomerInformation" ORDER BY CustomerID DESC LIMIT 1 """
                                    cur.execute(sql2)
                                    response = 0
                                    response = cur.fetchone()
                                    newcustid = response[0] + 1
                                    query = QtSql.QSqlQuery()     #Input user data into database
                                    sql = """INSERT INTO "CustomerInformation" ("CustomerID", "FirstName", "Surname", "HouseNumber", "Postcode",
"MobileNumber", "DeliveryType", "UserName", "Password", "AccountType", "AccountStatus", "SecurityQuestion", "SecurityAnswer") VALUES ( ?, ?, ?, ?, ?, ?,
?, ?, ?, "Customer", "Active", ?, ?)"""
                                    cur.execute(sql, (newcustid, firstnamebox, surnamebox, housenumberbox, postcodebox, mobilenumberbox, deliverybox,
usernamebox, hashedpass, question, answer, ))
                                    con.commit()
                                    sqlname = """SELECT "FirstName" FROM CustomerInformation WHERE CustomerID = '%d'""" % (newcustid)
                                    global firstnamereq
                                    cur.execute(sqlname)
                                    firstnamereq = cur.fetchone()
                                    self.hide()
                                else:
                                    QtGui.QMessageBox.information(self, "Error!", "The current postcode entered is different to the verified one!")
                            except:
                                QtGui.QMessageBox.information(self, "Error!", "You need to verify your postcode first!")
    def cancel(self):
        self.hide()
        self.homescreen = SecondWindow()
        self.homescreen.show()

global secondorder
secondorder = False

class FourthWindow(QtGui.QMainWindow, win4):    #GUI to order pizza
    def __init__(self, parent=None):          #Initialises window
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.POCancel.clicked.connect(self.cancel)
        self.POOrderDone.clicked.connect(self.orderdone)
        self.POAddAnother.clicked.connect(self.another)
        global usernameoptionsid
        sqlname = """SELECT "FirstName" FROM CustomerInformation WHERE CustomerID = '%d'""" % (usernameoptionsid[0])  #Fetches the logged in users
name
        global firstnamereq
        cur.execute(sqlname)
        firstnamereq = cur.fetchone()
        self.NameBox.setPlainText(firstnamereq[0])    #Displays the users name in the GUI
    def cancel(self):
        self.hide()
        self.homescreen = TenthWindow()
        self.homescreen.show()
    def orderdone(self):
        global secondorder
        sizebox = self.Size.currentIndex()      #Takes order inputs from GUI
        crustbox = self.StuffedCrust.currentIndex()
        toppingbox = self.Topping.currentIndex()
        extrasbox = self.Extras.currentIndex()
        drinksbox = self.Drink.currentIndex()
        oktogo=True
        if sizebox==0 and crustbox==0 and toppingbox==0 and extrasbox==0 and drinksbox==0:        #Validation stopping empty order
            QtGui.QMessageBox.information(self, "Error!", "You have not chosen anything to order!")
            oktogo=False
        if sizebox > 0 or crustbox > 0 or toppingbox > 0:
            if sizebox==0 or crustbox == 0 or toppingbox == 0: #Ensures that an order is complete before submitting
                QtGui.QMessageBox.information(self, "Error!", "You must choose the Size, Crust Type and Topping if you are ordering pizza.")
                oktogo=False
        if oktogo==True:
```

```
            sql2 = """SELECT "ProductLineID" FROM "ProductLine" ORDER BY ProductLineID DESC LIMIT 1 """
            cur.execute(sql2)
            response = 0
            response = cur.fetchone()
            newproductid = response[0] + 1
            sql4 = """SELECT "OrderID" FROM "ProductLine" ORDER BY OrderID DESC LIMIT 1 """
            cur.execute(sql4)
            response1 = 0
            response1 = cur.fetchone()
            if secondorder == True:
                neworderid = response1[0]
            else:
                neworderid = response1[0] + 1
                secondorder = True
            query = QtSql.QSqlQuery()  #Inserts ordered items into database 'ProductLine' table
            sql = """INSERT INTO "ProductLine" ("ProductLineID", "OrderID", "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" ) VALUES ( ?, ?, ?, ?,  ?, ?, ?)"""
            cur.execute(sql, (newproductid, neworderid, sizebox, crustbox, toppingbox, extrasbox, drinksbox, ))
            con.commit()
            self.hide()
            self.done = FifthWindow()
            self.done.show()
    def another(self):   #This function is run if the user wants to order more than one pizza
        global secondorder
        sizebox = self.Size.currentIndex()
        crustbox = self.StuffedCrust.currentIndex()
        toppingbox = self.Topping.currentIndex()
        extrasbox = self.Extras.currentIndex()
        drinksbox = self.Drink.currentIndex()
        oktogo=True
        if sizebox==0 and crustbox==0 and toppingbox==0 and extrasbox==0 and drinksbox==0:  #This validates the order to ensure something has been entered
            QtGui.QMessageBox.information(self, "Error!", "You have not chosen anything to order!")
            oktogo=False
        if sizebox > 0 or crustbox > 0 or toppingbox > 0: #Ensures that the size, crust type and topping of a pizza are all selected
            if sizebox==0 or crustbox == 0 or toppingbox == 0:
                QtGui.QMessageBox.information(self, "Error!", "You must choose the Size, Crust Type and Topping if you are ordering pizza.")
                oktogo=False
        if oktogo==True:
                sql2 = """SELECT "ProductLineID" FROM "ProductLine" ORDER BY ProductLineID DESC LIMIT 1 """
                cur.execute(sql2)
                response = 0
                response = cur.fetchone()
                newproductid = response[0] + 1
                sql5 = """SELECT "OrderID" FROM "ProductLine" ORDER BY OrderID DESC LIMIT 1 """
                cur.execute(sql5)
                response = 0
                response = cur.fetchone()
                if secondorder == True:
                    neworderid = response[0]
                else:
                    neworderid = response[0] + 1
                    secondorder = True
                sql = """INSERT INTO "ProductLine" ("ProductLineID", "OrderID", "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" ) VALUES ( ?, ?, ?, ?, ?, ?, ?)"""
                cur.execute(sql, (newproductid, neworderid, sizebox, crustbox, toppingbox, extrasbox, drinksbox, ))
                con.commit()
                self.hide()
                self.anotheragain = FourthWindow()
                self.anotheragain.show()

class FifthWindow(QtGui.QMainWindow, win5):    #This is the Order Confirmation window
    def __init__(self, parent=None):  #Constructor method
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.updatebutton.clicked.connect(self.update)
        self.returnhome.clicked.connect(self.home)
        self.printbutton.clicked.connect(self.printit)
        items  =[]
        overallpizza = 0
        drinkslist = 0
        extraslist = 0
        sql5 = """SELECT "OrderID" FROM "ProductLine" ORDER BY OrderID DESC LIMIT 1 """   #Retrieves the order that has just been saved
        cur.execute(sql5)
        orders = cur.fetchone()
        sql4 = """SELECT "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" FROM ProductLine WHERE OrderID = '%d'""" % (orders[0])
        cur.execute(sql4)
        items = cur.fetchall()
        repeat = len(items)-1
        global ordernumber
        ordernumber = orders[0]
        sql9 = """SELECT "ProductLineID" FROM "ProductLine" WHERE "OrderID" = '%d'  """ %(ordernumber)
```

```
cur.execute(sql9)
con.commit()
productlines = []
productlines = cur.fetchall()
timedate=time.strftime("%H:%M")+ " " + time.strftime("%d/%m/%Y")   #Takes the system time when saving the order into the 'Order' table
for each in productlines:   #Adds each productline to the main orders table
   currenteach = each[0]
   sqladdproductline = """INSERT INTO "Orders" ("FullOrderID", "ProductLineID", "CustomerID", "TimeandDate", "StatusID") VALUES(?, ?, ?, ?, "1")"""
   cur.execute(sqladdproductline, (ordernumber, currenteach, usernameoptionsid[0][0], timedate))
   con.commit()
drinks = []
drinkprice = 0
for eachdrink in items: #Selects the drinks ordered inorder to display them to the customer and calculate the cost
   sqldrinks = """SELECT "Drink" FROM Drinks WHERE DrinkID = '%d'""" % (eachdrink[4])
   cur.execute(sqldrinks)
   currentdrinks = cur.fetchone()
   drinks.append(currentdrinks)
drinkslist = ""
for each in drinks:
   try:
      drinkslist = drinkslist + " " + each[0]
      drinkprice = drinkprice + 2.99
   except:
      pass
   else:
      pass
extras = []
extrasprice = 0
currentextra = []
for eachextra in items:  #Selects the extras ordered inorder to display them to the customer and calculate the cost
   sqlextras = """SELECT "Extras" FROM Extras WHERE ExtrasID = '%d'""" % (eachextra[3])
   cur.execute(sqlextras)
   currentextra = cur.fetchone()
   extras.append(currentextra)
extraslist = ""
for each in extras:
   try:
      extraslist = extraslist + " " + each[0]
      extrasprice = extrasprice + 3.99
   except:
      pass
   else:
      pass
pizzas = []
pizzaprice = 0
for eachpizza in items:   #Selects the pizzas ordered inorder to display them to the customer and calculate the cost
   sqlsize = """SELECT "Size" FROM Sizes WHERE SizeID = '%d'""" % (eachpizza[0])
   sqltopping = """SELECT "Topping" FROM Toppings WHERE ToppingID = '%d'""" % (eachpizza[2])
   sqlcrust = """SELECT "Crust" FROM StuffedCrust WHERE CrustID = '%d'""" % (eachpizza[1])
   cur.execute(sqlsize)
   currentsize = cur.fetchone()
   cur.execute(sqltopping)
   currenttopping = cur.fetchone()
   cur.execute(sqlcrust)
   currentcrust = cur.fetchone()
   pizzas.append(currentsize)
   pizzas.append(currenttopping)
   try:
      if currentcrust[0] == "Yes":
         newcrustlist = []
         newcrustlist.append("Stuffed Crust")
         pizzaprice=pizzaprice + 2.99
         pizzaprice=pizzaprice + 7.99
   except:
      newcrustlist = []
   else:
      if currentcrust[0] == "No":
         newcrustlist = []
         newcrustlist.append("NOT Stuffed Crust")
         pizzaprice=pizzaprice + 7.99
   pizzas.append(newcrustlist)
pizzaslist = ""
for each in pizzas:
   try:
      pizzaslist = pizzaslist + " " + each[0]
   except:
      pass
   else:
      pass
```

```
sql = """SELECT "ProductLineID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(ordernumber)
cur.execute(sql)
productlines = cur.fetchall()
fullorder = []
overallpizza = ""
for eachline in productlines:
    sql2 = """SELECT "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" FROM ProductLine WHERE ProductLineID = '%d'""" % (eachline[0])
    cur.execute(sql2)
    order=cur.fetchall()
    fullorder.append(order)
extraslist= ""
drinkslist=""
for each in fullorder:
    size = each[0][0]
    sizesql = """SELECT "Size" FROM "Sizes" WHERE "SizeID" = '%d' """ %(size)
    cur.execute(sizesql)
    actualsize = cur.fetchone()
    crust = each[0][1]
    crustsql = """SELECT "Crust" FROM "StuffedCrust" WHERE "CrustID" = '%d' """ %(crust)
    cur.execute(crustsql)
    actualcrust = cur.fetchone()
    try:
        if actualcrust[0]=="Yes":
            finalcrust = "Stuffed Crust"
        else:
            finalcrust = "NOT Stuffed Crust"
    except:
        pass
    topping = each[0][2]
    toppingsql = """SELECT "Topping" FROM "Toppings" WHERE "ToppingID" = '%d' """ %(topping)
    cur.execute(toppingsql)
    actualtopping = cur.fetchone()
    try:
        overallpizza = overallpizza + actualsize[0] + " " + actualtopping[0] + " " + finalcrust + "                    "
    except:
        pass
    else:
        pass
    extras = each[0][3]
    extrasql = """SELECT "Extras" FROM "Extras" WHERE "ExtrasID" = '%d' """ %(extras)
    cur.execute(extrasql)
    actualextras = cur.fetchone()
    try:
        extraslist = extraslist + actualextras[0] + "                    "
    except:
        pass
    else:
        pass
    drinks = each[0][4]
    drinkssql = """SELECT "Drink" FROM "Drinks" WHERE "DrinkID" = '%d' """ %(drinks)
    cur.execute(drinkssql)
    actualdrinks = cur.fetchone()
    try:
        drinkslist = drinkslist + actualdrinks[0] + "                    "
    except:
        pass
    else:
        pass
self.pizzatable.setPlainText(overallpizza)    #Displays the ordered pizzas
self.extrastable.setPlainText(extraslist)    #Displays the ordered extras
self.drinkstable.setPlainText(drinkslist)    #Displays the ordered drinks
cost = drinkprice + extrasprice + pizzaprice
cost2dp = "{0:.2f}".format(cost)       #Rounds cost of order, to 2 decimal places
coststr = str(cost2dp)
self.costbox.setPlainText(coststr)     #Displays the total cost
sql = """UPDATE "Orders" SET "Cost"='%s' WHERE "FullOrderID"= '%s' """ %(coststr, ordernumber) #Saves cost of order
cur.execute(sql)
con.commit()
sqlpayment = """SELECT "DeliveryType" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ %(usernameoptionsid[0])
cur.execute(sqlpayment)
method=cur.fetchone()
methodstr=str(method[0])
self.paymentmethodbox.setPlainText(methodstr)
statussql = """SELECT "StatusID" FROM Orders WHERE "FullOrderID" = '%d' """ %(ordernumber)
cur.execute(statussql)
currentstatus = cur.fetchone()
status = currentstatus[0]
statusnamesql = """SELECT "Status" FROM Status WHERE "StatusID" = '%d' """ %(status)
cur.execute(statusnamesql)
```

```
        currentstatusname = cur.fetchone()
        status = str(currentstatusname[0])
        self.OrderStatus.setPlainText(status)
    def update(self):   #Function to refresh order status
        statussql = """SELECT "StatusID" FROM Orders WHERE "FullOrderID" = '%d' """ %(ordernumber)
        cur.execute(statussql)
        currentstatus = cur.fetchone()
        status = currentstatus[0]
        statusnamesql = """SELECT "Status" FROM Status WHERE "StatusID" = '%d' """ %(status)
        cur.execute(statusnamesql)
        currentstatusname = cur.fetchone()
        status = str(currentstatusname[0])
        self.OrderStatus.setPlainText(status)
    def home(self): #Function to return to home menu
        self.hide()
        self.newwindow = TenthWindow()
        self.newwindow.show()
    def print1(self, printer = None):
            if(printer is None):
                printer = QtGui.QPrinter()
                if(QtGui.QPrintDialog(printer).exec_() != QtGui.QDialog.Accepted):
                    return
            self.label.print_(printer)
    def printit(self):  #This function is run if the user wishes to print the order confirmation
        printer=QtGui.QPrinter()   #This calls the system print window
        dialog = QtGui.QPrintDialog(printer, self)
        if(dialog.exec_() != QtGui.QDialog.Accepted):
            return
        p=QtGui.QPixmap.grabWidget(self.frame)   #This specifies what is to be printed, in this case the frame called "frame"
        printLabel = QtGui.QLabel()
        printLabel.setPixmap(p)
        painter = QtGui.QPainter(printer)
        printLabel.render(painter)
        painter.end()

class EighthWindow(QtGui.QMainWindow, win8): #This loads the GUI that is the staff's homescreen
    def __init__(self, parent=None):   #Setting up the window
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.selectbutton.clicked.connect(self.select)    #These connect the function to the appropiate button press
        self.chooseorder.returnPressed.connect(self.select)
        self.backbutton.clicked.connect(self.back)
        con.cursor()
        s=""" SELECT DISTINCT Orders.FullOrderID, Orders.TimeandDate, CustomerInformation.FirstName, CustomerInformation.Surname, Status.Status
        FROM Orders
        INNER JOIN Status
        ON Orders.StatusID=Status.StatusID
        INNER JOIN CustomerInformation
        ON Orders.CustomerID=CustomerInformation.CustomerID
        WHERE Orders.StatusID < 6
        """                 #This is one of my Inner Join queries, which joins together OrderID, order time, customer names and the order status
        cur.execute(s)
        self.data = cur.fetchall() #This fetches all the specified data, and then outputs it into a table
        if len(self.data)>0:
            for i in range(len(self.data)-1,-1):
                self.data.pop(i)
                self.model.removeRow(index.row(self.data[i]))
        self.model=QtGui.QStandardItemModel(self)     #This section of code creates the table seen in the GUI
        self.tableView.setModel(self.model)
        for row in self.data:
                items = [
                    QtGui.QStandardItem(str(field))
                    for field in row
                ]
                self.model.appendRow(items)
    def select(self): #This function is for selecting a specific order to load its details
        global chosenorder
        chosenorder = self.chooseorder.text()
        sql = """SELECT "ProductLineID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)
        cur.execute(sql)
        productlines = cur.fetchall()
        if productlines==[]:
            pass
        else:
            self.hide()
            self.newwindow = NinthWindow() #Loads the order window
            self.newwindow.show()
    def back(self):
        self.hide()
```

```
        self.newwindow = SecondWindow()
        self.newwindow.show()

class NinthWindow(QtGui.QMainWindow, win9):   #This class controls the order view GUI for admin and staff
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.savebutton.clicked.connect(self.save)
        self.backbutton.clicked.connect(self.back)
        self.deleteorder.clicked.connect(self.delete)
        self.deleteorder.hide()
        self.label_3.setText("Staff Mode")
        backcheck = """SELECT "AccountType" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ %(usernameoptionsid[0])
        cur.execute(backcheck)
        ggj = cur.fetchone()
        if ggj[0]=="Admin":
            self.deleteorder.show()
            self.label_3.setText("Admin Mode") #The program checks which account type is logged in, and changes the label accordingly (Admin or staff)
        sql = """SELECT "ProductLineID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)
        cur.execute(sql)
        productlines = cur.fetchall()
        fullorder = []
        overallpizza = ""
        for eachline in productlines: #Retrieves the requested order
            sql2 = """SELECT "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" FROM ProductLine WHERE ProductLineID = '%d'""" % (eachline[0])
            cur.execute(sql2)
            order=cur.fetchall()
            fullorder.append(order)
        extraslist= ""
        drinkslist=""
        for each in fullorder:
            size = each[0][0]
            sizesql = """SELECT "Size" FROM "Sizes" WHERE "SizeID" = '%d' """ %(size)
            cur.execute(sizesql)
            actualsize = cur.fetchone()
            crust = each[0][1]
            crustsql = """SELECT "Crust" FROM "StuffedCrust" WHERE "CrustID" = '%d' """ %(crust)
            cur.execute(crustsql)
            actualcrust = cur.fetchone()
            try:
                if actualcrust[0]=="Yes":
                    finalcrust = "Stuffed Crust"
                else:
                    finalcrust = "NOT Stuffed Crust"
            except:
                pass
            topping = each[0][2]
            toppingsql = """SELECT "Topping" FROM "Toppings" WHERE "ToppingID" = '%d' """ %(topping)
            cur.execute(toppingsql)
            actualtopping = cur.fetchone()
            try:
                overallpizza = overallpizza + actualsize[0] + " " + finalcrust + " " + actualtopping[0] + "                          "
            except:
                pass
            else:
                pass
            extras = each[0][3]
            extrasql = """SELECT "Extras" FROM "Extras" WHERE "ExtrasID" = '%d' """ %(extras)
            cur.execute(extrasql)
            actualextras = cur.fetchone()
            try:
                extraslist = extraslist + actualextras[0] + "                                    "
            except:
                pass
            else:
                pass
            drinks = each[0][4]
            drinkssql = """SELECT "Drink" FROM "Drinks" WHERE "DrinkID" = '%d' """ %(drinks)
            cur.execute(drinkssql)
            actualdrinks = cur.fetchone()
            try:
                drinkslist = drinkslist + actualdrinks[0] + "                                    "
            except:
                pass
            else:
                pass
        self.pizzatable.setPlainText(overallpizza) #Displays the order details
        self.extrastable.setPlainText(extraslist)
        self.drinkstable.setPlainText(drinkslist)
```

```
        global chosenorderint
        chosenorderint = int(chosenorder)
        sql7 = """SELECT "StatusID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorderint)
        cur.execute(sql7)
        orderstatus = cur.fetchone()
        if orderstatus[0]==1:                #This uses the current order status to activate the correct radio button
            self.radiowaiting.setChecked(True)
        if orderstatus[0]==2:
            self.radiopreparing.setChecked(True)
        if orderstatus[0]==3:
            self.radiocooking.setChecked(True)
        if orderstatus[0]==4:
            self.radiodelivering.setChecked(True)
        if orderstatus[0]==5:
            self.radiofailed.setChecked(True)
        if orderstatus[0]==6:
            self.radiosuccessful.setChecked(True)
    def save(self):  #Upon saving the order, the status is saved by setting the status to whatever radio button has been selected
        if self.radiowaiting.isChecked()==True:
            updatestatus = """UPDATE "Orders" SET "StatusID" = 1 WHERE "FullOrderID" = '%d' """ %(chosenorderint)
            cur.execute(updatestatus)
        if self.radiopreparing.isChecked()==True:
            updatestatus = """UPDATE "Orders" SET "StatusID" = 2 WHERE "FullOrderID" = '%d' """ %(chosenorderint)
            cur.execute(updatestatus)
        if self.radiocooking.isChecked()==True:
            updatestatus = """UPDATE "Orders" SET "StatusID" = 3 WHERE "FullOrderID" = '%d' """ %(chosenorderint)
            cur.execute(updatestatus)
        if self.radiodelivering.isChecked()==True:
            updatestatus = """UPDATE "Orders" SET "StatusID" = 4 WHERE "FullOrderID" = '%d' """ %(chosenorderint)
            cur.execute(updatestatus)
        if self.radiofailed.isChecked()==True:
            updatestatus = """UPDATE "Orders" SET "StatusID" = 5 WHERE "FullOrderID" = '%d' """ %(chosenorderint)
            cur.execute(updatestatus)
        if self.radiosuccessful.isChecked()==True:
            updatestatus = """UPDATE "Orders" SET "StatusID" = 6 WHERE "FullOrderID" = '%d' """ %(chosenorderint)
            cur.execute(updatestatus)
        con.commit()
        backcheck = """SELECT "AccountType" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ %(usernameoptionsid[0])
        cur.execute(backcheck) #This is necessary to return the user to the correct previous screen, depending on whether they are admin or staff
        ggj = cur.fetchone()
        if ggj[0] == "Admin":
            self.hide()
            self.newwindow = FourteenthWindow()
            self.newwindow.show()
        if ggj[0] == "Staff":
            self.hide()
            self.newwindow = EighthWindow()
            self.newwindow.show()
    def back(self):
        backcheck = """SELECT "AccountType" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ %(usernameoptionsid[0])
        cur.execute(backcheck)
        ggj = cur.fetchone()
        if ggj[0] == "Admin":
            self.hide()
            self.newwindow = FourteenthWindow()
            self.newwindow.show()
        if ggj[0] == "Staff":
            self.hide()
            self.newwindow = EighthWindow()
            self.newwindow.show()
    def delete(self): #This deletes the order from the database
        sqldelete = """DELETE FROM Orders
            WHERE FullOrderID= '%s' """ %(chosenorder)
        cur.execute(sqldelete)
        sqldelete2 = """DELETE FROM ProductLine
            WHERE OrderID= '%s' """ %(chosenorder)  #The order is deleted from both the Orders and ProductLine tables, to maintain data integrity
        cur.execute(sqldelete2)
        con.commit()
        self.hide()
        self.newwindow = FourteenthWindow()
        self.newwindow.show()
        QtGui.QMessageBox.information(self, "Deleted", "Order deleted.")

class TenthWindow(QtGui.QMainWindow, win10): #This GUI is the homepage for the customer accounts
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)                #Setting up UI / Button commands
        self.orderpizza.clicked.connect(self.order)
        self.backbutton.clicked.connect(self.back)
```

```
        self.personaldetails.clicked.connect(self.details)
        self.selectbutton.clicked.connect(self.select)
        self.chooseorder.returnPressed.connect(self.select)
        global usernameoptionsid
        sqlname = """SELECT "FirstName" FROM CustomerInformation WHERE CustomerID = '%d'""" % (usernameoptionsid[0])
        cur.execute(sqlname)
        firstnamereq = cur.fetchone()
        self.NameBox.setPlainText(firstnamereq[0])
        self.NameBox.setPlainText(firstnamereq[0])
        self.data=[]
        self.model = QtGui.QStandardItemModel(self)
        self.tableView.setModel(self.model)
        self.load_data()
    def load_data(self):
        cur = con.cursor()
        s=""" SELECT DISTINCT Orders.FullOrderID, Orders.TimeandDate, Orders.Cost, Status.Status
FROM Orders
INNER JOIN Status
ON Orders.StatusID=Status.StatusID
WHERE Orders.CustomerID = '%d'
        """ %(usernameoptionsid[0])   #This is another Inner Join query, bringing together relevant details from multiple tables
        cur.execute(s)
        self.data = cur.fetchall() #This creates the table in the GUI
        if len(self.data)>0:
            for i in range(len(self.data)-1,-1):
                self.data.pop(i)
                self.model.removeRow(index.row(self.data[i]))
        self.model=QtGui.QStandardItemModel(self)
        self.tableView.setModel(self.model)
        for row in self.data:
                items = [
                    QtGui.QStandardItem(str(field))
                    for field in row
                ]
                self.model.appendRow(items)
    def select(self):
        global chosenorder
        chosenorder = self.chooseorder.text()
        sql = """SELECT "ProductLineID", "CustomerID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)
        cur.execute(sql)
        productlines = cur.fetchall()
        if productlines==[]:
            pass
        elif productlines[0][1] != usernameoptionsid[0][0]:
            pass
        else:
            self.hide()
            self.newwindow = EleventhWindow()
            self.newwindow.show()
    def order(self):
        self.hide()
        self.newwindow = FourthWindow()
        self.newwindow.show()
    def back(self):
        self.hide()
        self.newwindow = SecondWindow()
        self.newwindow.show()
    def details(self):
        self.hide()
        self.newwindow = ThirteenthWindow()
        self.newwindow.show()

class EleventhWindow(QtGui.QMainWindow, win11):  #GUI for customer to see order
    def __init__(self, parent=None):            #Constructor Method
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.backbutton.clicked.connect(self.back)
        self.viewconfirmation.clicked.connect(self.view)
        sql = """SELECT "ProductLineID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)   #Finds all productlines associated with the order
        cur.execute(sql)
        productlines = cur.fetchall()
        fullorder = []
        overallpizza = ""
        for eachline in productlines:   #Finds each item in each productline
            sql2 = """SELECT "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" FROM ProductLine WHERE ProductLineID = '%d'""" % (eachline[0])
            cur.execute(sql2)
            order=cur.fetchall()
            fullorder.append(order)
        extraslist= ""
```

```
        drinkslist=""
        for each in fullorder:
            size = each[0][0]
            sizesql = """SELECT "Size" FROM "Sizes" WHERE "SizeID" = '%d' """ %(size)
            cur.execute(sizesql)
            actualsize = cur.fetchone()
            crust = each[0][1]
            crustsql = """SELECT "Crust" FROM "StuffedCrust" WHERE "CrustID" = '%d' """ %(crust)
            cur.execute(crustsql)
            actualcrust = cur.fetchone()
            try:
                if actualcrust[0]=="Yes":
                    finalcrust = "Stuffed Crust"
                else:
                    finalcrust = "NOT Stuffed Crust"
            except:
                pass
            topping = each[0][2]
            toppingsql = """SELECT "Topping" FROM "Toppings" WHERE "ToppingID" = '%d' """ %(topping)
            cur.execute(toppingsql)
            actualtopping = cur.fetchone()
            try:
                overallpizza = overallpizza + actualsize[0] + " " + actualtopping[0] + " " + finalcrust + "                    "
            except:
                pass
            else:
                pass
            extras = each[0][3]
            extrasql = """SELECT "Extras" FROM "Extras" WHERE "ExtrasID" = '%d' """ %(extras)
            cur.execute(extrasql)
            actualextras = cur.fetchone()
            try:
                extraslist = extraslist + actualextras[0] + "                    "
            except:
                pass
            else:
                pass
            drinks = each[0][4]
            drinkssql = """SELECT "Drink" FROM "Drinks" WHERE "DrinkID" = '%d' """ %(drinks)
            cur.execute(drinkssql)
            actualdrinks = cur.fetchone()
            try:
                drinkslist = drinkslist + actualdrinks[0] + "                    "
            except:
                pass
            else:
                pass
        self.pizzatable.setPlainText(overallpizza)  #Outputs all pizza in the order
        self.extrastable.setPlainText(extraslist)   #Outputs all extras in the order
        self.drinkstable.setPlainText(drinkslist)   #Outputs all drinks in the order
        global chosenorderint
        chosenorderint = int(chosenorder)
        sql7 = """SELECT "StatusID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorderint)
        cur.execute(sql7)
        orderstatus = cur.fetchone()    #Sets the correct radio button, depending on order status
        if orderstatus[0]==1:
            self.radiowaiting.setChecked(True)
        if orderstatus[0]==2:
            self.radiopreparing.setChecked(True)
        if orderstatus[0]==3:
            self.radiocooking.setChecked(True)
        if orderstatus[0]==4:
            self.radiodelivering.setChecked(True)
        if orderstatus[0]==5:
            self.radiofailed.setChecked(True)
        if orderstatus[0]==6:
            self.radiosuccessful.setChecked(True)
    def back(self):
        self.hide()
        self.newwindow = TenthWindow()
        self.newwindow.show()
    def view(self):    #Function to view order confirmation
        self.hide()
        self.newwindow = FifteenthPlusOneWindow()
        self.newwindow.show()

class TwelthWindow(QtGui.QMainWindow, win12):  #Account Creation Confirmation window
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
```

```
        self.setupUi(self)                      #Setup UI
        self.loginbutton.clicked.connect(self.login) #Controls Login button
    def login(self):    #Function runs when called by button press
        self.hide()       #Hides itself
        self.newwindow = SecondWindow()   #Opens login screen
        self.newwindow.show()

class ThirteenthWindow(QtGui.QMainWindow, win13):    #Window class to for customer to update details
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)                      #Setup UI
        self.submitbutton.clicked.connect(self.submit)   #Setup buttons
        self.cancelbutton.clicked.connect(self.cancel)
        sql = """SELECT "FirstName", "Surname", "HouseNumber", "Postcode", "MobileNumber", "DeliveryType", "UserName", "Password"  FROM
CustomerInformation WHERE CustomerID = '%d'""" % (usernameoptionsid[0][0])
        cur.execute(sql)      #Selects all the details about the logged in user
        global currentdetails       #Makes the fetched details global so they can be accessed from other functions
        currentdetails = cur.fetchall()
        global mobilenumstr
        global housenumstr
        housenumstr = str(currentdetails[0][2])
        mobilenumstr = str(currentdetails[0][4])
        self.firstname.setPlainText(currentdetails[0][0])    #Set each textbox with the relevant info
        self.surname.setPlainText(currentdetails[0][1])
        self.housenum.setPlainText(housenumstr)
        self.postcode.setPlainText(currentdetails[0][3])
        self.mobilenum.setPlainText(mobilenumstr)
        self.payment.setPlainText(currentdetails[0][5])
        self.username.setPlainText(currentdetails[0][6])
        self.password.setPlainText("Password hidden")
    def cancel(self):   #Function to return to previous screen without saving
        self.hide()
        self.newwindow = TenthWindow()
        self.newwindow.show()
    def submit(self):   #Submit details
        newhousenum = self.housenumnew.text()    #Takes the input from the input boxes and stores them
        newpostcode = self.postcodenew.text()
        newmobilenum = self.mobilenumnew.text()
        newpayment = self.paymentmethodnew.currentText()
        newpassword = self.passwordnew.text()
        hashedpass = hasher(newpassword)
        if newhousenum == "":                   #This checks to see if any of the boxes are left blank,
            newhousenum = housenumstr           #and if they are, the new variables are set back to
        if newpostcode == "":                   #what they previously were
            newpostcode = currentdetails[0][3]   #so that empty values are not put in the database
        if newmobilenum == "":
            newmobilenum = mobilenumstr
        if newpayment == "":
            newpayment = currentdetails[0][5]
        if newpassword == "":
            newpassword = currentdetails[0][7]
        else:
            newpassword=hasher(newpassword)
        postcoderule = re.compile(r'^([A-PR-UWYZ0-9][A-HK-Y0-9][AEHMNPRTVXY0-9]?[ABEHMNPRVWXY0-9]? {1,2}[0-9][ABD-HJLN-UW-Z]{2}|GIR 0AA)$')
        if not postcoderule.search(newpostcode):    #REGEX validation on postcode
            QtGui.QMessageBox.information(self, "Error!", "Invalid Postcode \n Must be in format: 'XX12 3XX'")
        else:
            mobilerule = re.compile(r'^(07\d{9,9})$')    #REGEX validation on mobile number
            if not mobilerule.search(newmobilenum):
                QtGui.QMessageBox.information(self, "Error!", "Invalid Mobile Number \n Can only contain 11 numbers, starting with '07'")
            else:
                if len(newpassword) < 6:    #Validation on length of password
                    QtGui.QMessageBox.information(self, "Error!", "Password must contain at least 6 characters!")
                else:
                    sql = """UPDATE "CustomerInformation" SET HouseNumber='%s', Postcode='%s', MobileNumber ='%s', DeliveryType='%s', Password =  '%s' WHERE
"CustomerID" = '%s' """ %(newhousenum, newpostcode, newmobilenum, newpayment, newpassword, usernameoptionsid[0][0])
                    cur.execute(sql)    #Places the updated info back into the database
                    con.commit()
                    self.hide()
                    self.newwindow = TenthWindow()
                    self.newwindow.show()

class FourteenthWindow(QtGui.QMainWindow, win14):     #Admin home screen
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)                          #Setting up UI
        self.editcustomerbutton.clicked.connect(self.edit)     #Setting up buttons
        self.selectbutton.clicked.connect(self.select)
        self.chooseorder.returnPressed.connect(self.select)    #Making pressing return act the same as pressing the button
```

```
        self.databutton.clicked.connect(self.data)
        self.backbutton.clicked.connect(self.back)
        self.data=[]
        self.model = QtGui.QStandardItemModel(self)
        self.tableView.setModel(self.model)
        self.load_data()
    def load_data(self):       #Creates a table in the GUI for the admin
        cur = con.cursor()
        s="""SELECT DISTINCT Orders.FullOrderID, Orders.TimeandDate, CustomerInformation.FirstName, CustomerInformation.Surname, Status.Status
        FROM Orders
        INNER JOIN Status
        ON Orders.StatusID=Status.StatusID
        INNER JOIN CustomerInformation
        ON Orders.CustomerID=CustomerInformation.CustomerID
                """
        cur.execute(s)      #Selects the details of all orders in database using an inner join query
        self.data = cur.fetchall()
        if len(self.data)>0:        #Creates the table
            for i in range(len(self.data)-1,-1):
                self.data.pop(i)
                self.model.removeRow(index.row(self.data[i]))
        self.model=QtGui.QStandardItemModel(self)
        self.tableView.setModel(self.model)
        for row in self.data:
                items = [
                    QtGui.QStandardItem(str(field))
                    for field in row
                ]
                self.model.appendRow(items)
    def select(self):    #Selecting a specific order
        global chosenorder
        chosenorder = self.chooseorder.text()
        sql = """SELECT "ProductLineID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)   #Querying database for specific order
        cur.execute(sql)
        productlines = cur.fetchall()
        if productlines==[]:    #Does nothing if the order cannot be found
            pass
        else:
            self.hide()
            self.newwindow = NinthWindow()   #otherwise, it opens the order view window
            self.newwindow.show()
    def back(self):   #FUnction to go back to login screen
        self.hide()
        self.newwindow = SecondWindow()
        self.newwindow.show()
    def edit(self):     #View all customer details
        self.hide()
        self.newwindow = FifteenthWindow()
        self.newwindow.show()
    def data(self):     #View data analytics
        self.hide()
        self.newwindow = SeventeenthWindow()
        self.newwindow.show()

class FifteenthWindow(QtGui.QMainWindow, win15):  #GUI to view all user details
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)                  #Setup UI
        self.backbutton.clicked.connect(self.back)
        self.deletebutton.clicked.connect(self.delete)
        self.data=[]
        self.model = QtGui.QStandardItemModel(self)
        self.tableView.setModel(self.model)
        self.load_data()
        self.selectbutton.clicked.connect(self.upd_record)
    def upd_record(self):
        index = self.tableView.currentIndex() #returns currently selected cell
        cell_contents=index.data()#returns the contents of the currently selected cell
        idcust=self.model.data(self.model.index(index.row(), 0))
        firstname=self.model.data(self.model.index(index.row(), 1))
        surname=self.model.data(self.model.index(index.row(), 2))
        house=self.model.data(self.model.index(index.row(), 3))
        postcode=self.model.data(self.model.index(index.row(), 4))
        mobile=self.model.data(self.model.index(index.row(), 5))
        pay=self.model.data(self.model.index(index.row(), 6))
        user=self.model.data(self.model.index(index.row(), 7))
        password=self.model.data(self.model.index(index.row(), 8))
        hashedpass = hasher(password)
        typeaccount=self.model.data(self.model.index(index.row(), 9))
```

```
        statusaccount=self.model.data(self.model.index(index.row(), 10))
        question=self.model.data(self.model.index(index.row(), 11))
        answer=self.model.data(self.model.index(index.row(), 12))
        self.model.data(self.model.index(4,1))
        s='UPDATE CustomerInformation SET FirstName=?,Surname=?,HouseNumber=?,Postcode=?,MobileNumber=?,DeliveryType=?, Password=?,
AccountType=?, AccountStatus=?, SecurityQuestion=?, SecurityAnswer=? WHERE CustomerID=?'
        cur.execute(s,(firstname,surname,house, postcode, mobile, pay, hashedpass, typeaccount, statusaccount, question, answer, idcust))
        con.commit()
        self.load_data()
    def load_data(self):    #Loads the info and creates a table
        cur = con.cursor()
        s='select * from CustomerInformation' #Selects entire table
        cur.execute(s)
        self.data = cur.fetchall()
        if len(self.data)>0:        #Creates table in GUI
            for i in range(len(self.data)-1,-1):
                self.data.pop(i)
                self.model.removeRow(index.row(self.data[i]))
        self.model=QtGui.QStandardItemModel(self)
        self.tableView.setModel(self.model)
        for row in self.data:
                items = [
                    QtGui.QStandardItem(str(field))
                    for field in row
                ]
                self.model.appendRow(items)
    def back(self):     #Back function
        self.hide()
        self.newwindow = FourteenthWindow()
        self.newwindow.show()
    def delete(self):   #Suspend users function
        index = self.tableView.currentIndex()
        typeaccount=self.model.data(self.model.index(index.row(), 9))   #Works out customer ID of selected account
        if typeaccount == "Admin":
            QtGui.QMessageBox.information(self, "Error!", "Cannot suspend Admin or Staff accounts!")  #Stops admin suspending admin/staff accounts
        elif typeaccount == "Staff":
            QtGui.QMessageBox.information(self, "Error!", "Cannot suspend Admin or Staff accounts!")
        else:
            typeactive=self.model.data(self.model.index(index.row(), 10)) #Determines whether account is active or already suspended
            if typeactive == "Active":
                idcust=self.model.data(self.model.index(index.row(), 0))
                sqlupdate="""UPDATE CustomerInformation SET AccountStatus="Suspended" WHERE CustomerID= '%s'""" %(idcust)  #Update database
                cur.execute(sqlupdate)
                con.commit()
                self.load_data()
                QtGui.QMessageBox.information(self, "Suspended", "Account Suspended.")
            else:
                idcust=self.model.data(self.model.index(index.row(), 0))
                sqlupdate="""UPDATE CustomerInformation SET AccountStatus="Active" WHERE CustomerID= '%s'""" %(idcust) #Update database
                cur.execute(sqlupdate)
                con.commit()
                self.load_data()
                QtGui.QMessageBox.information(self, "Reactivated", "Account Re-activated.")

class FifteenthPlusOneWindow(QtGui.QMainWindow, win5):  #previous order GUI
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)   #Setup UI
        self.returnhome.clicked.connect(self.back)
        self.printbutton.clicked.connect(self.printit)
        self.updatebutton.clicked.connect(self.update)
        sql = """SELECT "ProductLineID" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)  #Finds all productlines in order
        cur.execute(sql)
        productlines = cur.fetchall()
        fullorder = []
        overallpizza = ""
        for eachline in productlines:  #Finds items in each productline
            sql2 = """SELECT "SizeID", "CrustID", "ToppingID", "ExtrasID", "DrinkID" FROM ProductLine WHERE ProductLineID = '%d'""" % (eachline[0])
            cur.execute(sql2)
            order=cur.fetchall()
            fullorder.append(order)
        extraslist= ""
        drinkslist=""
        for each in fullorder:
            size = each[0][0]
            sizesql = """SELECT "Size" FROM "Sizes" WHERE "SizeID" = '%d' """ %(size)
            cur.execute(sizesql)
            actualsize = cur.fetchone()
            crust = each[0][1]
```

```
            crustsql = """SELECT "Crust" FROM "StuffedCrust" WHERE "CrustID" = '%d' """ %(crust)
            cur.execute(crustsql)
            actualcrust = cur.fetchone()
            try:
                if actualcrust[0]=="Yes":
                    finalcrust = "Stuffed Crust"
                else:
                    finalcrust = "NOT Stuffed Crust"
            except:
                pass
            topping = each[0][2]
            toppingsql = """SELECT "Topping" FROM "Toppings" WHERE "ToppingID" = '%d' """ %(topping)
            cur.execute(toppingsql)
            actualtopping = cur.fetchone()
            try:
                overallpizza = overallpizza + actualsize[0] + " " + actualtopping[0] + " " + finalcrust + "                    "
            except:
                pass
            else:
                pass
            extras = each[0][3]
            extrasql = """SELECT "Extras" FROM "Extras" WHERE "ExtrasID" = '%d' """ %(extras)
            cur.execute(extrasql)
            actualextras = cur.fetchone()
            try:
                extraslist = extraslist + actualextras[0] + "                    "
            except:
                pass
            else:
                pass
            drinks = each[0][4]
            drinkssql = """SELECT "Drink" FROM "Drinks" WHERE "DrinkID" = '%d' """ %(drinks)
            cur.execute(drinkssql)
            actualdrinks = cur.fetchone()
            try:
                drinkslist = drinkslist + actualdrinks[0] + "                    "
            except:
                pass
            else:
                pass
        self.pizzatable.setPlainText(overallpizza)   #Outputs pizzas ordered
        self.extrastable.setPlainText(extraslist)    #Outputs extras ordered
        self.drinkstable.setPlainText(drinkslist)    #Outputs drinks ordered
        sqlcost = """SELECT "Cost" FROM "Orders" WHERE "FullOrderID" = '%s' """ %(chosenorder)
        cur.execute(sqlcost)
        method1=cur.fetchone()
        self.costbox.setPlainText(method1[0])  #Finds and outputs cost stored in database
        sqlpayment = """SELECT "DeliveryType" FROM "CustomerInformation" WHERE "CustomerID" = '%d' """ %(usernameoptionsid[0])
        cur.execute(sqlpayment)
        method=cur.fetchone()
        methodstr=str(method[0])
        self.paymentmethodbox.setPlainText(methodstr)    #Finds and outputs payment method stored in database
        statussql = """SELECT "StatusID" FROM Orders WHERE "FullOrderID" = '%s' """ %(chosenorder)
        cur.execute(statussql)
        currentstatus = cur.fetchone()
        status = currentstatus[0]
        statusnamesql = """SELECT "Status" FROM Status WHERE "StatusID" = '%d' """ %(status)
        cur.execute(statusnamesql)
        currentstatusname = cur.fetchone()
        status = str(currentstatusname[0])
        self.OrderStatus.setPlainText(status)  #Finds and outputs order status in database
    def back(self):   #Go back function
        self.hide()
        self.newwindow = TenthWindow()
        self.newwindow.show()
    def print1(self, printer = None):
            if(printer is None):
                printer = QtGui.QPrinter()
                if(QtGui.QPrintDialog(printer).exec_() != QtGui.QDialog.Accepted):
                    return
            self.label.print_(printer)
    def printit(self):     #Print function
        printer=QtGui.QPrinter()
        dialog = QtGui.QPrintDialog(printer, self)
        if(dialog.exec_() != QtGui.QDialog.Accepted):
            return
        p=QtGui.QPixmap.grabWidget(self.frame)  #Prints the specified frame
        printLabel = QtGui.QLabel()
        printLabel.setPixmap(p)
```

```
        painter = QtGui.QPainter(printer)
        printLabel.render(painter)
        painter.end()
    def update(self):    #Updates the order status
        statussql = """SELECT "StatusID" FROM Orders WHERE "FullOrderID" = '%d' """ %(chosenorderint)
        cur.execute(statussql)
        currentstatus = cur.fetchone()
        status = currentstatus[0]
        statusnamesql = """SELECT "Status" FROM Status WHERE "StatusID" = '%d' """ %(status)
        cur.execute(statusnamesql)
        currentstatusname = cur.fetchone()
        status = str(currentstatusname[0])
        self.OrderStatus.setPlainText(status)

class SixteenthWindow(QtGui.QMainWindow, win16):   #Password recovery GUI
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)
        self.back.clicked.connect(self.cancel)
        self.search.clicked.connect(self.searchpressed)
        self.usernamebox.returnPressed.connect(self.searchpressed)
        self.answerbox.returnPressed.connect(self.submitpressed)
        self.newpassword.returnPressed.connect(self.changepressed)
        self.submit.clicked.connect(self.submitpressed)
        self.change.clicked.connect(self.changepressed)
        self.tabWidget.setTabEnabled(1, False)
        self.tabWidget.setTabEnabled(2, False)
    def searchpressed(self):    #Search for username function
        search = self.usernamebox.text()
        searchsql = """SELECT * FROM CustomerInformation WHERE UserName = '%s' """ %(search)   #Searches Customer table for username
        cur.execute(searchsql)
        global details
        details = cur.fetchall()
        if details==[]:    #If no account found
            QtGui.QMessageBox.information(self, "Account Not Found", "This username is not stored in the system. \n Please try again...")
            self.hide()
            self.reload= SixteenthWindow()
            self.reload.show()
        else:
            if details[0][10] == "Suspended":    #If account suspended, denies entry
                QtGui.QMessageBox.information(self, "Account Suspended", "This account has been suspended. \n Contact the system administrator.")
                self.hide()
                self.reload= SixteenthWindow()
                self.reload.show()
            elif details[0][9] == "Admin" or details[0][9] == "Staff":   #Can only be used by customer accounts, not admin/staff
                QtGui.QMessageBox.information(self, "Error!", "The password recovery tool is for Customer accounts only. \n Contact the system administrator.")
                self.hide()
                self.reload= SixteenthWindow()
                self.reload.show()
            else:
                self.tabWidget.setTabEnabled(1, True)  #If account found, enables next tab and disables current one
                QtGui.QMessageBox.information(self, "Account Found", "Answer the security question on the next tab to access your account.")
                self.tabWidget.setCurrentIndex(1)
                self.tabWidget.setTabEnabled(0, False)
                self.questionbox.setPlainText(details[0][11])   #Outputs security question
    def submitpressed(self):   #Function to check security question
        answer=self.answerbox.text()
        if answer == details[0][12]:   #Compares inputted answer to actual answer
            self.tabWidget.setTabEnabled(2, True)    #If correct, eanbles the next tab
            QtGui.QMessageBox.information(self, "Success", "You have correctly answered the security question. \n Reset your password on the next screen...")
            self.tabWidget.setCurrentIndex(2)
            self.tabWidget.setTabEnabled(1, False)    #And disables the current tab
        else:
            QtGui.QMessageBox.information(self, "Incorrect Details", "You have incorrectly answered the security question. \n Try again...")
    def changepressed(self):                #Function to change password
        passwordnew = self.newpassword.text()
        if passwordnew == "":
            QtGui.QMessageBox.information(self, "Error!", "You need to enter a password...")
        elif len(passwordnew) < 6:   #Password validation
            QtGui.QMessageBox.information(self, "Error!", "Password must contain at least 6 characters!")
        else:
            hashedpass=hasher(passwordnew)
            updatepassword = """UPDATE CustomerInformation SET Password='%s' WHERE CustomerID = '%s' """ %(hashedpass, details[0][0])
            cur.execute(updatepassword)  #Saves password to database
            con.commit()
            QtGui.QMessageBox.information(self, "Password Changed", "Your password has been changed.")
            self.hide()
            self.newwindow=SecondWindow()   #Returns to login window
            self.newwindow.show()
```

```
    def cancel(self):   #Cancels password recovery
        self.hide()
        self.newwindow = SecondWindow()
        self.newwindow.show()

class SeventeenthWindow(QtGui.QMainWindow, win17):       #Data Analytics window
    def __init__(self, parent=None):
        QtGui.QMainWindow.__init__(self, parent)
        self.setupUi(self)                          #Setup UI
        self.backbutton.clicked.connect(self.back)
        sqlorders=  """SELECT COUNT( DISTINCT FullOrderID) FROM Orders"""    #Counts total orders
        cur.execute(sqlorders)
        ordertotal=cur.fetchone()
        self.totalorders.setPlainText(str(ordertotal[0]))
        cur.execute("SELECT DISTINCT FullOrderID FROM Orders")
        distinctorders= cur.fetchall()
        revtotal=0
        for each in distinctorders:   #Adds up cost of each order
            sqlrev=  """SELECT Cost FROM Orders WHERE FullOrderID='%s'""" %(each[0])
            cur.execute(sqlrev)
            rev=cur.fetchone()
            revtotal=revtotal+float(rev[0])
        revtotal="{0:.2f}".format(revtotal)     #Rounds total cost to 2 decimal places
        self.totalrev.setPlainText("£ "+str(revtotal))
        sqlaccounts=  """SELECT COUNT(CustomerID) FROM CustomerInformation  WHERE AccountType= 'Customer'"""   #Counts the customer accounts
        cur.execute(sqlaccounts)
        acctotal=cur.fetchone()
        self.totalaccounts.setPlainText(str(acctotal[0]))
        avgcost=float(revtotal)/ordertotal[0]    #Works out average cost of order
        avgcost="{0:.2f}".format(avgcost)       #Rounds average cost to 2 dp
        self.avgrev.setPlainText("£ "+str(avgcost))
        sqlactive="""SELECT COUNT(DISTINCT CustomerID) FROM Orders"""
        cur.execute(sqlactive)
        activeacc=cur.fetchone()
        self.activeaccounts.setPlainText(str(activeacc[0]))    #Outputs number of active accounts
        sqltoppings="""SELECT ToppingID FROM ProductLine"""     #Fetches all toppings ever ordered
        cur.execute(sqltoppings)
        toppingdata=cur.fetchall()
        margcount=0
        peppcount=0
        chickcount=0
        meatcount=0
        vegcount=0
        hawcount=0
        for each in toppingdata:    #Tallies up each type of topping
            if each[0]==1:
                margcount=margcount+1
            if each[0]==2:
                peppcount=peppcount+1
            if each[0]==3:
                chickcount=chickcount+1
            if each[0]==4:
                meatcount=meatcount+1
            if each[0]==5:
              vegcount=vegcount+1
            if each[0]==6:
                hawcount=hawcount+1
                #All HTML charts from amcharts:
        toppings="""
<!DOCTYPE html>
<html>
                <head>

                                <script type="text/javascript" src="https://www.amcharts.com/lib/3/amcharts.js"></script>
                                <script type="text/javascript" src="https://www.amcharts.com/lib/3/pie.js"></script>


                                <script type="text/javascript">
                                                AmCharts.makeChart("chartdiv",
                                                                {
                                                                                "type": "pie",
                                                                                "angle": 11.7,
                                                                                "balloonText": "[[title]]<br><span style='font-size:14px'><b>[[value]]</b>
([[percents]]%)</span>",
                                                                                "depth3D": 15,
                                                                                "innerRadius": 20,
                                                                                "labelRadius": 10,
                                                                                "minRadius": 0,
                                                                                "marginBottom": 0,
```

```
                                                "marginTop": 0,
                                                "outlineThickness": 2,
                                                "pullOutOnlyOne": true,
                                                "startDuration": 4,
                                                "startEffect": "elastic",
                                                "titleField": "category",
                                                "valueField": "column-1",
                                                "theme": "default",
                                                "allLabels": [],
                                                "balloon": {
                                                        "animationDuration": 0,
                                                        "borderThickness": 1,
                                                        "fadeOutDuration": 0,
                                                        "fontSize": 0,
                                                        "maxWidth": 0,
                                                        "pointerWidth": 0
                                                },
                                                "legend": {
                                                        "enabled": false,
                                                        "accessibleLabel": "",
                                                        "align": "center",
                                                        "labelText": "",
                                                        "markerType": "circle",
                                                        "rollOverGraphAlpha": 0
                                                },
                                                "titles": [],
                                                "dataProvider": [
                                                        {
                                                                "category": "Margherita",
                                                                "column-1": z1
                                                        },
                                                        {
                                                                "category": "Pepperoni",
                                                                "column-1": z2
                                                        },
                                                        {
                                                                "category": "Chicken",
                                                                "column-1": z3
                                                        },
                                                        {
                                                                "category": "Meat Feast",
                                                                "column-1": z4
                                                        },
                                                        {
                                                                "category": "Vegetable",
                                                                "column-1": z5
                                                        },
                                                        {
                                                                "category": "Hawaiian",
                                                                "column-1": z6
                                                        }
                                                ]
                                        }
                                );
                        </script>
                </head>
                <body>
                        <div id="chartdiv" style="width: 100%; height: 400px; background-color: #FFFFFF;" ></div>
                </body>
</html>
"""
    toppings=toppings.replace("z1",str(margcount))    #Replaces 'z1' (etc) in the HTML string above with actual value
    toppings=toppings.replace("z2",str(peppcount))
    toppings=toppings.replace("z3",str(chickcount))
    toppings=toppings.replace("z4",str(meatcount))
    toppings=toppings.replace("z5",str(vegcount))
    toppings=toppings.replace("z6",str(hawcount))
    self.webViewtopping.setHtml(toppings)    #Sets the webview in the GUI to display the HTML pie chart in the string above
    sqldrinks="""SELECT DrinkID FROM ProductLine"""    #Fetches all drinks ever ordered
    cur.execute(sqldrinks)
    drinkdata=cur.fetchall()
    colacount=0
    lemoncount=0
    pepsicount=0
    fantacount=0
    for each in drinkdata:    #Tallies up each type of drink
      if each[0]==1:
        colacount=colacount+1
      if each[0]==2:
```

```
            lemoncount=lemoncount+1
        if each[0]==3:
            pepsicount=pepsicount+1
        if each[0]==4:
            fantacount=fantacount+1
    drinks="""
<!DOCTYPE html>
<html>
        <head>

                <script type="text/javascript" src="https://www.amcharts.com/lib/3/amcharts.js"></script>
                <script type="text/javascript" src="https://www.amcharts.com/lib/3/pie.js"></script>


                <script type="text/javascript">
                        AmCharts.makeChart("chartdiv",
                                {
                                        "type": "pie",
                                        "angle": 11.7,
                                        "balloonText": "[[title]]<br><span style='font-size:14px'><b>[[value]]</b>
([[percents]]%)</span>",
                                        "depth3D": 15,
                                        "innerRadius": 20,
                                        "labelRadius": 10,
                                        "minRadius": 0,
                                        "marginBottom": 0,
                                        "marginTop": 0,
                                        "outlineThickness": 2,
                                        "pullOutOnlyOne": true,
                                        "startDuration": 4,
                                        "startEffect": "elastic",
                                        "titleField": "category",
                                        "valueField": "column-1",
                                        "theme": "default",
                                        "allLabels": [],
                                        "balloon": {
                                                "animationDuration": 0,
                                                "borderThickness": 1,
                                                "fadeOutDuration": 0,
                                                "fontSize": 0,
                                                "maxWidth": 0,
                                                "pointerWidth": 0
                                        },
                                        "legend": {
                                                "enabled": false,
                                                "accessibleLabel": "",
                                                "align": "center",
                                                "labelText": "",
                                                "markerType": "circle",
                                                "rollOverGraphAlpha": 0
                                        },
                                        "titles": [],
                                        "dataProvider": [
                                                {
                                                        "category": "Coca Cola",
                                                        "column-1": z1
                                                },
                                                {
                                                        "category": "Lemonade",
                                                        "column-1": z2
                                                },
                                                {
                                                        "category": "Pepsi",
                                                        "column-1": z3
                                                },
                                                {
                                                        "category": "Fanta",
                                                        "column-1": z4
                                                },

                                        ]
                                }
                        );
                </script>
        </head>
        <body>
                <div id="chartdiv" style="width: 100%; height: 400px; background-color: #FFFFFF;" ></div>
        </body>
</html>
```

```
"""
    drinks=drinks.replace("z1",str(colacount))     #Replaces 'z1' (etc) in the HTML string above with actual value
    drinks=drinks.replace("z2",str(lemoncount))
    drinks=drinks.replace("z3",str(pepsicount))
    drinks=drinks.replace("z4",str(fantacount))
    self.webViewdrink.setHtml(drinks)    #Sets the webview in the GUI to display the HTML pie chart in the string above
    sqlsize="""SELECT SizeID FROM ProductLine"""    #Fetches all pizza sizes ever ordered
    cur.execute(sqlsize)
    sizedata=cur.fetchall()
    ninecount=0
    twelvecount=0
    fifteencount=0
    eighteencount=0
    for each in sizedata:    #Tallies up each type of pizza size
      if each[0]==1:
        ninecount=ninecount+1
      if each[0]==2:
        twelvecount=twelvecount+1
      if each[0]==3:
        fifteencount=fifteencount+1
      if each[0]==4:
        eighteencount=eighteencount+1
    sizes="""
<!DOCTYPE html>
<html>
        <head>

                <script type="text/javascript" src="https://www.amcharts.com/lib/3/amcharts.js"></script>
                <script type="text/javascript" src="https://www.amcharts.com/lib/3/pie.js"></script>


                <script type="text/javascript">
                        AmCharts.makeChart("chartdiv",
                                {
                                        "type": "pie",
                                        "angle": 11.7,
                                        "balloonText": "[[title]]<br><span style='font-size:14px'><b>[[value]]</b>
([[percents]]%)</span>",
                                        "depth3D": 15,
                                        "innerRadius": 20,
                                        "labelRadius": 10,
                                        "minRadius": 0,
                                        "marginBottom": 0,
                                        "marginTop": 0,
                                        "outlineThickness": 2,
                                        "pullOutOnlyOne": true,
                                        "startDuration": 4,
                                        "startEffect": "elastic",
                                        "titleField": "category",
                                        "valueField": "column-1",
                                        "theme": "default",
                                        "allLabels": [],
                                        "balloon": {
                                                "animationDuration": 0,
                                                "borderThickness": 1,
                                                "fadeOutDuration": 0,
                                                "fontSize": 0,
                                                "maxWidth": 0,
                                                "pointerWidth": 0
                                        },
                                        "legend": {
                                                "enabled": false,
                                                "accessibleLabel": "",
                                                "align": "center",
                                                "labelText": "",
                                                "markerType": "circle",
                                                "rollOverGraphAlpha": 0
                                        },
                                        "titles": [],
                                        "dataProvider": [
                                                {
                                                        "category": "9 Inches",
                                                        "column-1": z1
                                                },
                                                {
                                                        "category": "12 Inches",
                                                        "column-1": z2
                                                },
                                                {
```

```
                                                                        "category": "15 Inches",
                                                                        "column-1": z3
                                                                },
                                                                {
                                                                        "category": "18 Inches",
                                                                        "column-1": z4
                                                                },

                                                        ]
                                                }
                                        );
                                </script>
                        </head>
                        <body>
                                <div id="chartdiv" style="width: 100%; height: 400px; background-color: #FFFFFF;" ></div>
                        </body>
</html>
"""
    sizes=sizes.replace("z1",str(ninecount))       #Replaces 'z1' (etc) in the HTML string above with actual value
    sizes=sizes.replace("z2",str(twelvecount))
    sizes=sizes.replace("z3",str(fifteencount))
    sizes=sizes.replace("z4",str(eighteencount))
    self.webViewsize.setHtml(sizes)    #Sets the webview in the GUI to display the HTML pie chart in the string above
    sqlcrust="""SELECT CrustID FROM ProductLine"""    #Fetches all crusts ever ordered
    cur.execute(sqlcrust)
    crustdata=cur.fetchall()
    yescount=0
    nocount=0
    for each in crustdata:    #Tallies up each type of crust
      if each[0]==1:
        yescount=yescount+1
      if each[0]==2:
        nocount=nocount+1
    crusts="""
<!DOCTYPE html>
<html>
                <head>

                        <script type="text/javascript" src="https://www.amcharts.com/lib/3/amcharts.js"></script>
                        <script type="text/javascript" src="https://www.amcharts.com/lib/3/pie.js"></script>


                        <script type="text/javascript">
                                AmCharts.makeChart("chartdiv",
                                        {
                                                "type": "pie",
                                                "angle": 11.7,
                                                "balloonText": "[[title]]<br><span style='font-size:14px'><b>[[value]]</b>
([[percents]]%)</span>",
                                                "depth3D": 15,
                                                "innerRadius": 20,
                                                "labelRadius": 10,
                                                "minRadius": 0,
                                                "marginBottom": 0,
                                                "marginTop": 0,
                                                "outlineThickness": 2,
                                                "pullOutOnlyOne": true,
                                                "startDuration": 4,
                                                "startEffect": "elastic",
                                                "titleField": "category",
                                                "valueField": "column-1",
                                                "theme": "default",
                                                "allLabels": [],
                                                "balloon": {
                                                        "animationDuration": 0,
                                                        "borderThickness": 1,
                                                        "fadeOutDuration": 0,
                                                        "fontSize": 0,
                                                        "maxWidth": 0,
                                                        "pointerWidth": 0
                                                },
                                                "legend": {
                                                        "enabled": false,
                                                        "accessibleLabel": "",
                                                        "align": "center",
                                                        "labelText": "",
                                                        "markerType": "circle",
                                                        "rollOverGraphAlpha": 0
                                                },
```

```
                                                "titles": [],
                                                "dataProvider": [
                                                        {
                                                                "category": "Stuffed",
                                                                "column-1": z1
                                                        },
                                                        {
                                                                "category": "Regular",
                                                                "column-1": z2
                                                        },


                                                ]
                                        }
                                );
                        </script>
                </head>
                <body>
                        <div id="chartdiv" style="width: 100%; height: 400px; background-color: #FFFFFF;" ></div>
                </body>
</html>
"""
    crusts=crusts.replace("z1",str(yescount))   #Replaces 'z1' (etc) in the HTML string above with actual value
    crusts=crusts.replace("z2",str(nocount))
    self.webViewcrust.setHtml(crusts)    #Sets the webview in the GUI to display the HTML pie chart in the string above
    sqlextra="""SELECT ExtrasID FROM ProductLine"""    #Fetches all extras ever ordered
    cur.execute(sqlextra)
    extrasdata=cur.fetchall()
    garliccount=0
    wedgescount=0
    for each in extrasdata:    #Tallies up each type of extra
      if each[0]==1:
        garliccount=garliccount+1
      if each[0]==2:
        wedgescount=wedgescount+1
    extras="""
<!DOCTYPE html>
<html>
                <head>

                        <script type="text/javascript" src="https://www.amcharts.com/lib/3/amcharts.js"></script>
                        <script type="text/javascript" src="https://www.amcharts.com/lib/3/pie.js"></script>


                        <script type="text/javascript">
                                AmCharts.makeChart("chartdiv",
                                        {
                                                "type": "pie",
                                                "angle": 11.7,
                                                "balloonText": "[[title]]<br><span style='font-size:14px'><b>[[value]]</b>
([[percents]]%)</span>",
                                                "depth3D": 15,
                                                "innerRadius": 20,
                                                "labelRadius": 10,
                                                "minRadius": 0,
                                                "marginBottom": 0,
                                                "marginTop": 0,
                                                "outlineThickness": 2,
                                                "pullOutOnlyOne": true,
                                                "startDuration": 4,
                                                "startEffect": "elastic",
                                                "titleField": "category",
                                                "valueField": "column-1",
                                                "theme": "default",
                                                "allLabels": [],
                                                "balloon": {
                                                        "animationDuration": 0,
                                                        "borderThickness": 1,
                                                        "fadeOutDuration": 0,
                                                        "fontSize": 0,
                                                        "maxWidth": 0,
                                                        "pointerWidth": 0
                                                },
                                                "legend": {
                                                        "enabled": false,
                                                        "accessibleLabel": "",
                                                        "align": "center",
                                                        "labelText": "",
                                                        "markerType": "circle",
```

```
                                                    "rollOverGraphAlpha": 0
                                                },
                                                "titles": [],
                                                "dataProvider": [
                                                    {
                                                        "category": "Garlic <br> bread",
                                                        "column-1": z1
                                                    },
                                                    {
                                                        "category": "Potato <br> Wedges",
                                                        "column-1": z2
                                                    },


                                                ]
                                            }
                                        );
                                </script>
                        </head>
                        <body>
                                <div id="chartdiv" style="width: 100%; height: 400px; background-color: #FFFFFF;" ></div>
                        </body>
</html>
"""
      extras=extras.replace("z1",str(garliccount))    #Replaces 'z1' (etc) in the HTML string above with actual value
      extras=extras.replace("z2",str(wedgescount))
      self.webViewextra.setHtml(extras)     #Sets the webview in the GUI to display the HTML pie chart in the string above
   def back(self):    #Back function
      self.hide()
      self.newwindow=FourteenthWindow()
      self.newwindow.show()

class EighteenthWindow(QtGui.QMainWindow, win18):   #Postcode Checker GUI
   def __init__(self, parent=None):
      QtGui.QMainWindow.__init__(self, parent)
      self.setupUi(self)    #Setup UI
      self.okbutton.clicked.connect(self.ok)
      global verifiedpostcode
      verifiedpostcode=True
      gmaps="""
<!doctype html>
<html>
  <head>
    <style type="text/css">
      #map
      {
        height:285px;
        width:100;
        display:block;
      }
    </style>


    <script src="http://maps.googleapis.com/maps/api/js?key=AIzaSyDRok8mK95-H3pDP2jvOB4DF9XtXLsCP2s&sensor=false&libraries=geometry"
type="text/javascript"></script>
                        <script>
                          var geocoder;
                          var map;


                          function initialize() {
                            geocoder = new google.maps.Geocoder();
                            var latlng = new google.maps.LatLng(51.6409256,-0.73800);
                            var mapOptions = {
                              zoom: 11,
                              center: latlng,
                              disableDefaultUI: true,
                              zoomControl: true,
                              mapTypeControl: false,
                              scaleControl: true,
                              streetViewControl: false,
                              rotateControl: false,
                              fullscreenControl: false,
                              mapTypeId: google.maps.MapTypeId.ROADMAP
                            }

                            map = new google.maps.Map(document.getElementById('map-canvas'), mapOptions);
                            var shoplocation = new google.maps.LatLng(51.6409256,-0.73800);
                            var shopmarker = new google.maps.Marker({position:shoplocation});
```

```
                    shopmarker.setMap(map);
                    var radius = new google.maps.Circle({
                    center: shoplocation,
                    radius: 10000,
                    strokeColor: "#0000FF",
                    strokeOpacity: 0.4,
                    strokeWeight: 1,
                    fillColor: "#0000FF",
                    fillOpacity: 0.05
                    });
                    radius.setMap(map);

                    codeAddress();

                   }


                   function codeAddress() {
                    var address = 'z1';
                    geocoder.geocode( { 'address': address}, function(results, status) {
                      if (status == google.maps.GeocoderStatus.OK) {
                        map.setCenter(results[0].geometry.location);
                        var newmarker = results[0].geometry.location;


                        var marker = new google.maps.Marker({
                          map: map,
                          position: results[0].geometry.location
                        });
                        var shopcoord = new google.maps.LatLng(51.6409256,-0.73800);

                        var p2 = new google.maps.LatLng(51.6409256,-0.73800);



                        //traveldi=calcDistance(shopcoord, newmarker);
                        //alert(traveldi)



                    }))
                   }

                   //calculates distance between two points in km's
                   function calcDistance(shopcoord, newmarker) {
                    //alert(shopcoord)
                    //alert(newmarker)
                    //alert("HH")
                    //alert(calcDistance)
                    return (google.maps.geometry.spherical.computeDistanceBetween(shopcoord, newmarker) / 1000).toFixed(2);
                   }



                    google.maps.event.addDomListener(window, 'load', initialize);
                  </script>

                  <div id="map-canvas" style="width:100%;height:285px;"></div>

  </head>

</html>

"""
    gmaps=gmaps.replace("z1", postcodetoverify)   #Replaces 'z1' in the HTML string above with the postcode that needs verifying
    self.webView.setHtml(gmaps)    #Sets the webview in the GUI to display the HTML Google Maps API in the string above
    postcode=postcodetoverify
    try:
      from urllib.request import urlopen    #import modules to access a webpage
      response = urlopen("https://api.postcodes.io/postcodes/%s" %(postcode))  #Calls separate postcode API
    except:
      verifiedpostcode=False    #HTML webpage displayed in case of failure
      self.rangelabel.setText("Failed to determine eligibility.")
      self.webView.setHtml("""
                  <center><h1>An Error Occured</h1></center>
```

```
                    <center><p>The Postcode Checker cannot find this address.</p></center>

                    <p>This could be for a number of reasons:</p>
                    <ul>
                    <li>The postcode you entered is invalid - Check your postcode and try again.</li>
                    <br>

                    <li>You have no internet connection - Check your connection and try again.</li>
                    </ul>
                    """)

        else:
            full=response.read().decode("utf8")  #If response received, it is decoded
            import json  #import json module
            result1 = json.loads(full)
            lat=result1['result']['latitude']  #Pick out logitude and latitude from data
            long=result1['result']['longitude']
            from math import radians, cos, sin, asin, sqrt  #import maths functions
            def haversine(lon1, lat1, lon2, lat2):       #Function for haversine formula, used to work out distance between 2 coords
                lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
                dlon = lon2 - lon1
                dlat = lat2 - lat1
                a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
                c = 2 * asin(sqrt(a))
                r = 6371
                return c * r
            traveldist=haversine(long, lat, -0.73800, 51.6409256)      #Calls function and passes in parameters
            if traveldist < 10:      #Sets verifiedpostcode to true/false depending on whether postcode is less than 10km away or not
                verifiedpostcode=True
            else:
                verifiedpostcode=False
            if verifiedpostcode==True:  #Changes label and button text depending on whether verified or not
                self.rangelabel.setText("Your postcode is within our delivery range! \n Distance from shop: %s km (10 km max)" %(format(traveldist, '.2f')))
                self.okbutton.setText("Continue")
            else:
                self.rangelabel.setText("Your postcode is outside our delivery range! \n We deliver up to 10 km, you are %s km away." %(format(traveldist, '.2f')))
                self.okbutton.setText("Go back")
    def ok(self):  #Back button
        self.hide()

def hasher(password):  #The hash function used to hash a password
    salt = uuid.uuid4().hex
    return hashlib.sha256(salt.encode()+password.encode()).hexdigest()+":"+salt

app = QtGui.QApplication(sys.argv)
homescreen = SecondWindow(None)     #Loads the first window
homescreen.show()
app.exec_()
```

Evaluation:

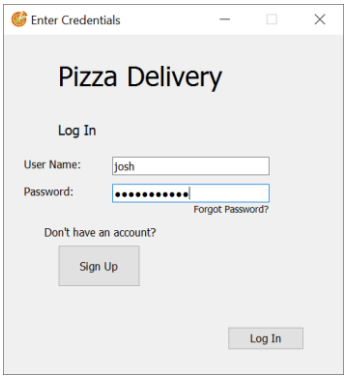
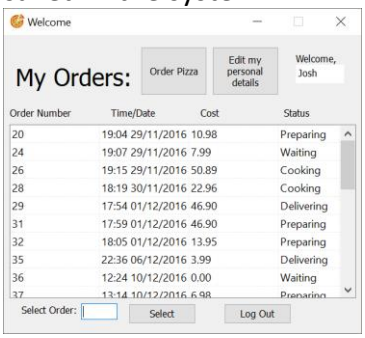
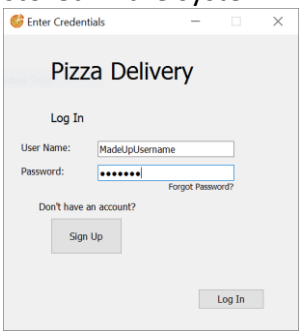
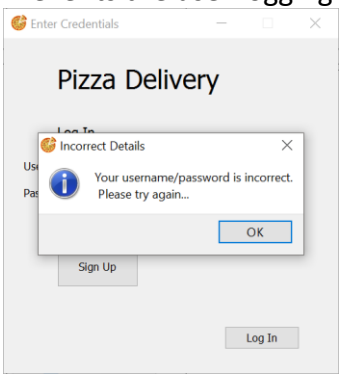
Testing to inform evaluation:

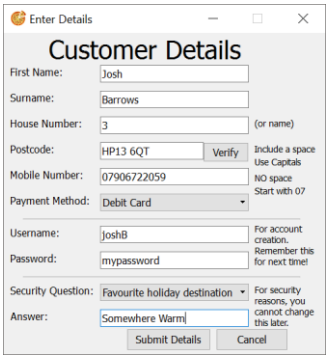
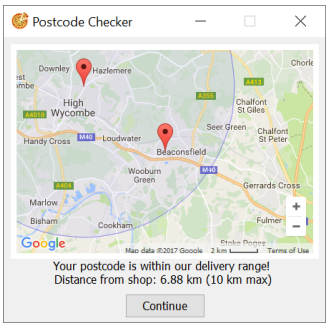
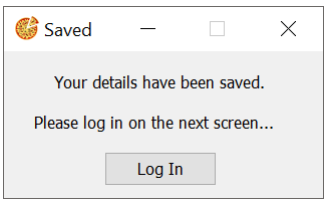
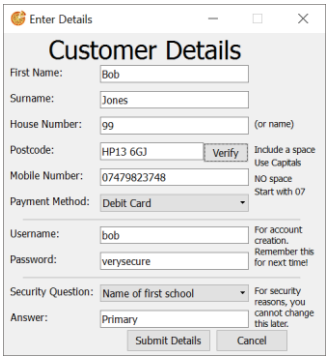
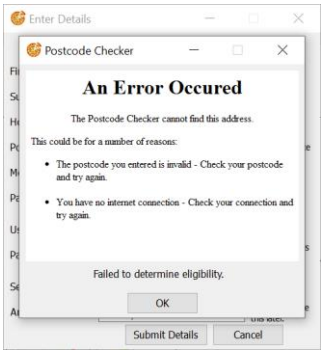
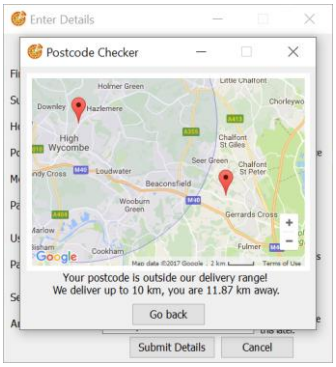
Post-development Testing:

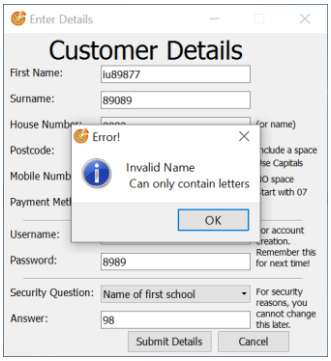
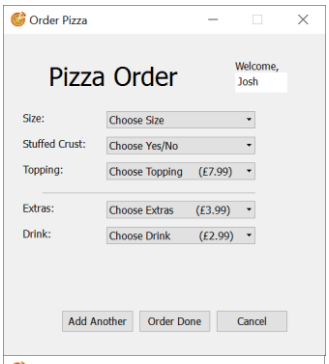
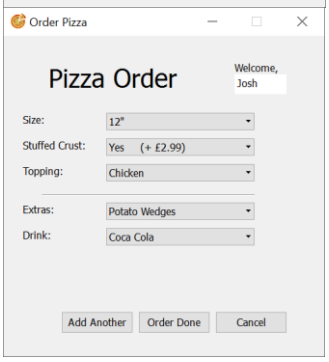
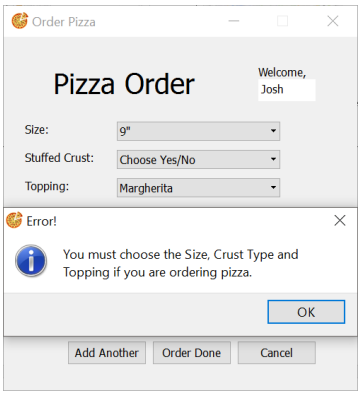
Since the completion of my program, I have carried out rigorous and thorough post-development testing, which covers all aspects and features of my program, and also tests the robustness of my software by entering invalid data where possible and seeing whether my validation stops it before a serious error occurs.

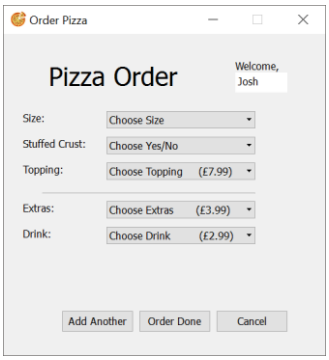
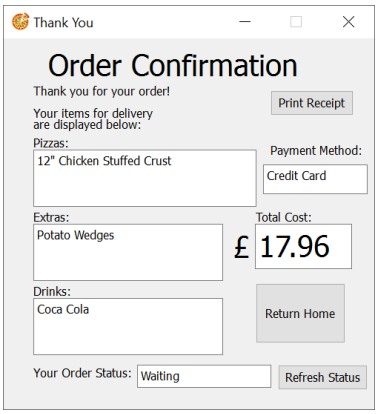
Post-development testing for function:

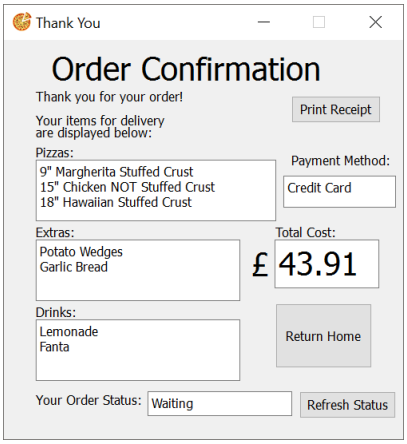
In this section of my testing I have listed all the different functions of my program and ensured they work as expected. I have tested everything I set out to do as defined in my Success Criteria in my Analysis.

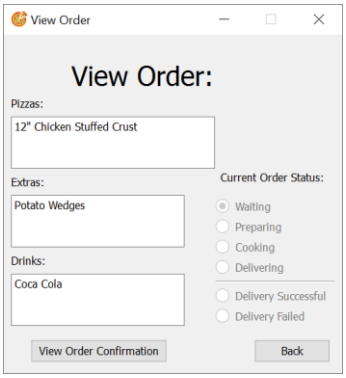
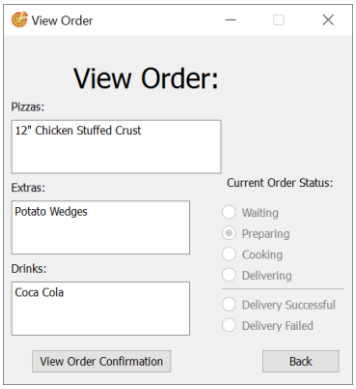
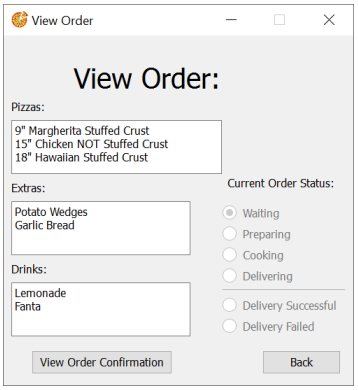
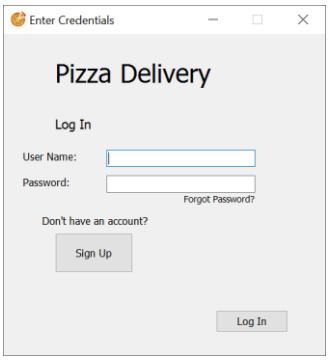
What testing (Success Criteria):	**Evidence of testing: Test 1:**	**Evidence of testing: Test 2: (if applicable)**	**Does it work?**
The customer can…			
Login with their details, which will bring up their details previously entered into the system	Entering the correct login details:  Loads the user's correct orders, saved in the system: 	Logging in with details not stored in the system:  Prevents the user logging in: 	Yes

New users can sign up to the system and enter their details, such as their name and address	The user can enter correct/validated information into the program:  The Google Maps API is used to determine eligibilty based on distance to the pizza shop (RGSHW):  And an account is created:  If the details entered are invalid the user cannot create an account without updating them: (EG: Invalid name)	Making another account:  An invalid postcode is entered, as it does not exist:  The program displays the preprogramed error message, with some useful diagnostics. If a postcode is entered that is too far away for delivery:  The program will show their location which is outside the delivery radius, and inform	Yes

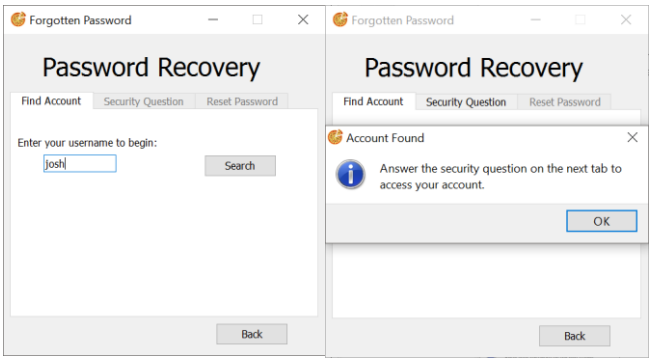
		them they are too far away and give them the actual distance.	
Order from the options provided, they can choose: pizza size, the option of stuffed crust, topping, extras and drinks	Ordering food items – options include choice of pizza (size, stuffed crust and topping), extras and drinks:  	Not complete pizza order:  The screenshot above demonstrates what happens if an order is not complete – i.e. the user orders a pizza topping and size but does not specify crust type	Yes

<table>
<tr><td>Order more than one pizza or drink</td><td colspan="2">Pressing “Add Another” reloads the above page after saving their first choices so that more items can be ordered.
 
</td><td>Yes</td></tr>
<tr><td>Receive a full output as a confirmation of their order, including all items ordered and a total cost</td><td> 

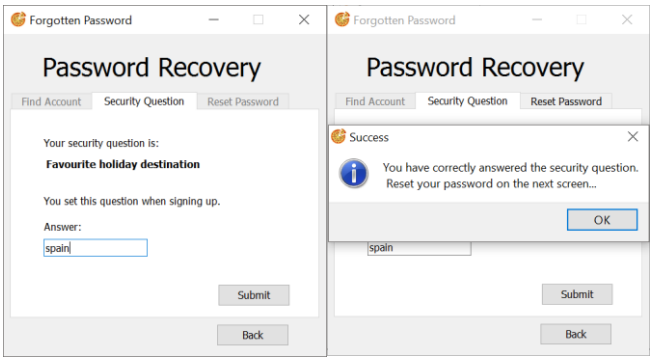
This is the order confirmation for the order shown above. It contains all items ordered and a total cost, as specified in the success criteria.</td><td>The following order confirmation includes multiple pizzas on one order:
 
</td><td>Yes</td></tr>
<tr><td>Receive a discount if they are a recurring customer</td><td colspan="2">No evidence available.</td><td>No</td></tr>
<tr><td>See an up to date order status, which tells the user the progress of their pizza</td><td>Pressing “Refresh Status” on the screenshot above will fetch the latest status update for that current order, so may change if the production of the pizza has moved forwards.
Also, the user can view any of their previous orders which shows them the order status. The below order details are from the order above:</td><td>The order above can be accessed later on, once the order has been submitted and the user has moved on:</td><td>Yes</td></tr>
</table>

	 When a staff member changes the current order status, this changes the order status that the user sees: 		
Recover their password if they forget it	The user can press the “Forgot Password?” button found on the login screen:  The new window that loads contains 3 tabs that the program automatically shifts between as the user completes each stage, as shown below.		Yes

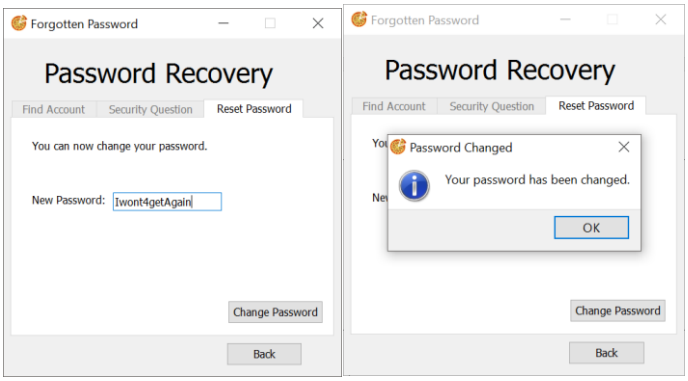
First stage – finding account in database:

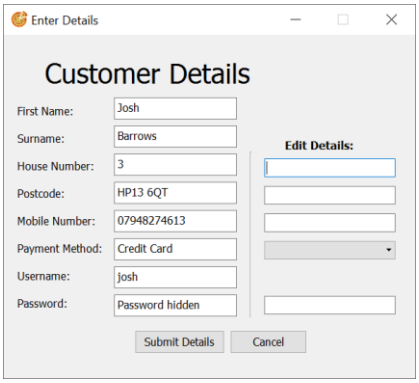
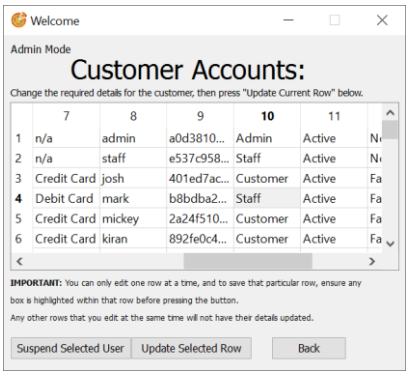


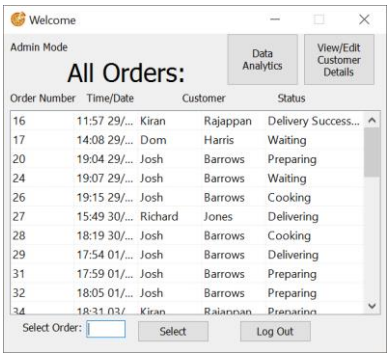
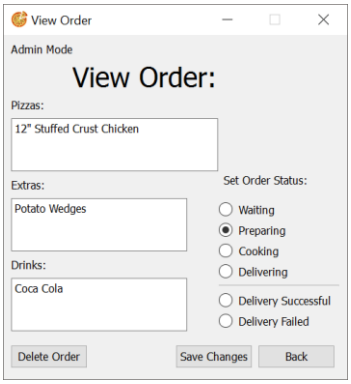
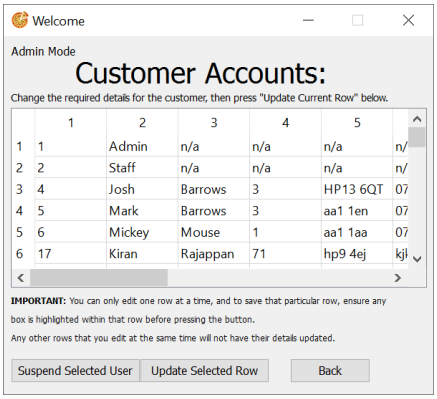
Second stage – answering the security question:

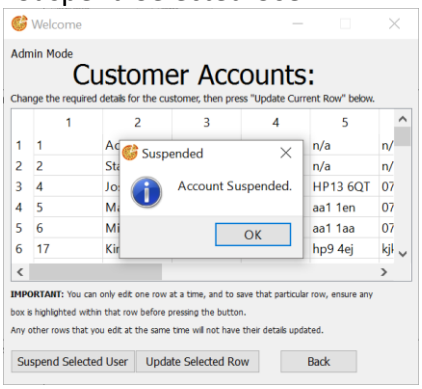
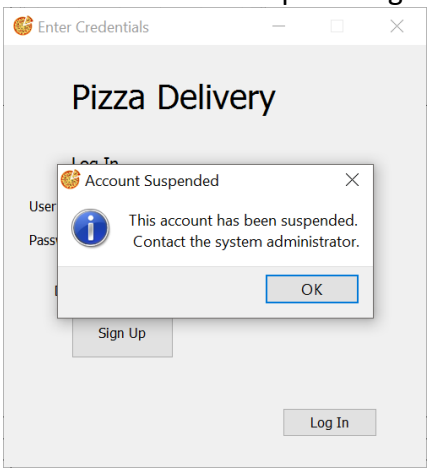
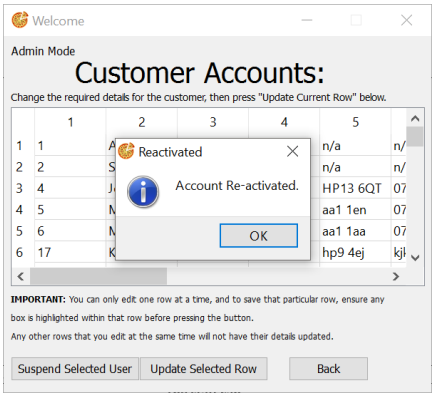


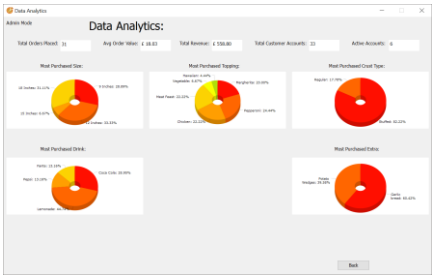
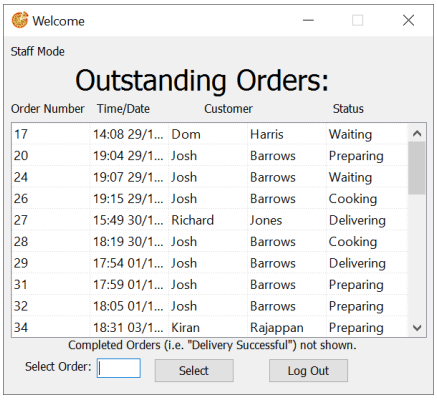
Third Stage – setting a new password:

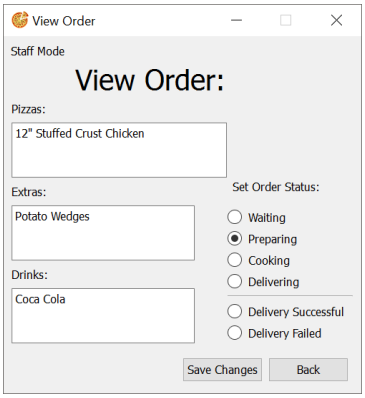
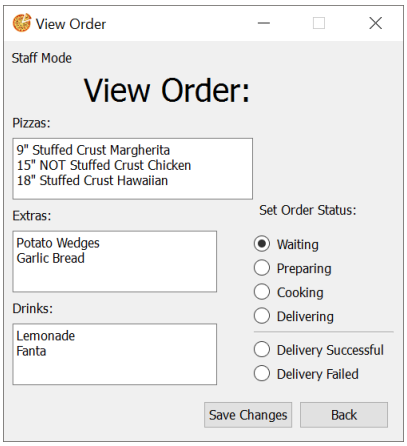


Update their personal information	The user can update most of their personal information:  The current details are displayed with the option to change your address, payment method and password. Customer name and username cannot be changed retrospectively because the username must be unique to avoid conflicts when logging in.	Yes
The administrator can…		
Add special accounts such as chefs and delivery men logins, which would have access to the full orders table	To promote a standard account to staff or administrator privileges, the administrator has to load the user accounts table and change the “Account Type” column to “Staff” or “Admin” for the required account.  	Yes

View all the outstanding orders	The administrator's dashboard shows all orders stored in the database:  Each order can be viewed and its status updated as required.	Yes
Delete Orders	The administrator can view each order, and has the special privilege of deleting each order. This maintains data referential integrity because the order is removed from both the Orders Table and the ProductLine table. The admin can press "Delete Order", should they need to:  	Yes
Edit Customer details	The administrator has direct access to the customer details stored in the table, and can edit customer's details one at a time by pressing "Update Selected Row" when the changes have been made to that row:  	Yes

Suspend users	Users can be suspended by selecting the user and pressing “Suspend Selected User”:  When the user attempts to logon they receive this message:  Accounts can be re-activated at any stage by pressing the same button on a selected suspended user:  	Yes

View Data Analytics	The administrator has a wide range of data analytics at their disposal, which can show the total orders placed, the average order value, total revenue, total customer accounts and number of active accounts. There also 5 HTML graphs that take data from the ProductLine table and display it in an easy to read format of a pie chart: 	Yes
The staff can…		
Login to the system with their credentials and be recognised as a higher-profile user, so are presented with a different screen	When logging in, the program detects whether the user is a staff member. Staff members are shown a different dashboard to that of a customer or administrator, as shown in the next test.	Yes
See orders that need to be cooked and delivered	The staff member can see all the outstanding orders:  Any orders with status as “Delivery Successful” are hidden from this view so the staff can only see the orders that need dealing with.	Yes

Change the status of orders so the customer can get an update on order progression	The staff can see each individual order so they know what to prepare and cook, and can update the order status: 	Another example of staff viewing an order: 	Yes

Post-development testing for robustness:

In this section of my testing I have listed all the different inputs found within my program and have entered test data that is invalid to see how the program's validation holds up. I have used the test data examples that can be found in the validation table in the Design section.

Where validated (Input):	**Test Data used:**	**Did the program withstand?**
Signing up – First Name	Josh1234 [Blank Input] Joshjoshjoshjosh	Yes
Signing up – Surname	Barrows9876 [Blank Input] barrowsbarrowsbarrows	Yes
Signing up – House Number	[Blank Input] 12345678901234567	Yes
Signing up – Postcode	HP136QT (no space) hiosfh898 H9 1DE 97G E35	Yes
Signing up – Mobile Number	0790fhsiche 0790123456789 99070707077	Yes
Signing up - Username	[Enter a username already known to be stored in database]	Yes
Signing up – Security Question	[Blank Input]	Yes

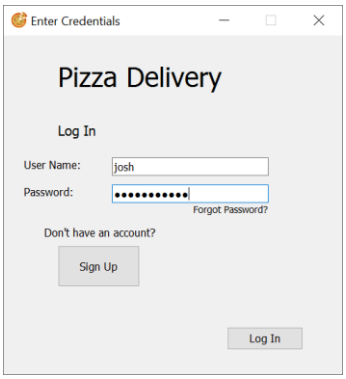
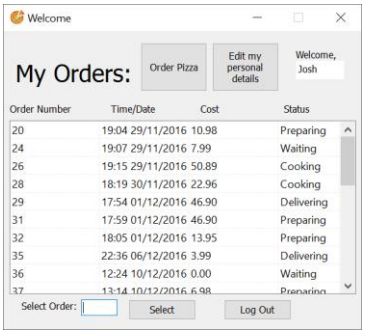
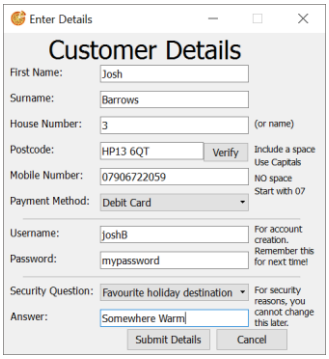
Signing up – Security Answer	[Blank Input]	Yes
Signing up – Password	Passw	Yes
Order Overview Screen – Select Order	Fdijij [Order number that doesn't exist]	Yes
Ordering Pizza – Size, Stuffed Crust, Topping	[Attempt to order a pizza topping and not specify its size or crust type]	Yes
Edit personal details – House Number, Postcode, Mobile Number, Password	As above	Yes
Password recovery – Setting a new password	Passw	Yes

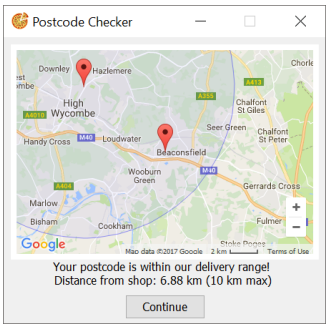
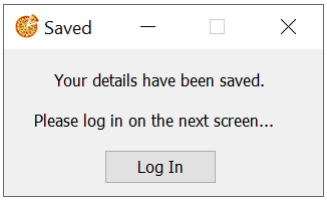
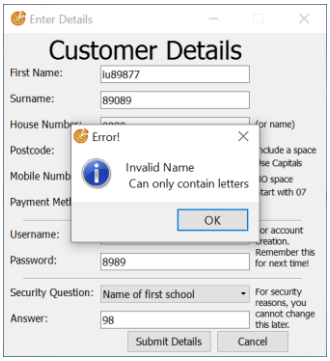
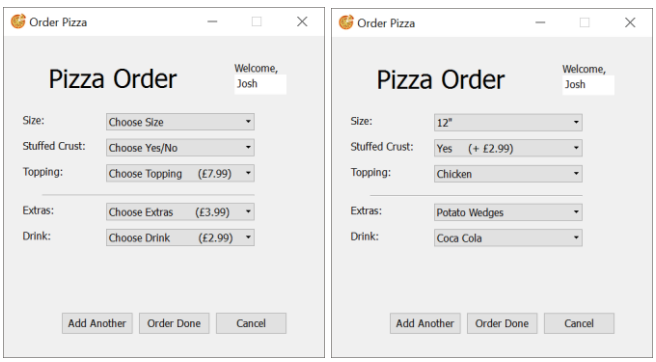
Usability Features:

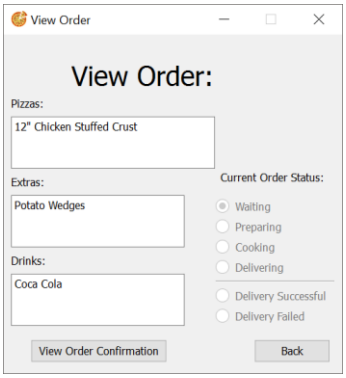
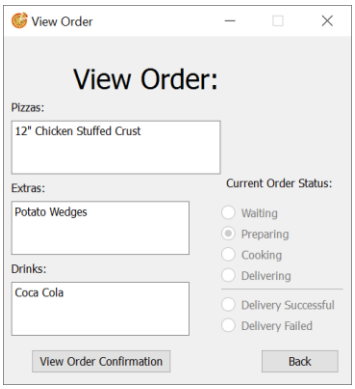
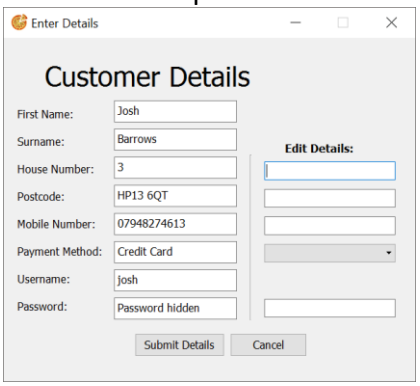
The usability features of my program were defined in the Design section of my program and each of these have been developed into working solutions. They are listed below:

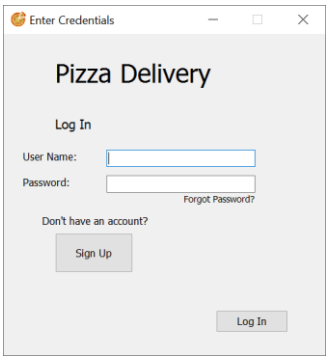
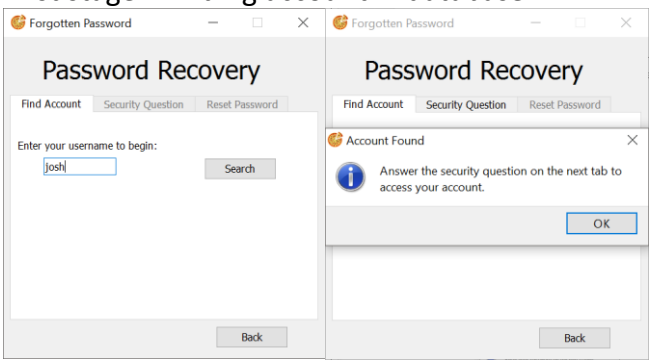
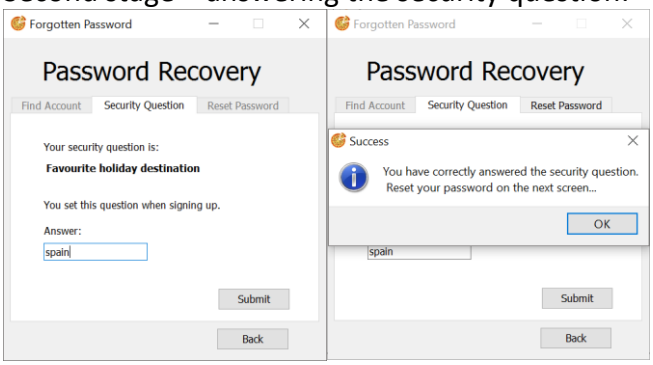
1) Logging in to the system
2) Creating a user account
3) Placing an order
4) Customer reviewing previous order
5) Customer editing personal details
6) Password Recovery
7) Staff View Orders
8) Admin Options

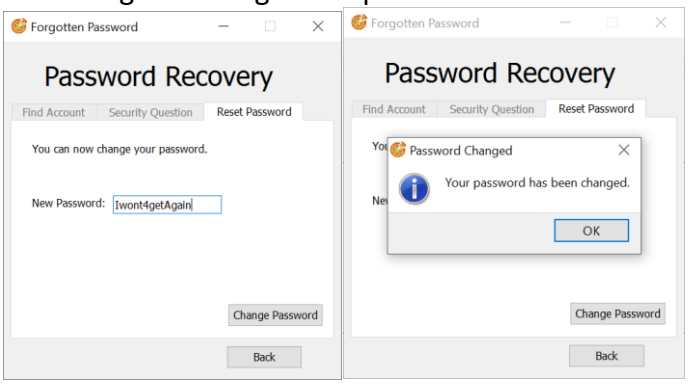
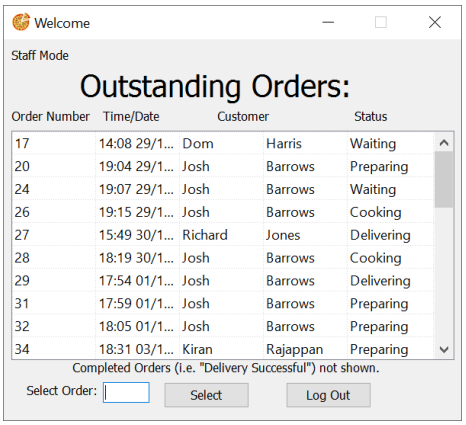
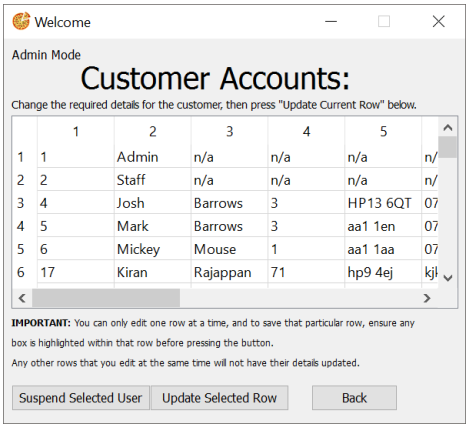
These features have also been tested in my 'Post-Development testing for robustness' above, so are only duplicated below to demonstrate that they work as effective Usability Features.

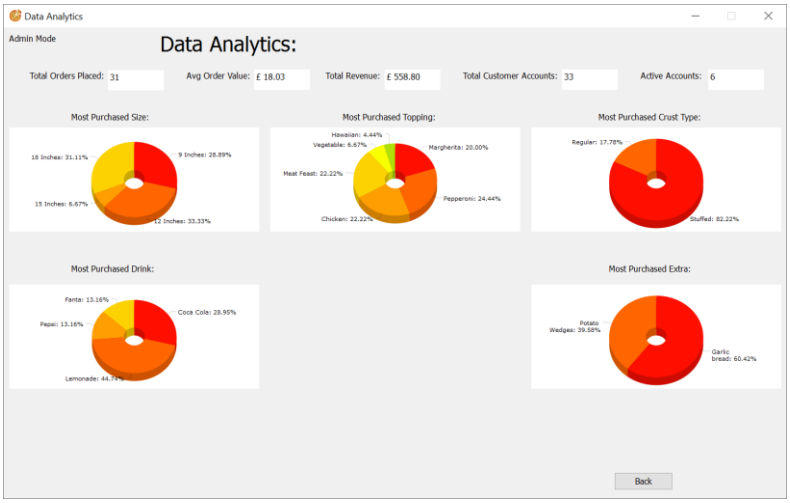
Usability Feature:	Annotated Evidence of Usability Testing:	Does the Usability Feature work as intended?
Logging in to the system	Entering the correct login details:  Loads the user's correct orders, saved in the system:  	Yes
Creating a user account	The user can enter correct/validated information into the program:  	Yes

	The Google Maps API is used to determine eligibilty based on distance to the pizza shop (RGSHW):  And an account is created:  If the details entered are invalid the user cannot create an account without updating them: (EG: Invalid name) 	
Placing an order	Ordering food items – options include choice of pizza (size, stuffed crust and topping), extras and drinks: 	Yes

Customer reviewing previous order	The user can view any of their previous orders which shows them the order status. The below order details are from the order above:  When a staff member changes the current order status, this changes the order status that the user sees:  	Yes
Customer editing personal details	The user can update most of their personal information:  The current details are displayed with the option to change your address, payment method and password. Customer name and username cannot be changed retrospectively because the username must be unique to avoid conflicts when logging in.	Yes

Password Recovery	The user can press the “Forgot Password?” button found on the login screen:  The new window that loads contains 3 tabs that the program automatically shifts between as the user completes each stage, as shown below. First stage – finding account in database:  Second stage – answering the security question: 	Yes

	Third Stage – setting a new password: 	
Staff View Order	The staff member can see all the outstanding orders:  Any orders with status as “Delivery Successful” are hidden from this view so the staff can only see the orders that need dealing with.	Yes
Admin Options	The administrator has direct access to the customer details stored in the table, and can edit customer’s details one at a time by pressing “Update Selected Row” when the changes have been made to that row: 	Yes

	The administrator has a wide range of data analytics at their disposal, which can show the total orders placed, the average order value, total revenue, total customer accounts and number of active accounts. There also 5 HTML graphs that take data from the ProductLine table and display it in an easy to read format of a pie chart:  	

User Feedback from End User:

The customer interviewee in the Analysis, Ben Smith has since tested the software and provided user feedback on the final program from a customer's point of view. Ben has had no input into the coding or other design of the program and hence does not understand how the actual program works. It is up to him to test the program and provide feedback on how well it works. It is essentially "black box testing" because he can only see the inputs and outputs with little idea of what happens between.

After using the program, Bens feedback was:

- The program provides the functions originally set out
- There is a wide range of items to be delivered
- The user interface could be improved by adding images of the available pizza choices to help pick one

- The delivery radius of the shop was generous for free delivery and likely to attract more customers as a result
- The textboxes and other buttons on the Order Confirmation screen are all different sizes which was confusing and not particularly aesthetically pleasing, which limits the usability of the program

Furthermore, the entire program has been tested by the prospective buyer of my software, Adam Jones, who has the final say over whether the program meets his requirements and is fit for purpose. Likewise to Ben Smith, Adam Jones has provided feedback based on using the program for a while:

- It is easy to manage the system orders, because each can be viewed, updated and deleted as necessary
- The data analytics window is very useful for showing the statistics of the business and also for displays the most popular items to order using easy to understand interactive pie charts
- The text on the Customer Details screen is quite small, which makes it difficult to read. This damages the usability of the program and the font size should be increased in future development of the program
- Also on the customer details GUI, the table to view all the details is quite small, so only a few accounts can be seen at once. This limits the productivity of the application because I have to keep scrolling to find the required information. Furthermore, as more accounts are added it will become increasingly difficult to find the required account due to a lack of search function
- Overall, the program is fit for purpose and meets all the requirements we originally agreed upon

Evaluation of solution:

Comparing the final product to initial success criteria:

As my testing above demonstrates, the development of my program has been a success. I have tested each of my success criteria, and all the functions of my program have been implemented successfully. Furthermore, my robustness testing has also been a success and shows that my program has thorough validation which means the program cannot be crashed if an invalid input is entered. This is very important for final software, so that the program cannot be broken by a malicious user or by an unknowing user who enters incorrect data.

All success criteria listed and successfulness evaluated:

- *The customer can effectively use the program to its fullest extent. They can:*
    - *Login with their details, which will bring up their details previously entered into the system*
        - **This has been successfully implemented – the user can enter their username and password to login, and then they can view all the details stored about them**
    - *New users can sign up to the system and enter their details, such as their name and address*
        - **This has been successfully implemented – a signup window exists to request all the required data about a new user, who must enter their information to get an account**
    - *Can then order from the options provided, they can choose: pizza size, the option of stuffed crust, topping, extras and drinks*
        - **This has been successfully implemented – the user can choose from many different sizes of pizza, crust types and toppings, as well as choosing extras and drinks**
    - *Order more than one pizza or drink*
        - **This has been successfully implemented – the user can press "Add Another" when ordering to add more items to the order**
    - *Receive a full output as a confirmation of their order, including all items ordered and a total cost*
        - **This has been successfully implemented – the order confirmation GUI shows all this information once the order is complete**
    - *Receive a discount if they are a recurring customer*
        - **This has not been implemented – due to time constraints, this success criteria has not been met. It has however been recognised as**

a limitation of the program and could be included in future development

- *See an up to date order status, which tells the user the progress of their pizza*
  - **This has been successfully implemented – the user can see the progress of their order in the order confirmation GUI and also when reviewing an order**
- *Recover their password if they forget it*
  - **This has been successfully implemented – a 3 stage recovery process has been developed to find the account, then ask for the answer to the predetermined security question and then finally allowing them to change their password**
- *Update their personal information*
  - **This has been partially successfully implemented – the user can change most of their details even once an account has been created, however to change certain details the administrator must do it for them, such as in the rare event of changing their name**

- *The program will be regulated by an administrator account:*
  They can:
  - *Add special accounts such as chefs and delivery men logins, which would have access to the full orders table*
    - **This has been successfully implemented – whilst viewing all accounts in the system the administrator can change any of the account types to "Staff" or "Admin" or back to "Customer"**
  - *View all the outstanding orders*
    - **This has been successfully implemented – all orders in the system can be seen by the admin on the admin dashboard**
  - *Delete Orders*
    - **This has been successfully implemented – each order can be individually deleted by the admin**
  - *Edit Customer details*
    - **This has been successfully implemented – the whole customer details table can be viewed by the admin account, and each customer detail can be amended**
  - *Suspend users*
    - **This has been successfully implemented – each user can be suspended and hence prevented from logging in. Accounts can also be reactivated**

  - *View Data Analytics*
    - **This has been successfully implemented – the admin can see facts such as total accounts, total orders, total revenue, average order value and number of active accounts. There are also 5 3D pie charts which show amounts of each pizza size, crust type, topping, drink and extra sold**

- *Another special account is the staff's logon:*
  They can:
  - *Login to the system with their credentials and be recognised as a higher-profile user, so are presented with a different screen*
    - **This has been successfully implemented – when the user logs in, the program checks their account type and opens the correct window**
  - *See orders that need to be cooked and delivered*
    - **This has been successfully implemented – the program shows all orders that are not set to "Delivery Successful" on the staff dashboard**
  - *Change the status of orders so the customer can get an update*
    - **This has been partially successfully implemented – the staff can select any order, view its items and change its order status as the order progresses. However, the customer can see the changes if they look for them, but will not get a notification update as the success criteria suggests**

Possible future additional features:

Given the time constraints surrounding the project, my program still manages to include lots of features, however there is always room for improvement. Future additional features could see the program expanding out into other ventures including other forms of takeaway food, for example.

Furthermore, the system could be expanded to have time controls on orders, such as set times for ordering pizza so that food cannot be ordered at midnight, and allowing pizza to be ordered in advance from when its wanted. The idea of stock management could also be considered as this would allow the program to manage the shops inventory and automatically stop selling products that have sold out and alert the administrator that more must be ordered in.

Evidence and justification of Usability Features:

As my usability testing in the above section demonstrates, the usability features of my program have been shown to work as they are supposed to and hence there is sufficient evidence for my usability features.

The usability features of my program are justified because they are the underlying procedures and functions that allow the program to work as required.

Maintenance:

The program should be able to be maintained quite easily by the administrator of the system. The program should function alone without any issues, however over time a backlog of orders may appear that could make it challenging for staff to see the newest orders as the table they see starts from the oldest first. The administrator could therefore be required to archive all older orders, although this would be slow and laborious.

Other maintenance issues for the programmer to consider are that is the shop wishes to change its range of goods or prices, this cannot be achieved through the GUI and therefore the developer would have to be on hand to change the coding to reflect the differences.

Should the shop change location, the Google Maps API would also have to be updated to reflect the new position of the marker on the map when signing up. This is hard coded so the programmer would also have to update this if the shop ever moved.

Big-O Complexity of algorithms:

Logging in uses a linear complexity because the program searches the database for the inputted username by using linear search. So as more users are stored in the system, the program will increase the amount of time it takes to find an account at the same rate that new accounts are added, assuming that the requested account is the last in the list. The user will likely not notice any delays anyway.

The algorithms used to add up the cost of an order when it is being confirmed use linear complexity, because the program loops for each extra item in the order, so the time to process the cost of an order increases linearly with the number of items ordered.

The algorithms used in the data analytics use different complexities depending on the sum. The algorithm to add up all the items in the previous orders is also linear because the amount of orders in the database directly correlates to the length of time taken to work out the ordered quantities.

The Big-O complexity of verifying a postcode is constant complexity, because only one postcode is searched for at a time and it takes the same length of time to determine a postcodes eligibility regardless of their distance from the shop.

The algorithms used to create the tables seen within the GUIs that show the orders/customer information/etc. use polynomial complexity, because they utilise nested 'for' loops, and hence for every column in the table it takes much longer to perform the calculations.

Limitations of the current solution:

There is only one admin and one staff logon and therefore no reports can be created to show best employee etc. which a business owner may find useful. My current version of the program does not allow for customers to receive discounts, so this is something that could be factored into a future release. Offering discounts would of course be at the discretion of the business owner.

Further limitations that must be recognised are the limited selection of platforms the software runs on. Currently the system is only designed for using on a computer, which is fine for the chef to use, but the delivery man would not be able to remotely update order statuses from their mobile to set the order status as delivery success or failure, and the current solution does not also accept external connections into the database so cannot be accessed or modified remotely which may be necessary for customer ordering and remote staff to use. The program only connects to the database when it is stored locally on the same computer currently, so when deployed to all customers to use would require a way of remotely accessing the database that is used to store orders.

Porting program to other platforms:

As mentioned just above, the current program is only designed to work on large screened devices that can run Python, PyQt and SQLite. The supported platforms are hence only Windows and MacOS. Porting the program to a mobile application would require lots of additional work, firstly because the language is not supported by either Android or iOS, the 2 main mobile operating systems around today, and secondly because each and every GUI window would have to be redesigned to fit the pixel densities and aspect ratios of smaller screens.

Designing the system to work with a Siri-like interface is a much more realistic possibility, because ordering takeaway food could easily be done by voice and would provide a user friendly way of doing so. Using a virtual assistant such as Siri or Alexa would require using their respective APIs but if set up correctly could allow users to place orders using their voice, such as "Order 2 stuffed crust 18inch Meat Feast Pizzas and a lemonade" would connect to the database and add the orders under the account holders name.

How the program might be modified to meet additional/changing requirements:

As mentioned in the maintenance section, the program may require future modifications to be viable for a specific business to use, if for example, their product range is altered.

The program could in the future be modified so that no maintenance would be required. This could be achieved by putting every single setting within the GUI so it can be modified easily. In my program, the shop location and delivery distance could both be edited by the administrator if these details were not hard coded into the program like they are currently.

How unmet/partially met Success Criteria could be addressed in future development:

Despite completing the majority of my original success criteria, some criteria were not successfully met, which are listed below:

- Loyalty discounts
- Push Notifications for status update
- Updating all personal details

Loyalty discounts could be implemented in a future iteration of the programs development, by looking at a user's order history and determining if they are eligible. The requirements to be eligible could be based upon the amount of orders ever placed, or it could be activated once a customer has spent a specific amount of money through the system. Alternatively, the system could analyse all the data in the system to identify the top 10% of spenders to pick out the most loyal customers and then given them a discount by taking money off at the order confirmation.

Push notifications for status updates could be addressed in future development by creating a notification panel on the side of the customer's dashboard, which would allow them to easy see the progress of their current order without having to manually search for their most recent order to track its progress like they have to currently.

The final partially unmet success criteria is for the user to be able to change all their details. To fix this, the current GUI that allows the details to be changed would have to have more input boxes added to it which would then allow the user to change their name, etc. instead of just their address and password.