INTERNATIONAL BACCALAUREATE DIPLOMA

Computer Science Higher Level Dossier

By Luke Geeson

VIK3

2012-13

Word Count: 49,333

Candidate number: dwt 611

School Number: 002769

Contents

Introduction

Many businesses will be familiar with methods of storing customer information. They are necessary for the operation of the business and law (data protection act). As a result, businesses will need a reliable way to input, process, output and store customer information. Whilst the old paper based system is workable when there is little information; paper based systems will often become dysfunctional and costly over time for the business and therefore a liability. A local company based in Ashford, Kent, has this problem; they are called "Pete's Pets." In this dossier, I have analysed exactly what the problem is; I have designed, implemented and maintained an electronic system that aims to tackle the task of storing customer information for this company whilst reducing the time and space required to do so. Thus making a system that is of value to Pete's Pets as well as a safety buffer for possible future expansion.

Stage A: Investigation and Analysis of the Problem

The information in stage A has been collected by observation of the business functioning, interviews (see appendix A Interviews) and asking employers how the existing system did work. The notes taken have been copied onto this document (see section A1: Description of the existing system). The sample data was taken from a clean document which would be filled in with the customer information; for reasons of customer confidence, I had to use the fictional character *John Smith* when illustrating the use of sample data in the old system.

A1: Description of the existing system

- Pete's Pets is a sole trader located in Ashford, in Kent, the business' main form of income is from the sale of domestic birds and animal food. Customers can order special goods through this company which are not normally displayed for the public. Items include prescription foods for pets and rare exotic birds which are legally raised and sold through the business. The business also sells standard pet goods such as *whiskers* brands of cat food and various toys for domestic animals to use.
- The business uses a paper-based system to store customer information. Customer information including **Name, Email Address, Home address** and **telephone number** are recorded on a form.
  A customer information form is as follows (for a hard copy see Appendix B Sample Data):

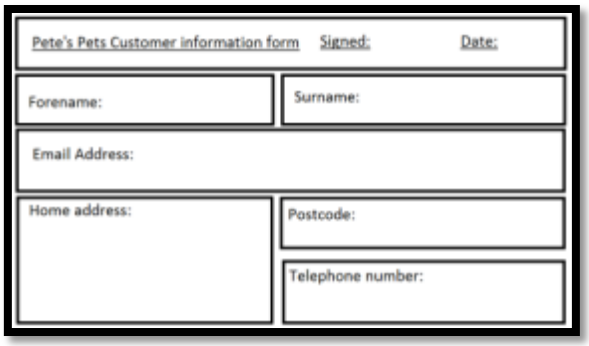


Figure 1: an empty customer form

- When a customer orders something, the business will fill in an order form which will be copied so that the business can store a copy with the customer file. Customer orders and receipts must be kept by law.
  A customer order form is as follows:

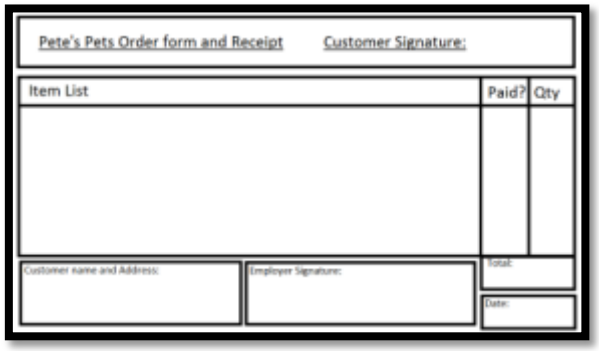
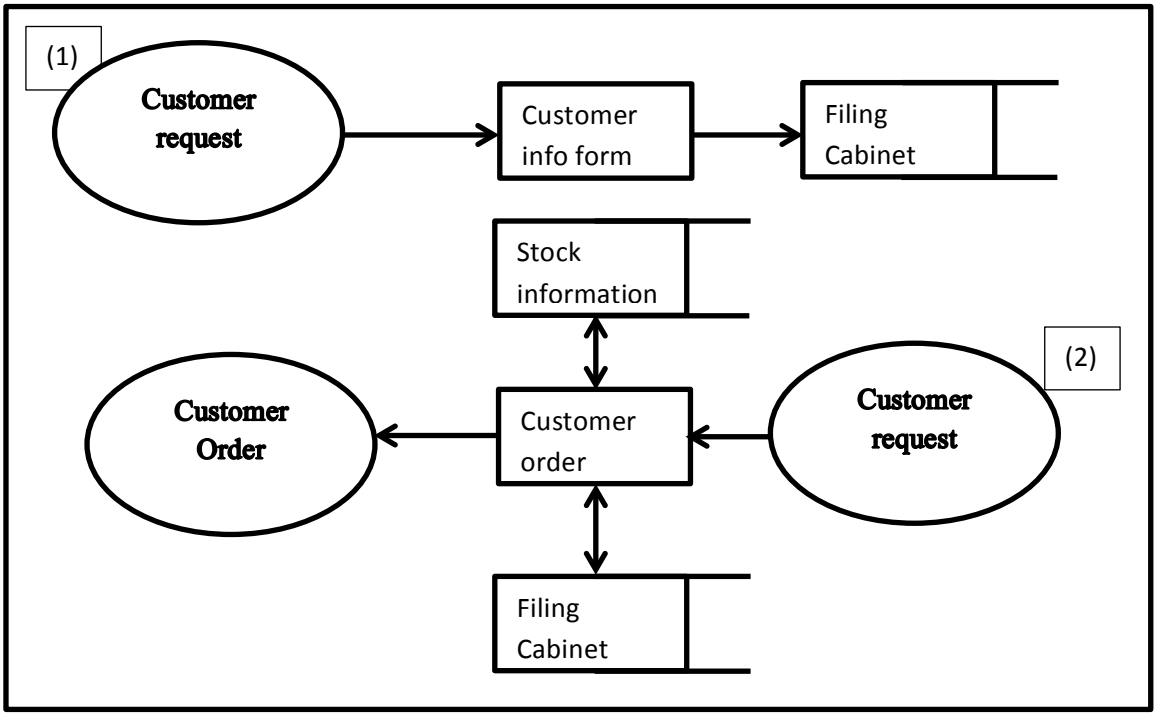


Figure 2: the customer order form

- Data is collected through **face-to-face** or **over-the-phone** customer interaction. Data obtained this way is recorded on A4 size **forms** which have specific fields that require filling.
- Once a form is completed with the required information, it is **stored with other forms.** Each customer has their own plastic sleeve to store forms in. Should a file need to be removed; it is located and **shredded with a cross-cut shredder**. This is **disposed of by general refuse collectors**. The waste is burnt in an industrial oven. Should data on a form need to be modified, the form containing the old data is located, the non-changing **data is copied onto a new form** and the **new data is input manually**. The old data may also be stored at the request of the customer but will otherwise be disposed of to save space.
- These forms are stored in a filing cabinet in a cupboard. The forms are **stored alphabetically** according to surname; "John Smith" would go after "James Burns." The order in which the forms are stored is checked monthly. The cupboard and cabinet are located inside the shop itself.
- When data from a form is required; the staff will **manually search** through the cabinet in order to find the form with the customer information.
- When a customer orders an item, the customer information is obtained from the forms or the customer themselves (should it be a first time customer or single sale). An order form is then filled out with the customer information, stock and item information and the date of purchase. Once the customer signature is acquired, a copy of this order form is **stored with the customer record** and the **copy sent/given to the customer as a receipt**. Standard in shop purchases do not apply to this; a standard checkout terminal is used, a small receipt is given and no information is needed.
- The forms are secured by a set of **keys** which locks both the cupboard and the cabinet. The cupboard and cabinet are located inside the shop itself which requires the keys to unlock. The shop is monitored by **CCTV** and the area surrounding the shop is **lit with street lamps**. CCTV is also in the shop itself. Recordings of the CCTV footage are stored on an external computer (owned by the security company) which is **backed-up** and a copy taken to the house of the business owner **every week**. The CCTV security is on a closed circuit and so the risk of data being intercepted is minimal. The **backed-up CCTV footage is stored in a safe** at the house. The keys are held by the manager who opens/closes the business daily to ensure data security. A fire detection system is installed in the shop and the filling cabinet itself is airtight to reduce the risk of a loss of data. The business is ensured. No back up of the paper-based system has ever been made due to the quantity of data and the risk of random error when copying the data.
- The business **employees are trained** to use the paper-based system so that they are aware of how to communicate with the customer and acquire appropriate information. This information is used to fill out the customer forms. This ensures a standard level of **data integrity** is met; employees are re-trained annually to ensure they are familiar with the system. There is no documentation or a user guide for the system. Training takes up to a week.
- Files are **checked annually by the manager and distinguished members** of staff to ensure that the data on the forms is consistent and valid. When a form is completed, it is **validated and signed** by employee that completed the form and the date of completion.

- There are over **200 individual forms** that have been completed (not including order forms which are kept as well) and the cabinet in which the forms are stored is almost full. The cupboard in which the cabinet is situated does not have the space for a second cabinet and the business does not have the funds required to build an expansion. Furthermore, there is no other area in the shop which has the level of security or the physical safe space that the cupboard does.
- Here are 2 data flow Diagrams to better show the flow of data in the old system.



1. Data flow for the creation/modification of customer files.
2. Data flow for a customer order.

- 3 of the 5 **employees are trained to use computers** and have at least GCSE (or O level) equivalent in ICT Qualifications. Currently the manager has a desktop (1 year old at the time of analysis - 2012) which is used to print the forms.
- The paper used is conventional A4 paper and the **business has been affected by the increase in prices of paper.** When the paper runs out, the manager must use money from the business to replenish resources. The printer used has expensive ink and is also costly for the business.
- All of employees have been affected by the paper-based system and all of them have said that they wish for a new system to replace the old. See appendix A Interviews for the staffs' thoughts on the paper-based system.
- The manager feels that the current system is not economically sound when compared to modern systems.
- All of the employees feel that the current system is slow and difficult to use.
- They have said that they would like a system that can manage the current database quickly and easily.
- Comments were made by employees about how the paper-based system is "out-dated" and "slow".

- The employees report that customer service has been affected by the speed at which they can serve as a result of the paper-based system.
- It is also reported that a capable computer system is installed but not effectively used in the shop.
- Overall, the interviews with the employees suggested that the employees were dissatisfied with the current paper-based system and that business was being affected as a result. Emphasis has been placed on the difficulty of using the customer forms as opposed to the order forms and the delivery process. The order forms must be kept by law. See Interviews for more information)

Sample data

One of the typical products that the system will stock is *Badminton Country Feeds- Poultry Corn*, this is a product typically bought by small businesses and families which own poultry (such as chickens) but do not own enough poultry to justify purchase of industrial amounts of feed. For reasons of customer confidence, in this sample of data I will be using a false name to illustrate the role of the paper-based system in a typical order. In this example, I will use *John Smith* as the name of the customer. (For hard copies, see appendix B Sample Data)

1. The customer phones up the business to order 1 bag of *badminton Country Feeds – Poultry Corn.* The employer fills in the customer form (or retrieves a previous one) as follows:

Figure 3: a customer form filled in

2. The employer then prepares the order for delivery and records the details on the order form. Two copies are created; one is stored with the customer file while the other is given to the customer with orders.

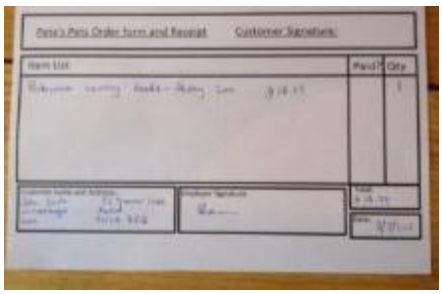


Figure 4: an order form filled in, a copy is sent with the order and a copy is paper clipped to the customer file

3. Once the customer has paid, signed for and received the order; the second order form is transferred to the customer file.

See appendix B Sample Data *for the hard copies of the sample data*

A1: Analysis of the existing system

The current paper-based system used has fundamental problems that prevent the business from functioning to optimum capability; however the system does also have some redeeming features that benefit the business.

Firstly, the format of data capture is a diverse and effective way to keep customer information. The fact that a telephone number and email address is used suggests that the business is making use of modern technology to an extent. However, there is the potential for human error when inputting customer information. This is tackled by annual training of staff to use the system, which costs time and money. Furthermore the annual data integrity check is also done manually which raises the potential for human error as well as taking time and therefore money to do. Secondly, the format of sorting the data is an effective one. Sorting databases by surname is widely used in many databases and serves to be both employee and customer friendly. However the method of sorting is widely open to human error. There is a chance that employers will put files away in the incorrect place to save time; this makes it harder to find the file later.

The existing paper-based system is generally time inefficient because of the very nature of the system. If a customer contacts the business to order a product, the employee has to leave the customer, go to the filling cabinet, and then search through 200+ files to find the right one. This generally takes minutes (more if items are not in order) and has the potential to lose sales and customer confidence. The files are sorted monthly to ensure this is prevented (for a while), however the very act of this will take time which the business could better use to improve its services. More time is wasted every year as a result of the need to retrain the staff to use this system and no documentation has been produced which would be better for the employees as they can "top up" their knowledge when they feel it is necessary rather than wasting a week every year for training.

Completing a form is inefficient in terms of time because an employer must manually fill in details whilst communicating with customers. The time of form completion depends on the speed of employer handwriting which is of course varied and an inconsistent and potentially negative experience for the customer. Furthermore, there is the potential for human error when filling in forms because the employer must decide to concentrate on the customer or the form. If mistakes are made, it will cost the business more time and money to rectify later on as they will have to contact the customer to validate or acquire information. Modification of information is also inefficient in because more paper is used to record new information and the old is often discarded (into waste).

The system is quite secure as it requires 3 keys to get to the customer information, uses fire precaution and is also monitored by closed circuit CCTV. This does not excuse the fact that the data is not backed-up. This could be very problematic in the future should they lose data. The system is secured from electronic threats however as the system is not connected to a network or any electronic system.

The methods of removing and disposing of files are both inefficient in terms of time and money. This is because it takes time to manually shred/prepare the waste appropriately and money because the refuse collectors charge an additional monthly fee of £50 to dispose of this waste. The waste is burnt

by the refuse company and therefore the system has a carbon footprint. The carbon footprint is made larger by the fact that the paper purchased is an unsustainable source which is further damaging to the environment. Forms are printed out by a standard inkjet printer which is costing the business money to run and is damaging to the environment because of the method of ink cartridge removal. Currently a capable computer system is being used for menial tasks and is wasteful for the business as well.

The storage of the files is another problem, although secure with CCTV, keys and fire precautions, the storage is becoming difficult for the employers because they are running out of room and cannot afford an expansion. An expansion that would mean the paper-based system would be even slower to operate and would require more resources. In addition, customer receipts must be stored by law which further reduces the space that the business has for new customers which will limit the potential capabilities of the business and potential clients in the future. The fact that 2 copies of receipts are made is also damaging to the environment.

The employee experience has also been affected by this system. The end-users of the current system are non-technical staffs that are qualified with O-levels in computing (or equivalent). In the interviews conducted (appendix A Interviews), various complaints were made about how the system was slow to use. The system was also called "tedious" and "out dated" which would suggest that the system is repetitive, the employers are not being stretched to their potential and the technology available is not being used. This is reinforced by the fact that the employers possess qualifications in ICT which are not being used effectively. The employer morale is evidently affected by this system for the above reasons. This is reinforced by the fact that all the employers of this business have voted in favour of a new system to replace the old. This is why I feel it is necessary to design, produce, install and maintain a new system for Pete's pets pet store.

A1: Description of how this problem has been solved in the past

- The existing paper-based system was replaced with an electronic system.
- Although **this is a different type of system**, it essentially s**hows the change from a paper based system to an electronic system** and shows a similar project which did the same thing
- I have decided to look at this because it shows how this problem has been solved in a similar fashion before
- A system has been developed that replaces paper-based Health records with electronic health records. HIM (health information management) has been moved on to electronic systems. This system is used in various forms throughout the world.
- An example of the NHS (National Health Service) electronic health record system is found here: http://www.nhscarerecords.nhs.uk/
- a detailed dossier can be found at: http://www.ironmountain.com/~/media/Files/Iron%20Mountain/Knowledge%20Center/Reference%20Library/White%20Paper/Sponsored/IDC/Moving%20from%20Paper%20to%20Electronic%20Health%20Records%20IDC%20Whitepaper.pdf
- Customer health records are stored on a computer system that is connected to a network so that other computers in the network can access customer information. These were previously paper records which had to be transferred via transportation.

- There is a hierarchical security system in which certain members of the organisation can access information that others cannot. This is also guarded by password protection.
- The volume of the data is in the millions, statistics of the volumes of data is not available to public view.
- A customer record is added, modified or removed through the use of a GUI in which employees can manage data in a quick and easy fashion. This is backed up to a secure copy of the data which is held on a secure server and an external host will manage. In England the NHS pays for British Telecomm to host NHS customer records on servers so that live and backup versions are constantly running at the Reigate and Harmondsworth data centres.
- When an appointment is made with a customer, the employee will link the customer record to said appointment. When the appointment is commenced, a medical professional will make appropriate changes to customer data (this is confidential) and save it so that it can be uploaded and updated on the server.
- The software has validation functions and data exceptions built in so that when the medical professionals input incorrect data it is flagged and not saved so that it doesn't crash the system.
- Customer record searches can be done by name, date of birth, address and national insurance number as well as many other fields linked to government records.

A1: Analysis of how this problem has been solved in the past

- The existing paper-based system was replaced with an electronic system.
- Although **this is a different type of system**, it essentially s**hows the change from a paper based system to an electronic system** and shows a similar project which did the same thing
- This is beneficial to the health company because they can store the data on a secure server which is much harder to access for malicious influences. Each hospital or health clinic can access customer health records and so data is much easier to transport and will save the company the time and money which would have been needed to transport paper-based health records.
- Managing customer health records was made much easier to do as they now have a system that is consistent throughout the clinics and hospitals, also the GUI was quicker and much easier to use than the paper based system.
- The potential for human error was reduced exponentially as validation functions were put in the system so that mistakes were reduced so much that a standard of data integrity was met. A standard of data integrity that may not have been present before.
- The health service will save money and resources as they do not have to print out individual records for each customer.
- The system was quicker to use and so more customers could be served, increasing the efficiency of the service.
- Data was made more secure by a hierarchical system that gave access to files based on rank in the company- this meant that customer confidentiality was met.

A2: Criteria for Success

The new system must address all of the problems highlighted in the analysis stage whilst keeping the good features of the existing system. The new system must be economically sound and capable of improving for future improvements and expansion. The new system must be quicker than the

previous and easier to use whilst being capable of storing all of the customer information with a lot of spare room for expansion. The new system must be able to do all of the functions of the previous system in a more efficient fashion. The new system must be secure and the data must be integral.

A2: The End-User

The end user will be the employees/employer interviewed in appendix A (see Interviews). The employer (Peter Smith) is largely concerned with the economics of the system but has experience using the system. However, I think it will be better to aim the system at the users who use it the most, such as Mary Kershaw or James Turner, as they will likely be the main users of the system in the future. Of course I will aim to satisfy the requirements of the other users as well so that any of the end-users can confidently use the new system and be happy with it.

A2: Requirement Specification

1. A system where the user can **add, modify, remove, sort and search** for customer information much like the user could with the old system but more efficiently.
    - See the interviews, the description of the old system and the analysis section above for reasons on why the users wants these features – they were the things that the users could do with the old system and need to be in the new one and business depends on it
2. A system that uses the **same fields of data input**: these are first name, last name, email address, home address, telephone number and postcode**.** These must be arranged so that the **interface looks similar to the previous system** but does not possess the problems associated with filing the records in the old system.
    - See the interviews, sample data and the analysis section above – these features were also in the old system and need to be in the new one as these were fields of data input which stored the customer records.
3. The system must be **easy to back-up** and **capable of upgrades** in the future. These upgrades include the capability to improve system speed and space as to ensure that the system will not become "out-dated" soon. This will tackle the problems of the old system including the ability to back-up data, the ability to upgrade the system and the ability to optimise the system which was not possible with the old system.
    - See the interviews and the analysis above. See the technical specification of the current computer system in section B. this was asked for specifically in the interview
4. The system must be **quick and easy to use** so that level data integrity is met as well as a higher standard of customer service. The old system was not quick as per the quantity of data that had to be dealt with; this will deal with it.
    - See the interviews which explicitly talk of this problem and the analysis of it above
5. A system that is **sorted alphabetically by surname**. The system must be able to sort itself independent of employee involvement such as when a record is added; the system must also be able to sort at the user's request
    - See the sample data, the interviews, the description of the data and the analysis of the data; this will ensure that the system is similar to the old system and the user will be somewhat familiar with the new one. This is also partially related to requirement 2 and the users requested that the new system must sort itself independently of their involvement.

6. A system that can **store all of the customer data as well as much more** whilst not taking up any more space than the current filling cabinet. There have been comments made on the preferred **use of the businesses computer** for the task.
    - See the interviews, the analysis and description of the old system for reasons for this requirement. The old system had not used this computer and it was becoming difficult to store many more records.
7. The system must have some form of **data validation** that checks the input fields so that no incorrect data is put in. Precautions are put in place so that the system will not stop working when an error occurs.
    - See the interviews which state there is a preferred use of a computer system and the analysis section above which provides reasons for this. With the old system, this is done manually and annually which raises the possibility of random/human error.
8. A **user manual must be supplied** so that re-training is not required annually as was the problem with the previous system.
    - See the interviews that state this is needed; they stated that training was costly for the business and time consuming.
9. **Levels of security must be put in place** so that the data is secure from malicious influences. This will be achieved through password protection for the system. This will replace the security that the old system had.
    - see the interviews which state this is required; also the analysis which states the paper based system has physical security which must be replaced with other forms.
10. The ability for the user to **search by surname** in a quick and easy fashion as well.
    - Currently, files are sought after manually and it is not time efficient and therefore bad/not economical for business; with the satisfaction of requirement 6, the search routine will be very quick

Now that the problem has been analysed, it is clear as to what the customer requires for their system. Now considerations must be made for what type of system the customer wants.

A3: Alternative Solutions

When developing the criteria for success, I found that there was a number of different ways in which the problem could be solved. These included: a more efficient paper-based system, a generic system or a tailored system. The requirement specifications were used to determine which system will in fact be the best system for the job. The paper-based system can be defined as the use of the original system but with a few minor changes that will make the system much easier to use. The generic system is defined as a piece of pre-made software that is used to make a pseudo-custom experience whilst still using a piece of software such as Microsoft access or Microsoft Excel. A tailored system is defined as a system that is custom built for the requirements and satisfies them appropriately such as the NHS EHR system in section A1 (see A1: Analysis of how this problem has been solved in the past)

Please refer to A2: Requirement Specification *for the requirements in this table*

For the following table, the text will be colour coded according to whether they achieve the requirements of the requirement specification. That means, a red if they haven't satisfied the requirements, a yellow if the requirements are partially satisfied and a green if they are satisfied.

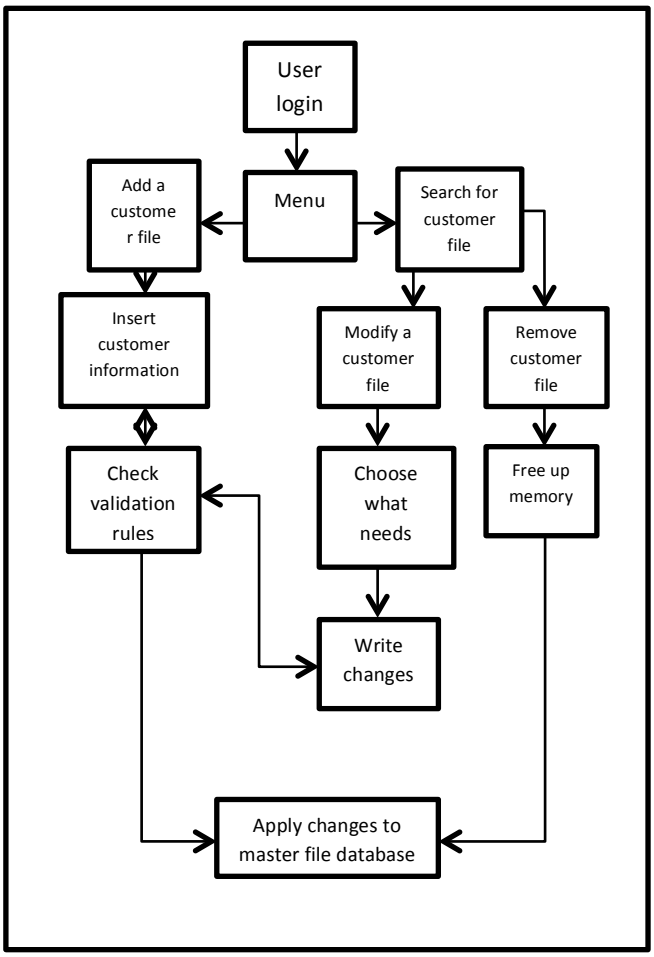
Requirement	A paper-based system	A generic system	A tailored system
1. A system where the user can add, modify, remove, sort and search for customer information.	The existing paper-based system can do this.	Software such as Microsoft Access can do this	A tailored system can be made to do exactly this
2. The same fields of data input.	This is possible	Access can be programed to accept those fields	A tailored system can be designed to accept these fields
3. The system must be easy to back-up and capable of upgrades in the future.	The paper-based system cannot be backed up easily and will require funds to prepare for the future upgrades	Access can be backed up easily onto an external drive.	A tailored computer system can be backed up easily onto an external drive. As long as the system code is not too abstract and the program follows some universal laws of programming (such as java's fundamental OOP rules) then the system should be easy enough to back up
4 .The systems must be quick and easy to use.	This system is not quick and easy, there is a degree of human error	Access is used on a computer and will be quick. It has validation.	This will be very quick on a computer and especially so if there is a dedicated program made to do this and only this
5. A system that is sorted alphabetically by surname	This system can be sorted but it is not quick to do and requires human interaction.	Access can be programmed to do this and will be quick since it is on a computer.	A tailored system can be designed to do this; it will be quick and can sort independent of human interaction.
6. A system that can store all of the current customer information as well many more customer files. There have been comments made on the preferred use of the businesses computer for the task.	This system cannot store much more and does not use the computer	Access can store many records quickly. There is plenty of space on a typical computer.	A tailored computer system will fulfil this requirement.
7. The system must have some form of data validation.	This system has manual validation which may have human error.	Access has validation functions which can pick out specific errors.	Tailored systems can be designed so that exact validation rules are met.
8. A user manual must be supplied so that re-training is not required annually.	This system requires annual training- it does not have a user manual.	This system has a few online resources but not any user manual.	A tailored system can have a manual included.
9. Levels of security	This system has	Access can be	Specific security

must be put in place so that the data is secure from malicious influences.	security already.	programmed for password protection.	routines can be designed so that the data is secure.
10. The ability for the user to search by surname	This system is sorted by name only	This system can do this	This system can be designed to do this.

A3: Initial design

Possible solutions were evaluated above and I feel that a custom built electronic system would be best for this company as it would ensure that their specific needs are catered for. The new system will have the following specifications:

- It will be a text-based system so that it is easy for the user to use – using a command line interface.
- The system will store customer information in a list which will be sorted at the input of a user command. The data will also be sorted alphabetically by surname every time the user adds/removes/modifies a customer file.
- The system will have password protection that will dictate whether the user can use the system. This password will be changeable at a user command.
- The system will use the computer in the shop as express interest was placed on this factor.
- The system will use object oriented Java (5.0 or above) because it is my strongest programming language and it offers many features which are suitable for the requirements.
- The system will have the same fields as the sample data.
- The system will run windows 7 as it is the most recent (2012) and most reliable system for a user.
- There will be specific commands to “add”, “remove”, “change”, search”, “sort” etc… so that the user can easily (and quickly as it is a computer) alter the database and its data.
- A User-guide and documentation will be included so the users can learn how to use the system and fix problems with the system themselves. This will also ensure that technical staff can maintain to new system in the future.
- The new system will have the capacity (hard-drive space) of 500 GB so that the user will have plenty of space to store many more customer files.
- Custom classes will be used to create custom validation methods so that the system will not crash when there is an error.
- User friendly operating system will be used so that the user can competently use the computer and this database. Anti- virus software will be used to ensure that the system itself is protected from cyber threats. Here is a diagram to illustrate this initial design (this serves as a basic model for the prototype where functions such as “sort” are removed for simplicity):

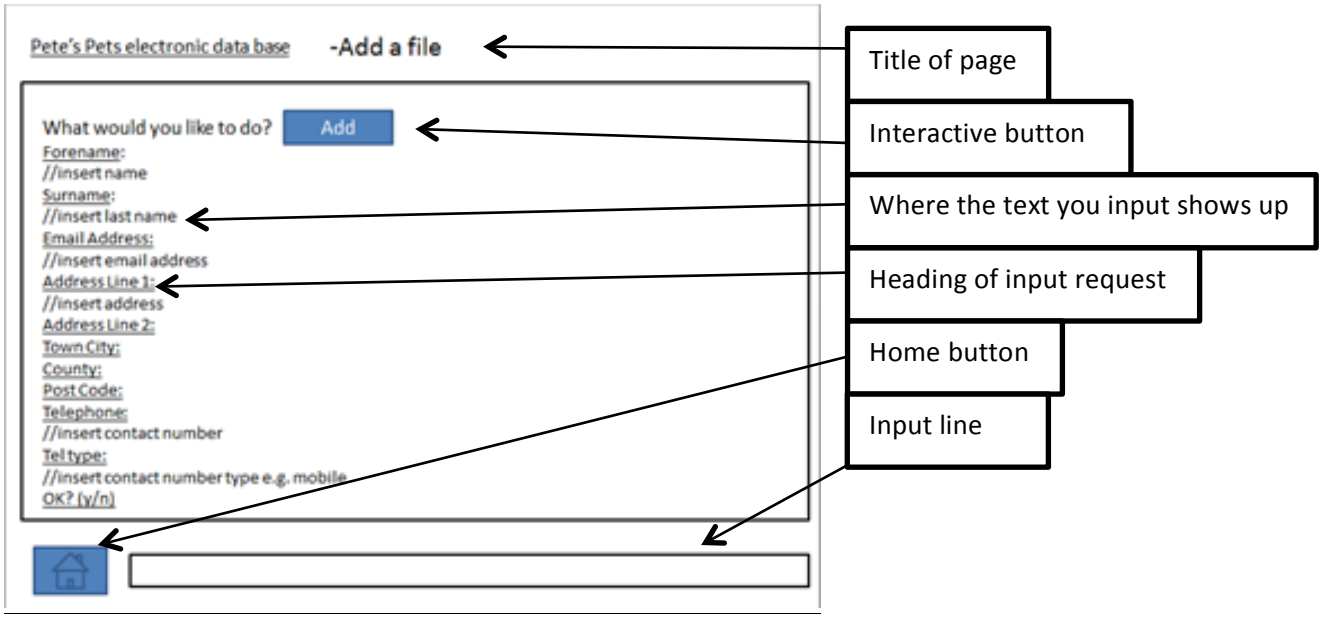

User login
Menu
Add a custome r file
Search for customer file
Insert customer information
Modify a customer file
Remove customer file
Check validation rules
Choose what needs
Free up memory
Write changes
Apply changes to master file database

A3: Prototype of new system with aims

A prototype system was created using Microsoft PowerPoint (it was based off of the criteria for success), showing a possible design for the new system. This was shown to the user and their feedback is documented. These are screenshots of the PowerPoint.

Note: this prototype is a "throw away" prototype and so it will not be incorporated into the final product, however the initial design that it follows will be used.

Key



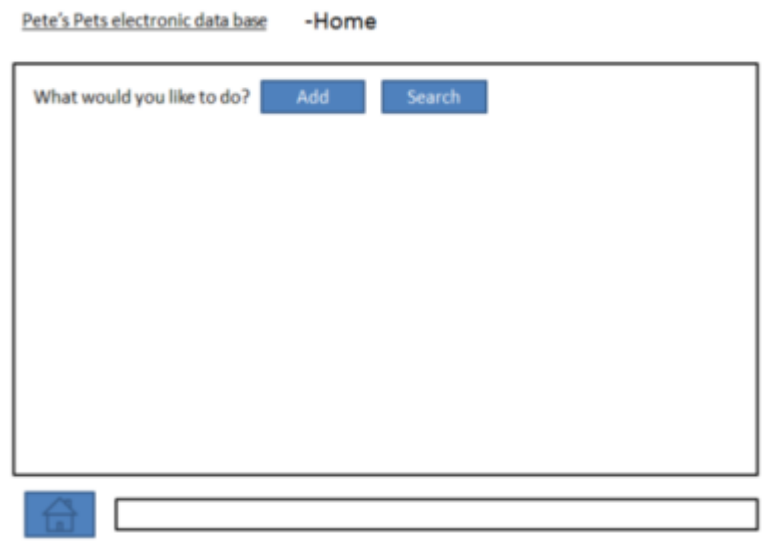


Figure 6: this would be the first page that the user will see; it will be a text based system.

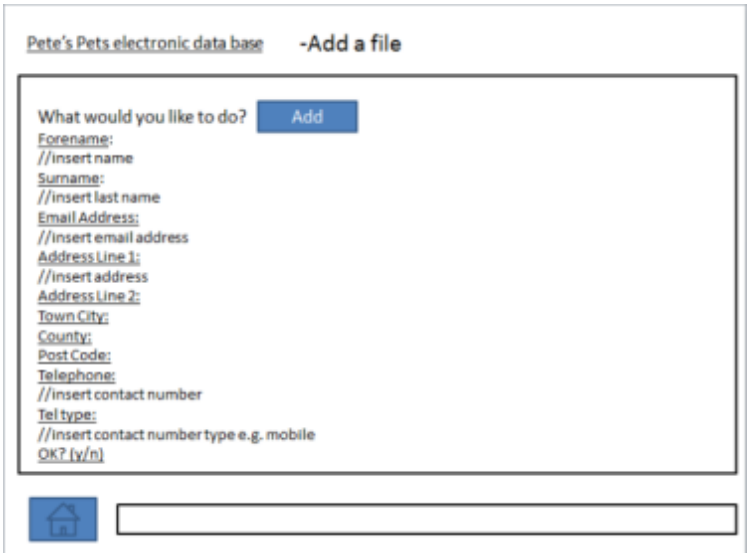


Figure 7: this will the text that the user will see, should they choose to add a customer information file.

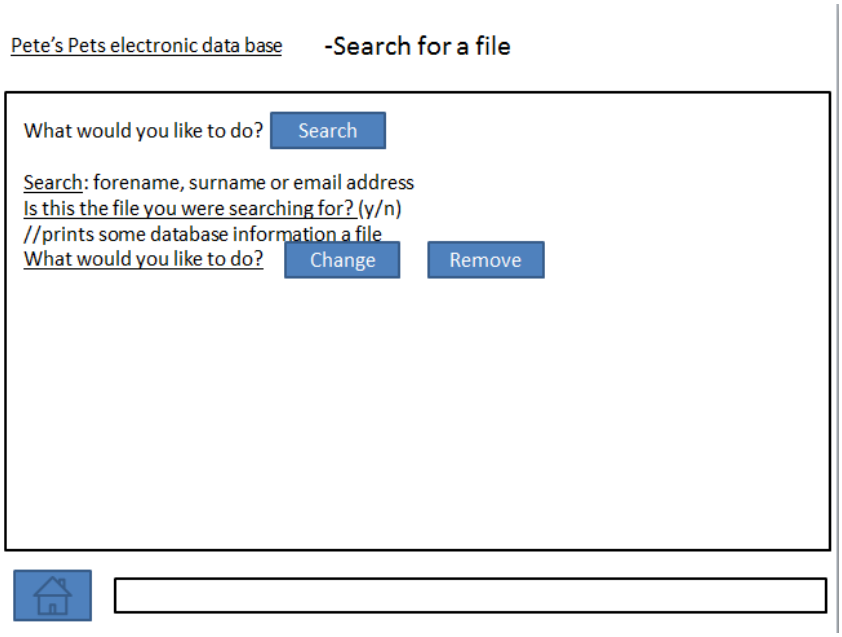


Figure 8: this is the text that the user will see if they choose to search for a file: it will provide ask for a y/n response to confirm whether you have found the correct file.

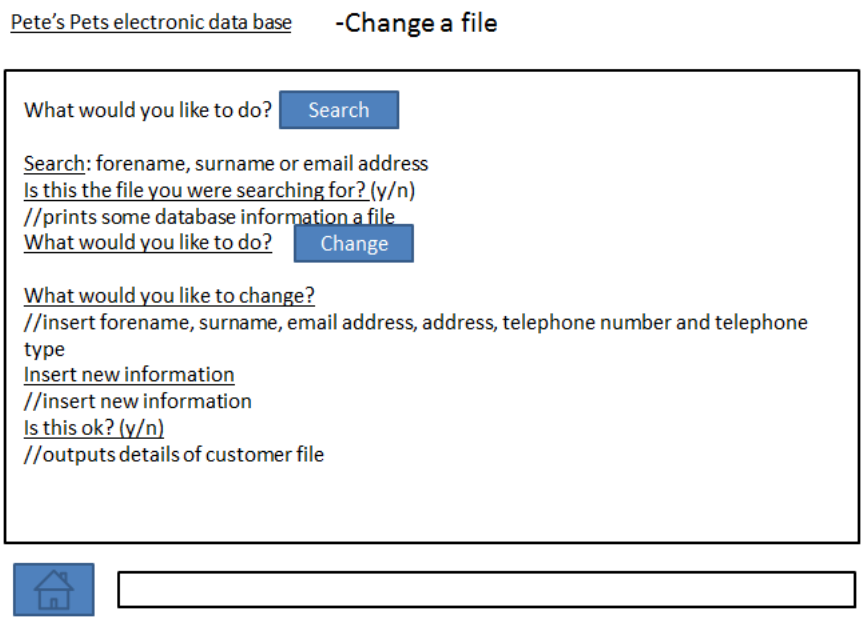


Figure 9: this is the text the user will see should they choose to change a file. Note: they must first search for the file in order to change or remove a file or its information.

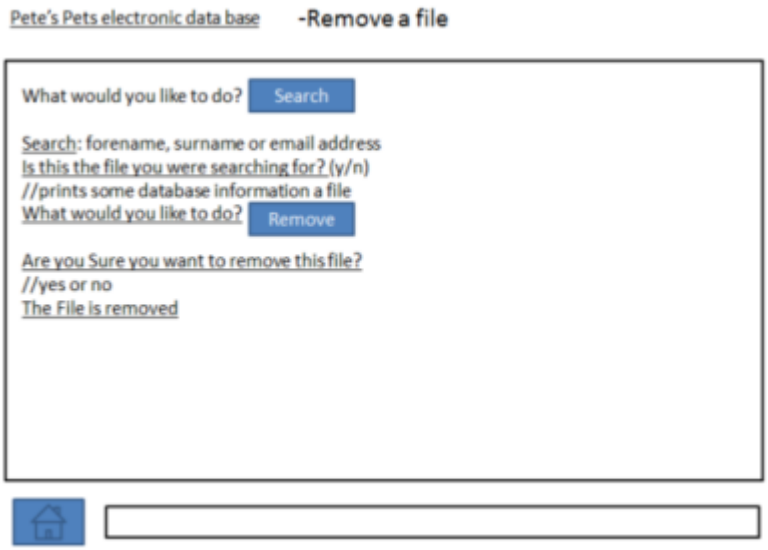


Figure 10: This is the text that the user will see when they wish to remove a file; note: the user must search and find a file before they can remove it.

Note: there will be a list of User commands such as "size" which will serve a purpose (for "size" it would tell you how many items there are in the database)

A3: User response to Prototype

This presentation was shown to the 5 employers individually. In this section I have summarised the opinions of the staff; please refer to Appendix C: User response of prototype for the full staff response. The summary of the user response is as follows:

- Overall, the user was happy with the system prototype: they were happy with its simple style and were reassured that it would be quicker to use.

- The manager was pleased that there would be less waste to throw away but made comments about how it may be easier to input single characters such as “c” (for change). This idea is feasible and will be taken into account at the design stage.
- The staff were happy that the system would be secure with a password protection and anti-virus software. The staff were also reassured that the system will be relatively failsafe when finished so that they could not crash it.
- They appreciated that the new system would take up less room and function quicker as well as the ability to back-up data easily.
- Users were reassured that the system will have a user-guide so they could learn to use the system more effectively.

A3: result of prototype

- The initial design above will be carried over to be used with the final product as the customer likes the way it works
- The prototype system was useful in showing that the user would accept a program that was driven by text input
- The prototype itself will not be used but the new system will follow the initial design from which it is based.

Stage B: Detailed Design

For the hardware and software rules please refer to A2: Criteria for Success

Inputs and Output

Choosing Hardware

- The system will use the computer system that is already in the shop as per requirement 6 of the requirement specification. As a result, there will be no need to purchase a computer system. The computer system has the hardware as follows (note the 64 bit system was custom built by another company in 2011):
    1. Asus M4a88t-m motherboard with am3 socket and the capability to install R.A.M that operates up to 1600MHz frequency. It has 6 SATA II ports on the board.
    2. 4 GB (2x2modules) Corsair DDR3 R.A.M that operates at 1333MHz frequency.
    3. An AMD Athlon II X3 455 tri-core processor that operates at 3.2GHz (AM3 socket) with a stock heat sync
    4. A 1Tb Western digital HDD (7200 rpm) with SATA II connections (both power and data).
    5. A Liteon optical drive that has SATA II connections (both power and data)
    6. An L.G LCD monitor 22”
    7. A Corsair 550W Version 2 PSU with SATA II connections.
    8. 2x 80mm cooling fans with Molex connections (use a Molex to SATA converter for the power) and a 120mm cooling fan.
    9. An internal card reader placed onto the front panel of the computer.
    10. A cooler master 342 system case to hold all of the above.
    11. Microsoft keyboard 3000 (with wireless mouse) that connects via USB 2.0 to the system.
    12. An Epson D78 series inkjet printer that takes up to A4 size prints and connects to the computer system via USB 2.0

        This hardware is relatively new and will be able to operate a java system easily. Furthermore, the system can easily be upgraded by purchasing a larger hard drive and R.A.M in the future. Moreover, this system is powerful enough to manage the customer database and therefore eliminates the need to purchase a new system. This hardware partially satisfies criterion 3, 4 and 6 of the criteria for success.
- The user will use a 500Gb external hard disk drive to back-up the customer data; this will connect to the computer by USB 2.0

Choosing Software

The computer system used above will have the following software installed so that the user can effectively use this system:

1. The users’ computer system currently runs a 64 bit version of windows XP but the new system will run Windows 7 home premium 64 bit. I have chosen this operating system because the user will be familiar with the layout of windows 7 as it is similar to windows XP. Furthermore, A GUI-based system will be easier for these users to use (it is intuitive if the user is familiar with it) as opposed to a CLI-based system (such as UNIX and its operating system LINUX) because CLI-based systems require more training to use than a GUI-based system. This satisfies criterion 4.

a. Note: this is the operating system, not the program being made; the program itself will use a command line interface

2. The System will have Java 7 and Java runtime environment so that the system can use the database program that I am designing.
3. The system will use AVG 2012 anti-virus software so that the system is protected from viruses and other harmful programs. This satisfies criterion 9 of the criteria for success.

Input devices

Input method	Input type	description
QWERTY Keyboard	Manual data entry	The keyboard will be used to input customer information into the system so that the program may use this information in the database
Mouse	Manual data entry	The mouse will be used to navigate around the operating system with ease

Output devices

Output method	output type	Description
Monitor	Temporary display	The monitor will display the program and the user input
Printer	Permanent display	The printer will be used to print out order forms for the business; the printer will not print out customer records
Hard Drive (HDD)	Permanent display	The hard drive will store all of the OS data, the program and the customer records.
External drive (HDD)	Permanent storage	Will store copies of database data, OS data and important code for the program

B1: Data Structures

B1: Primitive Data Types

In order to satisfy requirement 2 of the requirement specification (same fields of data input), there were some standard data types that were necessary for the customer records. These were:

Name	Data Type	Size	use
Forename	Private String	1 byte per character	Used to store the first name of the customer
Surname	Private String	1 byte per character	Used to store the last name of the customer
Phone number	Private String	13 bytes (including spaces)	Used to store the customer phone number
Email address	Private String	1 byte per character	Used to store the customer email address
Home address	Private String	1 byte per character	Used to store the customer home address
Post code	Private String	8 bytes (including space)	Used to store the customer postcode.

*note the data types are private because the business must adhere to the data protection act to protect confidential customer information. This also partially satisfies requirement 9. Methods will be used within the program itself which will be able to access the customer information when needed (encapsulation).

B1: Progress to a file structure

In this section I will be discussing the different data structures that this program will use. This program may be written in any language other than Java. The program will have a class and methods to create a "customer file" object using a customer file class. The data will be sorted into a singly linked list and will be sorted in ascending order by the key value (the surname: A-Z). Within this project there will be a singly-linked list class with "Add", "remove" and "modify" methods as well as "sort", "search", "merge" and other important additional classes and methods needed to use this system. This section will follow the same design (and diagram) that is present in the initial design stage (see: A3: Initial design) and the modular design section will analyse the program in a broader sense to show the overall system plan. Initial Considerations were made for a parallel Array based file structure which later developed into the singly-linked list file structure which I am using as a result.

Initially, I thought of using the parallel Array file structure to organise the customer information alphabetically (by surname). In this data structure, String Arrays would contain the data types. Each Array would be initialised to 1000. To add a new customer record, the data would be input via keyboard and data would be stored in its respective Array. Each record would have an index which would correspond to the appropriate index of the customer information in each Array. This would link each piece of data to form a "customer file" of sorts. A sorting routine could be used to compare data Strings so that customer files could be sorted and stored alphabetically by surname (and theoretically, by any other field). This would involve a "temp" String variable which would be used in the swapping process as a platform for the sort routine. The Array would have been sorted so that *Geeson* would precede *Smith*. This way the customer information would be sorted by surname.

This system would be good as it is relatively simple and would ensure that the information is stored and sorted appropriately. Furthermore, the system is often used and will be easier to upgrade in the future as arrays are widely recognised as a fundamental feature by programmers. Another important feature of an array based system will be that it has a notoriously fast runtime which would satisfy requirement 4. However, the parallel Arrays system has a number of limitations that would not be practical as a replacement for the paper-based system. Firstly, the parallel Array file structure will not allow the user to input any more data past the values that the String Arrays are initialised to (if the Arrays are each initialised to 1000, then the maximum index will be 999 including 0, so that the user may store only 1000 customer records). This means that the customer database will be limited to 1000 files and this will limit the total information that the business can store with this program which in turn does not allow for the completion of requirements 1, 3 and 6 of the requirement specification. It would also be impossible to predict the maximum size of the array list as the business circumstances may change which would be very problematic in the future. This problem could be tackled by the reallocation of the data in the Arrays to larger Arrays, however this will be a resource heavy feat for a system that would contain multiple Arrays and the problem will still remain (albeit with larger Arrays). Also the parallel Array file structure will obscure the relationship between the data types (this may be worsened by a bad programming style), this will make it difficult to upgrade in the future (thus, further limiting requirement 3). In addition the Array will reserve empty space in memory, even when it is not full (such as an integer Array that automatically initialises all elements to 0 which requires memory and hard drive space), this reduces the memory efficiency of the program.

To remove the issue of limited space (and the difficulty of upgrading to a larger file structure, I explored the idea of using a dynamic database. All the customer data for each respective customer could be stored in a unique object (1 object for each customer) which could then be stored in a singly linked list. The file would contain all of the customer information. This database would completely remove the problem of memory described above so that the physical and logical size of this list will be identical and only as big as the actual size of the list; the total size of the list will be dependent on the sum of the sizes of each record which will vary with the amount of data in the system. This removes the need to create a larger list if it fills up (as you would with an array-based system). Of course an array based system will be much quicker to use but I believe, with the technical specification of the computer system this system will run on, the speed of this system will not be a problem. Furthermore, the relationship between customer data will be much clearer as each unique object from the class will contain all of the data for each customer. Therefore, for this database, I believe the best system to use would be a singly linked list.

With this linked list system it would require a customer file class which would collect and contain all of the data for a customer. This means a method would be needed to *set* the respective data for the customer and a method to *get* it when it is needed elsewhere. For instance, if we had a customer called *Luke Geeson* we would need a method to *set* the forename and a method to *set* the surname; likewise, we would need methods that return this data so we can use it. When we consider essential data types we need (see B1: Primitive Data Types), that means we will need get and set methods for each. We would need methods to indirectly return this data because the data is confidential and needs to be protected by being set as *private* in order to adhere to the data protection act (encapsulation).

With this established, we must consider how the linked list will work. Each customer file would be part of a "node" of the linked list which could be placed into the linked List; the linked list would be sorted into ascending order by surname and the use of these "node" objects will form the elements of the list. The node objects themselves consist of 2 fundamental features; these are the data itself with an appropriate name or description (in this case it is the customer file object containing the data for each customer) and a pointer to the next object in the list. With this list, there is a "head" node object to represent the start of the list and an appointed "null" pointer to act as a sentinel and indicate the end of the list. Objects can be stored in ascending order (or not) according to a key value and each object is linked to the next by the next pointer of each node. In the example below, integer values are stored in ascending according to size but in my system I will be storing the customer files and their data in ascending order according to surname.

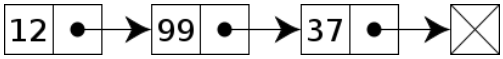


Figure 11: illustration of a singly linked list system that sorts integers in ascending order as found on *Wikipedia.org Link: http://en.wikipedia.org/wiki/Linked_list*

As required by requirement 1 of the requirement specification, I will therefore need add, modify, remove, sort and search functions to add, remove or alter the data contained within this linked list system and according to requirement 10, I would need to ensure that the sort function of this system sorts the list in alphabetical order (regardless of capitalised letters on the surname or not) and the search function uses the surname of each record in its search. This will be achieved in the following ways:

B1: The add function of the singly linked list

To add a record, the data will be input via the keyboard. Then this data will be saved in a new node as a customer file object, then in the node object and then placed in the list at the appropriate location. There are 5 main ways that this data structure will compare strings to ensure that the record is placed at the correct location in the list: these are if the new data is before the head of the list, if the new data is after the end of the list, if the list is empty (then the new data will be placed at the head of the list and will also be the end of the list to indicate that there are no more items), if the data item shares an identical key value with another node or if the new data is somewhere in between the head and the end of the list.

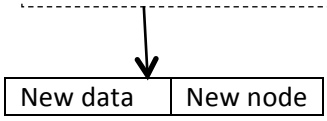
If the new data is before the head of the list, then the "next" pointer of the new data will be set to the current head and then the head of the list will be set to the new data. If the new data comes after the end of the list then the last item's "next" pointer will point to the new data so that the end of the list now points to the new data (by setting the new data "next" pointer of the new data as null, will then indicate that this is the last item in the list again). If the list is empty then the new data will be set as the head of the list (since there are no other items to compare it to) and the next pointer will be set to null to ensure that the system knows this is the only item in the list. If the nodes share an identical comparison key (in this case it is the surname) then the system will allocate the nodes so that the new nodes come after those nodes with the same surname since the system will be sorted in alphabetical order by surname, the linked list will compare the surname strings of data that are already stored with the surname string that is saved in the new node (more below). Then the Linked list will store the record in the right position. This is achieved by the reallocation of the pointers of the objects. This will ensure that the correct customer information is stored in the

correct location in the list. The following example will illustrate the storage of such a record in the Linked list if the linked list contains 3 records and we wish to insert an item in between the head and the end of the list.

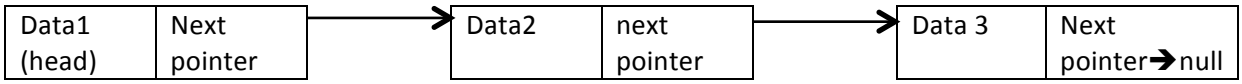
Note: a singly linked list does not have a tail node by definition and the end of the list will be indicated by a null "next" pointer as a result.

New node object

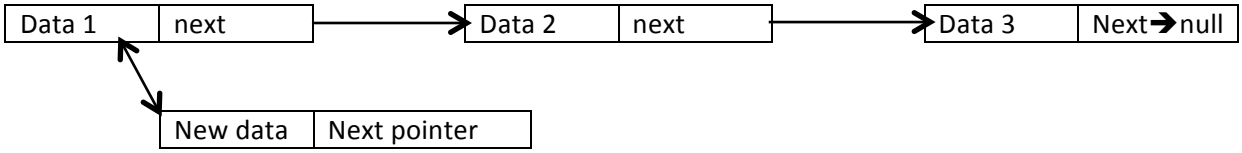
```
private node DATA
{
        //-Contains the customer information
        Forename←set Forename ("John")
        Surname←set Surname ("Smith")
        Telephone number←set phone number ("01233 566455")
        Home Address←set home address ("45 Downsview […]
        Kent")
        Post code← set post code ("TN24 OPL")
        Email Address← set email address ("JSmith@gmail.com")
}
```



Existing Linked list:

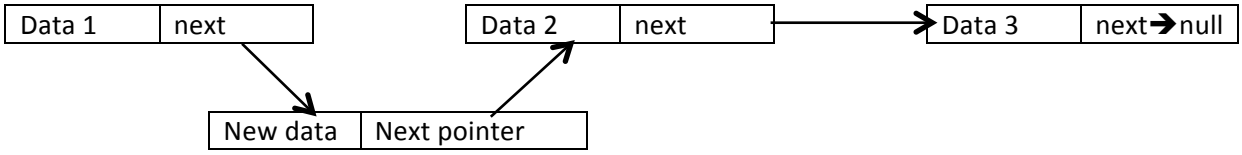


Adding it to the database



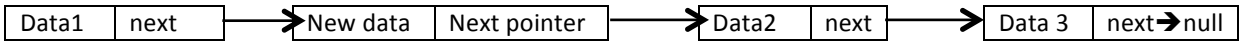
If data1→surname precedes new data→ surname→"smith" then proceed to compare new data with next item in the list (data2). Continue through list until the new data→surname→"smith" precedes data n→surname. It will also compare the new node data with the data at the end of the list to ensure that it does/does not come after the last node. For the ease of illustration let's assume that the last item of the list contains a surname string that proceeds the surname string stored in the new node so that the new node will be stored before the end.

When new data➔surname➔"smith" precedes data n➔surname



*Assuming this occurs between data 1 and data 2. Redirect the "next" pointer of data 1 to the new node. Also direct the "next" pointer of the new node object to data 2.

Result of adding this file

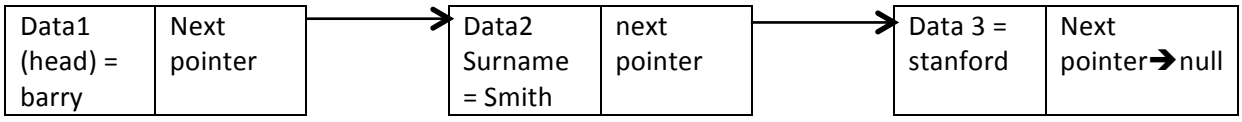


In effect the database will now be sorted alphabetically by surname.

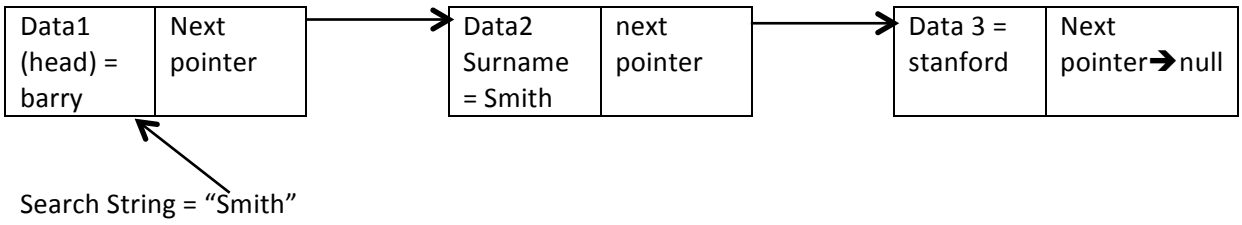
B1: The search function of the linked list

As per the initial plan in section in section A3, you must first search for a file before you can modify or remove it. This system will use a sequential search to find the correct customer record. This will satisfy requirement 10. In order to search for a customer record, the search will compare the surname input by the user with that of the records and when an identical one is found, it will print the details of the customer record and ask for confirmation from the user to ensure that the correct record is found. It will search from the start of the list. If the item is found then the record is returned so that it can be modified or removed from the list. If the item is not found then the search routine will continue through the linked list whilst searching for the record; if the record is not found at all and the user declines confirmation on other records then the system will indicate that the search has failed and will return to the main menu. The following example will demonstrate searching for the customer record of a customer called *John Smith* from the Linked list of 3 customers.

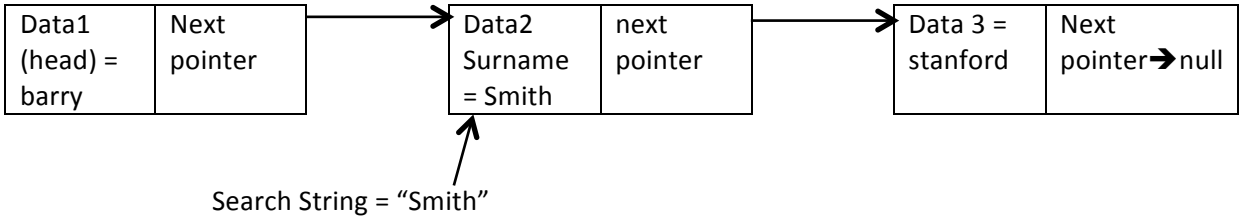
Linked list:



Surname we are searching for: "Smith"



Does the first node➔surname = the search string? No "barry" is not equal to "smith". Traverse through the list until the node is found.



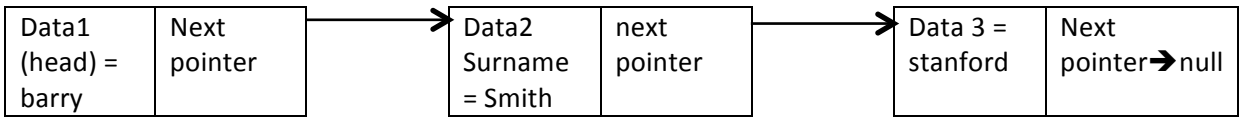
Does the next node➔surname = the search string? Yes "Smith" equals "Smith" so the system will return the record and print its details so that it can be modified or removed.

This is performing a sequential search on items in the list as it is reading through each item individually and comparing the data stored in the Surname String with the search string. Once we have found an item in the list, we can do one of the following: remove the record completely or modify specific items in the list.

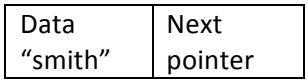
B1: The remove function of the linked list

The process of removing the records in the data base is relatively simple. The node we wish to remove will be in one of 4 positions: at the head of the list, at the end of the list, as the only item in the list or between the head and the end of the list. If the node we wish to remove is the head node, then we set the head as the next item (assuming there is one) and then delete the "old head" node. If the item is the last item in the list then we would set that as null (assuming there are other items before it). If the node is the only item in the list then, it should be, by definition, the head node – we can simply set the head node as null. If the node we wish to remove is within the list itself, then we can traverse through the list (using recursion) and once we have found the record to remove; we take the next pointer from the item preceding it in the list and redirect it to the item after it in the list. Once this is done the item to remove will no longer be part of the list and can be set as null, removing all customer data for that record and freeing up most of the memory that it had occupied. In the following example, we wish to remove a node between 2 other nodes in the list – there are 3 nodes in the list and we will use the same list from before.

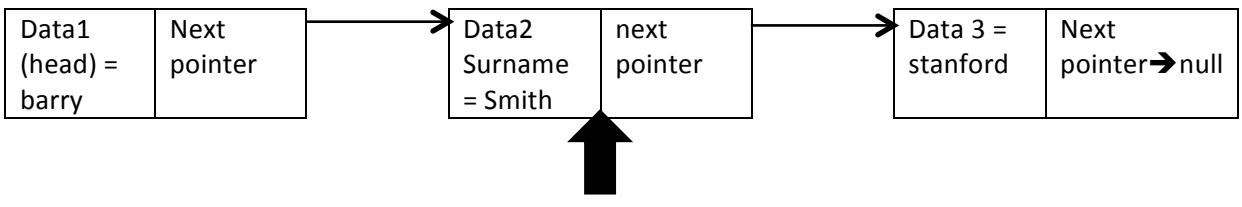
<u>The original list</u>



<u>The node we wish to remove</u> retrieved using the search function

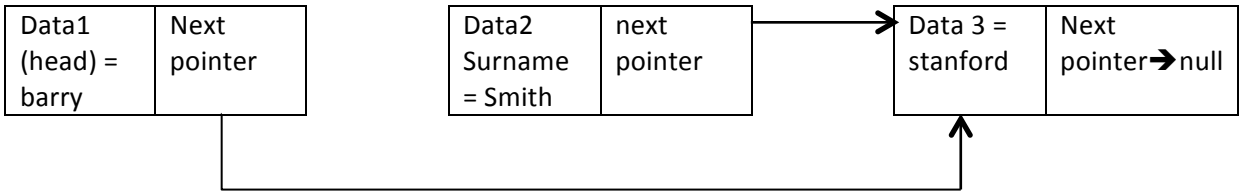


<u>The node's position in the list</u>

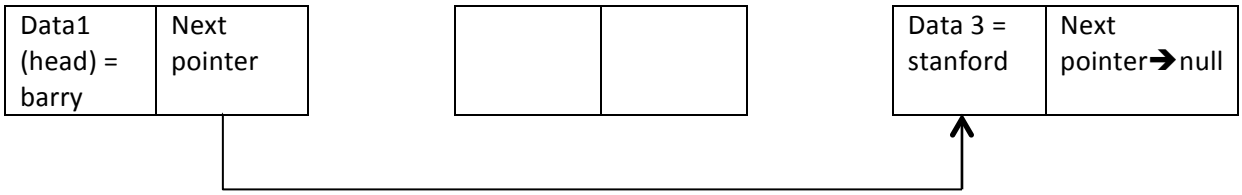


We would reallocate the nodes so that the previous node (data 1) next pointer will point to the node after the node we wish to remove (data 3)

Reallocated nodes in the list



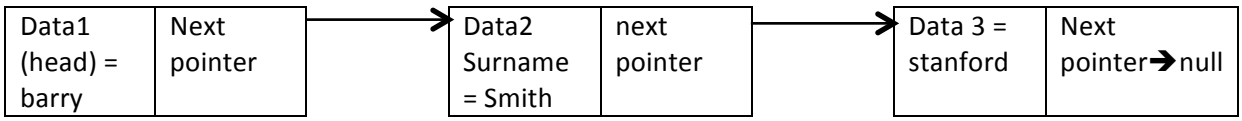
Once the nodes have been reallocated, we can simply remove the old node by setting it as null.



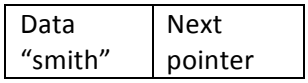
B1: The modify function of the linked list

To modify data of an item, the system will prompt the user to input the type of data which is to be changed; this will be one of the fundamental primitive data types outlined in primitive data types. Once the system has identified what is to be changed it will again prompt the user to input the new information to replace the old. Once this is complete, the record will be saved. When data is edited, the list will not change structurally unless the surname is changed (the key value), the sort method will then be called on the linked list so that the list will be sorted afterwards. In the following example we change the Surname of the node – using the previous list example again.

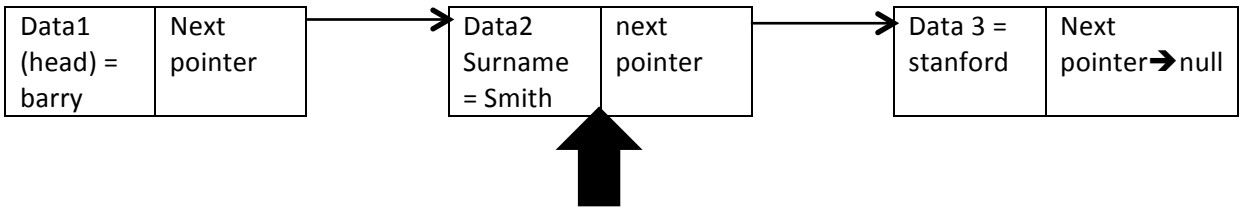
The previous list



The node we wish to change



The node's position in the list



Changing the node "Smith" to "Turner"

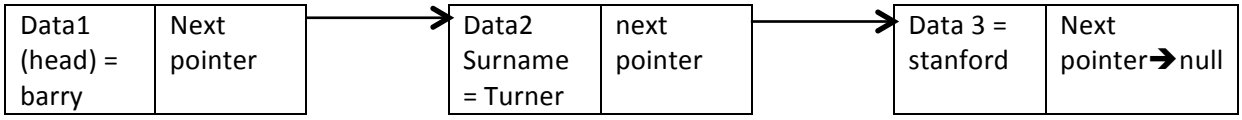
Data 2: smith	Next pointer

Changes to

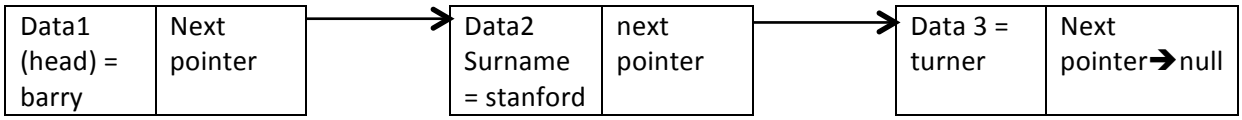
Data 2: Turner	Next pointer

Next pointers do not change with the modify method but the sort method will change the positions of the nodes.

States of the list after modifying



State of list after sorting



B1: Other functions of the linked list

There are some other standard functions of the linked list that are very similar to other linked lists and so demonstrations will not be required to illustrate how each will work. Most of these functions may not directly alter the linked list, but they may serve a purpose within the functioning of the system itself. The other functions of the linked list are summarised in the table below

Name of function	Description/use
Is empty	This function will return a Boolean value dependent of whether the list is empty.
size	This function will return an integer which is the size of the list
Sort	This function will sort the list, should it be out of order
Print list	This will print out the details of every item in the list
Print reverse list	This will do the above backwards
Get head	This will return the head node
Set head	This will set a new head node
Append last	This will add an item to the end of the list
Get last	This will return the last item in the list
To array	This will turn the linked list to an array, should it be needed in the future
Merge lists	This will merge 2 singly linked lists into 1 new one, should it be necessary in the future.

B1: Abstract Data Type – the Singly Linked List

As illustrated above, I will be using a Singly Linked List to store the customer data with each node representing a unique customer record and each node will contain a next pointer to the next item in

the list. Each node will contain all the **same fields of input** as those in the previous database. There will be a head node to indicate the start of the list and a null next pointer on the last item to indicate the last item of the list. This data structure is Sequential by nature and will therefore benefit from the use of recursive procedures which are precise, concise and more elegant (as visible code) when compared to their iterative counterparts. There will be **add, remove, modify, sort, search, merge lists, set/get head, append last, get last, is empty, size of list, to array, print list and print reverse list** functions which serve, I believe, as the essential functions required for all data structures and show that this data structure could be used with other databases which use a string as a keyword by which the list is sorted. This shows that this system is **capable of upgrades** in the future which will be aided by a good programming style to help technical staff to upgrade the system. This is reinforced by the fact that some of these functions will not be used in this system (such as the **to Array** function which is not needed as explained above) but they are available for future programmers to use at will. The singly Linked list will be **sorted in ascending order by the stored surname of each Node** and the **Search method will search for the Surname**. The Singly linked list will be read/written to an area on the secondary memory (an area of a disk if the secondary memory is a HDD) so that the **database can be stored and used on the business computer**. This also makes the system **easy to back-up** as the user can simply make a System image/copy of the database file on an external drive which can be encrypted with a password for data protection (that is not within the confines of this dossier, but it can be done). Furthermore, the computer system that the business is using has a high specification and will easily do the job of the database **quickly** and has a very large HDD (1TB) which will be able to store **all of the customer data as well as much more** and will in fact reduce the physical space taken up by the system as the whole database will be stored on the Hard-Drive opposed to a paper based system where paper took up physical space in the cupboard.

In terms of security, the database will be **stored on a HDD which cannot be read by humans** (we don't read in binary) without mechanical help (unlike the paper-based System). The **database will be saved as a special type of file in a specific file location which will reduce the possibilities of malicious users and viruses from accessing the confidential user data** as this program will be designed to pick up the specific file name. This is reinforced by the fact that **this is a custom built system for a small business (a sole-trader) which is highly unlikely to be the victim of an infection of a specifically made virus** as it is not a mainstream system. The system itself will run Windows 7, and the **computer will have anti-virus** (AVG 2012) installed as well as a **password for the OS to ensure the intended user can login and use the system** and access the files/programs. The **database itself will request a login password** which will determine whether the user can use the system or not and the customer **data will be set as private** ensure it cannot be seen outside the system without the use of appropriate methods.

Therefore the Singly linked List should fully satisfy requirements 1,2,3,4,5,6 and 10 with partial satisfaction of 7 (the code of the database itself will ensure there are no errors), satisfaction of requirement 8 (a user-manual will be supplied with this project) and partial satisfaction of requirement 9 (the supplied levels of security presented by Windows, AVG as well as the password protection of the database should ensure a satisfactory level of security).

B1: Class Definitions

B1: The customer record and Node classes

The Nodes of this linked list will contain 2 fundamental features, these are: the data and the next node reference. The data itself will consist of a **customer record** class in which the **forename, Surname, phone number, email address, home address** and **post code** are stored. This class will have **get and set** methods for these various data types which will ensure each customer record has data for them. These variables will be set as private to ensure that the system functions in accordance with the data protection act and the data can only be accessed by specific methods of the customer record class: **this adds another layer of data security (encapsulation).** The class will also have a **display all** method which will display all the information stored in the record and will allow the user to clearly see the data and its relations. The data itself will be input from another **main** method in another class with keyboard input and the **same fields of data have been used** to ensure familiarity with the paper-based system. The linked list itself will be sorted (and data added) by the key value of this data (which is the surname for this system).

The node class can store and return this data with the **get and set data** methods of the node class. The class will have **get and set** methods for the **next pointer** so that the value of the next pointer can be set or returned. Again the **data** and **next** variables will be set as private to ensure a **standard level of security.** This is a salient feature of linked lists and did not exist within the previous system.

B1: The Singly Linked List class

See above for details on how this works see: B1: Abstract Data Type – the Singly Linked List as well as the functions of the linked list

B1: The Main class

This class will contain all the methods/functions used to input data from the keyboard through the use of a CLI. This input is stored in the Customer File class and the customer file is stored in the Node class. The main class is also used to receive and interpret commands of the system which will use the **add, remove, modify, search, sort, clear list, print** and **get size** functions of the singly linked list class on the list saved in memory. This class will also deal with File I/O which will save the singly linked list to the disk and load the singly linked list to the disk so that data can be accessed later on. This class will act as the CLI interface between the user and the system. This class will have a **main** method, a **save list to file** method, a **load list from file** method, **set/get password** as well as **set/get file name** methods. Required by specification 9 (security) the **main** method will require a password from the user before they can make changes to the list. The **main** method will interpret all commands and receive the data using a Scanner (or equivalent) to acquire the input from the keyboard – it will interpret **Add, search, remove, modify, sort, clear list, print list** or **get size** and will receive the data input as Strings or integers. The **save list to file** and **load list from file** methods will receive the file from the location on the disk and will save the file to that location when done. The **get/set password** methods will "get" and "set" the password much like the get and set methods of the Customer File class function, however the **get password** method will be unique in the sense that it will ask for a **system passcode** in order to return the user's password. This passcode will be supplied with the system and cannot be changed; but will be necessary to retrieve the password.

B2: Algorithms

Note: for the following code, the symbol "🡐" represents initialising a variable whereas "🡒" represents the use of a method/function so that:

Int j 🡐4 - means we are initialising a primitive object of type "int" to be of value 4

Current🡒display all details - means we are accessing and using the **display all details** method/function used by the current object (it is a node object)

```
//-----------------------------------------------------------------
```

Customer file class – creates objects to be used in the linked list and node class, is public so it can be used with other classes

```
Variables:
String forename      //stores the customers first name
String Surname       //stores the customers surname
String phone number  //stores the customers phone number
String email address      //stores the customers email address
String home address  //stores the first line of the address
String Post code          //stores the post code of the customer
//these variables are private to ensure data protection
```

```
Methods/functions      //void = doesn't return anything
Set forename           //void
Get forename           //returns a string
Set surname            //void
Get surname            //returns a string
Get phone number       //void
Get phone number       //returns a string
Set email address      //void
Get email address      //returns a string
Set home address       //void
Get home address       //returns a string
Set post code          //void
Get post code          //returns a string
Convert and present//returns a String, takes a string
Display all details       //void
```

Development into Code:

Pseudo code for **set forename** function – a public function that allows it to be accessed from other classes, would be void, no return value, takes a String **input** as a parameter- initialising **forename**

```
Function set forename
{
     Forename 🡐 (convert and present🡒string input)
}    //-inserts and formats the string (see convert and present
function)
```

Pseudo code for **get forename** function – a public function that allows it to be accessed from other classes, would return string variable **forename** and **takes no parameters**

```
Function get forename
{
     RETURN forename
}
```

Pseudo code for **set Surname** function – a public function that allows it to be accessed from other classes, would be void, no return value, takes a String **input** as a parameter- initialising **Surname**

```
Function set Surname
{
     Surname ← (convert and present→String input)
}
```

Pseudo code for **get Surname** function – a public function that allows it to be accessed from other classes, would return string variable **Surname, takes no parameters**

```
Function get Surname
{
     RETURN Surname
}
```

Pseudo code for **set Phone number** function – a public function that allows it to be accessed from other classes, would be void, no return value, takes a String **input** as a parameter- initialising **Phone number**

```
Function set phone number
{
     phone number ← input
}
```

Pseudo code for **get phone number** function – a public function that allows it to be accessed from other classes, would return string variable **phone number** and **takes no parameters**

```
Function get phone number
{
```

```
    RETURN phone number
}
```

Pseudo code for **set email address** function – a public function that allows it to be accessed from other classes, would be void, no return value, takes a String **input** as a parameter- initialising **email address**

```
Function set email address
{
    email address ← input
}
```

Pseudo code for **get email address** function – a public function that allows it to be accessed from other classes, would return string variable **email address, takes no parameters**

```
Function get email address
{
    RETURN email address
}
```

Pseudo code for **set home address** function – a public function that allows it to be accessed from other classes, would be void, no return value, takes a String **input** as a parameter- initialising **home address, takes no parameters**

```
Function set home address
{
    home address ← input
}
```

Pseudo code for **get home address** function – a public function that allows it to be accessed from other classes, would return string variable **home address** and **has no parameters**

```
Function get home address
{
    RETURN home address
}
```

Pseudo code for **set post code** function – a public function that allows it to be accessed from other classes, would be void, no return value, takes a String **input** as a parameter- initialising **post code**

```
Function set post code
{
     post code ← input (as upper case)
}    //-ensures the post code has capitals (as in a UK postcode)
```

Pseudo code for **get post code** function – a public function that allows it to be accessed from other classes, would return string variable **post code, takes no parameters**

```
Function get post code
{
     RETURN post code
}
```

Pseudo code for the **Convert and present** function – a public function that allows it to be accessed from other classes, the function capitalises the String input e.g. Input: "luke" returns: "Luke". Passes the String to be converted as an parameter and returns the converted string

---

```
Function convert And Present
{
     remove whitespaces from the start and end of word
     Character first letter ← store the first letter of the
input   string
          String converted ← first letter (upper case) + the
rest of the String (minus the first character)
}
```

Pseudo code for **display all details** function – a public function that allows it to be accessed from other classes, would be void as it does not strictly return a value, however it will output (to the console) the details of the customer file object.it **takes no parameters** as it simply reads and prints the data already stored in the customer file.

```
Function Display all details
{
     output to console 'the name of the customer is '+ get
forename                  + get surname

     Output to console 'the email address is '+ get email address

     Output to console 'the home address is '+ get home address

     output to console 'the post code is ' + get post code
```

```
      Output to console 'the phone number is ' + get phone number
}

//----------------------End of class-------------------------------
```

Node class – creates the node which are used in the linked list – is public so it can be used in other classes

---

Variables:

```
Customer Node next    //assigns the next node in the list
Customer File data    //assigns customer file that is the data
```

Methods/functions:

```
Get data              //returns a customer file object, takes no
parameters
Set data              //void, takes a Customer Node object
Get next              //returns a Customer Node, takes no parameters
Set next              //void, takes a Customer Node object as a
parameter
```

Development into code:

Pseudo code for **set data** function – it is void so that it doesn't return anything and takes a Customer File object **input** as a parameter. The method assigns the data that forms one of two fundamental parts of the node – the customer file which is the data.

```
Function set data

{

      Data←customer file input

}
```

Pseudo code for **get data** function – it returns the Customer file object assigned to the Customer Node **data** variable. It does not take anything as a parameter.

```
Function get data

{

      RETURN data

}
```

Pseudo code for the **set next** function – it is void, it doesn't return anything but it takes a Customer Node object as a parameter and assigns this to the **next** object

---

```
Function set next
{
     Next←Node input node
}
```

Pseudo code for the **get next** function – it returns the value assigned to the Customer Node **next**; it takes no parameters

---

```
Function get next
{
     RETURN next
}
//------------------------End of Class-------------------------------
```

Singly Linked List Class – creates and implements a linked list data structure for the Customers of this system.

Variables:

```
Customer Node head    //the head of the list – private for data protection
```

Methods/functions:

```
is Empty              //returns a true if the list is empty
Size of List          //returns the size of the list
Add Record                 //adds a node to the linked list
Search List           //searches a list for the node needed
Remove Node           //removes the node which is found with search
Change Node           //modifies data within the node
Sort List             //sorts the list in ascending order
Merge and sort list//merges the list with another
Traverse and print //traverses and prints all items of the list
Print Reverse List //prints the list from the end to the start
Set head              //sets the head node of the list
Get head              //returns the head of the list
append last           //adds an item to the end of the list
get last              //returns the last item in the list
To array              //turns the linked list into an array
```

Development into code:

Pseudo code for the **is empty** function, determines whether the linked list is empty – returns a Boolean **empty** accordingly (true = list is empty). Takes no parameters

---

```
function is empty
{
     Boolean empty
     IF list head is equal to NULL
```

```
      {
            empty ← true
      }
      ELSE
      {
            empty ← false
      }
      RETURN empty
}
```

Pseudo code for the **size of list** function: determines the size of the list – takes a Customer Node **input** (pass the head of the list) as a parameter and recursively traverses as long as there is another item (incrementing 1 each time). Returns and int **size** which is the size of the list (null if empty)

```
function size of list
{
      IF input is equal to NULL
      {
            return 0;
      }
      ELSE
      {
            RETURN 1 + length➔(input➔get next)
      }
}
```

Pseudo code for the linked list **add record** function – it is void as it does not return anything and takes 2 node objects as parameters – it is a recursive procedure which takes a customer node **New data** and compares it with the customer Node **trail node**. It recursively runs through the list until a suitable position is found in the list to insert the node.

```
function add record
{
      IF trail node is equal to NULL
      {
            IF head us equal to NULL
            {
                  set head← new data
            }
            ELSE
            {
                  trail node ← new data
            }
            new data ➔ (set next ←NULL)
      }
      ELSE IF trail Node is equal to head AND new data precedes
it
      {
            new data ➔ (set next ← head)
            set head ← new data
      }
      ELSE IF trail node➔get next is equal to NULL
```

```
      {
            trail node ➔ (set next←new data)
            new data ➔ (set next← NULL)
      }
      ELSE IF trail node ➔ get surname is equal to new data ➔
get surname //will set data with the same surname after the first
one input
      {
            new data ➔ set next ← (trail node ➔ get next)
            trail node ➔ (set next ← new data)
      }
      ELSE IF new data comes before (trail node ➔ get next)
      {
            IF new data comes after trail node
            {
                  new data➔ (set next←(trail node➔get next))
                  trail node ➔ (set next←new data)
            }
      }
      ELSE
      {
            add record //pass new data and (trail node➔get
next)
      }
}
```

Pseudo code for the **search list** function – it returns a customer Node object, it takes String **search** as a parameter and a customer node object **comp Node**. This is a recursive procedure which will compare the **search** string with the surname stored in the **comp Node.** Returns the first occurrence of a match and returns a null if there is no match.

```
Function search list
{
      IF comp node is equal to NULL
      {
            RETURN NULL
      }
      ELSE IF comp node has a surname equal to search
      {
            RETURN comp node
      }
      ELSE
      {
            RETURN searchlist //pass comp node ➔ get next AND
search
      }
}
```

Pseudo code for the **remove node** function – will remove node at a specific location in the linked list. It takes 2 parameters, the node **to remove** (found by the search function) and the **comp node** which will be used to compare with the node that is to be removed.**no return values**

```
Function remove node
```

```
{
      IF the list is empty
      {
            output to console "the list is empty"
      }
      ELSE IF the node to remove is equal to the head of the list
      {
            IF the head is not empty AND the item after head is NULL
            {
                  head←NULL
            }
            ELSE IF
            {
                  to remove ← get head
                  head ← (head→get next)
                  to remove ← NULL
            }
      }
      ELSE IF comp node → get next is equal to to remove AND
the last item in the list
      {
            comp node → (set next ← NULL)
      }
      ELSE IF
      {
            to remove ← (comp node → get next)
            node rem prev ← comp node
            node rem after ← (to remove → get next)
            rem prev → (set next←rem after)
            to remove ← NULL
      }
      ELSE
      {
            remove node //pass to remove and (comp node→get
next)
      }
}
```

Pseudo code for the **change node** function – will modify any data stored in the customer file of object of the node. It does not sort the list when done – but simply changes data. It returns the changed node and takes 3 parameters: 1 Node and 2 strings. The Node passed is the **node to change** and the Strings passed are the **new data** and the **data field** to be changed.

```
Function change node
{
      IF data field equals "Forename"
      {
            node to change → (set forename ← new data )
      }
      ELSE IF data field equals "Surname"
      {
            node to change → (set surname ← new data )
      }
      ELSE IF data field equals "Phone number"
      {
            node to change → (set phone number ← new data )
```

```
      }
      ELSE IF data field equals "Email address"
      {
            node to change ➔ (set email address ← new data )
      }
      ELSE IF data field equals "home address"
      {
            node to change ➔ (set home address ← new data )
      }
      ELSE IF data field equals "Post code"
      {
            node to change ➔ (set post code ← new data )
      }
      RETURN node to change
}
```

Pseudo code for the **sort list** function – this function performs a bubble sort on the linked list – sorts the list in ascending order according to surname (A-Z). The list returns nothing as the function is performed on the list and takes nothing as a parameter as it performs the function on the list.

```
function sort list
{
      IF the list is empty
      {
            output to console "cannot sort – the list is empty"
      }
      ELSE IF there is one item in the list
      {
            output to console "List sorted"
      }
      ELSE
      {
            Node current ← get Head
            Boolean swap done ← true
            WHILE swap done = true
            {
                  IF the item after current should precede current
                  {
                        Customer File temp dat ← (current➔get
data)
                        current ➔ set data ← (current ➔ get
next ➔ get data)

                        current ➔ get next ➔ (set data ← temp
dat)
                        swap done ← TRUE
                  }
                  current ← (current ➔ get next)
            }
            current ← get head
      }
}
```

Pseudo code for the **merge list** function – this function will take 2 singly linked lists as parameters and will merge them. It will

create a new linked list **new list** formed from the merging of the 2 lists. It takes 2 singly linked list objects as parameters: **lst1** and **lst2**. It returns the new linked list.

```
function merge list
{
      IF both lst1 and lst2 is empty
      {
            new list ← NULL
            output to console "both lists are empty – cannot merge"
      }
      ELSE IF lst1 is empty but lst2 is not
      {
            new list ← lst2
      }
      ELSE IF lst2 is empty but lst1 is not
      {
            new list ← lst1
      }
      ELSE
      {
            new list ← lst1
            node current ← (lst2 → get head)
            FOR int j ← 0 where j<(lst→size) j++
            {
                  new list → add record //pass current and new
list
                  current ← (current → get next)
            }
      }
      RETURN new list

}
```

Pseudo code for the **traverse and print** function – this function recursively traverses through the singly linked list and prints the contents of each node – it is void as it does not return anything but it does print strings to the screen so the user can see the information stored in each node. It takes a Customer Node **input** as a parameter and will recursively print and traverse the list until the end. The head node should be passed in the **input** parameter to allow the list to fully traverse. It returns nothing

---

```
Function traverse and print
{
      IF input is NOT equal to NULL
      {
            input →(get data→display all details)
            traverse and print //pass input→get next
      }
}
```

Pseudo code for the **print reverse list** function – this function does the same as the above function but traverses to the end of the list first and then prints – this means that the list will be printed in reverse order. Again it is void as it does not strictly

return anything but it does print the data to the screen again. The head node should be passed in the **input** parameter.

```
Function print reverse list
{
     print reverse list //pass input➔get next
     input➔get data➔display all details
}
```

Pseudo code for the **set Head** function – this is another "set" method which simply initializes the **head** node with the data String passed in the parameter **input**. It is void as it does not return anything.

```
Function set head
{
     input ➔ (set next ← head)
     head node ← input node
}
```

Pseudo code for the **get Head** function – this is another "get" method which simply returns the value assigned to the Node **head.** It has no parameters.

```
Function get head
{
     RETURN head node
}
```

Pseudo code for the **append last** function.- This is an advanced "set" method which will recursively traverse to the end of the list and then append the Node **input** to the end of the list. Is void, no return value, takes a Node **element of list** which allows the traversal and a Node **input** which is the new node to be added

---

```
Function append last
{
     IF element of list is equal to NULL
     {
          IF the list is empty
          {
               set head ← input
               input ➔(set next ← NULL)
          }
          ELSE
          {
               element of list ←input
               input➔(set next←null)
          }
     ELSE IF element of list.next is equal to NULL
     {
          element of list ➔ (set next ← NULL)
```

```
            input → (set next ← NULL)
      }
      ELSE
      {
            append last //pass element of list→get next AND
input
      }
}
```

Pseudo code for the **get last** function - This method simply returns the last node in the list – returns the last node in the list, takes a node **element of list** to allow traversal to the end of the list where the last node is to be found.

```
Function get last
{
      IF element of list is equal to NULL
      {
            RETURN element of list
      }
      ELSE IF element of list→get next is equal to NULL
      {
            RETURN element of list
      }
      ELSE
      {
            RETURN get last //pass element of list→get next
      }
}
```

Pseudo code for the **to Array** function – this function will convert the singly linked list into a 1 dimensional array which will be the current size of the list and will have all elements of the list. It takes no parameters and returns a Node[] array

---

```
Function to Array
{
      int k = size of List←head
      node data array[] ← new node[k]
      node listNode←get head
      FOR int j ← 0, from j to k (0 to end of list) j++
            data array[j] ← list node
            list node ← (list node → get next)
      RETURN data array
}

//---------------------End of Class---------------------------------
```

Main Class – the main interface between the User and the database, uses scanners and File I/O to add, modify, remove, sort, search, print or check the size of the list. This is achieved through the use of specific commands which are interpreted and specific things are done.

Variables:

```
String file name          //the name of the file to which the
```

```
linked list is saved
String password file      //the password required to use the
system is stored here
final String PASSCODE     //the unique passcode that cannot be
changed and is used to retrieve lost passwords.
```

Methods/functions:

main
save list to file
load list from file
set password
get password
set file name
get file name

---

development into code:

Pseudo code for the **main** function, this is the main method that will receive commands from the user and assign values through the use of File I/O –you can add, remove, search, sort, print list, get size clear the list or EXIT the stop the program – it is void as it does not return any values and takes no values as parameters

---

```
function main
{
        Boolean done ← FALSE
        WHILE done != TRUE
        {
                load list from file → singly linked list lst
                output to console "do you want to add/sort/search/print list/clear list/get
size/change password or EXIT to close"
                String user input ← input from console
                WHILE done != true
                {//following  'if' statements are not case sensitive
                        IF user input equals "ADD"
                        {
                                output to console "input forename"
                                String forename ← input to console
                                output to console "input surname"
                                String surname ← input to console
                                output to console "input phone number"
                                String phone number ← input to console
                                output to console "input email address"
                                String email address ← input to console
                                output to console "input Home address"
                                String home address ← input to console
                                output to console "input post code"
                                String post code ← input to console
                                Customer File new data
                                new data → set forename
                                new data → set surname
                                new data → set phone number
                                new data → set home address
                                new data → set set email address
                                new data → set post code
                                node new node
                                new node → (set data ← new data)
                                lst → add record //pass new node
                        }
                        ELSE IF user input equals "sort"
                        {
                                lst → sort list
                        }
                        ELSE IF user input equals "search"
                        {
                                output to console "input surname of customer required"
                                String search ← input from console
                                WHILE the file is not found
                                {
                                        lst → search list //pass search
```

```
                                        node data found ← lst search result
                                        IF the list is empty OR the search failed
                                        {
                                                output to console “search failed”
                                                escape loop
                                        }
                                        ELSE
                                        {
                                                WHILE confirmation is not one the valid forms
of input
                                                {
                                                        output to console “is this the file you
need?”
                                                        String confirmation ← input from
console
                                                        IF confirmation equals “yes”
                                                        {
                                                                output to console “what would you
like to do? Remove or modify or EXIT”
                                                                String rem or mod ← input from
console
                                                                IF rem or mod equals “remove”
                                                                {
                                                                remove data found from lst
                                                                }
                                                                ELSE IF rem or mod equals
“modify”
                                                                {
                                                                        output to console “what do
you want to change? Forename, Surname, home, email, post code or phone number”
                                                                        String command ← input
from console
                                                                        output to console “input
new data”
                                                                        String new data ← input
from console
                                                                        lst → change node
//pass command and new data

                                                                }
                                                                ELSE IF rem or mod equals “EXIT”
                                                                {
                                                                        exit the loop and return
to the main menu
                                                                }
                                                                ELSE
                                                                {
                                                                        output to console “invalid
input” continue WHILE loop
                                                                }
                                                        ELSE IF confirmation equals “no”
                                                        {
                                                                retry search and loop but from
the current node until a file is found or the search fails
                                                        }
                                                        ELSE
                                                        {
                                                                output to console “invalid input
- try again”
                                                                continue WHILE loop
                                                        }
                                                }
                                        }
                                }
                        }
                        ELSE IF user input equals “print list”
                        {
                                lst → print list
                        }
                        ELSE IF user input equals “clear list”
                        {
                                Boolean done clearing ← FALSE
                                WHILE done clearing is not equal to TRUE
                                {
                                        output to console “are you sure you want to clear the
list”
                                        String confirmation ← input from console
                                        IF confirmation equals “yes”
                                        {
                                                lst → clear list
                                                done clearing ← TRUE
                                        }
                                        ELSE IF confirmation equals “no”
                                        {
                                                output to console “list not cleared”
```

```
                                        done clearing ← TRUE
                                }
                                ELSE
                                {
                                        output to console "invalid input try again"
                                }
                        }
                }
                ELSE IF user input equals "get size"
                {
                        output to console "the size of the list is lst ← get size"
                }
                ELSE IF user input equals "change password"
                {
                        Boolean password validation ← FALSE
                        WHILE password validation is not equal to NULL
                        {
                                output to console "input old password"
                                String old password ← input from console
                                IF old password is correct
                                {
                                        output to console "input new password"
                                        String new pass ← input from console
                                        set password ← new pass
                                        password validation ← TRUE
                                }
                                ELSE
                                {
                                        output to console "incorrect password please try
again"
                                }
                        }
                }
                ELSE IF user input equals "EXIT"
                {
                        output to console "system closing"
                        done ← TRUE
                }
                ELSE
                {
                        output to console "invalid input – try again"
                        done ← FALSE
                }
        }
        lst ➔ save list to file
    }
}
```

Pseudo code for the **save list to file** function, this function will save the linked list to the disk so that it can be accessed later on – it returns nothing, it is private to ensure the list is protected from outside the program – it takes the Singly linked list as a parameter which will update the current linked list saved in memory.

```
Function save list to file
{
     output stream OS ➔ new stream to file name
     OS ➔ write object to file name //file name being the
location on disk
     OS ➔ flush //to ensure the file is saved
     OS ➔ close //closes the stream to ensure it is written
}
```

Pseudo code for the **load list from file** function, this will load the list from the file so that it can be modified. It returns the singly linked list on file and takes nothing as a parameter.

```
Function load list from file
```

```
{
      input stream IS ➔ new stream to file name
      singly linked list  ➔ (IS ➔ read object)
      IS ➔ close //closes the stream to ensure it is finished
}
```

Pseudo code for the **set password** function, this sets the password used to access the system so that the user may set it to their preference – it returns nothing and sets the String variable **user password** according to the string **input** taken as a parameter

---

```
Function set password
{
      output stream OS Password ➔ new stream to password file
      OS password ➔ write object ➔ input //writes the new
password
      OS Password ➔ flush//ensures the file is saved
      OS password ➔ close//ensures the stream is closed

}
```

Pseudo code for the **get password** function, this function is unique in the sense that it returns the **user password** saved on a file but in order to retrieve it, you must pass a String **input passcode** as a parameter which will be compared with the variable stored in **passcode.** if these strings are not the same, then the function will not return the password. It returns password string

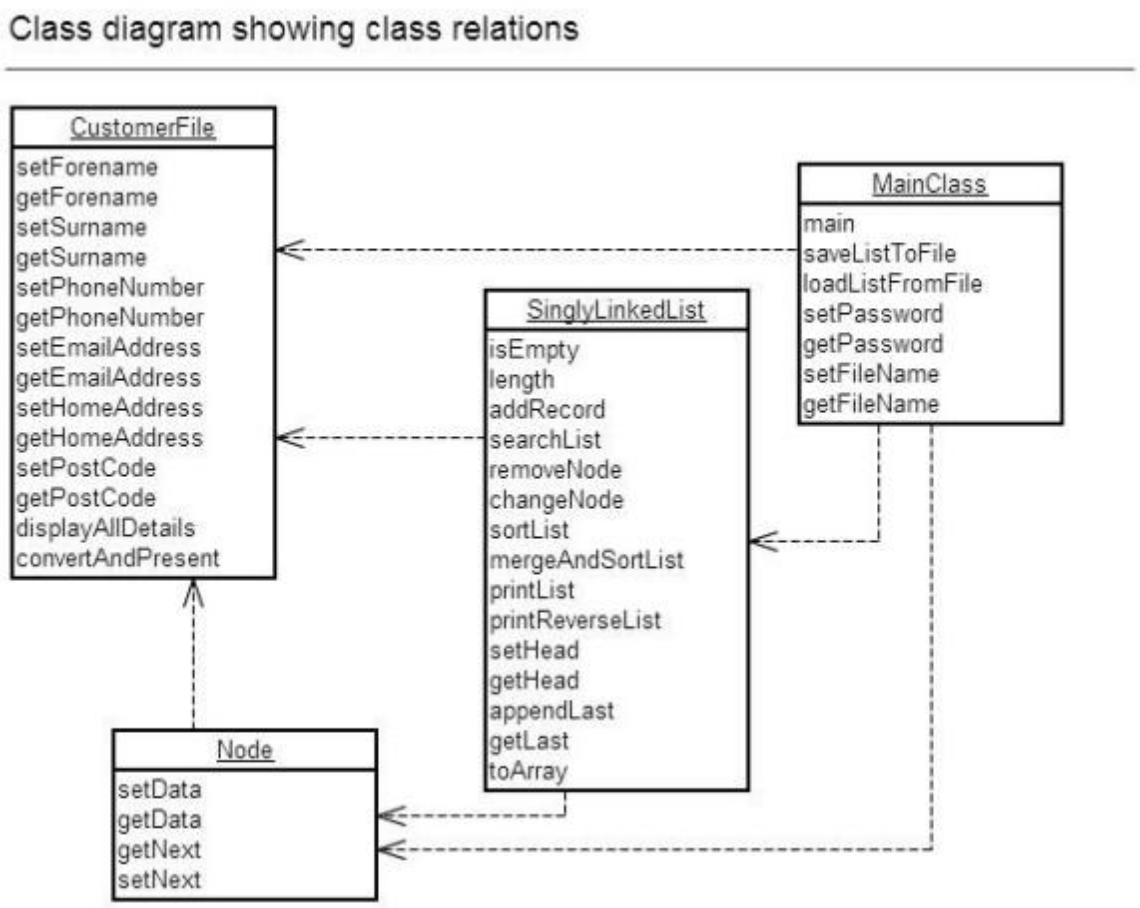
---

```
Function get password
{
      IF input passcode is equal to passcode
      {
            output stream IS password ➔ new output stream to
password file
            user password ➔ (IS Password ➔ read object)
            IS Password ➔ close
            output to console "correct system code, your password is
user password"
            RETURN user password
      }
      ELSE
      {
            output to console "incorrect system code - try again"
            RETURN NULL
      }
}

//--------------------End of class---------------------------------
```

B3: Modular Organisation

Consider the Following diagram: this shows how each class and the respective methods are linked and which class calls which.



As you can see from this diagram, modules are called by others that need them and all of the modules are called by the **main** class which is the CLI between the user and the system. The arrows represent the respective dependencies of the modules. In each module we are given the name of the module (the top box) and the various functions/methods of the module. These factors combined make sense when we consider the Linked List compares the strings of the CustomerFile using the **get Surname** function, thus the SinglyLinkedList module has a dependency on the CustomerFile module which contains this method/function.

B3: The CustomerFile Module

In this module contains the functions that will get and set the various data strings that are stored, there is no interface of any kind in this module and it is simply there to be used with other modules and store customer data. The module is called from the MainClass, the SinglyLinkedList and the Node modules. The module itself does not call any custom built modules.

called from	Links to
1. MainClass module • Main function 2. SinglyLinkedList module • addRecord function	-

• searchList function • removeNode function • changeNode function • sortList function • mergeAndSortList function • printList function • printReverseList function 3. Node module • getData function • setData function	

B3: The Node Module

The Node module contains the data and the next node. This is another get/set module and so it does not have an interface of any kind – it is called by other modules. It is called by the SinglyLinkedList module and the Main module and calls the CustomerFile module so it can store the data in the object when made.

Called from	Links to
1. SinglyLinkedList module • All functions 2. Main module • Main function	1. CustomerFile module • CustomerFile object

B3: The SinglyLinkedList Module

This module has the functions used to create and alter a SinglyLinkedList using the Node and CustomerFile modules. It calls the CustomerFile and Node modules; it is called by the Main module. It does not have an interface of any kind and is simply used for the methods and the SinglyLinkedList object.

Called from	Links to
1. Main module • Main function • saveListToFile function • loadListFromFile function	1. CustomerFile module • All functions 2. Node module • All functions

B3: The Main Module

This is the main module that deals with File I/O and user interface – it calls all of the other modules as it has to assign values to a CustomerFile object which is then stored in a Node object which is then stored in a SinglyLinkedList. It calls the CustomerFile module and the SinglyLinkedList module; it is not called by any other module.

Called from	Links to
-	1. CustomerFile module • All functions 2. SinglyLinkedList module • addRecord function • isEmpty function • length function • remove function

	• sortList function • searchList function • printList function • printReverseList function • changeNode function

Stage C: The Program

C1: Good Programming Style

Consider the following code from the CustomerFile class (screenshots of code in BlueJ):

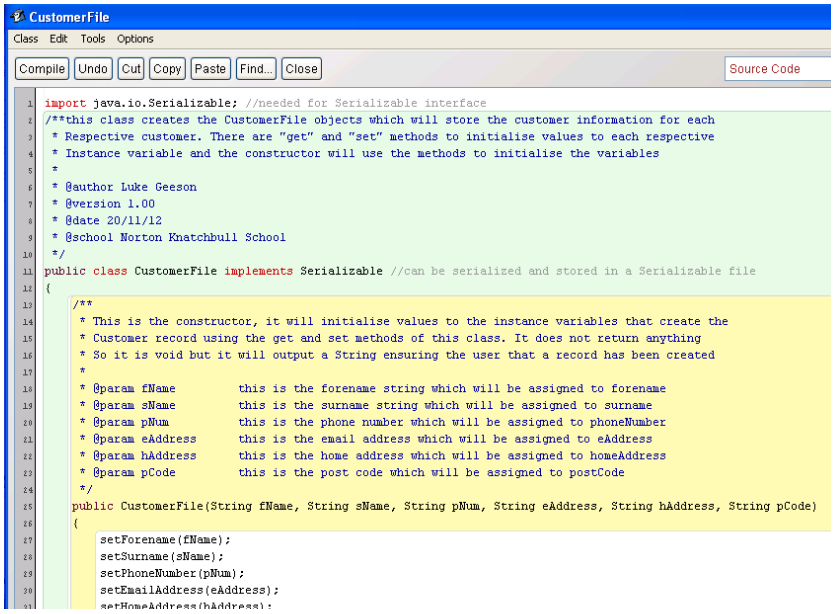
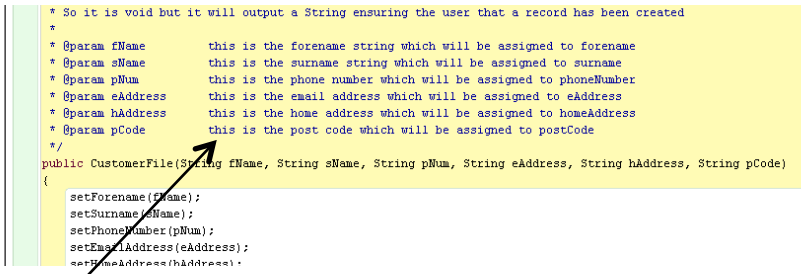
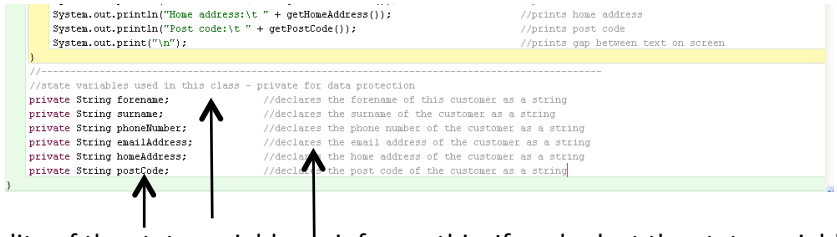


Figure 12 the screenshot of the top of the class and the constructor of the class (this is the screenshot of the top of the display – there is more but it cannot be fit on a page whilst being clearly visible.)



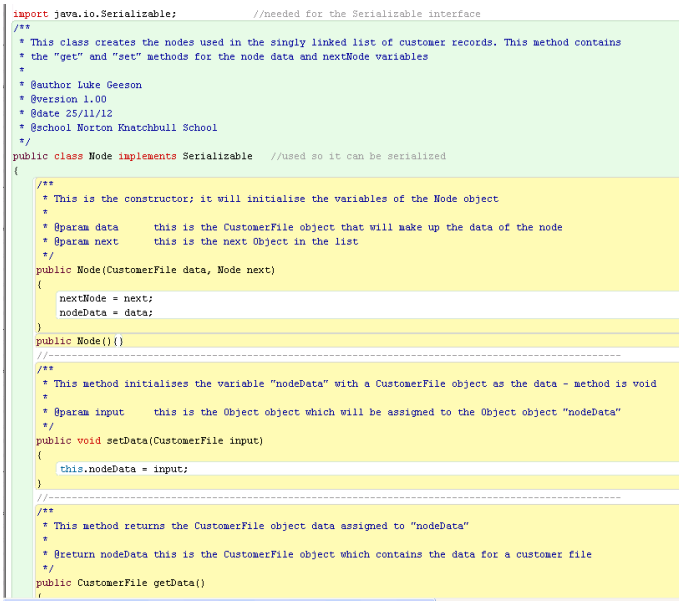
- Comments are included to describe the purpose and parameters of each method. In all methods the variables and parameters are given meaningful names (in this example "fName" for the forename parameter) which are easy to understand and the comments clarify this. The tag "@param" is used so that it is even clearer that these are the parameters ('param' indicating a parameter) and this enables code to be used with javaDoc documentation which is one of the most universal ways to present code from java in a HTML document that is clear to read.

•



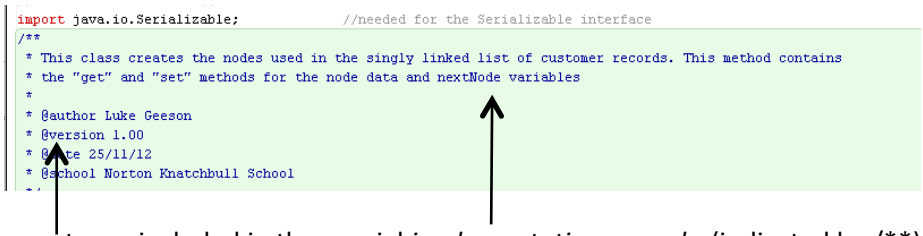
- The quality of the state variables reinforces this. If we look at the state variables, conventionally placed at the bottom of the class together as a trait of a good programming style, we see that the names are meaningful and obvious (it is clear that 'postCode' is a string that holds the post code of the customer).
- This is clarified by the comments next to each variable and the comment above the variables to state what they are and why they are private.
- Furthermore, I have named the identifiers with specific spelling ('postCode' as opposed to 'Post_Code' or 'POSTCODE') because this is one of the conventional ways of declaring something without indicating it as a FINAL object (all caps) and ensures they are not confused with class names (which start with a capital letter for each word). This shows an awareness of convention in coding for java as a good trait of a programmer and style.
- Also I believe the object types I have chosen are also appropriate as another factor that reinforces a good programming style: it is sensible to store words in a string (forename) and I had concluded that the best way to store numeric objects which were likely to be varying (phone numbers may be different) was in a string which indicates the initiative of a good programming style.
- Finally, I have declared the variables to be of type *private* so that the customer data is protected in accordance with the data protection act and it shows that I have taken the initiative in deciding on appropriate variables types. This also ensures that I can achieve the mastery factor of *encapsulation* as one of the fundamentals of OOP which highlights my awareness of great java technique and an adoption of a good programming style.
- Therefore, from this class alone you can see that I have demonstrated constant, type and variable declarations that have explanatory comments, identifiers with meaningful names and an awareness of conventional coding for java as traits of a good programming style.

Consider the following code from the Node class (screenshot taken in BlueJ):



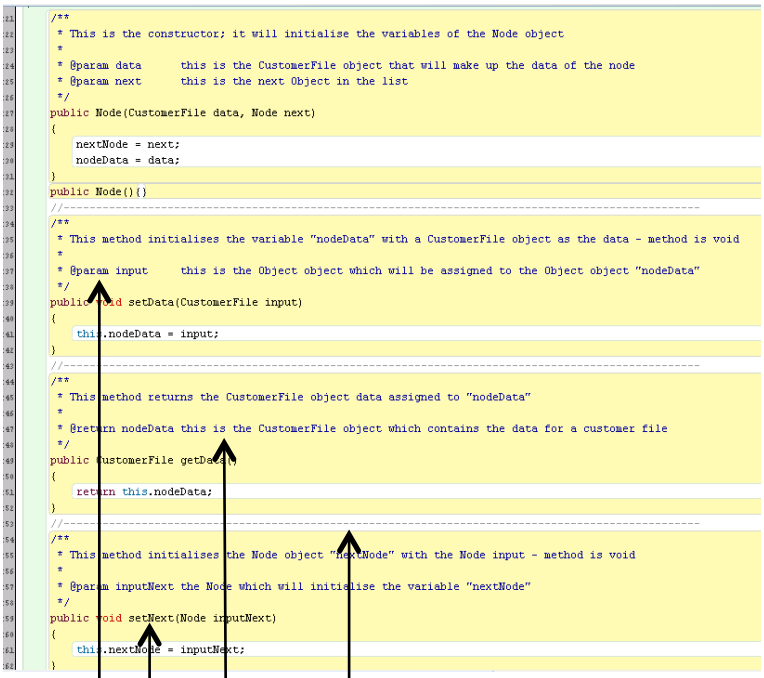
```
import java.io.Serializable;            //needed for the Serializable interface
/**
 * This class creates the nodes used in the singly linked list of customer records. This method contains
 * the "get" and "set" methods for the node data and nextNode variables
 *
 * @author Luke Geeson
 * @version 1.00
 * @date 25/11/12
 * @school Norton Knatchbull School
 */
public class Node implements Serializable   //used so it can be serialized
{
    /**
     * This is the constructor; it will initialise the variables of the Node object
     *
     * @param data      this is the CustomerFile object that will make up the data of the node
     * @param next      this is the next Object in the list
     */
    public Node(CustomerFile data, Node next)
    {
        nextNode = next;
        nodeData = data;
    }
    public Node(){}
    //-----------------------------------------------------------------------------------------
    /**
     * This method initialises the variable "nodeData" with a CustomerFile object as the data - method is void
     *
     * @param input     this is the Object object which will be assigned to the Object object "nodeData"
     */
    public void setData(CustomerFile input)
    {
        this.nodeData = input;
    }
    //-----------------------------------------------------------------------------------------
    /**
     * This method returns the CustomerFile object data assigned to "nodeData"
     *
     * @return nodeData this is the CustomerFile object which contains the data for a customer file
     */
    public CustomerFile getData()
```

Figure 13 this is the top of the Node class (cannot fit entire class on one page easily)



```
import java.io.Serializable;            //needed for the Serializable interface
/**
 * This class creates the nodes used in the singly linked list of customer records. This method contains
 * the "get" and "set" methods for the node data and nextNode variables
 *
 * @author Luke Geeson
 * @version 1.00
 * @date 25/11/12
 * @school Norton Knatchbull School
```

- Comments are included in the special *implementation remarks* (indicated by /**) which describes the purpose of the class and a comment is given to clarify what the purpose of the node class is as an example of good commenting. The use of implementation remarks allows this program to be used with *javaDoc* should it be necessary.
- The tags have been used again, but this time they indicate the *author, date, school* and version which is a good trait of a good programming style. This allows *author* and *version* to show up clearly if someone wished to use *javaDoc* with this program. It was not feasible to input the computer ID because this program was created on various computers and would be constantly changed.

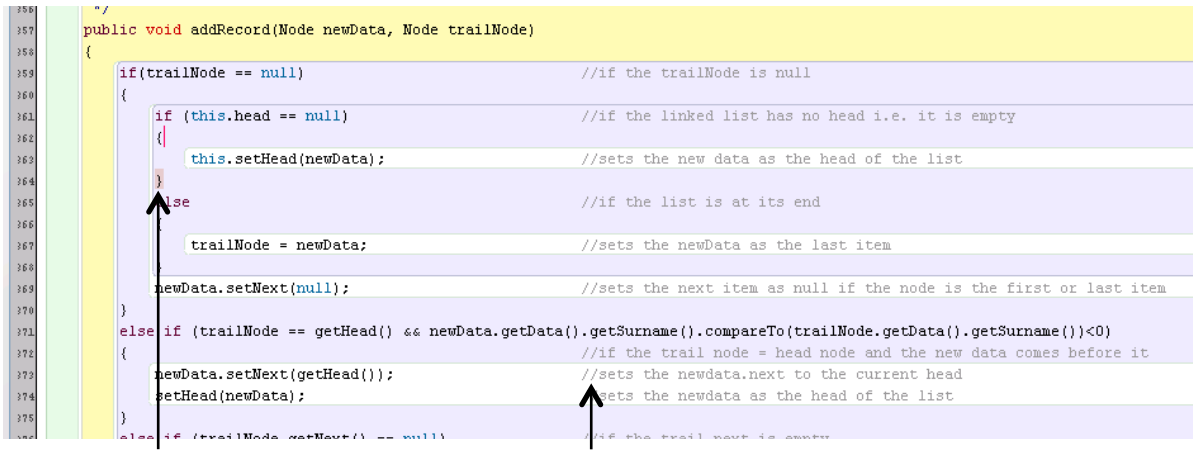


```
/**
 * This is the constructor; it will initialise the variables of the Node object
 *
 * @param data      this is the CustomerFile object that will make up the data of the node
 * @param next      this is the next Object in the list
 */
public Node(CustomerFile data, Node next)
{
    nextNode = next;
    nodeData = data;
}
public Node(){}
//-------------------------------------------------------------------------------------------
/**
 * This method initialises the variable "nodeData" with a CustomerFile object as the data - method is void
 *
 * @param input     this is the Object object which will be assigned to the Object object "nodeData"
 */
public void setData(CustomerFile input)
{
    this.nodeData = input;
}
//-------------------------------------------------------------------------------------------
/**
 * This method returns the CustomerFile object data assigned to "nodeData"
 *
 * @return nodeData this is the CustomerFile object which contains the data for a customer file
 */
public CustomerFile getData()
{
    return this.nodeData;
}
//-------------------------------------------------------------------------------------------
/**
 * This method initialises the Node object "nextNode" with the Node input - method is void
 *
 * @param inputNext the Node which will initialise the variable "nextNode"
 */
public void setNext(Node inputNext)
{
    this.nextNode = inputNext;
}
```

- This code shows that I have made active use of *implementation remarks* to describe the purpose of each method and what is taken as a parameter or returned. This is a good programming style that makes it easy to understand what this code does and once again allows for the use of *javaDoc*.
- The tags are used again to indicate what a parameter is (and what is returned) and clearly show what each parameter/return value signifies. The comments next to these indicate the roles of these variables within the method which is more evidence of a good programming style.
- Each method has been given a name that is meaningful and simple yet effective. For instance 'setNext' is concise and effective at describing what this method will do.
- As you can see, each method is separated by the 'comment barrier' (//-------) which is an eye-pleasing indicator that the method has finished and the code after would either be another method or the state variables.

Consider the following code in the SinglyLinkedList class for the add record method:

```
*/
public void addRecord(Node newData, Node trailNode)
{
    if(trailNode == null)                                   //if the trailNode is null
    {
        if (this.head == null)                              //if the linked list has no head i.e. it is empty
        {
            this.setHead(newData);                          //sets the new data as the head of the list
        }
        else                                                //if the list is at its end
        {
            trailNode = newData;                            //sets the newData as the last item
        }
        newData.setNext(null);                              //sets the next item as null if the node is the first or last item
    }
    else if (trailNode == getHead() && newData.getData().getSurname().compareTo(trailNode.getData().getSurname())<0)
    {                                                       //if the trail node = head node and the new data comes before it
        newData.setNext(getHead());                         //sets the newdata.next to the current head
        setHead(newData);                                   //sets the newdata as the head of the list
    }
    else if (trailNode.getNext() == null)                   //if the trail.next is empty
    {
        trailNode.setNext(newData);                         //set the next item in the list
        newData.setNext(null);                              //set the next item in the list as null
    }
    else if (newData.getData().getSurname().equalsIgnoreCase(trailNode.getData().getSurname()))
    {                                                       //if the surnames are identical - store new one after old
        newData.setNext(trailNode.getNext());               //set new data.next to trail node.next
        trailNode.setNext(newData);                         //set trailnode.next as new data
    }
    else if(newData.getData().getSurname().compareTo(trailNode.getNext().getData().getSurname())<0)
    {                                                       //if the new item comes before the trail.next
        if (newData.getData().getSurname().compareTo(trailNode.getData().getSurname())>0)
        {                                                   //if the new item comes after the trail node
            newData.setNext(trailNode.getNext());           //sets the new data as the node before the item after trail node
            trailNode.setNext(newData);                     //sets the new data as the item after the trail node
        }
    }
    else
    {                                                       //if the new data is not within the confines of the 2 nodes
        addRecord(newData,trailNode.getNext());             //recursive traversal through the list to the correct point
    }
```



```
*/
public void addRecord(Node newData, Node trailNode)
{
    if(trailNode == null)                                   //if the trailNode is null
    {
        if (this.head == null)                              //if the linked list has no head i.e. it is empty
        {
            this.setHead(newData);                          //sets the new data as the head of the list
        }
        else                                                //if the list is at its end
        {
            trailNode = newData;                            //sets the newData as the last item
        }
        newData.setNext(null);                              //sets the next item as null if the node is the first or last item
    }
    else if (trailNode == getHead() && newData.getData().getSurname().compareTo(trailNode.getData().getSurname())<0)
    {                                                       //if the trail node = head node and the new data comes before it
        newData.setNext(getHead());                         //sets the newdata.next to the current head
        setHead(newData);                                   //sets the newdata as the head of the list
    }
    else if (trailNode.getNext() == null)
```

- The following code demonstrates my ability to actively use effective and suitable indentation for the various programming constructs in my program. From the first 'if' statement, you can see this clearly and the arguments are clear.
- This is aided by comments in the code where it is difficult to understand – in this case, the method is rather difficult to interpret as it involves complex reallocation of nodes and so the comments are important – these are clearly laid out beside each line of code and contain meaningful descriptions. This factor is helped by the use of indentation on the comments so that the comments are easy to identify within the program itself.

C2: Handling errors

In this section, I shall document all of the error handling procedures present within my program which should cater for all errors input by the user and created by the system. I shall go about this by illustrating the methods and precautions put in place to catch each error created by the user and each potential error created by the system in each class of the program.

This section covers all input/out and file IO which is present in the system so that all code, classes, methods and functions have error handling procedures for input and output.

C2: The CustomerFile class

Consider the code of the **convertAndPresent** method which converts the data input so that it can be properly formatted for use within the system – this involves the removal of any extra whitespaces on each side of the String input and the capitalisation of the Forename and Surname strings so that they are sortable in the system. This is vitally important as the 'compareTo' method of the string class is used to compare and sort surnames which, would otherwise be unsorted and random; this is because the ASCII code for 'a' is different to the ASCII code for 'A' which means the list will be sorted with capitalised surnames of the records of the list.

```
public static String convertAndPresent(String toConvert)
{
    toConvert = toConvert.trim();
    if (toConvert.equals(""))
    {
        toConvert = "empty field";
    }
    char firstLetter = toConvert.charAt(0);
    String converted = String.valueOf(firstLetter).toUpperCase() +
toConvert.substring(1,toConvert.length());
    return converted;
}
```

Within this class, this method is applied to the following methods and the data associated with each: **setForename** and **setSurname**.

As all data set to the customer records are stored using these methods – the data will be formatted when the methods are called and these arguments passed – so from the **main** method of the main class, the data passed when adding to a file is formatted so it can be effectively sorted. Logically, this method does not work in the other data types of the class such as the **setPhoneNumber** method and the associated data type (as numbers do not have an upper case) or the **setPostCode** method in which the data must be all capitalised (as in a UK postcode). In these cases, methods such as the **toUpperCase()** method of the string class or the **trim()** method have been used to appropriately format the data for storage. For instance, consider the code of the **setPostCode** method which demonstrates but one of the methods in the class that ensures it is correctly stored.

```
public void setPostCode(String inputPCode)
{
    inputPCode = inputPCode.trim();
    inputPCode = inputPCode.toUpperCase();
    this.postCode = inputPCode;
}
```

As you can see, this method trims any extra whitespaces that the user may have accidentally added onto the end of the data. It also converts the String to its upper case variant which is how post codes

should be stored (UK). Finally, it assigns this formatted string to the private String **postCode** which is kept private for data protection.

Methods that do this, or similar, in the customer file class include **all the set methods** (where it is relevant)

As **there are many examples of this in this class** I have not included all of them for the sake of repetition.

C2: The Node class

The node class itself does not have an error handling procedures that deal with the user input or program because this class is a simple get/set class and the SinglyLinkedList, Main and CustomerFile deal with the user input errors at all other levels.

C2: The SinglyLinkedList class

The SinglyLinkedList class itself deals with all of the error handling for the linked list when items are added, removed modified or any of the methods are used within the class.

Consider the code for the **isEmpty** method of this class

```
public boolean isEmpty()
{
    boolean empty;
    if (this.head == null)
    {
        empty = true;
    }
    else
    {
        empty = false;
    }
    return empty;
}
```

As you can see from this method, the **isEmpty** function caters for the possibility of an empty list (or not) and returns a respective Boolean value.

Scenario	Expected result	Actual result
The list is empty	Returns a false	Returns a false
The list is not empty	Returns a true	Returns a true

Consider the following code for the **length** method:

```
public static int length(Node input)
{
    if (input == null)
    {
        return 0;
    }
    else
    {
        return 1 + length(input.getNext());
    }
}
```

This method has error testing for the list so that it can deal with an empty (NULL) list and would otherwise return the actual size of the list.

Scenario	Expected result	Actual result
The list is empty	Return 0	Return 0
The list has 4 items	Return 4	Return 4
The list has 10 items	Return 10	Return 10
The list has 2 items	Return 2	Return

Consider the following code of the **addRecord** method

```
public void addRecord(Node newData, Node trailNode)
{
    if(trailNode == null)
    {
        if (this.head == null)
        {
            this.setHead(newData);
        }
        else
        {
            trailNode = newData;
        }
        newData.setNext(null);
    }
    else if (trailNode == getHead() &&
newData.getData().getSurname().compareTo(trailNode.getData().getSurname())<0)
    {
        newData.setNext(getHead());
        setHead(newData);
    }
    else if (trailNode.getNext() == null)
    {
        trailNode.setNext(newData);
        newData.setNext(null);
    }
    else if
(newData.getData().getSurname().equalsIgnoreCase(trailNode.getData().getSurname())
)
    {
        newData.setNext(trailNode.getNext());
        trailNode.setNext(newData);
    }
    else if
(newData.getData().getSurname().compareTo(trailNode.getNext().getData().getSurname
())<0)
    {
        if
(newData.getData().getSurname().compareTo(trailNode.getData().getSurname())>0)
        {
            newData.setNext(trailNode.getNext());
            trailNode.setNext(newData);
        }
    }
    else
    {
```

```
            addRecord(newData,trailNode.getNext());
        }
    }
```

As you can see, I have facilitated for the 5 possible conditions of adding to the linked list – this is represented by the 'if, else' structure which will ultimately determine what to do when: the list is empty, if the linked list has no head, if the list is at its end, if the new data belongs at the head of the list, if the surnames are identical (it stores the new one after the old), or if the new data comes after the trail node and before the next (i.e. – within the list). There is also the recursive 'else' condition which will ensure traversal works appropriately so that the data is not simply tagged on the last item or before the first and that the system can find the correct position for the new data.

New data	Existing list	Expected new list	Actual new list
"Smith"	"Geeson, Johnson, Turner"	"Geeson, Johnson, Smith, Turner"	"Geeson, Johnson, Smith, Turner"
"Gatsby"	"Geeson, Johnson, Turner"	"Gatsby, Geeson, Johnson, Turner"	"Gatsby, Geeson, Johnson, Turner"
"Gates"	Null	"Gates"	"Gates"
"Geeson"	"Gatsby, Geeson, Turner" //different Geeson	"Gatsby, Geeson, Geeson, Turner" //old Geeson first	"Gatsby, Geeson, Geeson, Turner"

Consider the code for the **searchList** method of the class

```
public Node searchList(Node compNode, String search)
{
    if (compNode == null)
    {
        return null;
    }
    else if (compNode.getData().getSurname().equalsIgnoreCase(search))
    {
        return compNode;
    }
    else
    {
        return searchList(compNode.getNext(),search);
    }
}
```

As you can see, the code has been written so that it can manage if the list is empty or we have reached the lists end and would otherwise return the correct node or recursively traverse through the list.

Search string	List	Expected result	Actual result
"Smith"	"Geeson, Jones, Smith"	Return "Smith" node	Returns "Smith" node
"Jones"	"Geeson, Smith, Turner"	Return NULL	Returns null

Consider the code for the **removeNode** method

```
public void removeNode(Node toRemove, Node compNode)
{
```

```
    if (isEmpty() == true)
    {
        System.out.println("The list is empty - cannot remove this item as it does
not exist");
    }
    else if (toRemove == getHead())
    {
        if (getHead() != null && getHead().getNext() == null)
        {
            this.head = null;
        }
        else if (getHead().getNext() != null)
        {
            toRemove = getHead();
            this.head = getHead().getNext();
            toRemove = null;
        }
    }
    else if (compNode.getNext() == toRemove && compNode.getNext() ==
getLast(getHead()))
    {
        compNode.setNext(null);
    }
    else if (toRemove == compNode.getNext() )
    {
        toRemove = compNode.getNext();
        Node remPrev = compNode;
        Node remAfter = toRemove.getNext();
        remPrev.setNext(remAfter);
        toRemove = null;
    }
    else
    {
        removeNode(toRemove, compNode.getNext());
    }
}
```

As you can see from the method, it will search through the list until it finds the first occurrence of the node that we wish to remove – this method has the potential to throw multiple errors and complex selection has again been used to ensure that all possibilities are catered for so that no errors occur. The first 'If' statement will determine whether the list is empty, the second statement will determine whether the node we wish to remove is at the start of the list, the third statement determines whether the node we wish to remove is at the end of the list, the fourth statement determines whether the node is within the list (knowing that the statements beforehand have not been satisfied) and the fifth will recursively traverse through the list until one of the above statements is true

Surname of node to remove	Existing list	Expected result	Actual result
"Gatsby"	"Gatsby, Geeson, Marley, Phillip"	"Geeson, Marley, Phillip"	"Geeson, Marley, Phillip"
"Phillip"	"Geeson, Marley, Phillip"	"Geeson, Marley"	"Geeson, Marley"
"Marley"	"Gatsby, Geeson, Marley, Phillip"	"Gatsby, Geeson, Phillip"	"Gatsby, Geeson, Phillip"
"Gates"	NULL	Does nothing – list is	Does nothing – list is

		empty	empty
"Gates"	"Gatsby, Geeson, Marley, Phillip"	Does not remove what is not there	Does not remove what is not there

Consider the code for the **change node** method

```
public static Node changeNode(Node nodeToChange, String changeDecision, String
changeInput)
{
    if (changeDecision.equalsIgnoreCase("FORENAME") ||
changeDecision.equalsIgnoreCase("FIRSTNAME") ||
changeDecision.equalsIgnoreCase("FIRST NAME")||
changeDecision.equalsIgnoreCase("FIRST") || changeDecision.equalsIgnoreCase("F"))
    {
        nodeToChange.getData().setForename(changeInput);
    }
    else if (changeDecision.equalsIgnoreCase("SURNAME") ||
changeDecision.equalsIgnoreCase("SECONDNAME") ||
changeDecision.equalsIgnoreCase("SECOND NAME") ||
changeDecision.equalsIgnoreCase("S"))
    {
        nodeToChange.getData().setSurname(changeInput);
    }
    else if (changeDecision.equalsIgnoreCase("PHONENUMBER") ||
changeDecision.equalsIgnoreCase("PHONE NUMBER") ||
changeDecision.equalsIgnoreCase("PHONE"))
    {
        nodeToChange.getData().setPhoneNumber(changeInput);
    }
    else if(changeDecision.equalsIgnoreCase("EMAILADDRESS") ||
changeDecision.equalsIgnoreCase("EMAIL ADDRESS") ||
changeDecision.equalsIgnoreCase("EMAIL") || changeDecision.equalsIgnoreCase("E"))
    {
        nodeToChange.getData().setEmailAddress(changeInput);
    }
    else if (changeDecision.equalsIgnoreCase("HOMEADDRESS") ||
changeDecision.equalsIgnoreCase("HOME ADDRESS") ||
changeDecision.equalsIgnoreCase("HOME") || changeDecision.equalsIgnoreCase("H"))
    {
        nodeToChange.getData().setHomeAddress(changeInput);
    }
    else if (changeDecision.equalsIgnoreCase("POSTCODE") ||
changeDecision.equalsIgnoreCase ("POST CODE"))
    {
        nodeToChange.getData().setPostCode(changeInput);
    }
    return nodeToChange;
}
```

This method changes the unique data of each node, the code accounts for many different commands, so that the Surname can be changed if we request the following strings "Surname", "Secondname", "Second name" or "S" (Ignoring the case). This means that there is a degree of flexibility when inputting the field which is to be changed so that the program is not 'rigid' in the sense that it does not demand only one command when there are very many.

Consider the code for the **sort List**

```
public void sortList()
{
```

```
    if (isEmpty())
    {
        System.out.println("cannot sort list - list is empty\n");
    }
    else if (getHead().getNext() == null)
    {
        System.out.println("List sorted\n");
    }
    else
    {
        Node current = getHead();
        boolean swapDone = true;
        while (swapDone == true)
        {
            swapDone = false;
            while (current != null)
            {
                if (current.getNext() != null &&
current.getData().getSurname().compareTo(current.getNext().getData().getSurname())
>0)
                {
                    CustomerFile tempDat = current.getData();
                    current.setData(current.getNext().getData());
                    current.getNext().setData(tempDat);
                    swapDone = true;
                }
                current = current.getNext();
            }
            current = getHead();
        }
        System.out.println("List sorted\n");
    }
}
```

As you can see from the code, the method accounts for the possibility of an empty list, a 1 item list and would otherwise perform a bubble sort on the list so that the nodes are compared and swapped if necessary. This bubble sort will continue until it reaches the end of the list and will then decide whether or not to iterate through the list and sort it again which will be dependent on the Boolean *swapDone* variable being true. In each case it prints a message to indicate to the user whether the sort is successful or not.

Consider the code for the **mergeLists** method

```
public static SinglyLinkedList mergeLists(SinglyLinkedList lst1, SinglyLinkedList
lst2)
{
    SinglyLinkedList newList = new SinglyLinkedList();
    if (lst1 == null && lst2 == null)
    {
        newList = null;
        System.out.println("Both lists are empty - cannot merge");
    }
    else if (lst1 == null && lst2 != null)
    {
        newList = lst2;
    }
    else if (lst1 != null && lst2 == null)
    {
```

```
            newList = lst1;
        }
        else
        {
            newList = lst1;
            Node current = lst2.getHead();
            for (int j = 0; j < lst2.length(lst2.getHead()); j++)
            {
                newList.addRecord(current,newList.getHead());
                current = current.getNext();
            }
        }
        return newList;
}
```

The code ensures that measures are taken if one, or the other, or both lists are empty and would otherwise insert the items of the lists into one new one. Of course, like any merge method, this method requires that both lists are sorted initially so that minimum time and power is needed to sort and merge the lists.

Consider the code for the **printReverseList** method

```
public void printReverseList(Node input)
{
    if (input != null)
    {
        printReverseList(input.getNext());
        input.getData().displayAllDetails();
    }
}
```

With this recursive function, you can see that the list will continue until the 'if' statement is not satisfied so that it will call the items up the chain and not loop indefinitely.

Consider the code for the **printList** method

```
public void printList(Node input)
{
    if(input != null)
    {
        input.getData().displayAllDetails();
        printList(input.getNext());
    }
}
```

Much like the print reverse list method, this method also has an 'if' statement will only run for the length of the list

Consider the code for the **appendLast** method

```
public void appendLast(Node elementOfList,Node input)
{
    if (elementOfList == null)
    {
        if (isEmpty() == true)
        {
            setHead(input);
            input.setNext(null);
        }
```

```
        else
        {
            elementOfList = input;
            elementOfList.setNext(null);
        }
    }
    else if (elementOfList.getNext() == null)
    {
        elementOfList.setNext(input);
        input.setNext(null);
    }
    else
    {
        appendLast(elementOfList.getNext(), input);
    }
}
```

This code caters for the possibility of an empty list (the 'is empty' method is called) and would otherwise append to a null pointer which is interpreted as the last item in the list. It also accommodates for the possibility of the next item being null so that a null pointer exception is not thrown and would otherwise recursively traverse through the list which will ultimately be stopped by the one of the other statements.

Consider the code for the **getLast** method

```
public Node getLast(Node elementOfList)
{
    if (elementOfList == null)
    {
        return elementOfList;
    }
    else if (elementOfList.getNext() == null)
    {
        return elementOfList;
    }
    else
    {
        return getLast(elementOfList.getNext());
    }
}
```

The method caters for the possibility of an empty list (will return null), or if the next pointer is null which will return the last item and would otherwise recursively traverse through the list until the last item is found.

Consider the code for the **setHead** method

```
public void setHead(Node input)
{
    input.setNext(this.head);
    head = input;
}
```

The code ensures that the previous head is set as the next item so that the list is not lost when setting the next item as the head.

The **getHead** method does not take any value and simply returns the head variable of the linked list therefore there are no error routines needed for this method

C2: The main class

C2: Login code

Consider the section of code in the **main method of the main class** for the **login process**

```
boolean correctPas = false;
int attemptNo = 0;
String passwordAttempt = "";
Scanner kbReader = new Scanner (System.in);
while (correctPas != true)
{
    if ( attemptNo == 3)
    {
        passwordAttempt = null;
        while (passwordAttempt == null)
        {
            System.out.println("Login failed too many times, please input the
SYSTEM PASSCODE");
            String systemCode = kbReader.nextLine();
            passwordAttempt = getPassword(systemCode);
        }
    }
    else
    {
        System.out.println("Input login password - it is case sensitive ");
        passwordAttempt = kbReader.nextLine();
    }
    if (getPasswordWithoutCode().equals("") || getPasswordWithoutCode().equals("
") || getPasswordWithoutCode() == null)
    {
        setPassword("password");
    }
    if (passwordAttempt.equals(getPasswordWithoutCode()))
    {
        correctPas = true;
        Date date = new Date();
         System.out.println("Login successful - Welcome, the date is " +
date.toString() + "\n");
    }
    else
    {
        System.out.println("Incorrect password - please try again " + (2-
attemptNo) + " attempts remaining\n");
        if (attemptNo < 3)
        {
            attemptNo++;
        }
        correctPas = false;
    }
}
```

as you can see, the user is given 3 opportunities to input the correct password whereby the first statement will be satisfied if the user inputs the wrong password 3 times and then the loop will continue to iterate until the system code is input (this ensures the user has to input the system code if they input the wrong password). The second 'else' statement is to allow the user to input the password if they had not already done so and the second 'if' statement is to ensure that if there is no password saved to the disk then the default password will be 'password'. This allows first time

users to login. The third 'if' statement will be satisfied if the password input is correct which allows the user to escape the loop and login to the main menu of the system. If the password is wrong then the first 'if' statement will be satisfied whereby the user will be given 3 – n attempts (where n is attempNo). This allows for a secure login system where there are no loopholes that allow malicious users into the system

C2: Main menu commands code

Consider another segment of the code of the **main** method of the **main** class for the **main menu commands**

note: this has been paraphrased to show only the main menu commands as there is over 400 lines of other complex code; the other functions of the main class will be covered after the main menu commands (see below)

```
SinglyLinkedList a = loadListFromFile();
boolean done = false;
while (done != true)
{
    saveListToFile(a);
    System.out.println("What do you want to do? add/search/Print all/get
size/sort/clear list/change password or EXIT to quit");
    String choice = kbReader.nextLine();
    choice.trim();
    if(choice.equalsIgnoreCase("ADD") || choice.equalsIgnoreCase("A"))
    {
        //omitted code for demonstration (add function)
    }
    else if(choice.equalsIgnoreCase("SEARCH") || choice.equalsIgnoreCase("S"))
    {
        //omitted code for demonstration (search function)
    }
    else if(choice.equalsIgnoreCase("PRINT ALL") ||
choice.equalsIgnoreCase("PRINT") || choice.equalsIgnoreCase("P"))
    {
       //omitted code for demonstration (print function)
    }
    else if(choice.equalsIgnoreCase("GET SIZE") || choice.equalsIgnoreCase("SIZE")
|| choice.equalsIgnoreCase("GS"))
    {
       //omitted code for demonstration (get size function)
    }
    else if(choice.equalsIgnoreCase("SORT"))
    {
       //omitted code for demonstration (sort function)
    }
    else if(choice.equalsIgnoreCase("CLEAR LIST") ||
choice.equalsIgnoreCase("CLEAR") || choice.equalsIgnoreCase("C"))
    {
       //omitted code for demonstration (clear list function)
    }
    else if(choice.equalsIgnoreCase("EXIT") || choice.equalsIgnoreCase("E"))
    {
      System.out.println("System closing");
       done = true;
    }
```

```
    else if (choice.equalsIgnoreCase("CHANGEPASSWORD") ||
choice.equalsIgnoreCase("CHANGE PASSWORD") || choice.equalsIgnoreCase("PASSWORD"))
    {
       //omitted code for demonstration (change password function)
    }
    else
    {
       System.out.println("Invalid request, please try again\n");
       done = false;
       continue;
    }
}
saveListToFile(a);
kbReader.close();
```

As you can see, the main menu gives you numerous options (`add/search/Print all/get size/sort/clear list/change password or EXIT to quit`) and so the system caters for any one of these (or the abbreviations: "p" for "print all"). This main menu system is contained in a while loop whereby the system can continue to take commands to change the list and then loop back to the main menu request line (`What do you want to do? add/search/Print all/get size/sort/clear list/change password or EXIT to quit`) until "exit" is input (triggering the sentinel value of `done = true`) which allows the user to exit the loop and close the system when they are ready and not simply after they have done 1 thing. The system also caters for the input of invalid commands (the else statement at the end) which ensures that the system will not crash or loop indefinitely and can deal with invalid data input.

It is also worthy to note the code at the start of the loop and outside the loop: (`saveListToFile(a)`) this ensures that with each successive change to the list, the master file version of the list is updated so that no changes are missed. (See the **save list to file** method for more on how this works)

C2: Code for adding a record

Consider the code for the **add function** within the **main method of the main class** that is used to add a new record to the list (this is one of the sections of code which was omitted for demonstration purposes in the previous example)

```
if(choice.equalsIgnoreCase("ADD") || choice.equalsIgnoreCase("A"))
{
    System.out.println("Input Forename");
    String fName = new String (kbReader.nextLine());
    boolean hasSurname = false;
    String sName = "";
    while (hasSurname != true)
    {
        System.out.println("Input Surname");
        sName = new String (kbReader.nextLine());
        if (sName.equalsIgnoreCase("") || sName == null )
        {
            System.out.println("Invalid input - please insert a surname");
        }
        else
        {
            hasSurname = true;
```

```
        }
    }
    System.out.println("Input Phone Number");
    String pNum = new String (kbReader.nextLine());
    System.out.println("Input Email Address");
    String eAdd = new String (kbReader.nextLine());
    System.out.println("Input the first line of the Home Address");
    String hAdd = new String (kbReader.nextLine());
    System.out.println("Input Post Code");
    String pCode = new String (kbReader.nextLine());
    CustomerFile custDat = new CustomerFile(fName,sName,pNum,eAdd,hAdd,pCode);
    Node custNode = new Node(custDat, null);
    if (a == null || a.getHead() == null)
    {
        a = new SinglyLinkedList();
    }
    a.addRecord(custNode, a.getHead());
    System.out.println("File added successfully\n");
 }
```

As you can see, the user can use the scanner object to input the data for each field; the surname is the key value by which the record is sorted and so a while loop is used to ensure that something is input for the surname field – this eliminates the possibility of inputting a nameless record which may be incorrectly placed in the list. With this while loop, the system will iterate until something is input for the surname field (triggering the sentinel `hasSurname = true`); when this is done, the system will leave the loop to continue inputting data. Once all of the data is input, the system will create a customer file object which is placed inside a node object which is then added to the list. The 'if' statement at the end ensures the possibility of a null or empty list is catered for so that the program doesn't crash when the node is added to a *null* list.

Is noteworthy to add that the error handling for data input is very basic by nature as a complex data integrity function would take a very long time to make and would not be feasible with the time given for this project.

C2: Code for searching for a record

Consider the code section for the **search function** within the **main** method of the **main class** where the code which is irrelevant to the search function has once again been removed for demonstrative purposes (see below for the remove and modify functions)

```
else if(choice.equalsIgnoreCase("SEARCH") || choice.equalsIgnoreCase("S"))
{
    Node result = null, current = null;
    boolean foundFile = false;
    String search = "";
    while (foundFile != true)
    {
        if (result == null)
        {
           System.out.println("Input Surname of customer required");
           search = kbReader.nextLine();
           search = search.trim();
        }
        if (a == null)
        {
            System.out.println("Search failed - the list is empty\n");
            foundFile = true;
```

```
            continue;
        }
        else if (current != null)
        {
            if (current.getNext() == null)
            {
                System.out.println("search failed");
                foundFile = true;
                continue;
            }
            else
            {
                result = a.searchList(current.getNext(), search);
            }
        }
        else
        {
           result = a.searchList(a.getHead(), search);
        }
        if (result == null || search == "")
        {
            System.out.println("Search failed, please try again");
            foundFile = true;
            break;
        }
        else
        {
            System.out.println("search successful - printing details of file\n");
            result.getData().displayAllDetails();
        }
        System.out.println("Is this the record you need? yes/no");
        String validation = kbReader.nextLine();
        if (validation.equalsIgnoreCase("YES") ||
validation.equalsIgnoreCase("Y"))
       {
            //code omitted for demonstrative purpose(remove and modify functions)
        }
        else if (validation.equalsIgnoreCase("NO") ||
validation.equalsIgnoreCase("N"))
        {
            current = result;
        }
        else
        {
            System.out.println("Invalid request, please try again");
        }
    }
}
```

The search function utilises 2 node objects and a Boolean value (`foundFile`) whereby a while loop will ensure that the system searches for a record until the record is or the search fails. The first 'if' statement allows the user to input a surname which is trimmed to ensure an exact match can be found with surnames in the record (the convert and present method in the customer file class ensures this is so with surnames stored in the list). The second 'if' statement will not search the list if the list is empty and would otherwise search the list from a previous point in the list (if a record is found but is the incorrect one); if this is not the case, then the list will search from the beginning of the list which will occur the first time a search is done. The next if statement will indicate that the search has failed (if `result == null || search == ""`) or display the details of a record with a

matching surname to that searched. The next 'if','else if' and 'else' statement determines whether the record 'is' or 'is not' the correct record (with 'yes' and 'no' input values respectively). Notice the 'else' statement which ensures invalid data is catered for so that the system will print record with the same surname and should the user input invalid data at the confirmation (not yes or no) then the system will not crash but would simply reiterate through the loop and ask for confirmation once more.

C2: Code for deciding to remove or modify the record

This code determines whether the user changes or modifies the record which has been found with the search function (above). This code will take the input from the user which then determines what section of the while loop is satisfied (remove or modify or otherwise). The remove and modify functions within this code are complex in their own right and have been omitted from this section for demonstration purposes (they are each given their own section to demonstrate how they work (below)

```
boolean doneEdittingFile = false;
while (doneEdittingFile != true)
{
    System.out.println("What would you like to do with this record? remove/modify
or EXIT");
    String remOrMod = kbReader.nextLine();
    if (remOrMod.equalsIgnoreCase("REMOVE") || remOrMod.equalsIgnoreCase("R"))
    {
        //omitted code for demonstration purposes (remove function below)
    }
    else if (remOrMod.equalsIgnoreCase("MODIFY") || remOrMod.equalsIgnoreCase("M")
|| remOrMod.equalsIgnoreCase("CHANGE") || remOrMod.equalsIgnoreCase("C"))
    {
        //omitted code for demonstration purposes (modify function below)
    }
    else if(remOrMod.equalsIgnoreCase("EXIT") || remOrMod.equalsIgnoreCase("E"))
    {
        doneEdittingFile = true;
        continue;
    }
    else
    {
        doneEdittingFile = false;
        System.out.println("Invalid input - please try again\n");
        remOrMod = null;
    }
}
```

As you can see, the code will accept remove or its abbreviation, modify or its abbreviations, Exit (should they choose to not remove or modify the file) or other invalid input. The 'while' loop will iterate until one of the valid forms of input is detected by the 'if' statements (remove, modify or exit) and will also detect and reject invalid input (the 'else' statement). If valid input is detected then it will accept it and perform one of the functions (removing, modifying or exiting to the main menu) and the system would otherwise reject the invalid input and reiterate through the 'while' loop once more. This ensures only valid input is accepted at this stage.

C2: Code for the remove function

Consider the code section for the **remove function** which was omitted for demonstration in the previous example

```
current = result;
foundFile = true;
boolean doneEdittingFile = false;
while (doneEdittingFile != true)
{
    System.out.println("What would you like to do with this record? remove/modify
or EXIT");
    String remOrMod = kbReader.nextLine();
    if (remOrMod.equalsIgnoreCase("REMOVE") || remOrMod.equalsIgnoreCase("R"))
    {
        boolean finRemove = false;
        while (finRemove != true)
        {
            System.out.println("Are you sure that you want to remove this file?
yes/no");
            String confirm = kbReader.nextLine();
            if (confirm.equalsIgnoreCase("YES") || confirm.equalsIgnoreCase("Y"))
            {
                a.removeNode(current, a.getHead());
                System.out.println("Record successfully removed\n");
                finRemove = true;
            }
            else if(confirm.equalsIgnoreCase("NO") ||
confirm.equalsIgnoreCase("N"))
            {
                System.out.println("File not removed\n");
                result = null;
                current = null;
                finRemove = true;
            }
            else
            {
                System.out.println("Invalid input - please try again");
                finRemove = false;
            }
        }
        doneEdittingFile = true;
    }
}
```

The user is given the option to confirm whether they want to remove the file (yes or no). the use of a while loop and the sentinel value (**boolean** finRemove) means the system will exit the loop only when 'yes' or 'no' is input and would otherwise iterate until "yes" or "no" is input.

C2: Code for the modify function

Consider the code for the **modify** function which was the second function and was omitted from the **search function illustration** (above). This function allows the user to modify single data fields of the record which has been found with the **search function**

```
else if (remOrMod.equalsIgnoreCase("MODIFY") || remOrMod.equalsIgnoreCase("M") ||
remOrMod.equalsIgnoreCase("CHANGE") || remOrMod.equalsIgnoreCase("C"))
{
    boolean finChange = false;
    while (finChange != true)
```

```
    {
        System.out.println("What data within this record would you like to change?
\nforename/surname/email address/home address/post code/phone number or EXIT to
leave");
        String changeDecision = kbReader.nextLine();
        if (changeDecision.equalsIgnoreCase("FORENAME") ||
changeDecision.equalsIgnoreCase("FIRSTNAME") ||
changeDecision.equalsIgnoreCase("FIRST NAME")||
changeDecision.equalsIgnoreCase("FIRST") || changeDecision.equalsIgnoreCase("F"))
        {
            System.out.println("Input new forename");
            String newForename = kbReader.nextLine();
            current = a.changeNode(current, changeDecision, newForename);
            System.out.println("Forename changed\n");
            finChange = true;
        }
        else if (changeDecision.equalsIgnoreCase("SURNAME") ||
changeDecision.equalsIgnoreCase("SECOND NAME") ||
changeDecision.equalsIgnoreCase("SECONDNAME") ||
changeDecision.equalsIgnoreCase("S"))
        {
            boolean hasSurname = false;
            String newSurname = "";
            while (hasSurname != true)
            {
                System.out.println("Input New Surname");
                newSurname = new String (kbReader.nextLine());
                if (newSurname.equalsIgnoreCase("") || newSurname == null )
                {
                    System.out.println("Invalid input - please insert a surname");
                }
                else
                {
                    hasSurname = true;
                }
            }
            current = a.changeNode(current, changeDecision, newSurname);
            System.out.println("Surname changed");
            finChange = true;
            a.sortList();
        }
        else if ((changeDecision.equalsIgnoreCase("PHONENUMBER") ||
changeDecision.equalsIgnoreCase("PHONE NUMBER") ||
changeDecision.equalsIgnoreCase("PHONE") || changeDecision.equalsIgnoreCase("P")))
        {
            System.out.println("Input new phone number");
            String newNumber = kbReader.nextLine();
            current = a.changeNode(current, changeDecision, newNumber);
            System.out.println("Phone number changed\n");
            finChange = true;
        }
        else if ((changeDecision.equalsIgnoreCase("EMAILADDRESS") ||
changeDecision.equalsIgnoreCase("EMAIL ADDRESS") ||
changeDecision.equalsIgnoreCase("EMAIL") || changeDecision.equalsIgnoreCase("E")))
        {
            System.out.println("Input new Email Address");
            String newEmail = kbReader.nextLine();
            current = a.changeNode(current, changeDecision, newEmail);
            System.out.println("Email address changed\n");
```

```
            finChange = true;
        }
        else if ((changeDecision.equalsIgnoreCase("HOMEADDRESS") ||
changeDecision.equalsIgnoreCase("HOME ADDRESS") ||
changeDecision.equalsIgnoreCase("HOME") || changeDecision.equalsIgnoreCase("H")))
        {
            System.out.println("Input the new first line of house address");
            String newHAddress = kbReader.nextLine();
            current = a.changeNode(current, changeDecision, newHAddress);
            System.out.println("Home address changed\n");
            finChange = true;
        }
        else if (changeDecision.equalsIgnoreCase("POSTCODE") ||
changeDecision.equalsIgnoreCase ("POST CODE"))
        {
            System.out.println("Input new post code");
            String newPCode = kbReader.nextLine();
            current = a.changeNode(current, changeDecision, newPCode);
            System.out.println("Post code changed\n");
            finChange = true;
        }
        else if (changeDecision.equalsIgnoreCase("EXIT"))
        {
            finChange = true;
            continue;
        }
        else
        {
            finChange = false;
            System.out.println("Invalid input - please try again");
        }
    }
    doneEdittingFile = true;
}
```

The code demonstrates that I have catered for every field of data that the user wishes to input (and their abbreviations ("h" for "home address") as well as invalid data input or `EXIT`. These values are enclosed in a while loop (with a sentinel value `boolean finChange = false;`) that ensures the user must input valid data or the system will simply reiterate and request valid input again. It is also worthy to note that I have included the same while loop that is present in the add function for the surname as this ensures an empty surname is not input.

C2: Code for printing the list

Consider the code for the **print function** in the **main method of the main class**. This allows the user to print the list of items at request.

```
else if(choice.equalsIgnoreCase("PRINT ALL") || choice.equalsIgnoreCase("PRINT")
|| choice.equalsIgnoreCase("P"))
{
    if (a == null || a.isEmpty() == true)
    {
        System.out.println("The list is empty - cannot print list\n");
    }
    else
```

```
    {
        a.printList(a.getHead());
    }
}
```

The code demonstrates that I have catered for the possibility of an empty list and the program would otherwise print the list without problems

C2: Code for getting the size of the list

Consider the code for the **get size function** in the **main method of the main class**

```
else if(choice.equalsIgnoreCase("GET SIZE") || choice.equalsIgnoreCase("SIZE") ||
choice.equalsIgnoreCase("GS"))
{
    if (a == null || a.isEmpty() == true)
    {
        System.out.println("The list is empty, the size is 0\n");
        continue;
    }
    else
    {
        System.out.println("The size of the list is " + a.length(a.getHead()) + "
items\n");
    }
}
```

The code demonstrates that the system has catered for the possibility of an empty list (or not) and prints the appropriate message accordingly.

C2: Code for sorting the list

The error handling procedures for this method are contained within the **sort method of the singly linked list class** and so there are none needed here. To ensure that data items are kept in order the **sort function** is called when a record (and hence a surname) is added to the list or when the surname field of a record is changed (with the **modify function**)

C2: Code for clearing the list

Consider the code for the **clear function** of the **main method of the main class** where the user is given the choice whether they want to clear the list or not.

```
else if(choice.equalsIgnoreCase("CLEAR LIST") || choice.equalsIgnoreCase("CLEAR")
|| choice.equalsIgnoreCase("C"))
{
    boolean doClear = false;
    while (doClear != true)
    {
        System.out.println("Are you sure you want to clear the whole list?
yes/no");
        String toClear = kbReader.nextLine();
        if (toClear.equalsIgnoreCase("YES") || toClear.equalsIgnoreCase("Y"))
        {
            a = null;
            System.out.println("List cleared\n");
            doClear = true;
            toClear = null;
```

```
        }
        else if(toClear.equalsIgnoreCase("NO") || toClear.equalsIgnoreCase("N"))
        {
            System.out.println("List not cleared");
            doClear = true;
            toClear = null;
        }
        else
        {
            System.out.println("Invalid request - please try again");
            doClear = false;
        }
    }
}
```

Before the user can clear the list, they must first confirm their choice; this is achieved by the placement of a yes/no response which is enclosed in a while loop (with sentinel value `boolean doClear = false`). The system will exit the loop when the user inputs 'yes' or 'no' (or abbreviations) and would otherwise iterate the loop and request valid input again (invalid input is catered for by the else statement).

C2: Code for changing the password

Consider the code for the **change password** function **in the main method of the main class.** This allows the user to change the password which is saved on the file.

```
else if (choice.equalsIgnoreCase("CHANGEPASSWORD") ||
choice.equalsIgnoreCase("CHANGE PASSWORD") || choice.equalsIgnoreCase("PASSWORD"))
{
    boolean isPassword = false;
    String oldPassword = "";
    while (isPassword != true)
    {
        System.out.println("Input old password, it is case sensitive");
        oldPassword = kbReader.nextLine();
        if(oldPassword.equals(getPasswordWithoutCode()))
        {
            System.out.println("Input new password");
            String newPass = kbReader.nextLine();
            setPassword(newPass);
            System.out.println("Password Changed\n");
            isPassword = true;
        }
        else
        {
            boolean confirm = false;
            while (confirm != true)
            {
                System.out.println("Incorrect password, do you want to try again?
yes/no");
                String retry = kbReader.nextLine();
                if (retry.equalsIgnoreCase("YES") || retry.equalsIgnoreCase("Y"))
                {
                    isPassword = false;
                    confirm = true;
                }
                else if(retry.equalsIgnoreCase("NO") ||
retry.equalsIgnoreCase("N"))
```

```
                {
                    isPassword = true;
                    confirm = true;
                    System.out.println("Password not changed\n");
                }
                else
                {
                    System.out.println("Invalid input, please try again");
                    confirm = false;
                }
            }
        }
    }
}
```

The user is must input their old password before they can input the new one; this means the password input must be exactly the same to the one stored on the file - the first 'if' statement detects a correct input of the password and requests a new password to be input whereas the 'else' statement will be satisfied if the password input is incorrect. If this is the case, then the user can choose to try again (yes/no with a loop to detect yes/no/invalid input) where the system also caters for invalid input (the 'else' statement within the second while loop)

C2: Code for the saveListToFile method

Consider the code for the **save list to file** method of the **main class**. This method uses serialization in order to save the singly linked list to the disk.

```
private static void saveListToFile(SinglyLinkedList lst)
{
    try
    {
        ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream(fileName));
        os.writeObject(lst);
        os.flush();
        os.close();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

This code shows the use of the output stream objects and the appropriate use of a try/catch statement to ensure the system does not crash if a file is not found or there is an input exception/error. I have also used the appropriate **close()** and **flush()** methods to ensure memory is cleared and the stream is closed.

C2: Code for the loadListFromFile method

This is the method that loads the singly linked list from the disk; it is the **main class**

```
private static SinglyLinkedList loadListFromFile()
```

```
{
    SinglyLinkedList lst = null;
    try
    {
        ObjectInputStream is = new ObjectInputStream(new
FileInputStream(fileName));
        lst = (SinglyLinkedList) is.readObject();
        is.close();
    }
    catch(FileNotFoundException e)
    {
        e.printStackTrace();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
    catch(ClassNotFoundException e)
    {
        e.printStackTrace();
    }
    return lst;
}
```

As you can see, the code has the appropriate try/catch statements for IO errors, file not found errors and class errors. I have also used the **close()** method for the stream as well.

C2: Code for the getPassword method

Consider the code for the **getPassword** method of the **main class.** This method returns the password upon the correct input of the system code.

```
private static String getPassword(String systCode)
{
    if (systCode.equals(PASSCODE))
    {
        String userPassword = "";
        try
        {
            ObjectInputStream isPassword = new ObjectInputStream (new
FileInputStream (passwordFileName));
            userPassword = (String) isPassword.readObject();
            isPassword.close();
            System.out.println("Correct system code, your password is: " +
userPassword);
        }
        catch(FileNotFoundException e)
        {
            e.printStackTrace();
        }
        catch(IOException e)
        {
            e.printStackTrace();
        }
        catch(ClassNotFoundException e)
        {
            e.printStackTrace();
        }
```

```
        return userPassword;
    }
    else
    {
        System.out.println("That is the incorrect system code, please try again");
        return null;
    }
}
```

As you can see, there are 2 statements to which to method will respond: if the correct system code is input or if it is not (the 'if' and 'else' statement respectively. Notice in the first statement there is the appropriate try/catch statements for the input stream; notice the use of **close()** as well: this ensures the stream is closed so the IO is complete. Notice in the else statement, the system prints the appropriate message and returns `null`.

C2: Code for the getPasswordWithoutCode method

Consider the code for the **getPasswordWithoutCode method** that is used to obtain the user password without inputting the system code (only used within the program itself to avoid malicious access).

```
private static String getPasswordWithoutCode()
{
    String userPassword = "";
    try
    {
        ObjectInputStream isPassword = new ObjectInputStream (new FileInputStream
(passwordFileName));
        userPassword = (String) isPassword.readObject();
        isPassword.close();
    }
    catch(FileNotFoundException e)
    {
        e.printStackTrace();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
    catch(ClassNotFoundException e)
    {
        e.printStackTrace();
    }
    return userPassword;
}
```

As you can see the code has the appropriate try/catch statement and the use of a **close()** method for the input stream: this stops the program from crashing should there be a file IO error.

C2: Code for the setPassword method

Consider the code for the **setPassword** method of the **main class.** This class allows the user to save a password to the disk.

```
private static void setPassword(String newPassword)
```

```
{
    try
    {
        ObjectOutputStream osPassword = new ObjectOutputStream (new
FileOutputStream (passwordFileName));
        osPassword.writeObject(newPassword);
        osPassword.flush();
        osPassword.close();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

The code demonstrates the appropriate use of try/catch statements, the use of **flush()** and **close()** that enable the system to continue to work if IO errors or stream errors occur.

C2: Code for the getFilename method code for the setFilename method

These methods are simple *get* and *set* methods for the private variables (encapsulation) and do not have or need error handling procedures as a result.

C3: Success of the Program

For references in this section please refer to D1: Annotated hard copy of test output and all of the subsections within it (note the testing evidence will be there as there are many tests and it would not be feasible to copy them again into this section).

Requirement	Expected result	Actual result	Comments/evidence
1. A system where the user can add, modify, remove, sort and search for customer information.	the user can do all of these on the linked list that stores the customer data via a command line interface	the user can do all of these on the linked list that stores the customer data via a command line interface	See: D1 Annotated hard copy of test output, D1: Adding an item to the list, D1: Searching for an item in the list, D1: Removing an item, and D1: Sorting the list and D1: modifying an item in the list
2. The same fields of data input.	the same fields were used to input customer data: these were the forename, surname, email address, home address, phone number and post code	The same fields are used and the user can add a customer and input data for all of the fields or modify the data in a specific field	See D1: Annotated hard copy of test output - D1: Adding an item to the list and modifying an item in the list
3. The system must be easy to back-up and capable of upgrades in the future.	The file containing the customer data can be simply copied to a secure and/or external drive or the whole system can be backed up onto an external drive as a system image	The file containing the customer data can be saved onto another drive via the easy 'drag and drop' interface of windows 7	See D1: Annotated hard copy of test output - D1: Satisfying requirement 3
4 .The program/system must be quick and easy to use.	the business computer has high specifications for hardware and the program will run on this – should be quick to suit	I cannot provide evidence of how fast the system responds to commands (would require video evidence) but it does work quickly	See Stage B: Detailed Design - Inputs and Output - Choosing Hardware for system spec but a speed test of the computer itself cannot be done
5. A system that is sorted alphabetically by surname	The customer records should be sorted by surname and stored by surname	The **addRecord** and **sortList** methods ensure that the data is sorted as it is added and sorted by surname	See D1: Annotated hard copy of test output - D1: Adding an item to the list and D1: Sorting the list
6. A system that can store all of the current customer information as well as many more customer files. There have been comments	The business computer has a very large hard drive and will easily be able to store all of the customer data. The business computer will be used	The business computer is used, the program is stored on the computer and the hard-drive can easily store all of the customer data.	See Stage B: Detailed Design - Inputs and Output - Choosing Hardware for system spec – evidence of the use of the system cannot be provided by the business.

made on the preferred use of the businesses computer for the task.			
7. The system must have some form of data validation.	The system does not allow you to input an 'empty' surname when adding/applying changes to a customer record. This is important as the whole system is sorted by this.	All of the commands and menu choices must be validated before the system uses them so that the system does not crash forms of validation including: 'automatic capitalisation' of names, trimming data of whitespaces, capitalising UK postcodes and ensuring that illegal data types input do not crash the system as well as interpreting commands or otherwise catching and looping until correct input is input A1: Analysis of the existing system	See D1: Annotated hard copy of test output - D1: Login to the system, D1: The main menu, D1: Adding an item to the list, D1: Searching for an item in the list, D1: Removing an item, D1: Modifying an item in the list, D1: Clearing the list and D1: Changing the password
8. A user manual must be supplied so that re-training is not required annually.	A user manual is to be supplied with the system that allows the user to use the system.	a user manual is supplied with the system which allows the users to learn how to use the system. It is available at all times so that they may 'top-up' their knowledge whenever they feel it is necessary	See a copy of the user manual in Stage D: Documentation – User manual
9. Levels of security must be put in place so that the data is secure from malicious influences.	A login method must be used which will not allow the user to login unless they provide the password OR the system password, should they forget theirs.	There is a sequence of code at the start of the main method which ensures that the user must input the password or the System password if they forget theirs. It does not let you use the database otherwise.	See D1: Annotated hard copy of test output - D1: Login to the system and D1: Changing the password
10. The ability for the user to search by surname	A search method must be used which allows them to search for the customer by surname	The search method allows them to search by surname	See: D1: Annotated hard copy of test output - D1: Searching for an item in the list

Mastery Factors

Below is a table that indicates the use of mastery factors in the program. Please find the references to the code in the Appendix D: Hard copy of the program source code

Mastery Factor		How I demonstrate this mastery factor	Page no(s) of Where in my program to find this Mastery Factor	Page no(s) where the Explanation for using this Mastery Factor can be found
HL	recursion	Recursive **functions in the SinglyLinkedList class**	Pages 8-19 of appendix D; code lines 349, 398, 423, 467, 603, 619, 673 and 695	Page 29: section B
HL	Merging two or more data structures	**Merging 2 singly linked lists into one new one** – method in the SinglyLinkedList class.	Pages 15-16 of appendix D; code lines 555-592	Page 28: section B
HL	encapsulation	**Setting state variables as private and being able to access them from specific "get" and "set" methods – found in the CustomerFile Class, Node Class**, **SinglyLinkedList Class** and **Main Class**: most **emphasis placed on code in the Customer File and Node classes.**	Pages 1-8, 17, 35-36 of appendix D; code lines 40-164, 236-274	Page 30: section B
HL	Parsing a text file or data stream	**Implementing the Serializable interface** in order to save and load the Singly Linked List and modify it. **Using ObjectOutputStream, FileOutputStream, ObjectInputStream and FileInputStream methods**	Pages 31-35 of appendix D; code lines 1156-1305	
HL	Implementing a hierarchical composite data structure	**Use of the SinglyLinkedList** in which **each element of the list is a node** and **each node contains a data record (CustomerFile object)** and a pointer to the next Node in the list. The **customer file object contains all the data for a specific customer** – hence a record	Pages 1-20 of appendix D; code lines 1-719	Pages 28-30: Section B
HL	Any 5 SL mastery factors: user defined methods	Creating **custom methods to Get and set data (encapsulation)** in the CustomerFile	Every method	Every method in

	Any 5 SL mastery factors: User-defined methods with parameters	and Node classes, **specific methods to alter a singly linked list** whilst **using parameters and return values**, **specific methods to interpret the data input from the keyboard** and **alter the linked list** stored as a ".ser" file.	in the program (in appendix D)	the program (in appendix D)
	Any 5 SL mastery factors: Simple selection (if–else) Any 5 SL mastery factors: Complex selection (nested if, if with multiple conditions or switch)	The **methods used in the SinglyLinkedList class and Main class use both simple selection and complex selection with nested if statements as well as while loops**	Singly linked list: pages 8-20 (lines 353-400), main class pages 20-31 of appendix D (lines 838-1150)	Every method in the program (in appendix D)
	Any 5 SL mastery factors: User-defined objects Any 5 SL mastery factors: Objects as data records	**Custom made CustomerFile, Node and SinglyLinkedList Objects are created**. The **CustomerFile stores Customer data**, The **Nodes Store CustomerFile Objects and next Nodes**, The **SinglyLinkedList Object Stores Nodes**.	pages 1-31 of appendix D lines 1-306	Consult appendix D
	Any 5 SL mastery factors: Searching	The **SinglyLinkedList class has a Search method** which **searches the SinglyLinkedList for the first occurrence of a Node** containing the same String passed as an argument into the method.	Pages 11-12 of appendix D lines 401-425	Pages 25-26: section B
HL	Implementation of ADTs: the singly linked list	A **SinglyLinkedList class** that **uses a Node class** in which the key value by which data is compared **can be set dependent on what data is using the classes** – this LinkedList object can be **used to compare any Objects with a common String Key value and put them in ascending order (A-Z).** The class has **the following methods that universalise the class** so it can be widely used: **Add, remove, sort, search, modify, add Last, get Last, Get Head, set Head, isEmpty, Length, merge lists and sort lists, print list, print reverse list** and a **toArray** method.	Customer file and node classes: pages 1-8, singly linked list class: pages 8-20 of Appendix D (lines 1-720)	Pages 21-30: section B
HL	Implementation of ADTs			
HL	Implementation of ADTs			
HL	Implementation of ADTs			

Stage D: Documentation

D1: Annotated hard copy of test output

D1: Login to the system

In order to use the program, the user must first input a correct password so that the system is available for them to use. The following table summarises the expected and actual output of the login process. The user is given 3 attempts to input the correct password and would otherwise have to resort to inputting a SYSTEM CODE that comes with the system. Following this table I have included the screen grabs of the program showing these data being tested.

For this test run, I have set the password to “barley”

Test for:	Input	Data type	Expected input	Expected result	Actual result	Comments:
1.Correct input on first attempt	“barley”	Valid	“barley”	Login successful – continue to main menu	Login successful – continue to main menu	See evidence screen grab
2.Incorrect input on first attempt	“aloevera5”	Invalid	“barley”	Incorrect – retry with 2 attempts remaining	Incorrect – retry with 2 attempts remaining	
3.Correct input on second attempt	“barley”	Valid	“barley”	Login successful – continue to main menu	Login successful – continue to main menu	
4.Incorrect input on second attempt	“christmas43”	Invalid	“barley”	Incorrect – retry with 1 attempt remaining	Incorrect – retry with 1 attempt remaining	
5.Correct input on third attempt	“barley”	Valid	“barley”	Login successful – continue to main menu	Login successful – continue to main menu	
6.Incorrect input on third attempt	“WHALE88”	Invalid	“barley”	Attempts failed – continue to input system code	Attempts failed – continue to input system code	
7.Correct input of system code	"14GF5602C2"	Valid	"14GF5602C2"	Login successful – continue to main menu	Login successful – continue to main menu	Displays the password for next time
8.Incorrect	“panda”	Invalid	"14GF5602C2"	Incorrect –	Incorrect –	

input of system code				retries system code input	retry system code input	

Evidence that this testing works:

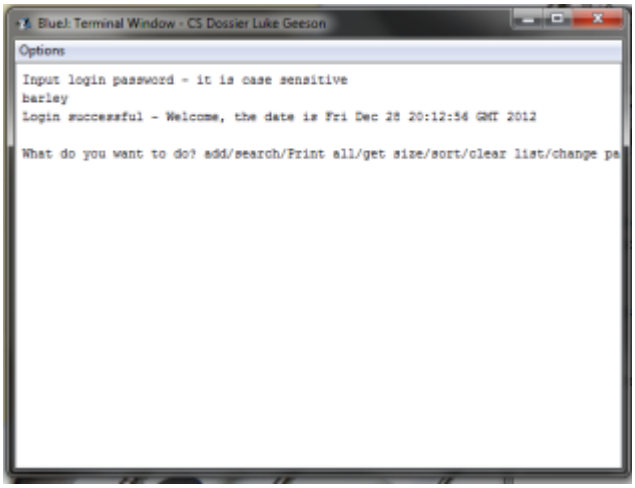


Figure 14: Evidence of test 1: to show correct input on the first attempt

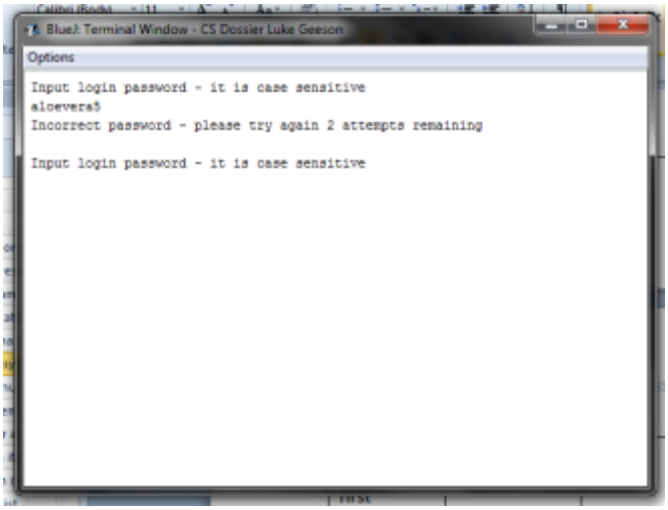


Figure 15: Evidence of test 2: to show incorrect input on the first attempt

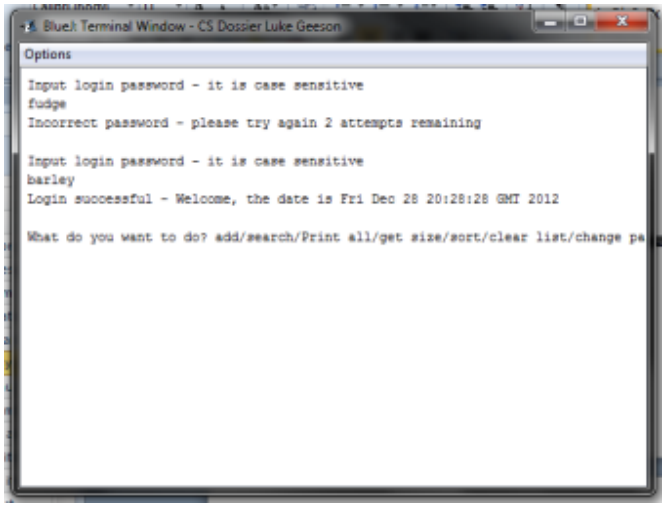


Figure 16: Evidence of test 3: to show correct input on the second attempt

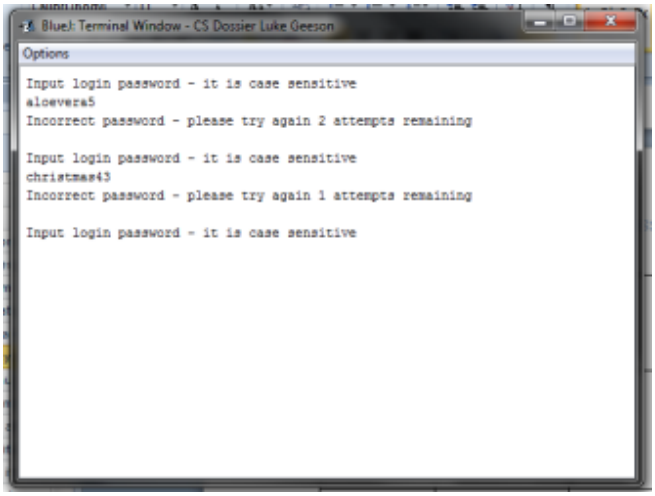


Figure 17: Evidence of test 4: to show incorrect input on the second attempt

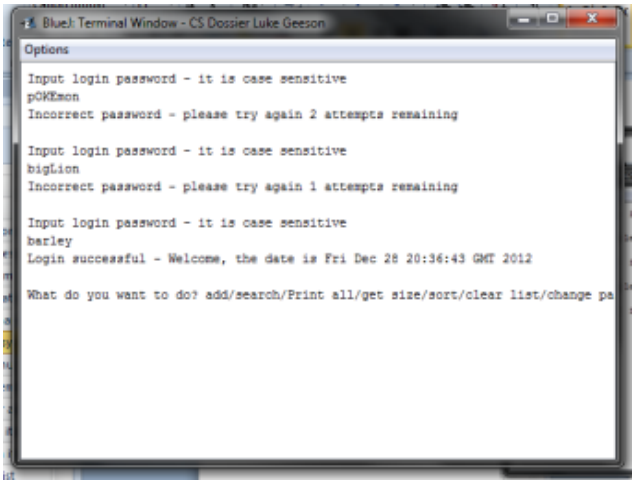


Figure 18: Evidence of test 5: correct input on third attempt

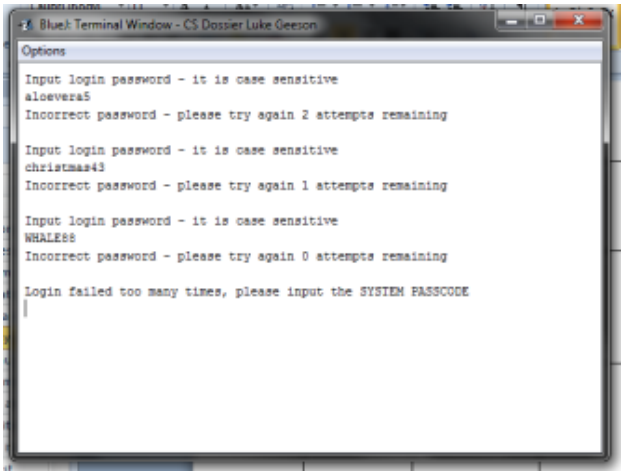


Figure 19: Evidence of test 6: to show incorrect input on the third attempt

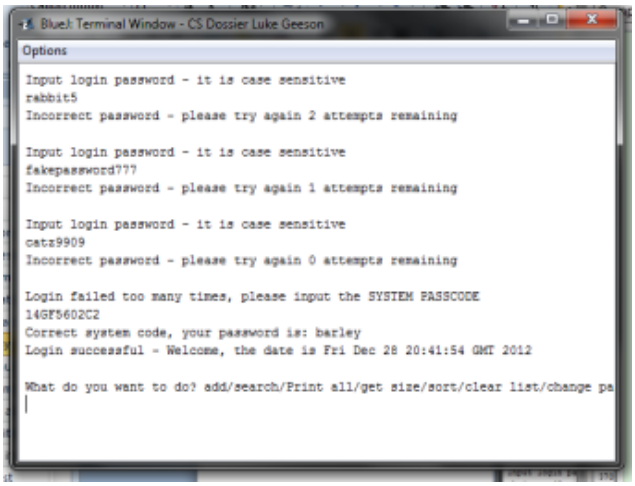


Figure 20: evidence of test 7: to show correct input of the system code

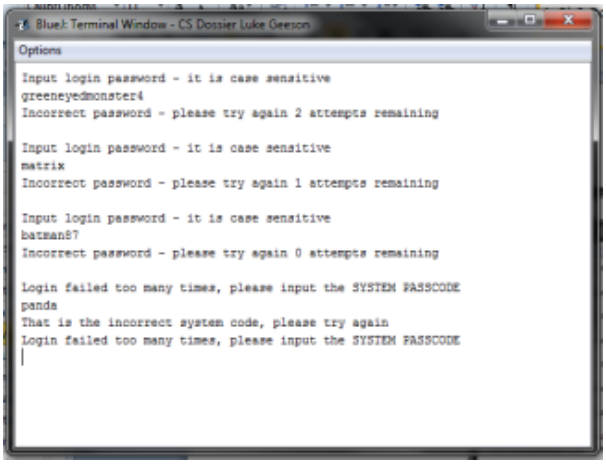


Figure 21: Evidence of test 8: to show incorrect input of the system code

D1: The main menu

This section tests the general functioning of the main menu – it tests the CLI commands to ensure that the commands (and their abbreviations) of the main menu work correctly when input.

Once the user has successfully logged in to the system, they will be given the option to "add/search/Print all/get size/sort/clear list/change password or EXIT to quit" this means the user can type in any of the above (or the abbreviations as highlighted in the user manual) and the program will interpret these commands and make changes to the list as follows: in the following table, I will show the process of inputting commands at the main menu itself. The specific commands such as the search command are equally complex in their own right and deserve closer attention in the sections below this one.

Note: the commands input are not case sensitive for this and so the user may input "add" or "ADD" for the same result

Test for:	Input	Data type	Expected input	Expected result	Actual result	Comments:
1.Accessing the "add" menu with "add"	"add"	valid	"add" or "a"	Proceed to add customer	Proceeds to add customer	User must now input customer information
2.Accessing the "add" menu with "A"	"a"	valid	"add" or "a"	Proceed to add customer	Proceeds to add a customer	User must now input customer information
3.Accessing the search menu with "search"	"search"	valid	"search" or "s"	Proceed to search	Proceed to search	User must now input surname of customer
4.Accessing the search menu with "s"	"s"	valid	"search" or "s"	Proceed to search	Proceed to search	User must now input surname of customer
5.Printing the list with "print all"	"print all"	valid	"print all" or "print" or "p"	Prints the list	Prints the list	
6.Printing the list with "print"	"print"	valid	Same as above	Prints the list	Prints the list	
7.Printing the list with "p"	"p"	valid	Same as above	Prints the list	Prints the list	
8.Getting the size of the list with "get size"	"get size"	valid	"get size" or "size" or "gs"	Returns the size	Returns the size	As a string
9.Getting the size of the list with "size"	"size"	valid	Same as above	Returns the size	Returns the size	As a string
10.Getting the size of the list with "gs"	"gs"	valid	Same as above	Returns the size	Returns the size	As a string
11.Sorting the	"sort"	valid	"sort"	Sorts the	Sorts the	Prompts

list with “sort”				list	list	the user
12.Clearing the list with “clear list”	“clear list”	valid	“clear list” or “clear” or “c”	Proceed to clear upon confirmation	Proceed to clear upon confirmation	Asks confirmation
13.Clearing the list with “clear”	“clear”	valid	Same as above	Proceed to clear upon confirmation	Proceed to clear upon confirmation	Asks confirmation
14.Clearing the list with “c”	“c”	valid	Same as above	Proceed to clear upon confirmation	Proceed to clear upon confirmation	Asks confirmation
15.Closing the program with “exit”	“exit”	valid	“exit” or “e”	Closes program	Closes program	
16.Closing the program with “e”	“e”	valid	Same as above	Closes program	Closes program	
17.Changing the password with “change password”//with space	“change password”	valid	“change password” or “changepassword” or “password”	Proceeds to change password	Proceeds to change password	Asks for old password first
18.Changing the password with “changepassword”	“changepassword”	valid	Same as above	Proceeds to change password	Proceeds to change password	Asks for old password first
19.Changing the password with “password”	“password”	valid	Same as above	Proceeds to change password	Proceeds to change password	Asks for old password first
20.Accounting for invalid data input	“twerp50”	invalid	Any of the above	Retry input	Retry input	
21.Accounting for invalid data input	“pandabear”	invalid	Any of the above	Retry input	Retry input	
22.Accounting for invalid data input	“hello world”	invalid	Any of the above	Retry input	Retry input	

Evidence that the testing works:

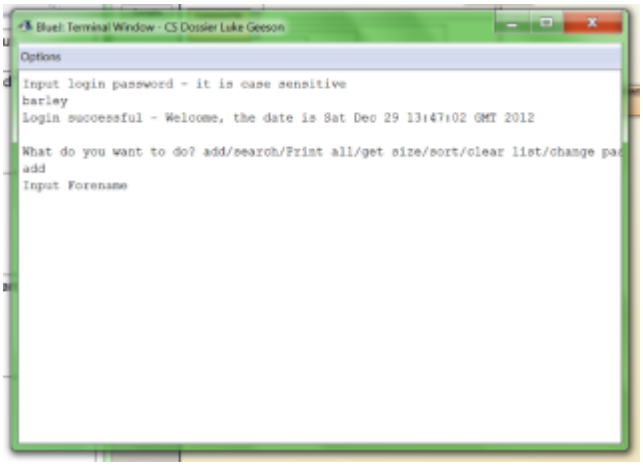


Figure 22: evidence of test 1: inputting "add" to add a record

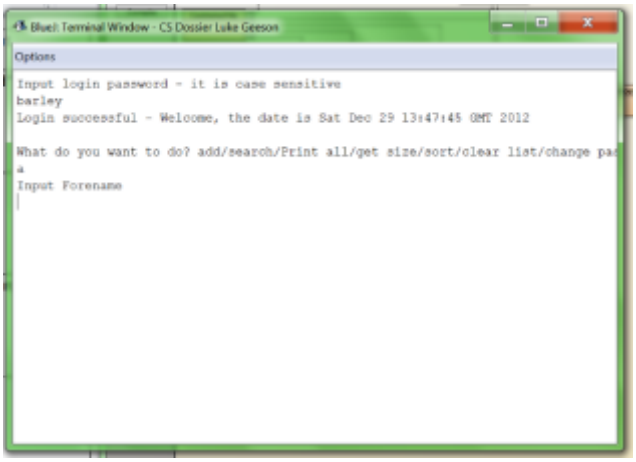


Figure 23: evidence of test 2: inputting "a" to add a record

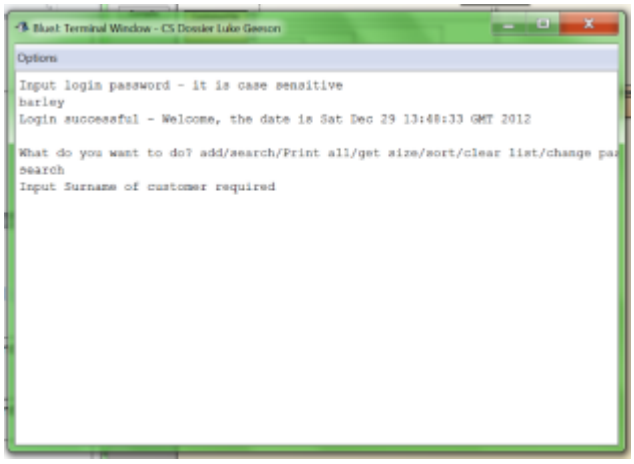


Figure 24: evidence of test 3: inputting "search" to search for a record

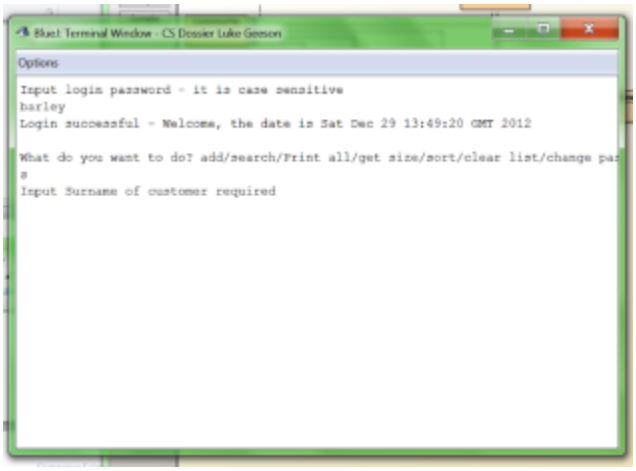


Figure 25: evidence of test 4: inputting "s" to search for a record

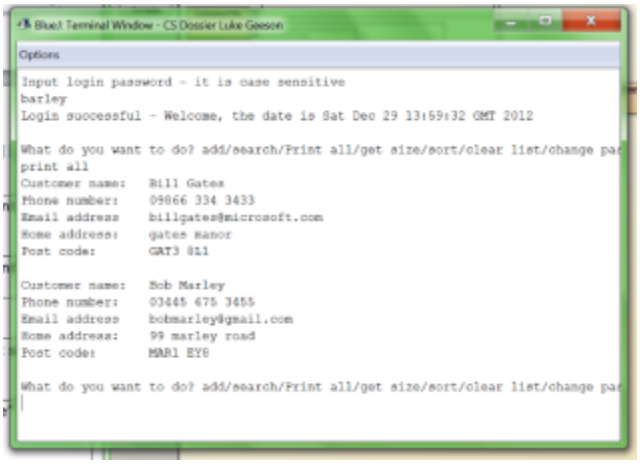


Figure 26: evidence of test 5: inputting "print all" to print the list

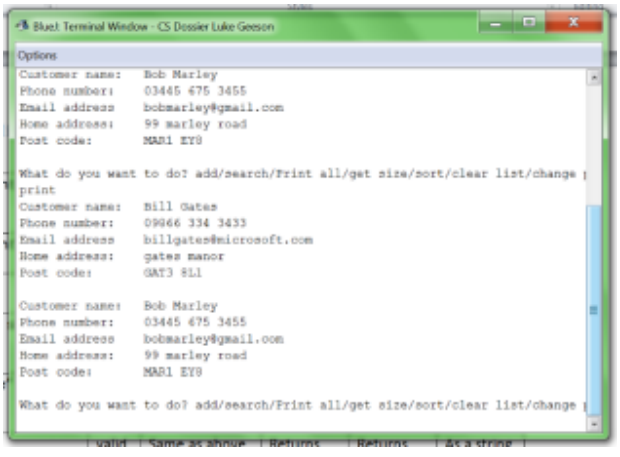


Figure 27: evidence of test 6: inputting "print" to print the list

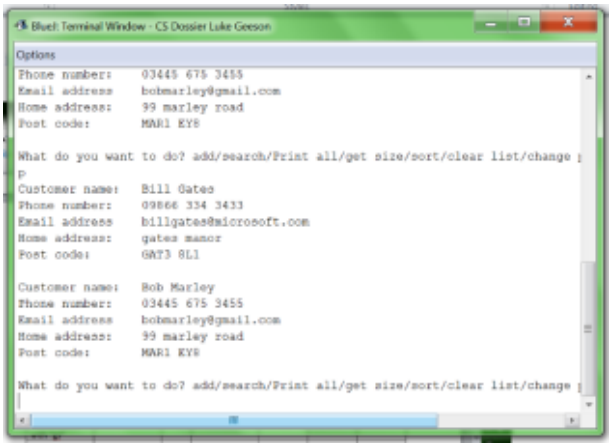


Figure 28: evidence of test 7: inputting "p" to print the list

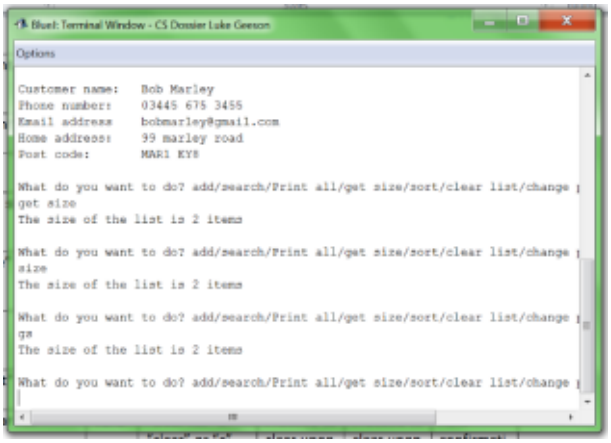


Figure 29: evidence of tests 8, 9 and 10: getting the size of the list by inputting "get size", "size" or "gs"

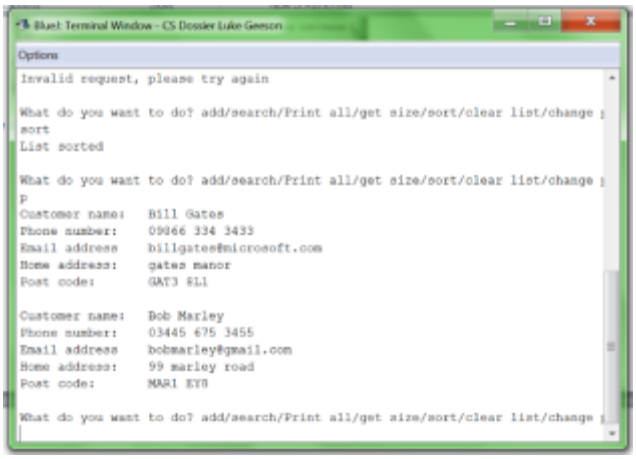


Figure 30: evidence of test 11: sorting the list, I have sorted and then printed the list for illustrative purposes

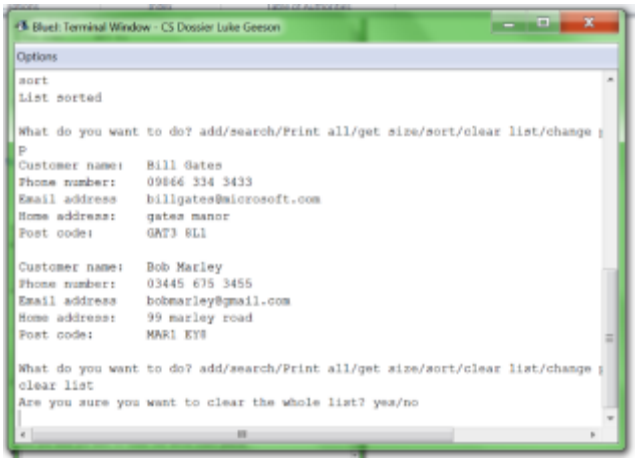


Figure 31: evidence of test 12: going to clear the list by inputting "clear list"

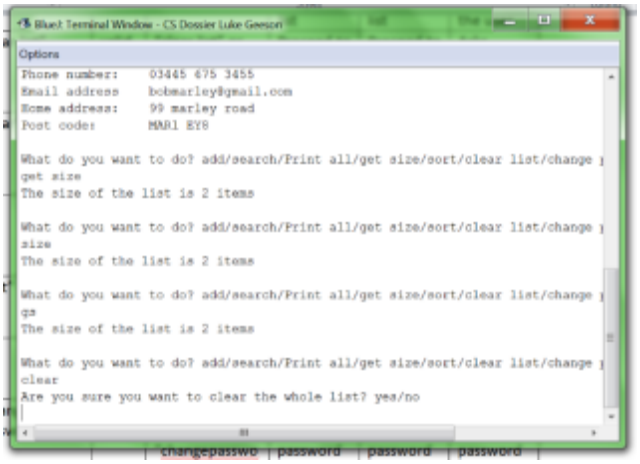


Figure 32: evidence of test 13: clearing the list by inputting "clear"

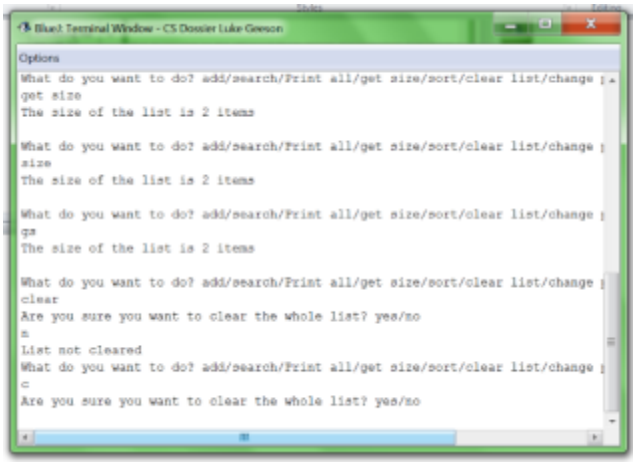


Figure 33: evidence of test 14: going to clear the list by inputting "c"

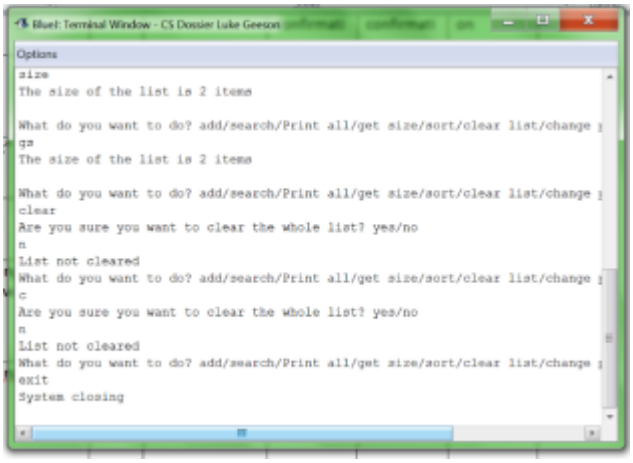


Figure 34: evidence of test 15: closing the program by inputting "exit"

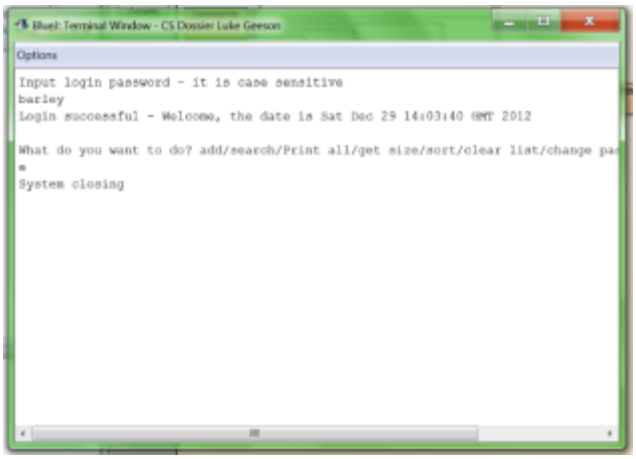


Figure 35: evidence of test 16: closing the program by inputting "e"

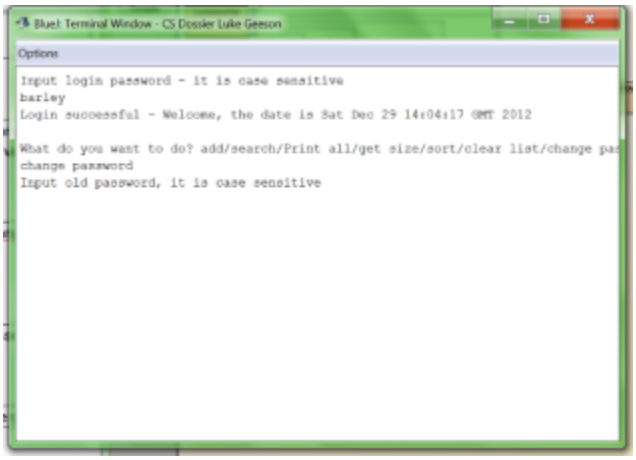


Figure 36: evidence of test 17: changing the password by inputting "change password"

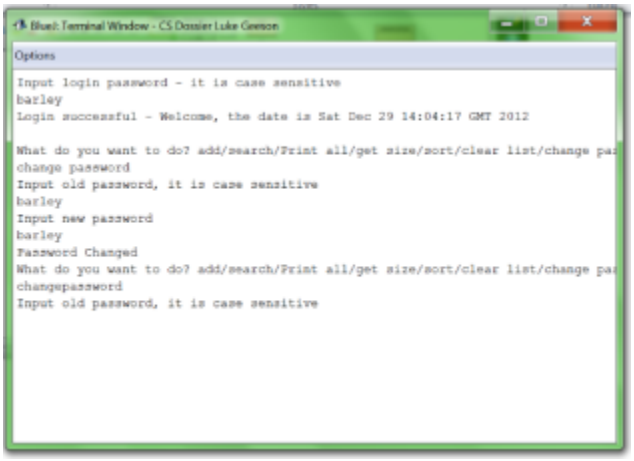


Figure 37: evidence of test 18: changing the password by inputting "changepassword"

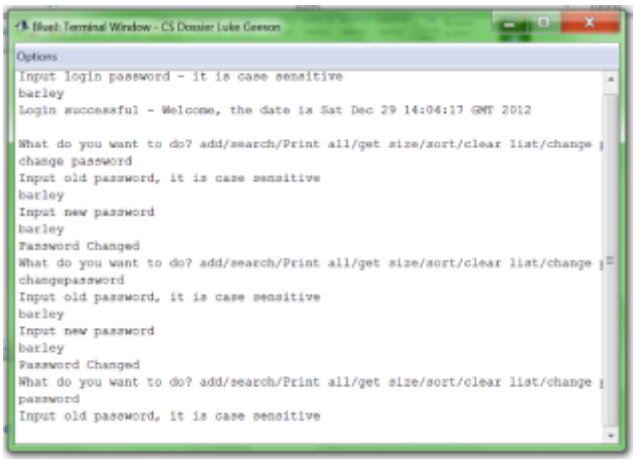


Figure 38: evidence of test 19: changing the password by inputting "password"

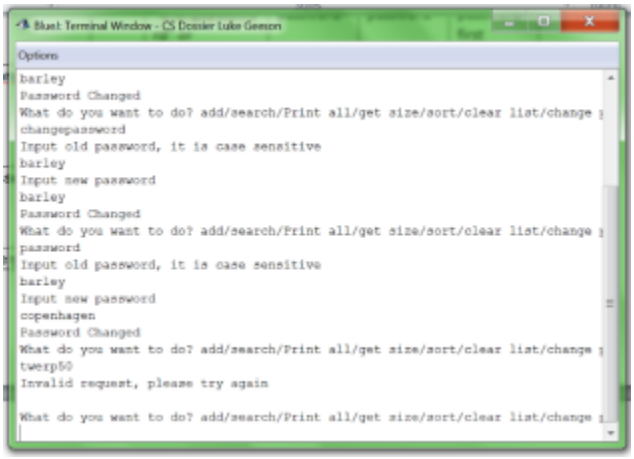


Figure 39: evidence of test 20: invalid input with “twerp50”

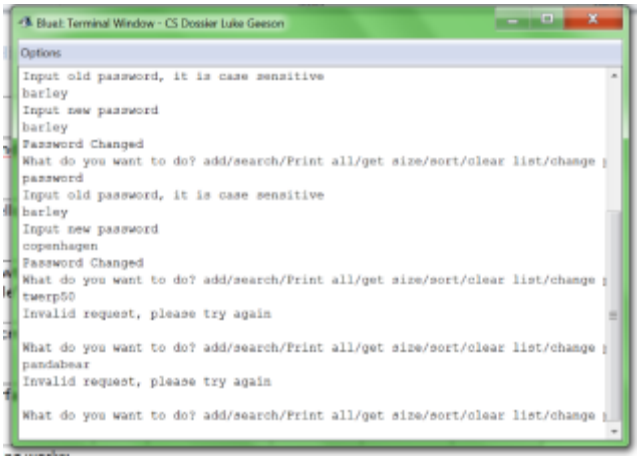


Figure 40: evidence of test 21: invalid input with "pandabear"

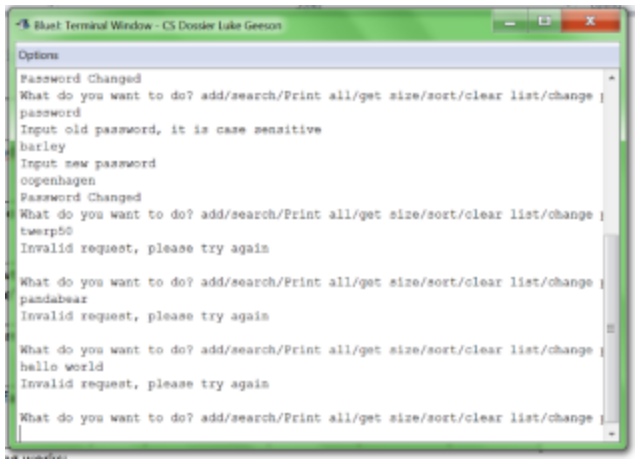


Figure 41: evidence of test 22: invalid input with “hello world”

D1: Adding an item to the list

This section shows the documentation and testing for the add function for this program. This shows partial satisfaction of requirement 1 of the specification as that requires the user to be able to “add” customer files, full satisfaction of requirement 2 as that requires the same fields of data input and partial satisfaction of requirement 5 as the customer file is sorted by surname as it is added and the list can otherwise be sorted by the sort list function (below)

From the following 2 images, you can see a screen shot of a customer file from the new system and the original copy of the customer file from the old. They both have the same fields of data input thus satisfying requirement 2:

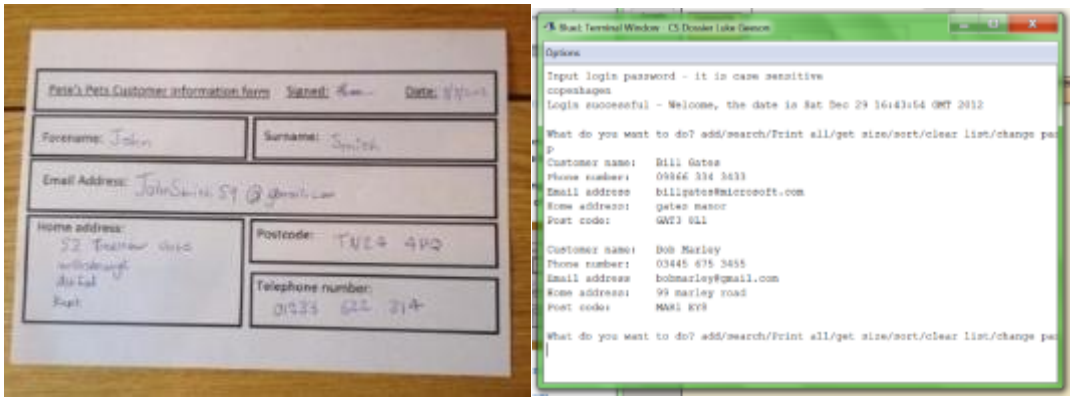


Figure 42: evidence that the new program has the same fields of input, both have forename, surname (forename and surname are stored as separate data items but appear as one on the terminal), email address, home address, post code and telephone number

From the following 2 images, you can see that the add function inserts the record in the correct position within the list itself – the first image shows the list before and the second image shows the list once a record is inserted. This shows partial satisfaction of requirement 5 as the list is sorted as each record is added and the list can only become unsorted when the surname is changed (as this is the key value by which the list is sorted) the sort function is called if the surname is changed on a record and so this is not a problem. See sorting the list for the justification for the full satisfaction of requirement 5.

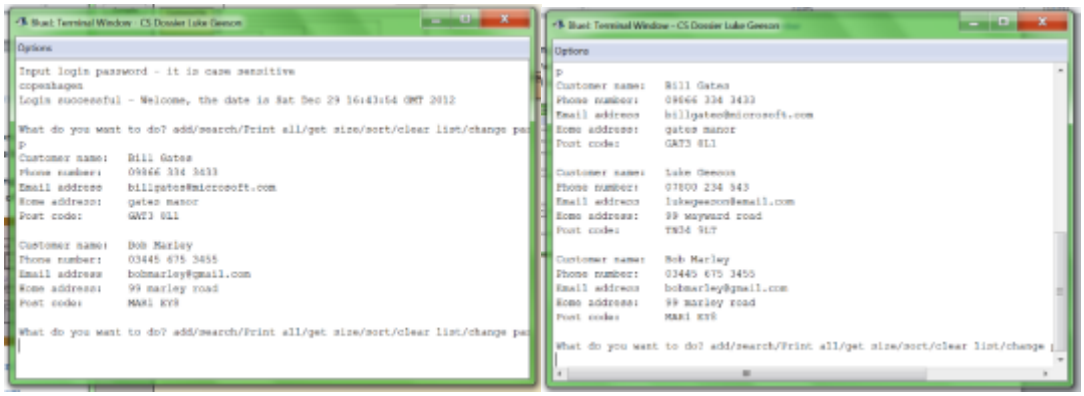


Figure 43: evidence that the add function correctly inserts data at the correct position in the list – 'Geeson' comes after 'Gates' and before 'Marley'

For the actual function of the list – there are multiple possibilities as to what may happen when a record is added. In the following table I have summarised the testing I have done in order to show that the add function works correctly. This table is followed by the evidence

Test for:	Data type	Expected result	Actual result	comments
1.Adding to an empty list	valid	Creates new list and sets record as head of list	Creates new list and sets record as head of list	
2.Adding to the head of the list	valid	Sets record as the head of the list	Sets record as the head of the list	

3.Adding to the tail of the list	valid	Sets record as last item in the list	Sets record as last item in the list	
4.Inserting within the list	valid	Inserts record at correct position in the list	Inserts record at correct position in the list	
5.Inserting after a record with an identical name	valid	Inserts record after item with identical name	Inserts record after item with identical name	
6.Invalid data insert	invalid	Sets either at head of list or tail of list (dependent on value of ASCII code)	Sets either at head of list or tail of list (dependent on value of ASCII code)	Invalid because it is not a proper surname (if it has numbers in it)

Evidence that the testing works:

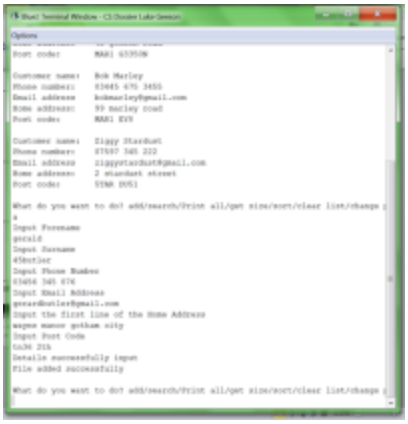


Figure 44: test evidence of the process of adding a record – this is what the user will see when they are inputting information – this data is also used in test 6 (that is why the surname had numbers in it – invalid input)

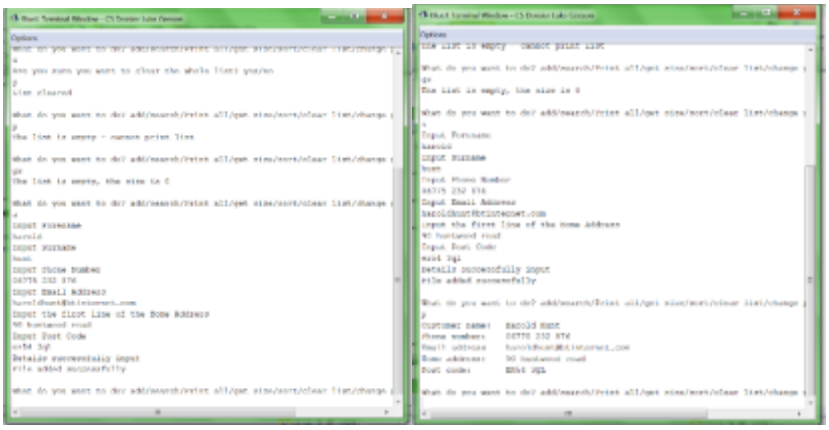


Figure 45: evidence of test 1: inserting into an empty list - the left shows the process of clearing a previous list (used in testing) and then inputting the new data to be stored in a node. This node is then added to the formerly empty list and set as the head – when the list is printed, you see that it is the only item in the list. Therefore this works.

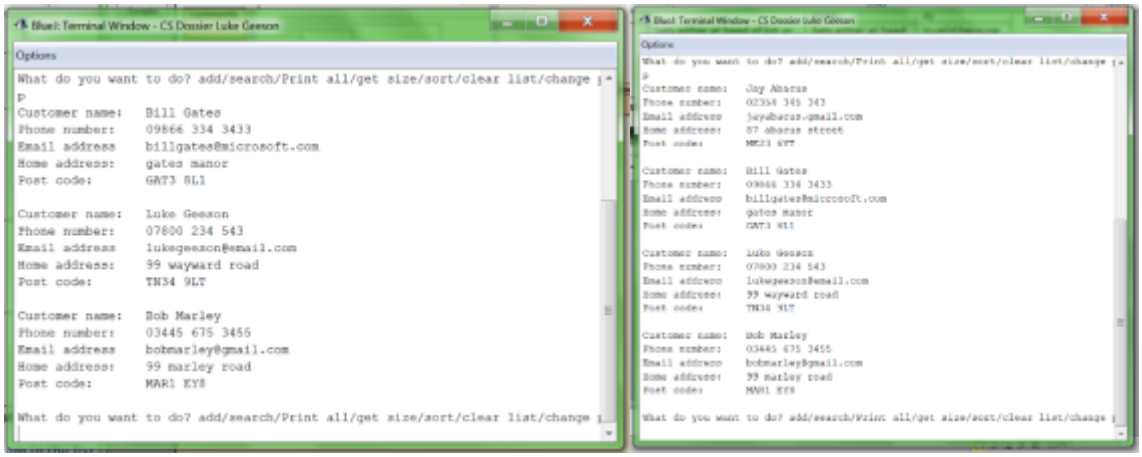


Figure 46: evidence that test 2 works: adding to the head of the list –these screen shots show the list before (left) and the list after (right). I have used the print function to print the lists for illustrative purposes and you can see that the record with the surname 'abacus' is placed at the head of the list as 'abacus' precedes 'gates'

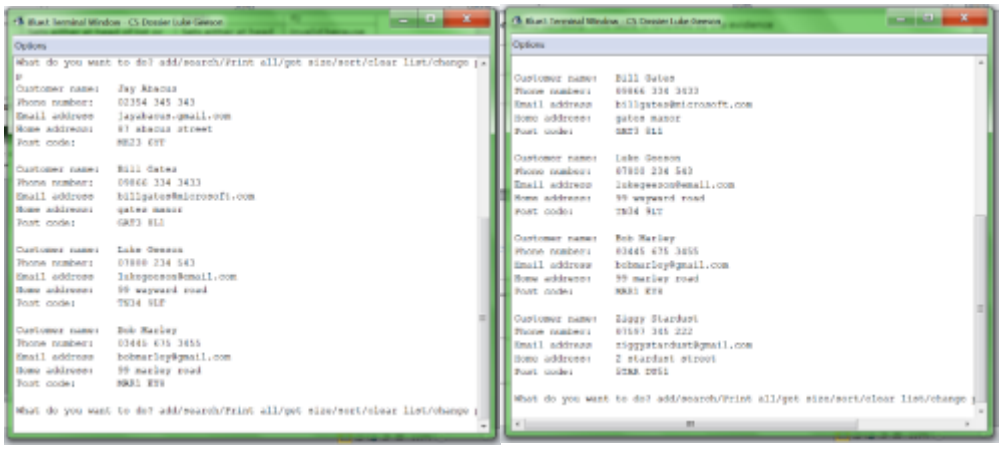


Figure 47: evidence that test 3 was a success: adding to the tail of the list. The old list (left) and the new list (right) have been displayed with the print function. After adding a record with the surname 'stardust' you can see that it has been placed at the end of the list as 'stardust' comes after 'marley.'

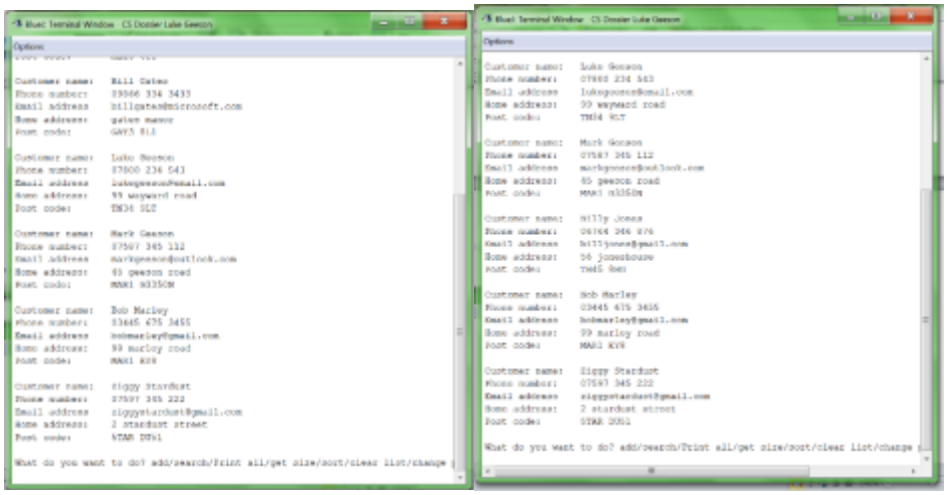


Figure 48: evidence that test 4 was a success: inserting an item within the list. The old list (left) and the new list (right) are displayed using the print method. An item with the surname 'Jones' has been added to the list and inserted into the correct position within the list (after Geeson and before Marley)

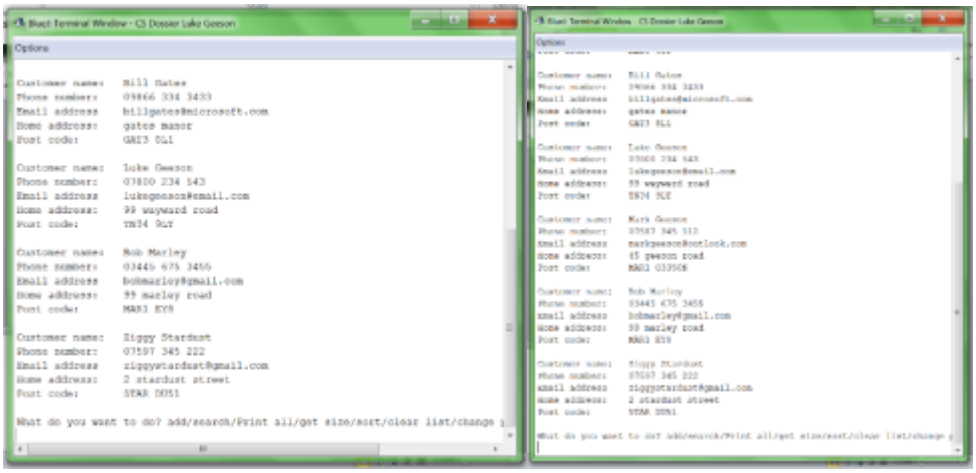


Figure 49: evidence of test 5 was a success: inserting a record with an identical name within the list. The old list (left) and the new list (right) have been printed with the print function. The new list contains 2 records with the surname, and the new record, with 'Mark Geeson' has been placed after the first

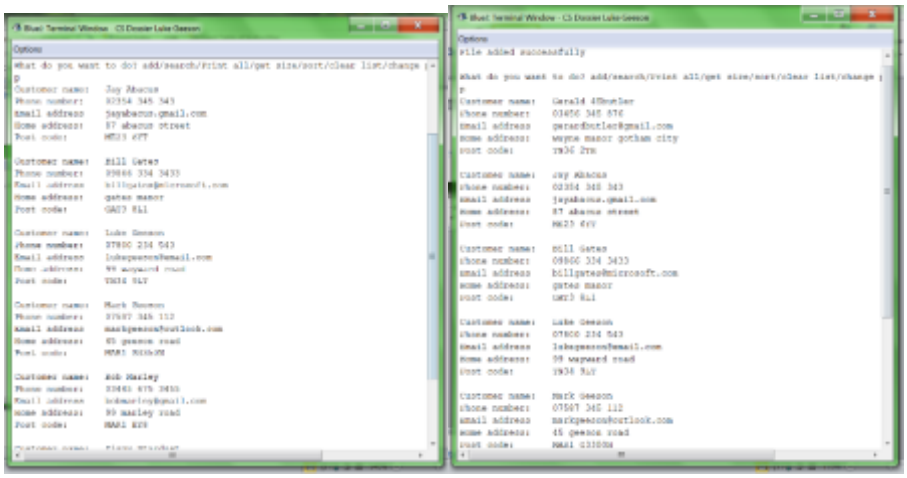


Figure 50: evidence of test 6: invalid data input. The old list (left) and the new list (right) were displayed using the print function. The invalid, yet accepted, with the surname '45butler' has been saved into the list at the head of the list. This is because the ASCII code of '45butler' is less than that of 'abacus' and therefore the invalid data has been stored there. Of course, this is evidence of the add record competently inserting the item at the head of the list – but a few more validation rules may have avoided this problem.

D1: Searching for an item in the list

In order to satisfy requirements 1 and 10 (especially 10) the user must be able to search for a record in the list by the surname of the customer, they should be able to do so without crashing the system, skipping the record they need or having to input the record surname twice if the search is unsuccessful the first time around. In order to test that this works, numerous tests were required to ensure this requirement is fully satisfied.

Test for	Input	Data type	Expected input	Expected result	Actual result	comments
1.Searching an empty list	"smith"	valid	Nothing – no surnames to find	Search failed – list is empty – should return to main menu	Search failed – list is empty and returns to main menu	Still requires surname to be input
2.Searching	"jones"	valid	The	Should return the	Record	Proceeds to

a 1 item list			surname in the list	found customer file if the surname matches that of the search input – should ask whether this file is the correct one (validation)	found, prints details from record and asks for confirmation	confirmation request
3.Searching a 1 item list where the surname sought after is not present	"marley"	valid	The surname in the list	Should display message – "search failed" and return to the main menu	Record not found as the search has failed to match a surname	Returns to main menu
4.Searching with a valid surname (list has at least 2 items)	"geeson"	valid	Any surname in the list	Should return the file if the surname matches that of the search input – should ask whether it is the correct record or not	Record found in the 2 item list - details are printed and confirmation required	Goes to confirmation request
5.Searching with an invalid surname	"pandaoc topus"	invalid	Any surname in the list	Should display message – search failed and return to the menu	Search failed – return to main menu	Returns to main menu
6.Searching with valid surname which is not in the list	"smith"	valid	Any surname in the list	Should display message – search failed and return to the main menu	Search failed – return to the main menu	Returns to the main menu
7.Searching with a valid surname where there are multiple records with the same surname	"Geeson"	valid	Any surname in the list	Should return the first instance of a record with the correct surname and ask whether this is the correct record (validation)	Returns first instance of record in the list with this surname, asks for confirmation	Goes to confirmation request
8.Confirming record found with "yes"	"yes"	valid	"yes" or "y"	Should hold record in memory for modification or removal	Accepts response – asks to 'remove modify or exit'	Goes to 'remove modify or exit request'
9.Confirming record found with "y"	"y"	valid	"yes" or "y"	Should hold record in memory for modification or removal	Accepts response – asks to 'remove modify or	Goes to 'remove modify or exit request'

					exit'	
10.rejecting record found with "no"	"no"	valid	"no" or "n"	Should continue search if there are items after the current or otherwise display message – search failed and return to the main menu	Accepts response and displays next item in the list with the same surname	Goes to next item in the list with the same surname
11.Rejecting record found with "n"	"n"	valid	"no" or "n"	Should continue search if there are items after the current or otherwise display message – "search failed" and return to the main menu	Accepts response – search fails and returns to main menu	No more items in the list with "geeson" as surname and so the search fails and returns to the main menu
12.Invalid input at the record confirmation stage	"asdf"	invalid	"yes" or "y" or "no" or "n"	Should display message - "invalid input" and loops until a correct command is input	Rejects input and asks user to input valid response	Asks for confirmation again

Evidence that the test works

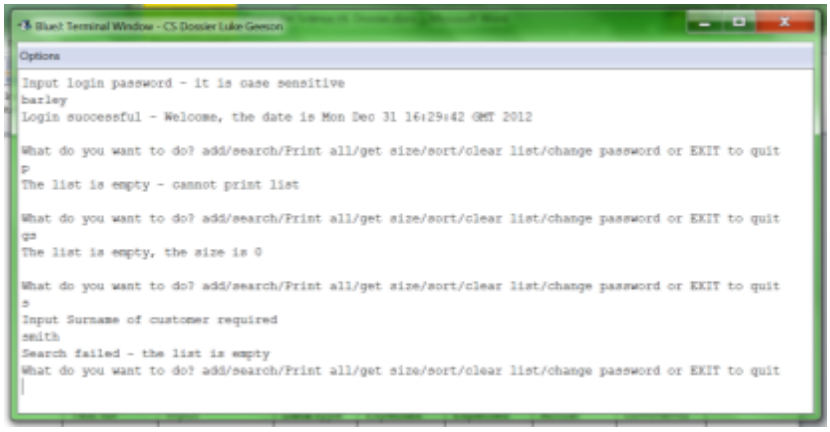


Figure 51: evidence of test 1: searching an empty list – there are no items in the list and so the search is not performed, you do however have to input a surname before the search is stopped and the program returns to the main menu

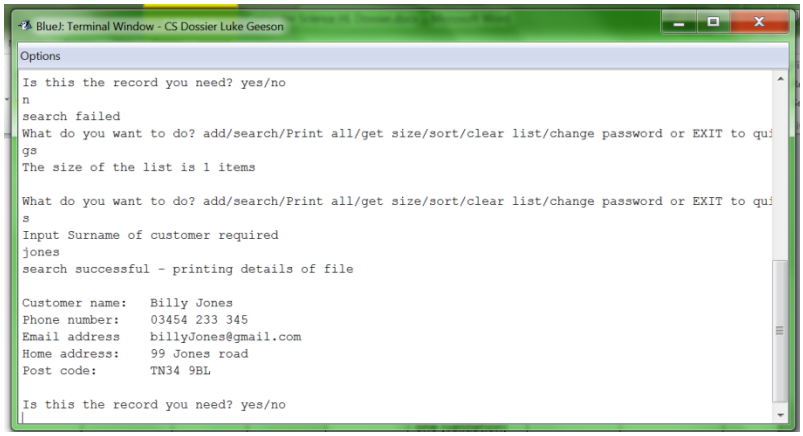


Figure 52: evidence of test 2: searching a 1 item list - as you can see, I have called the "gs" command to illustrate that it is a one item list and then I search for "jones" with the "s" command and the record found is returned

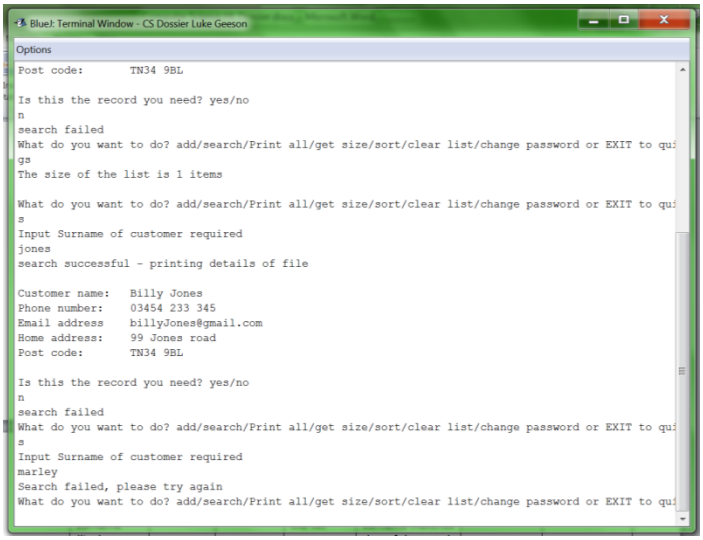


Figure 53: evidence of test 3 – as you can see I have searched for "marley" using the "s" command on the previous list which stated the search has failed as a result of the record not existing

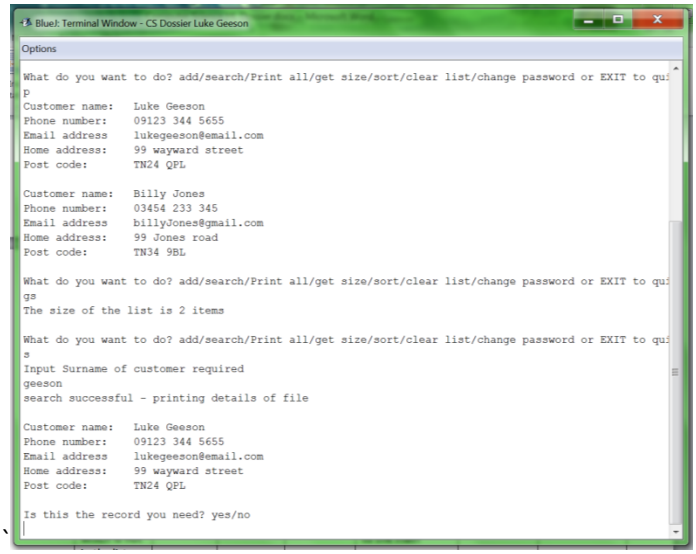


Figure 54: evidence of test 4 – as you can see I have printed the list to show the 2 items in the list and then called the "gs" command to show the list has 2 items, then I search for "geeson" with the "s" command and the correct record is found and the details printed

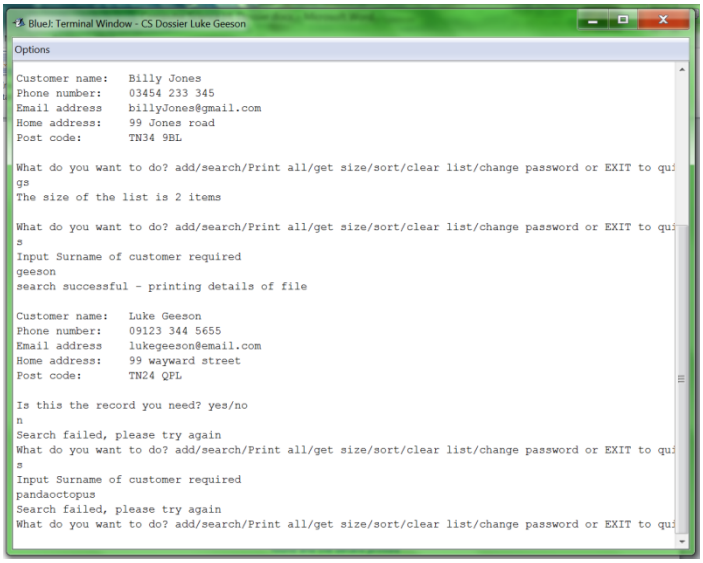


Figure 55: evidence of test 5 - I have used the list in the previous example and searched for "pandaoctopus" by calling the "s" function, this is an invalid name and no record has this surname stored so the search has failed

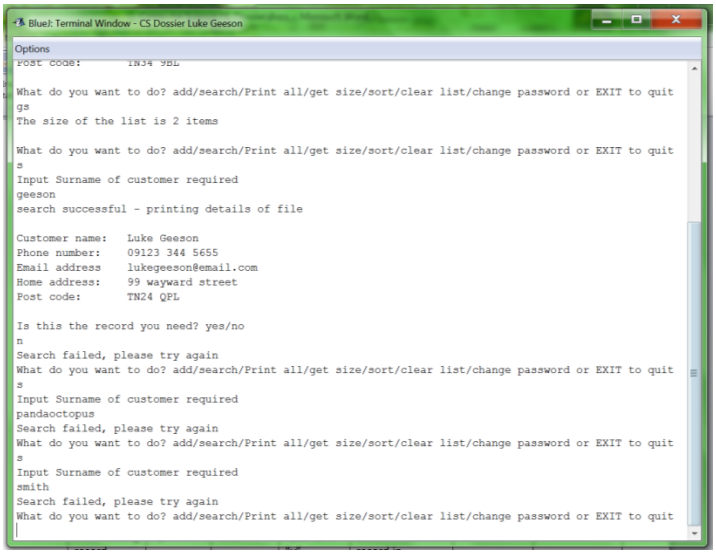


Figure 56: evidence of test 6 – I have used the example of the previous 2 examples whereby I search for a valid surname "smith" that is not present – the search fails and returns to the main menu

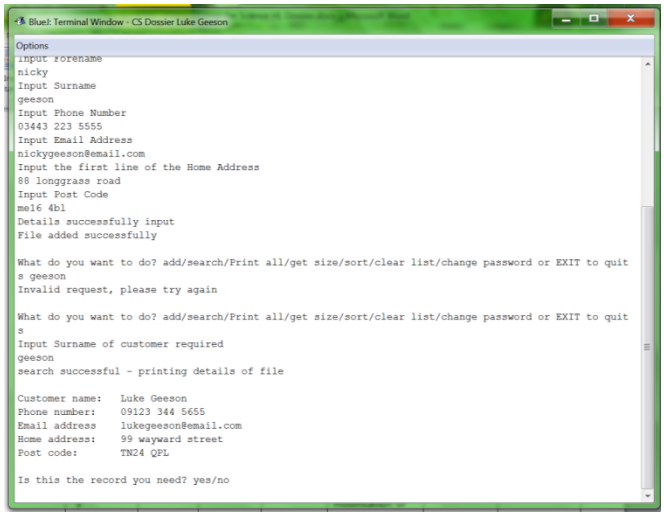


Figure 57: evidence of test 7 – as you can see I add the record with the name "nicky geeson" and then accidently search "s geeson" which is an invalid input for the main menu (reinforcing the tests done in the main menu section) then I search for geeson and the search returns the first record with the surname "geeson". As the system was programmed to store records with identical names in the order they are added, "luke geeson" is the first record and so it is the first to be found in the search

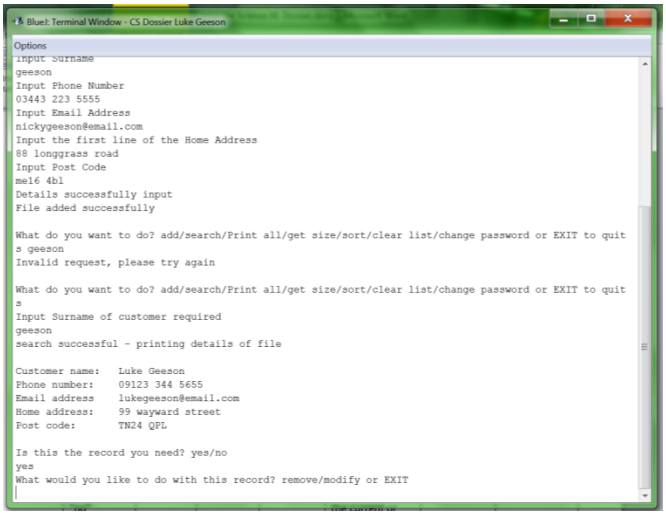


Figure 58: evidence of test 8, using the previous example, I searched for records containing "geeson" as the surname and confirmed it was the record with "yes" it then proceeds to ask me what I want to do with it

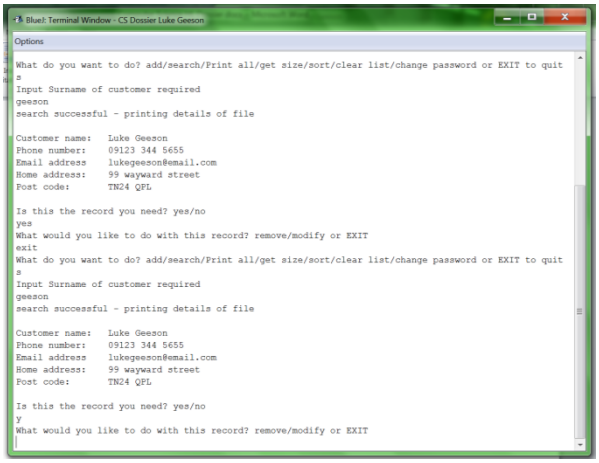


Figure 59: evidence of test 9 – using the previous example I have cancelled the previous search and re-searched for the same record in order to show that confirmation on records can be either "yes" or "y"

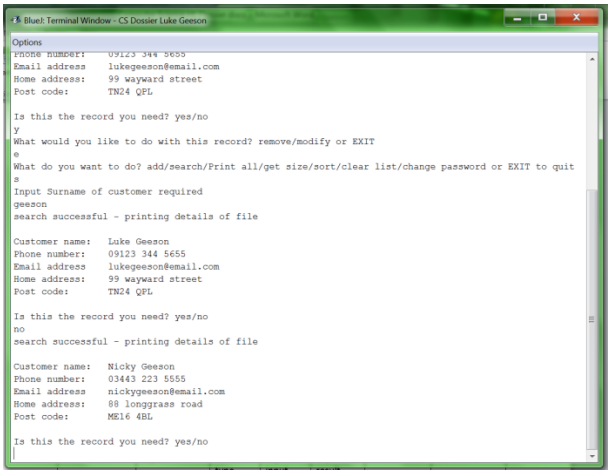


Figure 60: evidence of test 10 – using the previous example, I have once again cancelled the search and re-searched the for the same record whereby I cancel the confirmation with "no" and the list finds and returns the next item in the list.

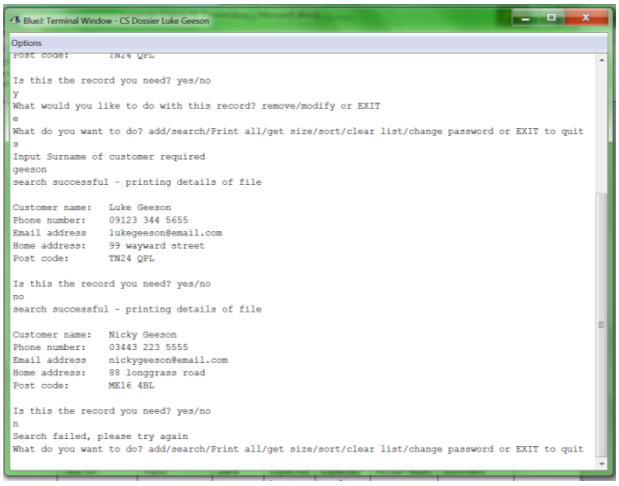


Figure 61: evidence of test 11 – using the previous example, I have declined confirmation on the second record with "n" and as there are only 2 items in the list with "geeson" the search fails and returns to the main menu

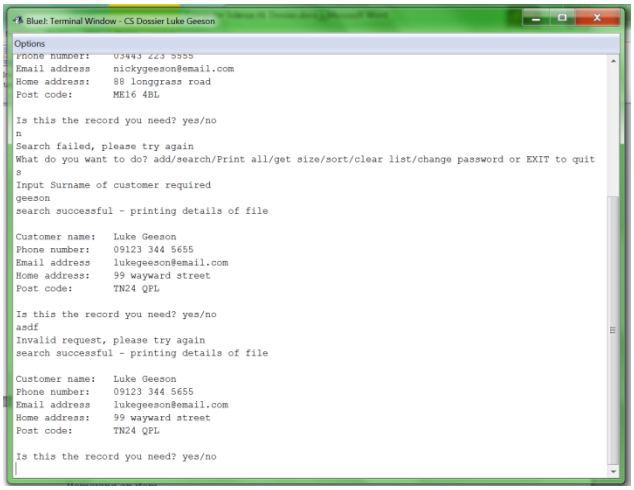


Figure 62: evidence of test 12 – using the previous example, I have searched for "geeson" and input invalid input to show how the system deals with it – the system recognises the invalid input and asks for the user to input a valid response

After a record has been found with the search function, the user has 3 options: they can 'remove', 'modify' or 'exit' back to the main menu. The testing for these input options (note this is not the testing for the function of these methods but rather the user interface that allows the user to access these functions - see the next 2 sections for testing of those functions)

Test for:	Input	Data type	Expected input	Expected result	Actual result	comment
1.Removing the record with "remove"	"remove"	Valid	"remove" or "r"	Accepts input – asks for confirmation of remove	Accepts input - Asks for remove confirmation	See next section for this
2.Removing the record with "r"	"r"	Valid	"remove" or "r"	Accepts input – asks for confirmation of remove	Accepts input - Asks for remove confirmation	See next section for this

3.Modifying the record with "modify"	"modify"	Valid	"modify" or "m" or "change" or "c"	Accepts input – asks what data is to be changed	Accepts input – asks what data is to be changed	See D1: Modifying an item in the list
4.Modifying the record with "m"	"m"	Valid	"modify" or "m" or "change" or "c"	Accepts input – asks what data is to be changed	Accepts input – asks what data is to be changed	See Modifying an item in the list
5.Modifying the record with "change"	"change"	Valid	"change" or "c" or "modify" or "m"	Accepts input – asks what data is to be changed	Accepts input – asks what data is to be changed	See Modifying an item in the list
6.Modifying the record with "c"	"c"	Valid	"change" or "c" or "modify" or "m"	Accepts input – asks what data is to be changed	Accepts input – asks what data is to be changed	See Modifying an item in the list
7.Exiting to the main menu with "exit"	"exit"	Valid	"exit" or "e"	Accepts input – exits to main menu	Accepts input – returns to the main menu	Returns to the main menu
8.Exiting to the main menu with "e"	"e"	valid	"exit" or "e"	Accepts input – exits to main menu	Accepts input – returns to the main menu	Returns to the main menu
9.Handling of invalid input with "qwertyuiop"	"qwertyuiop"	invalid	Any of the above	Rejects input – requests valid input – reiterates	Rejects input – requests valid data - reiterates	Reiterates

Evidence that this testing works:

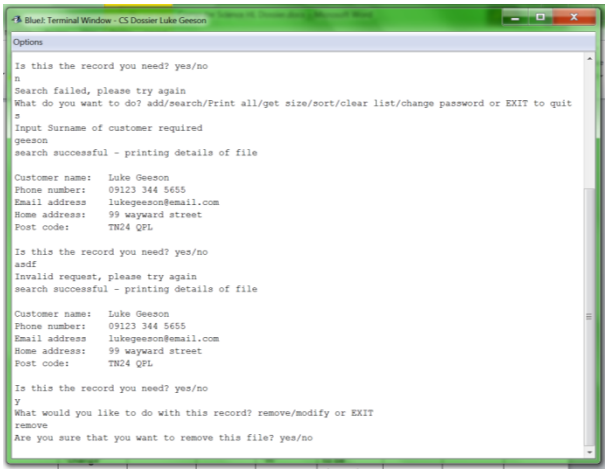


Figure 63: evidence of test 1 - I first search for "geeson", there are 2 items, first I test the invalid input of "asdf" for the previous test and then I decide to remove the record by inputting "remove" which brings me to the remove function confirmation (see next section)

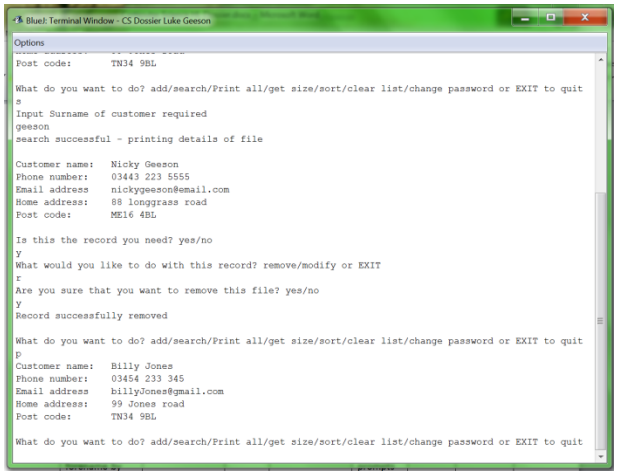


Figure 64: evidence of test 2 – first I search for the record with surname "geeson", then I remove it with "r" and print the list – which does not have the record I have just removed.

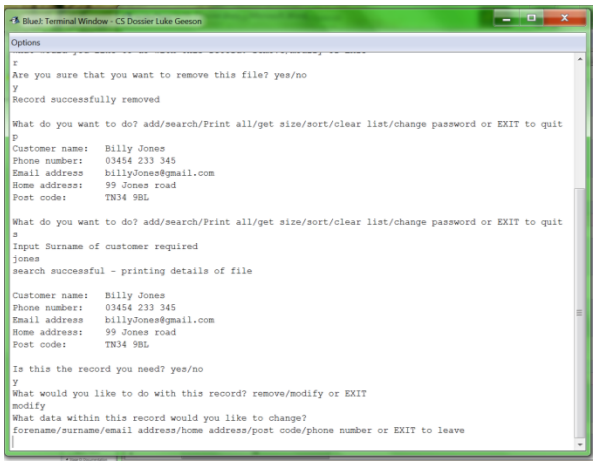


Figure 65: evidence of test 3 – following the previous example, I then decide to modify the last item in the list, first I search for the surname "jones" and then I choose the modify it with "modify", then the system asks me what data I wish to modify

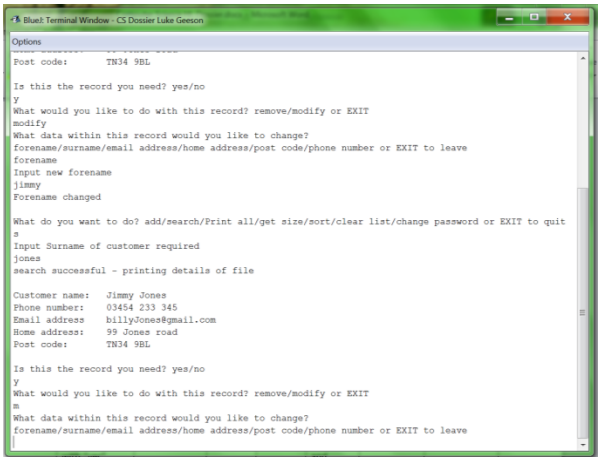


Figure 66: evidence of test 4 – following the previous example – whereby I searched for "jones" and then changed the surname, I then searched for the same record in order to modify it with "m" which then gave me the same options as before

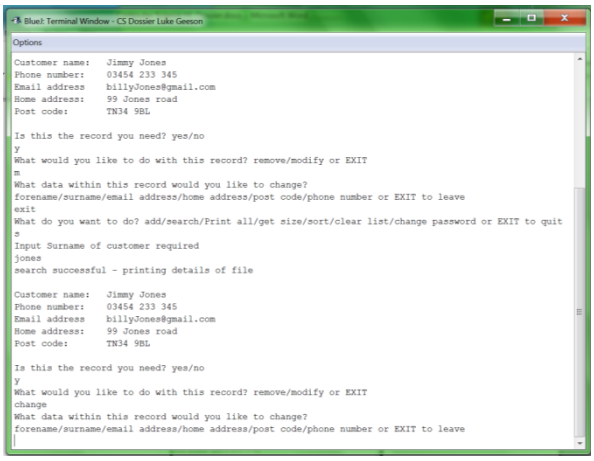


Figure 67: evidence of test 5 - following the previous example- I modified the same record again by searching for it and then inputting "change" whereby the system will detect that this means the same thing as inputting "modify" or "m" and will give the same options as a result

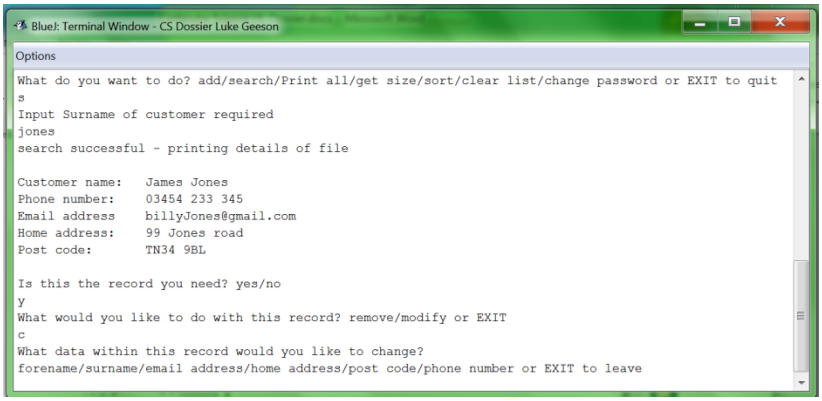


Figure 68: evidence for test 6 - following the previous example- I modified the same record again by searching for it and then inputting "c" whereby the system will detect that this means the same thing as inputting "modify" or "m" and will give the same options as a result

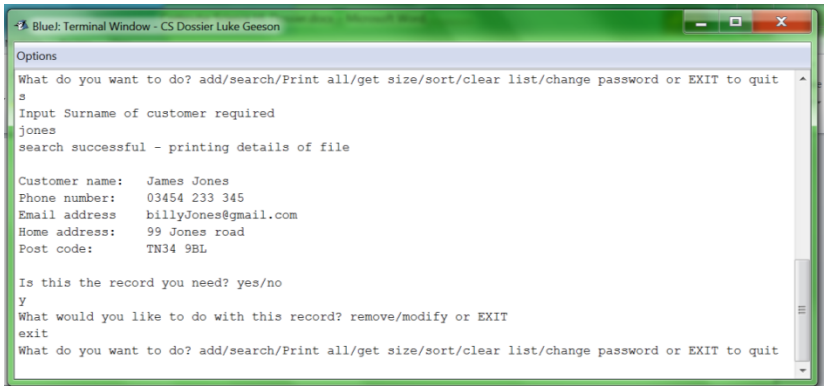


Figure 69: evidence for test 7 – I searched for the same record again and then returned to the main menu with "exit"

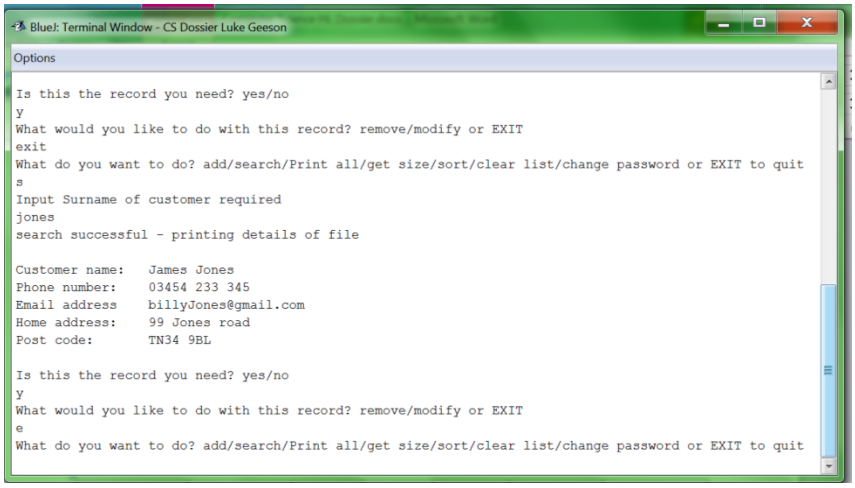


Figure 70: evidence of test 8 – after the previous example I searched for the same record and then returned to the main menu with "e"

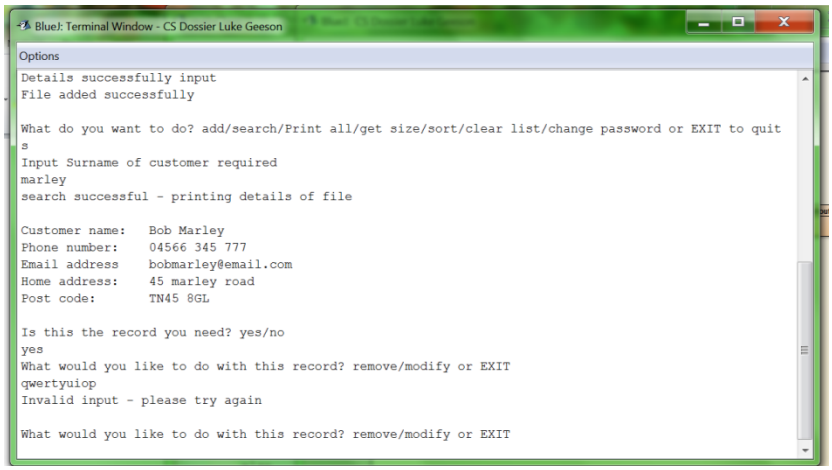


Figure 71: evidence of test 9 – I searched for a recently added file with the surname "marley" and then tested for invalid input by inputting "qwertyuiop" whereby the system rejected the input and reiterated to ask for the input again

D1: Removing an item

This function is part of the system that satisfies requirement 1; an item can only be removed once a record has been found using the search function.

Test for:	Input	Data type	Expected input	Expected result	Actual result	comment
1.Confirming you want to remove the file with “yes”	“yes”	Valid	“yes” or “y”	Should remove the file and return the main menu	Input accepted – file is removed	Returns to main menu
2.Confirming you want to remove the file with “y”	“y”	Valid	“yes” or “y”	Should remove the file and return to the main menu	Input accepted – file is removed	Returns to the main menu
3.Confirming you do not want to remove the file with “no”	“no”	Valid	“no” or “n”	Does not remove the file and returns to the main menu	Input accepted – file not removed – returns to main menu	Returns to main menu
4.Confirming you do not want to remove the file with “n”	“n”	Valid	“n”	Does not remove the file and returns to the main menu	Input accepted – file not removed – returns to main menu	Returns to main menu
5.Invalid data input when attempting to confirm whether to remove the file	“hats”	Invalid	“yes” or “y” or “no” or “n”	Asks user to re-enter command	Input rejected – file not removed – iterates again for valid input	Requests user to input valid input
6.Removing an item at the head of the list	n/a	Valid	n/a	Removes item from the head of the list	Head of list is removed – return to main menu	Returns to main menu
7.Removing an item at the tail of the list	n/a	Valid	n/a	Removes an item from the tail of the list	Tail item removed – return to main menu	Returns to main menu
8.Removing an item in between the head and end of the list	n/a	Valid	n/a	Should remove an item from within	Correct record removed – return to the main	Returns to the main menu

				the list	menu	

Evidence that the testing works:

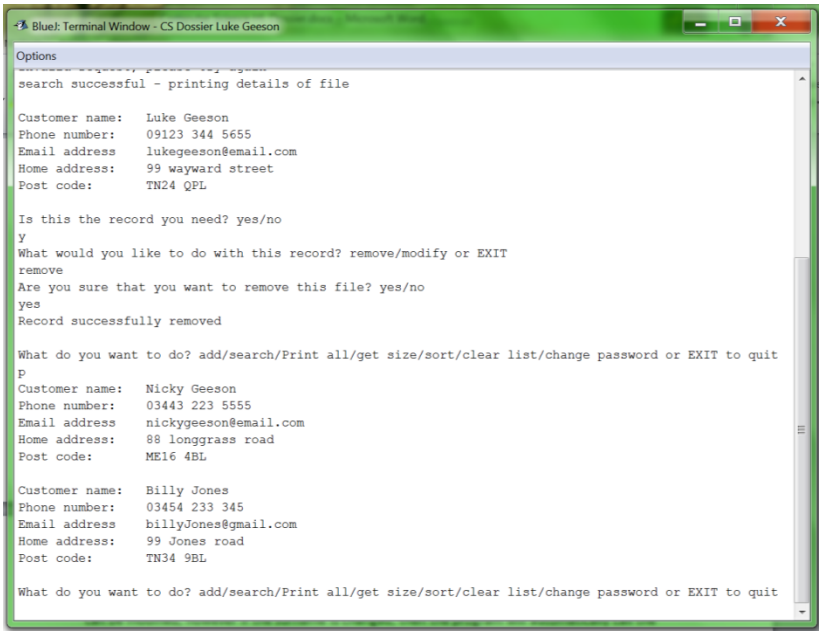


Figure 72: evidence of test 1 - as you can see, I have found a file and have chosen to remove it with "yes" then I call the print function in order to show the new list with the old record removed

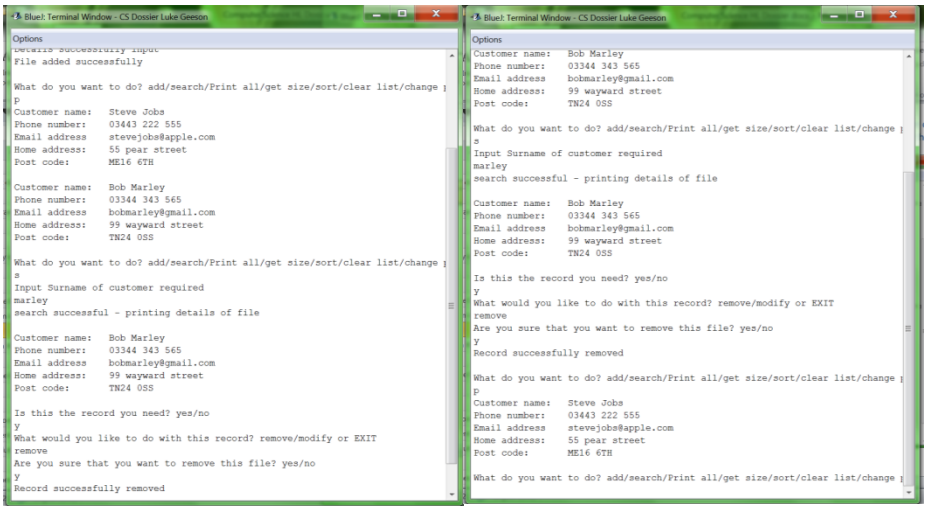


Figure 73: evidence of test 2 – after adding 2 items to an empty list, I then print the list to show the list contents (left) I then find and remove the record with the surname "marley", I confirm that I want to remove it with "y" and the file is removed. The new contents of the list are then displayed when I call the print function once again (right)

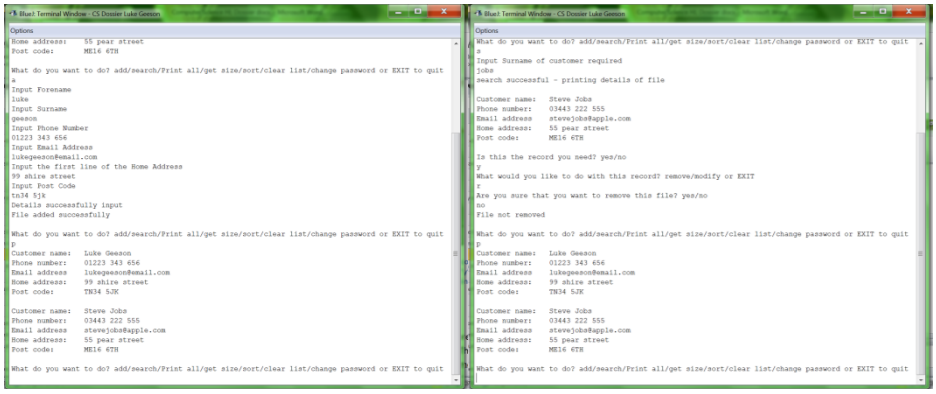


Figure 74: evidence of test 3 – following the previous example I added a file with the surname "geeson" (left) and then searched for the record with the surname "jobs". The record was found and then at the stage of choosing to remove the record, I input "no" and the record was not removed and the contents of the list were not changed (as illustrated by the use of the print function to show the list is no different before or after the input)

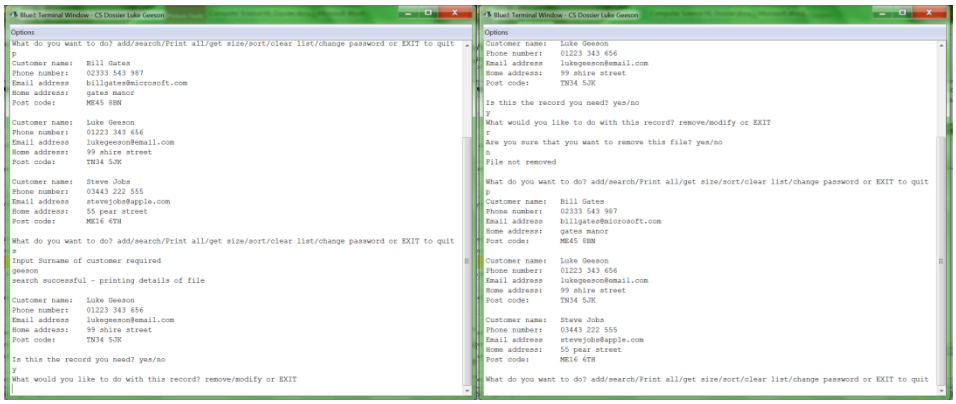


Figure 75: evidence of test 4 – following the previous example I added a record with the surname "gates" to the list and printed the list (left) then I searched for the record with the surname "geeson" and when given the choice whether or not to remove it, I input "n" and the record was not removed nor was the state of the list changed (illustrated by the use of the print command on the right).

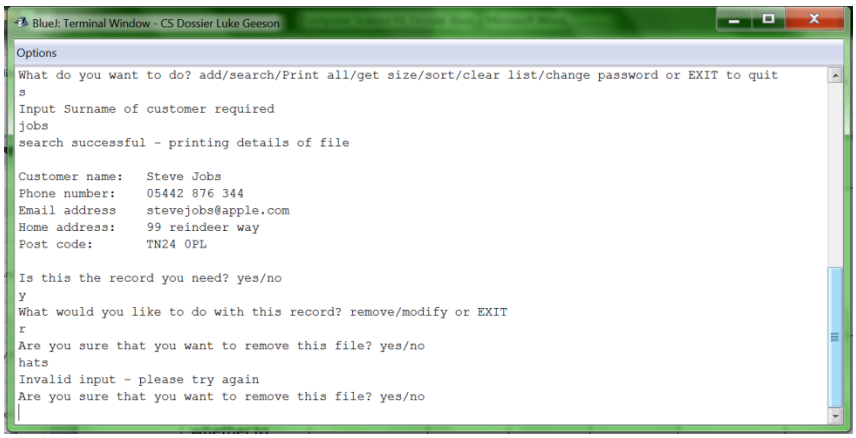


Figure 76: evidence that test 5 works – after creating a new list I searched for a record with the surname "jobs" then when I was given the choice as to remove it or not, I input "hats", as this is invalid input the system dealt with it appropriately and the system reiterated the loop so that I could input a valid command

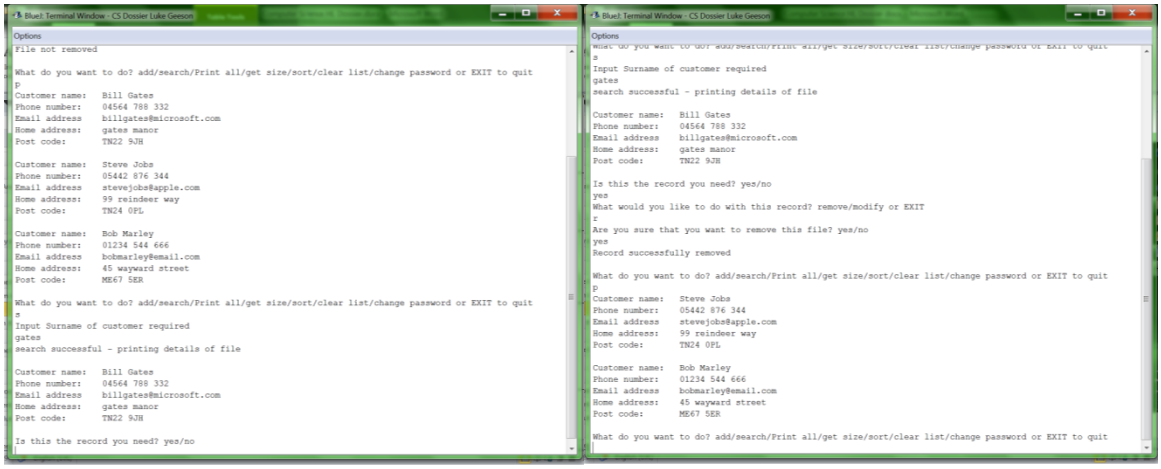


Figure 77: evidence that test 6 works – the list before removal is presented on the left. I search and find the record with the surname "gates" and proceed to remove it. Gates was the first item in the list and after removal it is not – the use of the print function before and after removal show the state of the list before and after the removal

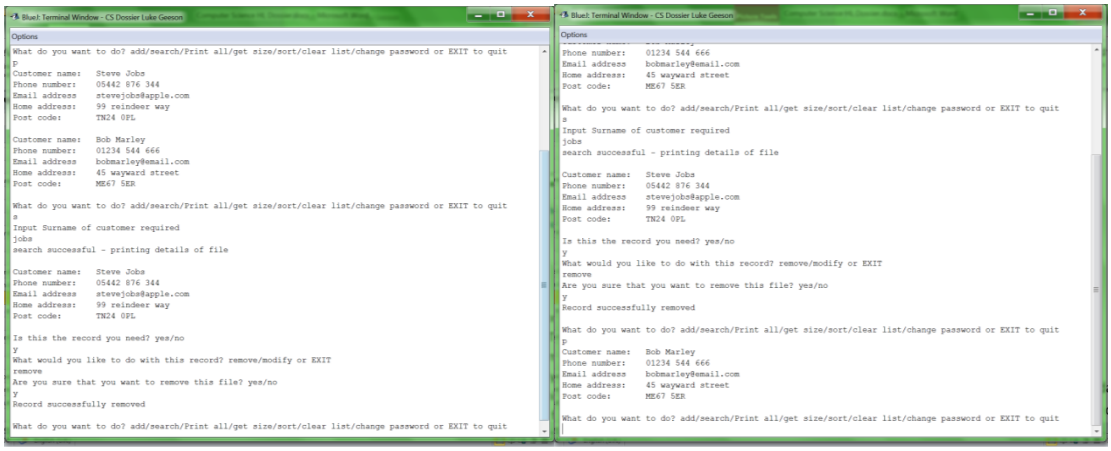


Figure 78: evidence of test 7 – the first image shows the items in the list (displayed with the print function) and the process of removing the last record in the list (the record with the surname "jobs") whereby it is successfully removed and the second slide shows the new state of the list (there were 3 items in the previous example, but since then we have removed 2 so now there is just the one)

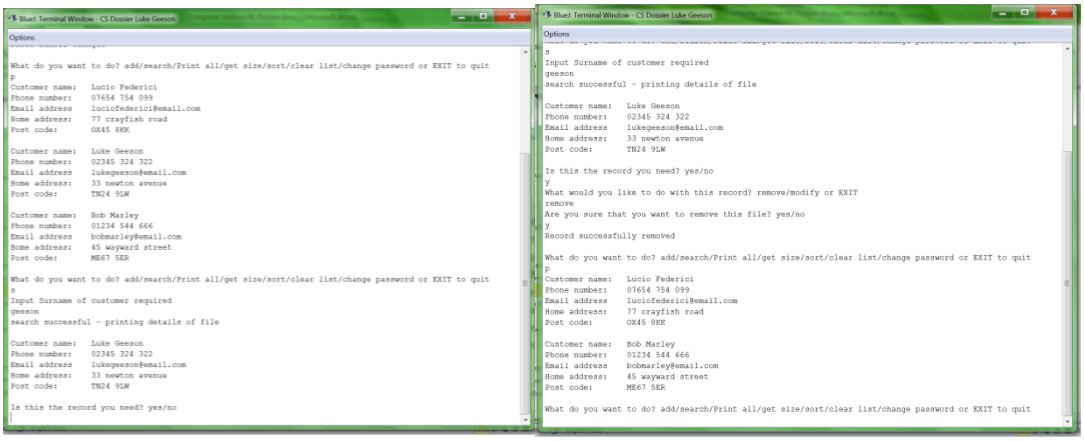


Figure 79: evidence of test 8 – using the previous example, I added 2 new records with surnames "geeson" and "federici" (see left image where the new list is displayed with the print function) whereby "geeson" is the item between the head and the end of the list. I search for "geeson" with the search function and choose to remove it; it is successfully removed and the new list is displayed with the print function (right)

D1: Modifying an item in the list

This section shows the testing for the part of the program which modifies the data of a pre-existing file in the list itself; the file must first be found using the search function before the user can make these changes. This function of the list contributes to the fulfilment of requirement 1. Any field of the data of a customer record can be modified, however if the surname is changed then the program will automatically call the sort routine in order to resort the list- this is because the surname is the key value by which the list is sorted. This is only called for this condition because the modification of any of the other fields will not make a difference to the order of the list. The following table summarises the testing performed to ensure that this function of the list works correctly so that it may contribute to the fulfilment of requirement 1.

Test for	Data input	Data type	Expected input	Expected result	Actual result	Comment
1. Choosing to change the forename by inputting "forename"	"forename"	Valid	"forename"	Accepts input and prompts user to input new data	Accepts input – requests new forename input	
2.Choosing to change the forename by inputting "first name"	"first name"	valid	"first name"	Accepts input and prompts user to input new data	Accepts input – requests new forename input	
3.Choosing to change the first name by inputting "firstname"	"firstname"	Valid	"firstname"	Accepts input and prompts user to input new data	Accepts input – requests new forename input	
4.Choosing to change the input by inputting "first"	"first"	Valid	"first"	Accepts input and prompts user to input new data	Accepts input – requests new forename input	
5.Choosing to change the firstname by inputting "f"	"f"	Valid	"f"	Accepts input and prompts user to input new data	Accepts input – requests new forename input	
6.Choosing to change the surname by inputting "surname"	"surname"	Valid	"surname"	Accepts input and prompts user to input new data	Accepts input – requests new surname input	
7.Choosing to	"second name"	Valid	"second name"	Accepts	Accepts	

change the surname by inputting "second name"				input and prompts user to input new data	input – requests new surname input	
8.Choosing to change the surname by inputting "secondname"	"secondname"	valid	"secondname"	Accepts input and prompts user to input new data	Accepts input – requests new surname input	
9.Choosing to change the surname by inputting "s"	"s"	Valid	"s"	Accepts input and prompts user to input new data	Accepts input – requests new surname input	
10.Choosing the phone number by inputting "phonenumber"	"phonenumber"	Valid	"phonenumber"	Accepts input and prompts user to input new data	Accepts input – requests new phone number input	
11.Changing the phone number by inputting the phone number	"phone number"	Valid	"phone number"	Accepts input and prompts user to input new data	Accepts input – requests new phone number input	
12.Changing the phone number by inputting "p"	"p"	Valid	"p"	Accepts input and prompts user to input new data	Accepts input – requests new phone number input	
13.Changing the email address with "email address"	"email address"	Valid	"email address"	Accepts input and prompts user to input new data	Accepts input – requests new email address input	
14.Changing the email address with "emailaddress"	"emailaddress"	valid	"emailaddress"	Accepts input and prompts user to input new data	Accepts input – requests new email address	

					input	
15.Changing the email address with “email”	“email”	Valid	“email”	Accepts input and prompts user to input new data	Accepts input – requests new email address input	
16.Changing the email address with “e”	“e”	valid	“e”	Accepts input and prompts user to input new data	Accepts input – requests new email address input	
17.Changing the home address with “homeaddress”	“homeaddress”	Valid	“homeaddress”	Accepts input and prompts user to input new data	Accepts input – requests new home address input	
18.Changing the home address with “home address”	“home address”	Valid	“home address”	Accepts input and prompts user to input new data	Accepts input – requests new home address input	
19.Changing the home address with “home”	“home”	valid	“home”	Accepts input and prompts user to input new data	Accepts input – requests new home address input	
20.Changing the home address with “h”	“h”	Valid	“h”	Accepts input and prompts user to input new data	Accepts input – requests new home address input	
21.Changing the post code with “postcode”	“postcode”	valid	“postcode”	Accepts input and prompts user to input new data	Accepts input – requests new post code input	
22.Changing the post code with	“post code”	Valid	“post code”	Accepts input and	Accepts input –	

"post code"				prompts user to input new data	requests new post code input	
23.Stopping the current command with "exit"	"exit"	Valid	"exit"	Accepts input and returns to main menu	Accepts input – returns to main menu	Returns to main menu
24.Invalid data input with "the surname"	"the surname"	invalid	"the surname"	Rejects input and asks user to retry input	Rejects input and requests retry – reiterates	
25.Invalid data input with "the post code"	"the post code"	invalid	"the post code"	Rejects input and asks user to retry input	Rejects input and requests retry – reiterates	

Evidence that the testing works:

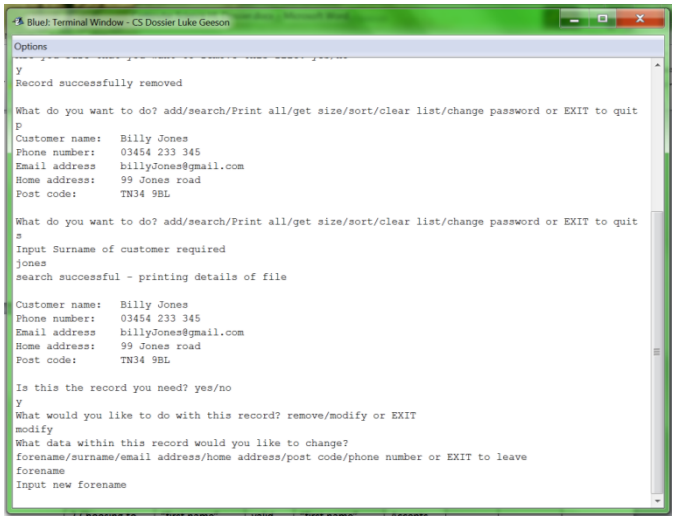


Figure 80: evidence that test 1 works: inputting "forename" to change the forename

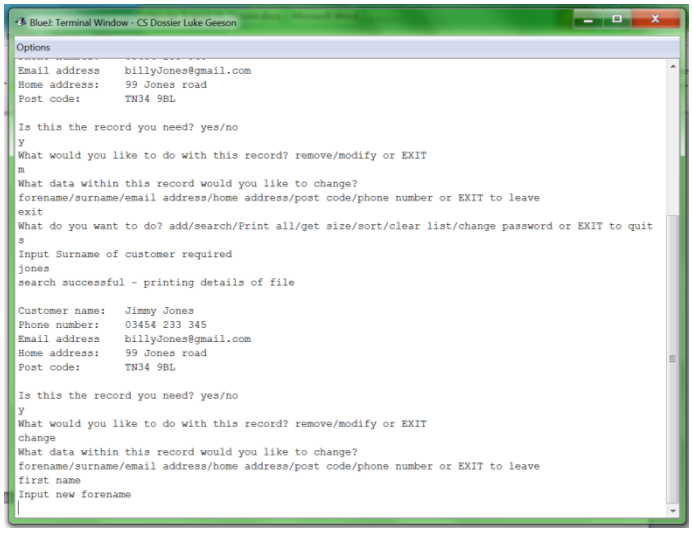


Figure 81: evidence that test 2 works: inputting "first name" to change the forename

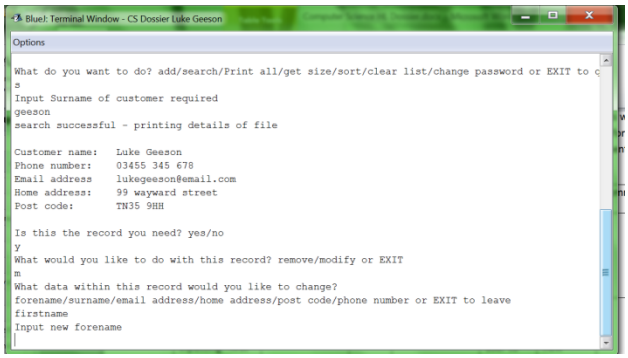


Figure 82: evidence that test 3 works – changing the forename by inputting "firstname"

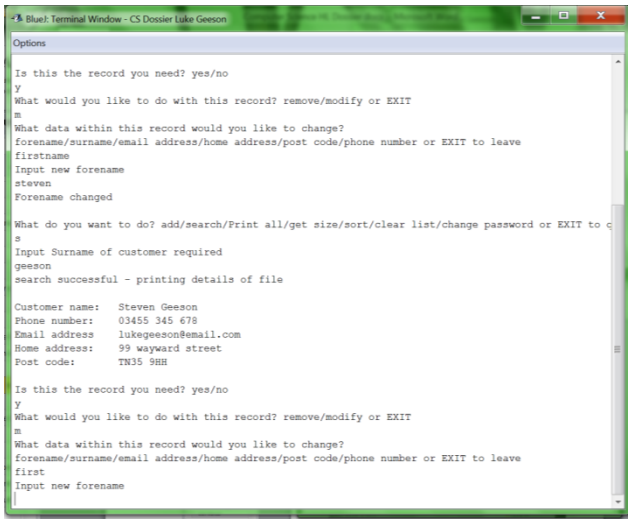


Figure 83: evidence of test 4 – following the previous example, I where I had changed the forename to "steven", I then searched for the same record and this time I chose to change the forename by inputting "first"

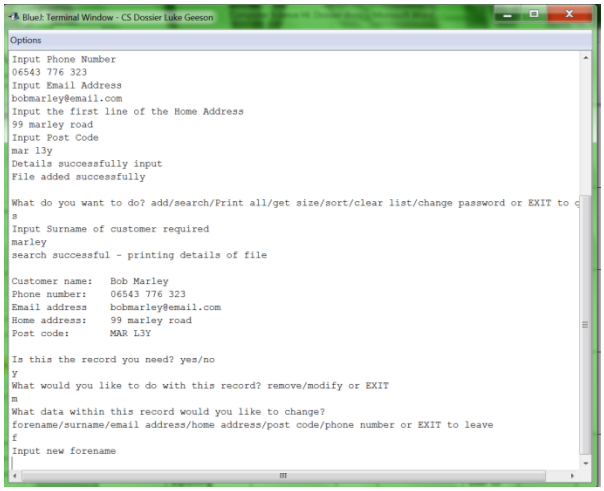


Figure 84: evidence of test 5 – following the previous example, I added another record with a surname "marley". I then searched for it and chose to change the forename by inputting "f"

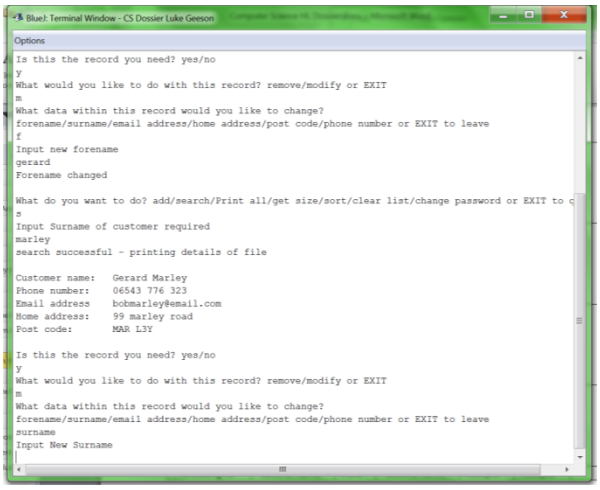


Figure 85: evidence of test 6 – following the previous example, I changed the forename to "gerard" and then searched for the same record where this time I chose to change the surname by inputting "surname"

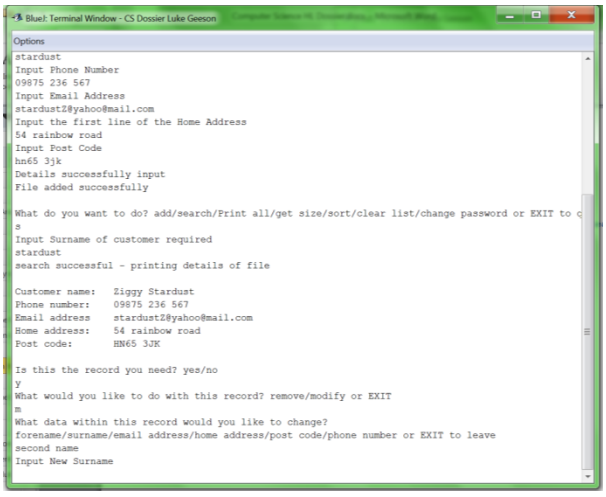


Figure 86: evidence of test 7 – following the previous example, I added "ziggy stardust" to the list and then searched for this record whereby I changed the surname by inputting "second name"

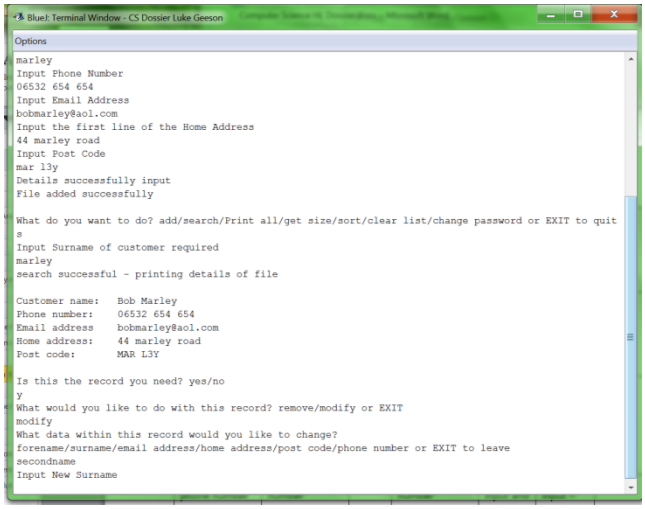


Figure 87: evidence of test 8 – I search for the record with the surname "marley" and then change the surname by inputting "secondname"

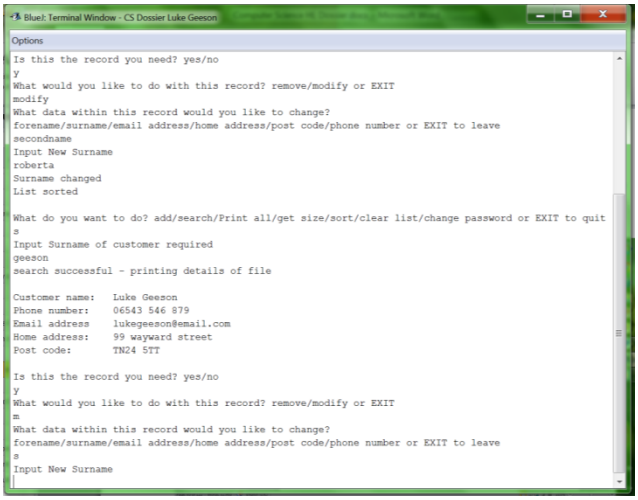


Figure 88: evidence that test 9 works – after the previous example (where I change the surname to "roberta"), I then search for the surname "geeson" whereby I input "s" in order to change the surname

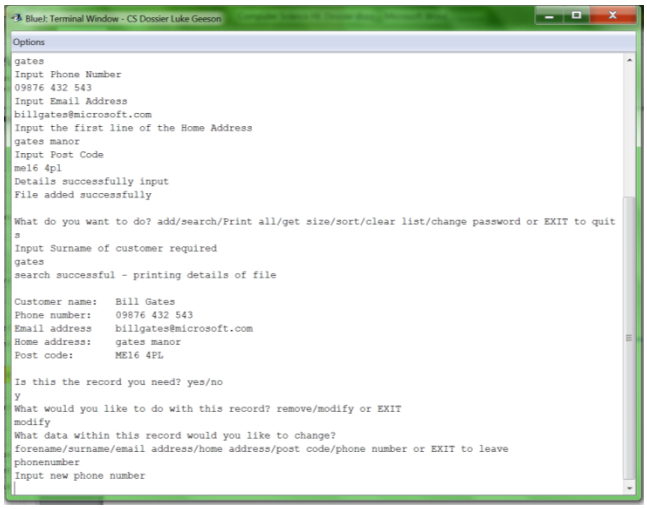


Figure 89: evidence that test 10 works - following the previous example, I add "bill gates" to the database. I then search for it and input "phonenumber" in order to change the phone number

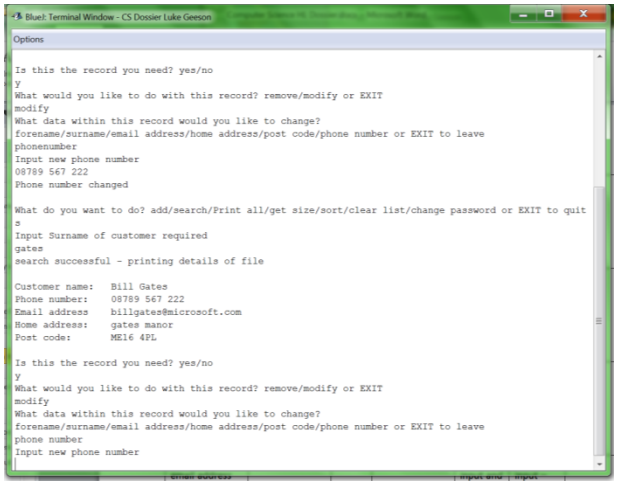


Figure 90: evidence that test 11 works – following the previous example, I searched for the same record again and input "phone number" to change the phone number

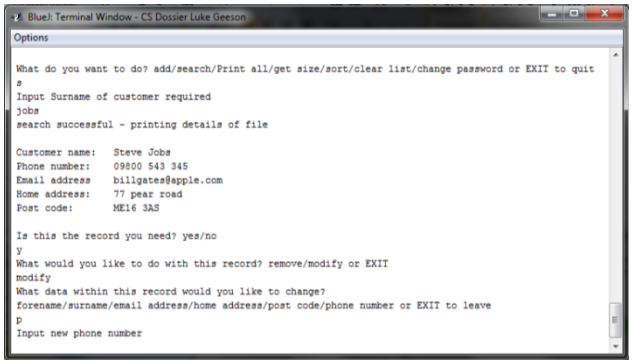


Figure 91: evidence of test 12 – I search the list for the record with the surname "jobs" and then input "p" so that I can change the phone number

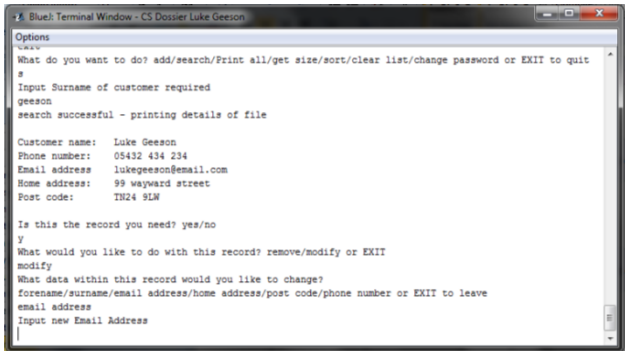


Figure 92: evidence that test 13 works – I search for the record with surname "Geeson" and then input "email address" so I can change the email address

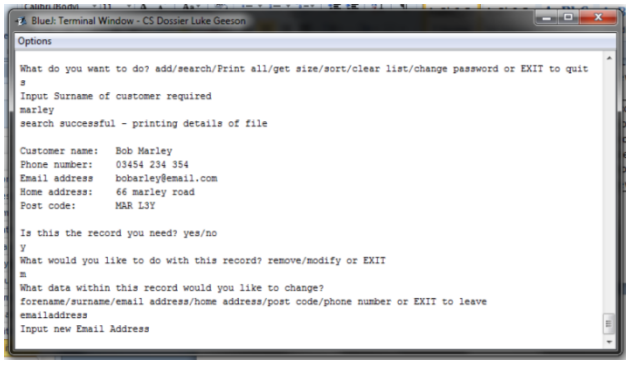


Figure 93: evidence of test 14 – I search for the record with the surname "marley" and then input "emailaddress" so that I can change the email address

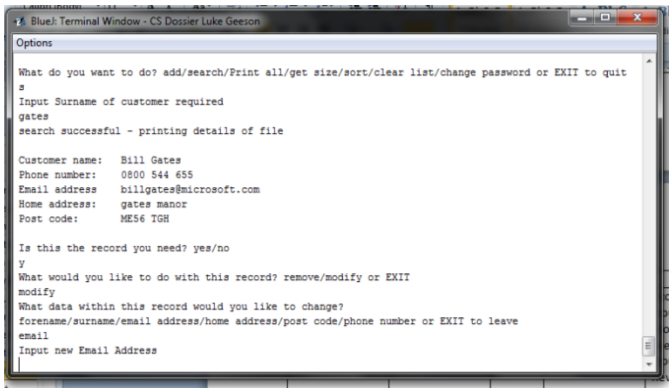


Figure 94: evidence of test 15 – changing the record by inputting "email"

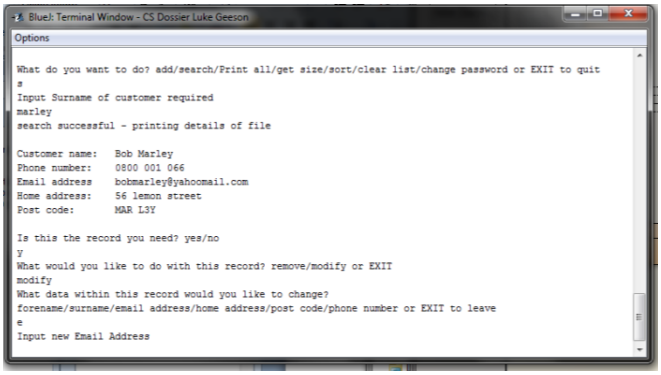


Figure 95: evidence of test 16 – changing the email address by inputting "e"

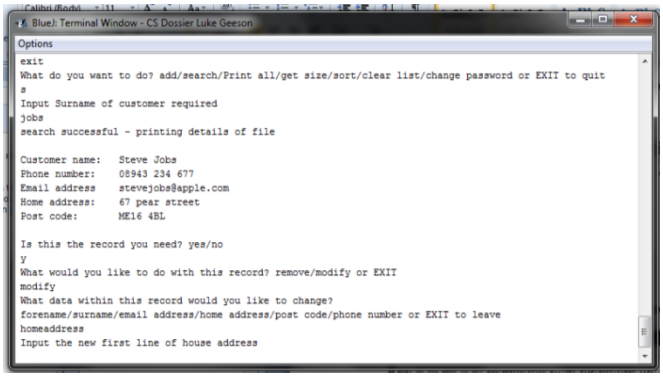


Figure 96: evidence of test 17 – changing the home address by inputting "homeaddress"

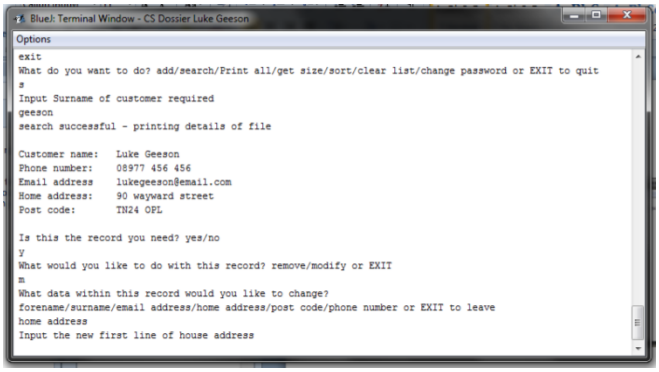


Figure 97: evidence of test 18 – changing the home address by inputting "home address"

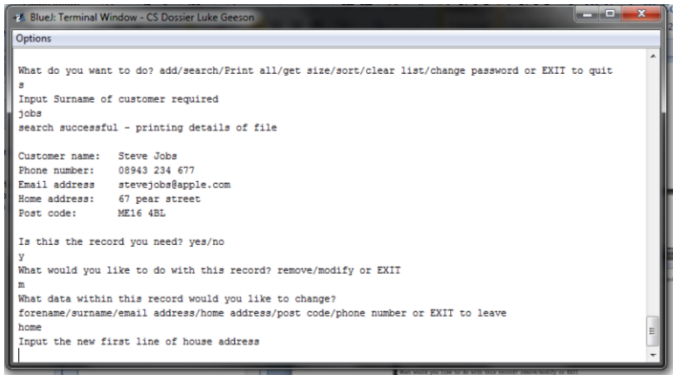


Figure 98: evidence of test 19 – changing the home address by inputting "home"

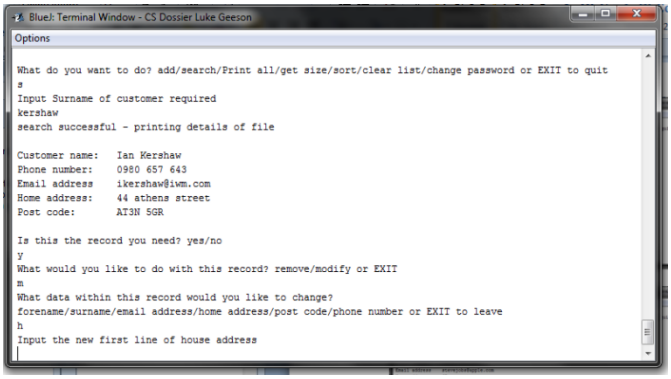


Figure 99: evidence of test 20 – changing the home address by inputting "h"

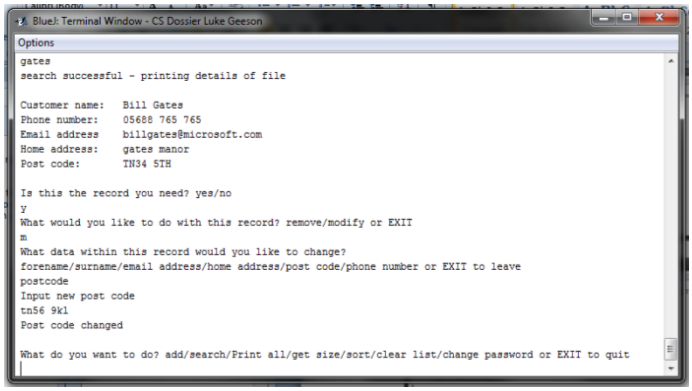


Figure 100: evidence of test 21 – changing the post code by inputting "postcode"

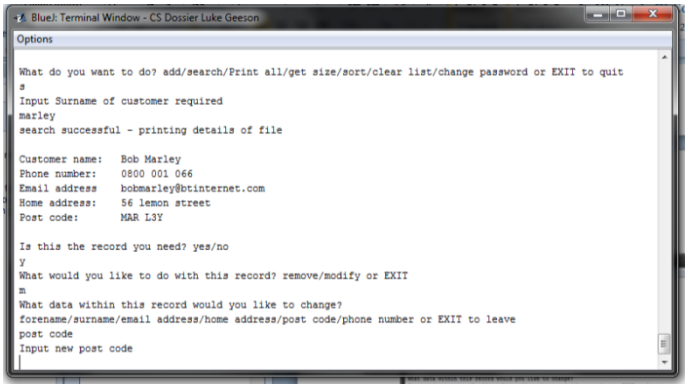


Figure 101: evidence of test 22 – changing the post code by inputting "post code"

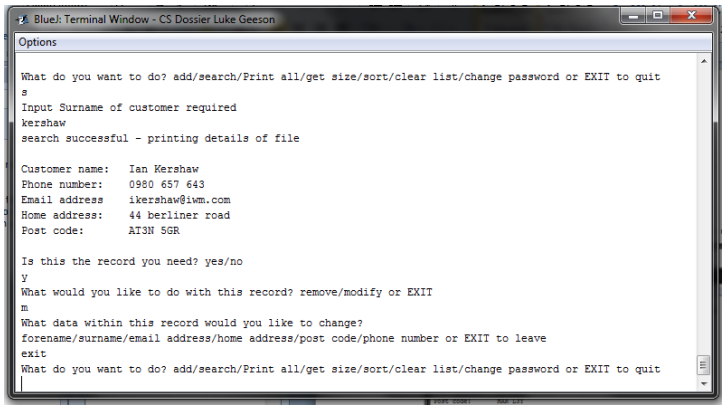


Figure 102: evidence of test 23 – returning to the main menu by inputting "exit"

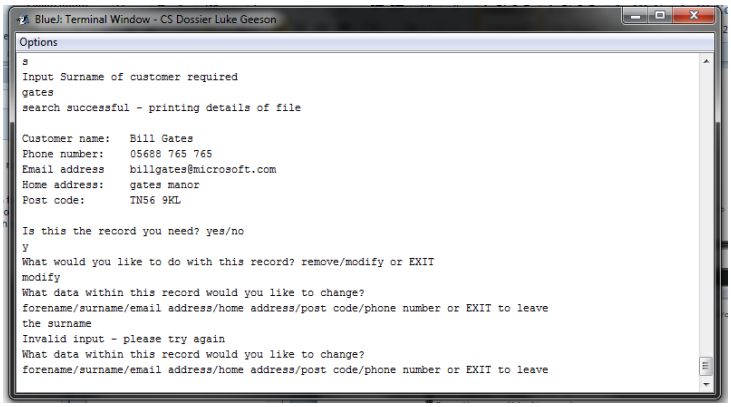


Figure 103: evidence of test 24 – proof that the system can handle errors at this stage as "the surname" may seem valid, but is not an accepted form of input for this system

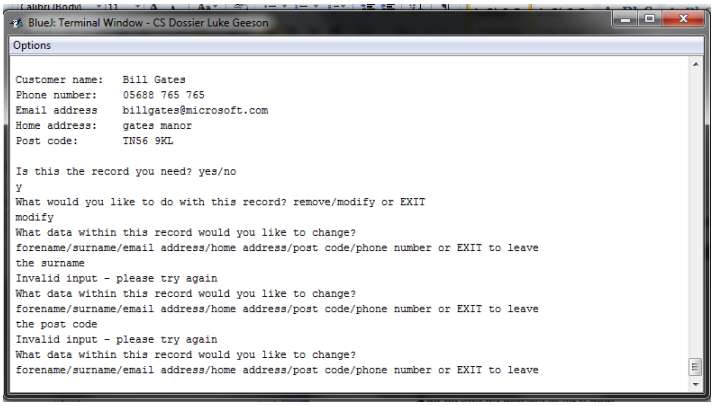


Figure 104: evidence of test 25 – proof that the system can deal with invalid input as "the post code" may seem valid but is not an acceptable form of input for this system

D1: Printing the list

This section will test another function of the program – it will print all of the items in the list

Test for	Expected result	Actual result
1.Printing an empty list	Should display a message – "cannot print an empty list"	Get message "cannot print an empty list" –does not print list
2.Printing a 1 item list	Should print the 1 item in the list	Prints the list

Evidence that the testing works:

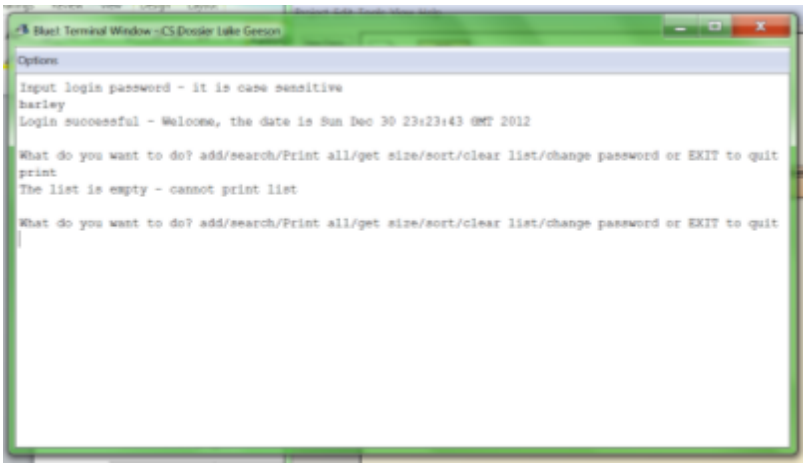


Figure 105: evidence that test 1 works: I have input the "print" command after clearing the list that was stored in the file

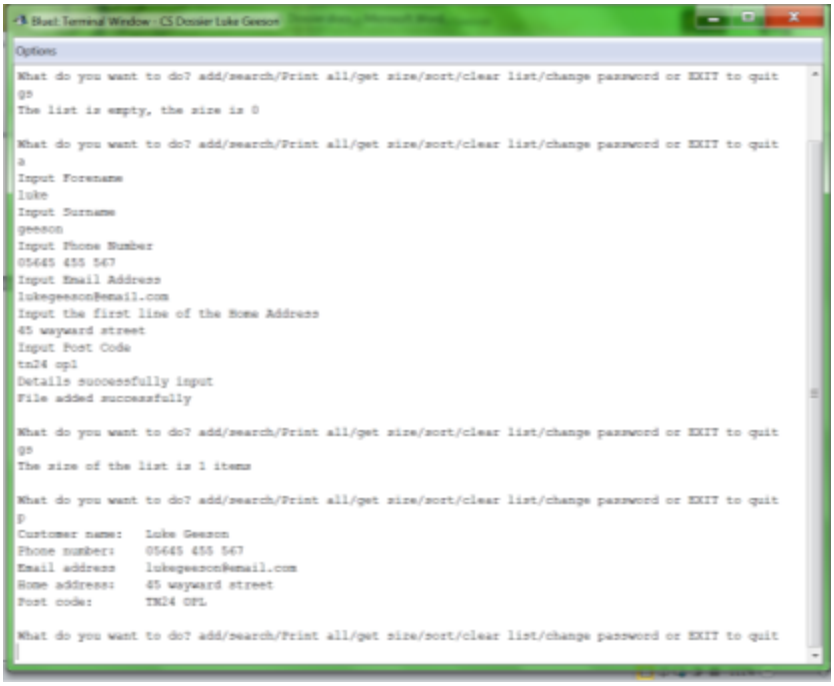


Figure 106: evidence that test 2 works. I add an item and then print the list – its prints the 1 item list

D1: Getting the size of the list

This section will test another function of the program –it will tell the user how many items there are in the list. This does not come as part of the criteria for success, but rather it is part of the function of the program. This table will summarise the testing, evidence follows:

Test for:	Size of list	Expected result	Actual result	Comments
1.Size of an empty list	0 items	Should display message – list is empty	Displays message "cannot print list" does not print what is not there	
2.Size of a 1 item list	1 item	Should display message –list has 1 item	Displays message – "list has 1 item"	

Evidence that the testing works:

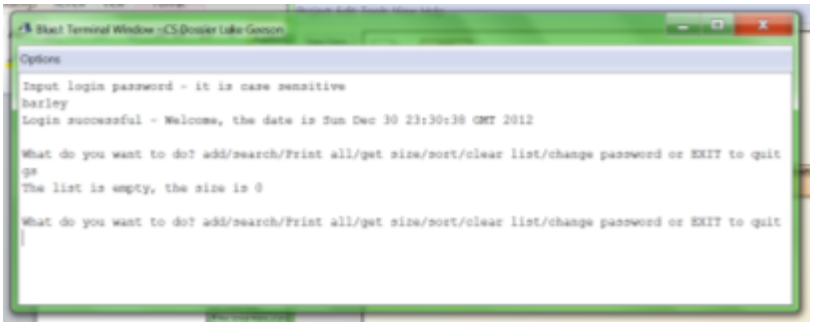


Figure 107: evidence that test 1 works: getting the size of an empty list after manually deleting the file containing the test data I had previously entered

Figure 108: evidence that test 2 works: getting the size of a list after adding an item. I add an item and then input the command for the size of the list

D1: Sorting the list

This section will test the sort function of the program which will sort the linked list in ascending order by surname. When customer records are added, the insert function ensures that the records are stored in ascending order. This means that the list will already be sorted when the user needs it; this means that requirement 5 is already partially satisfied. However, should a user modify the surname of a customer record then the list will not be sorted. This why the sort function is important – it allows the list to be sorted at the request of the user or when they edit the surname of the customer record. As a result, a combination of the sort routine and the insert record routine ensures that requirement 5 is fully satisfied. The following table will summarise the testing performed on the program- this is by sorting the list from the main menu or by indirectly sorting it when the user changes the surname.

Test for:	Input:	Data type:	Expected input	Expected result:	Actual result:	comment
1.Sorting from the main menu:	"sort"	Valid	The list should be sorted	The list is sorted	The list is sorted	Message is displayed prompting the user of a successful sort
2.Attempting the sort an empty list from the main menu	"sort"	Valid	"sort"	A message should be displayed – "cannot sort empty list" – no sort	Message displayed – "cannot sort empty list" – does not sort	
3.Attempting to sort a 1 item list	"sort"	Valid	"sort"	The list is already sorted –	The list Is already sorted –	Message displayed – list sorted

from the main menu				should not sort	does not perform sort	
4.Sorting when a new surname is input	New surname “smith”	Valid	Anything	The program should perform a sort on the list	The list is sorted	Message displayed – list sorted
5.Sorting when an invalid input is set as the surname	New surname	Invalid	Anything	List should be sorted	List is sorted	Message displayed – list is sorted

Evidence that this testing works:

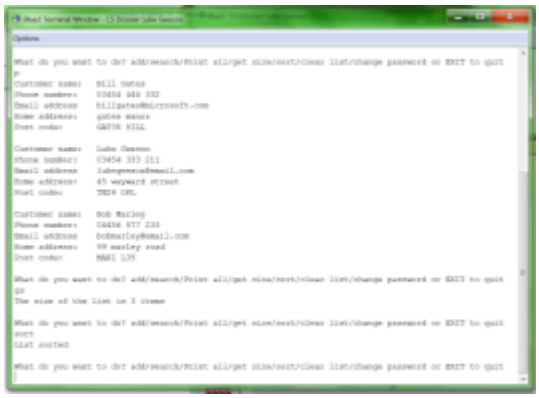


Figure 109: evidence that test 1 works. As you can see the list I am applying the sort to is already populated with 3 items which are stored and sorted by the add record function. This means the sort record is redundant as an option on the main menu – but the sort function can be called and executed from the menu should the user want it.

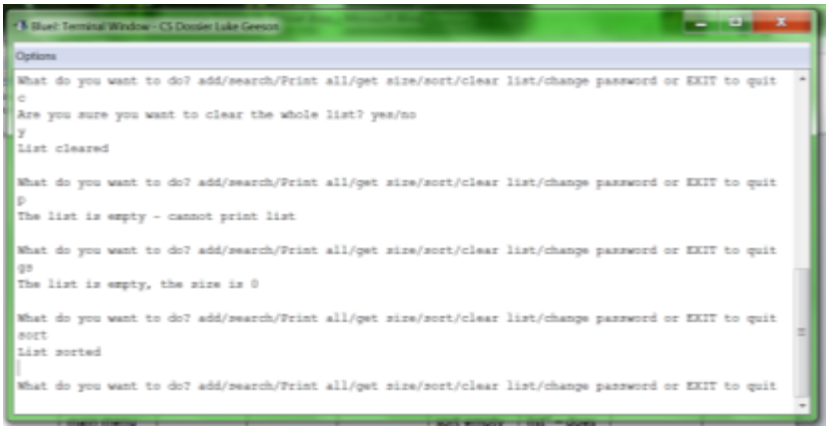


Figure 110: evidence that test 2 works – the list is empty and when the sort function is called – it displays a message to state that the list is sorted (for the peace of mind) but the code does not actually sort the list

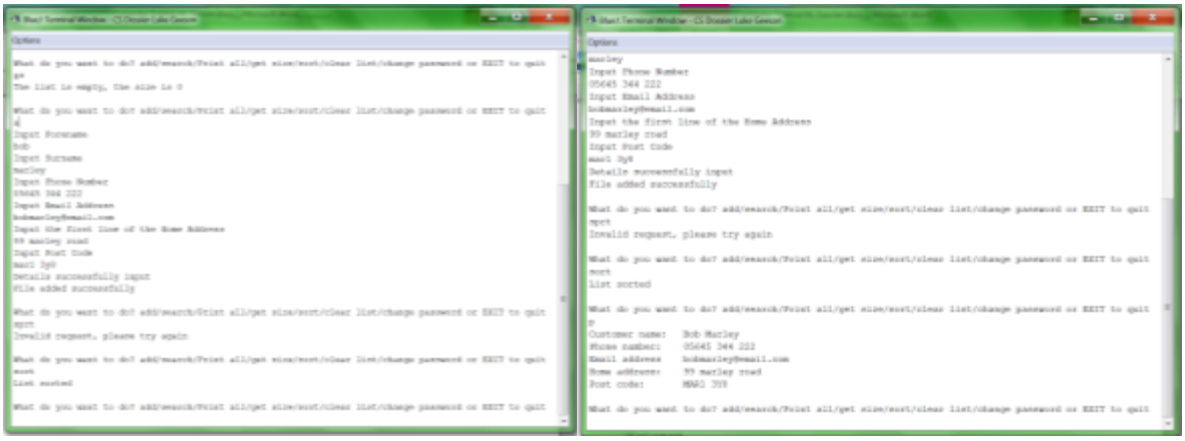


Figure 111: evidence that test 3 works. I get the size of the list which is 0, the list is empty. Then I add a record which and sort the list. Since there is only one item, a sort is not necessary but the program will nevertheless prompt the user that the list is sorted for the user's 'piece of mind'.

When a surname is changed in the list – the sort function is automatically called on the list so that the list is already sorted by the program and will not need the user to manually sort the list – this now satisfies requirement 5 (sort by surname at the request of the user and also independently of their involvement.

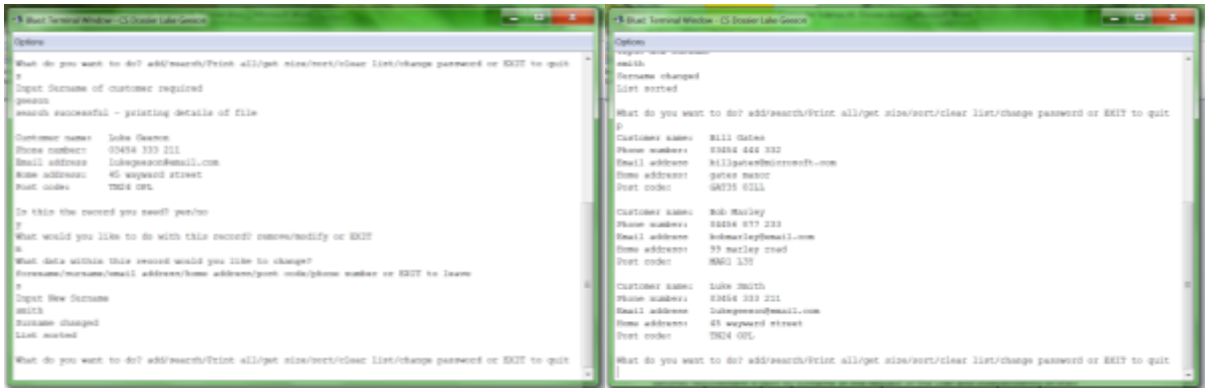


Figure 112: evidence that test 4 works. Using the same list as the one in test 1, I have searched for and found the file with the surname "Geeson". Then I have changed the surname to "smith" and the list has automatically called the sort function – evidence that the list is sorted independently of user interaction. The second image shows the newly sorted list and you can see that "Geeson" has been changed to "smith" and sorted so that "smith" comes after "Marley"

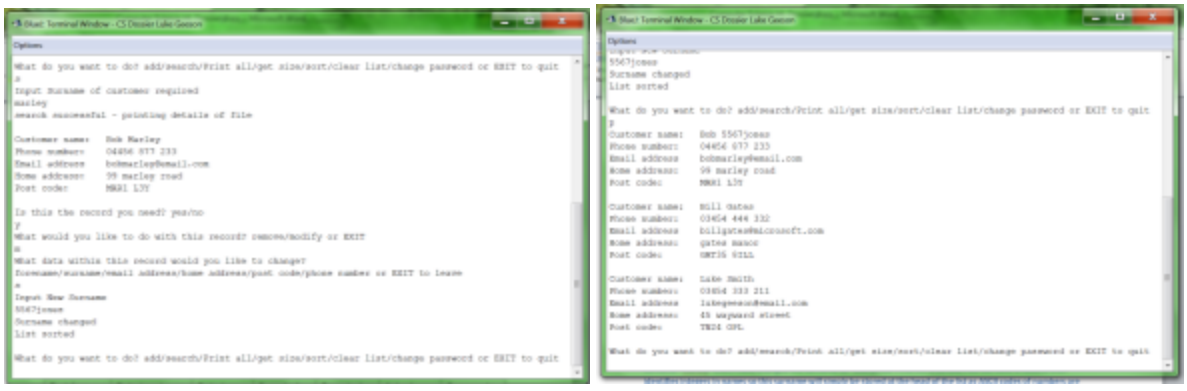


Figure 113: evidence of test 5: I have used the example from test 4 and searched for and found the record with the surname "marley" then I have changed the surname to "5567jones". The list is sorted, I do not have a routine that identifies integers in names so this surname will simply be stored at the head of the list as ASCII codes of numbers are less than those of letters. Since the compareTo method compares said ASCII codes, the list will place these invalid yet

accepted names at the front of the list. As you can see, the newly modified file is now at the head of the list because of the numbers

D1: Clearing the list

This section will test another feature of this system in more depth – this section is not testing to ensure the requirements are satisfied, but rather it is testing the clear function of the list to ensure that it works. The following table shows what happens once the clear function is called from the main menu.

The answers are not case sensitive and so typing 'YES' or 'yes' will yield the same result

Test for:	Data input	Data type	Expected input	Expected result	Actual result	Comments
1.Ensuring the program understands 'yes' as an answer	"yes"	Valid	"yes" or "y"	Clears the list and returns to the main menu	Clears the list and returns to the main menu	Displays a message to confirm to the user that the list is clear
2.Ensuring the program understands "y" as a substitute answer for "yes"	"y"	Valid	"yes" or "y"	Clears the list and returns to the main menu	Clears the list and returns to the main menu	Displays a message to confirm to the user that the list has been cleared
3.Ensuring the program understands "no" as an answer	"no"	Valid	"no" or "n"	Should not clear the list – should return to the main menu	Does not clear the list - returns to the main menu	Message displayed to indicate that the list has not been cleared
4.Ensuring the program understands "n"	"n"	Valid	"no" or "n"	Should not clear the list – should return to the main menu	Does not clear the list – returns to the main menu	Message is displayed to indicate the list has not been cleared.
5.Invalid input	"asdfqwerty50"	Invalid	All of the above	Should loop until valid answer is input	Does loop until a valid answer is input	Message to prompt the user that this is invalid
6.invalid input	"yes please"	Invalid	All of the above	Should loop until a valid answer is input	Does loop until a valid answer is input	Message to prompt the user that this is invalid

Evidence that the testing works:

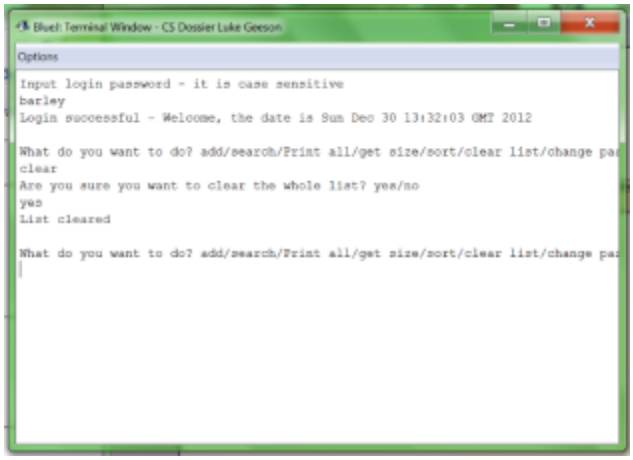


Figure 114: evidence that test 1 works: that "yes" is an accepted answer

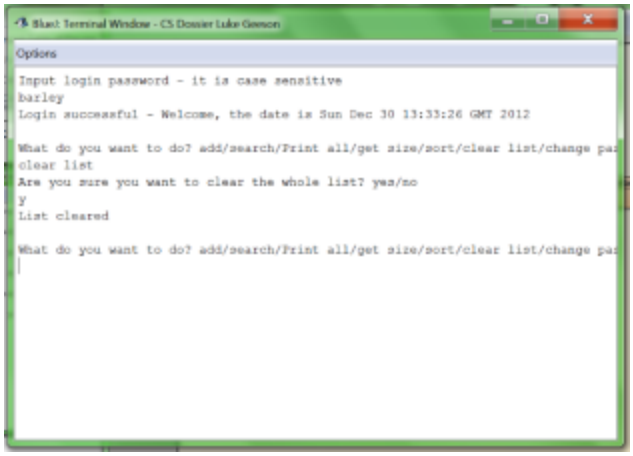


Figure 115: evidence that test 2 works: that "y" is an accepted answer

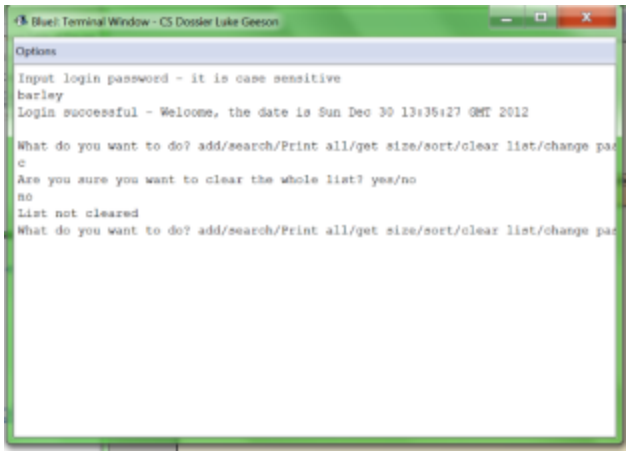


Figure 116: evidence that test 3 works: that "no" is an accepted answer

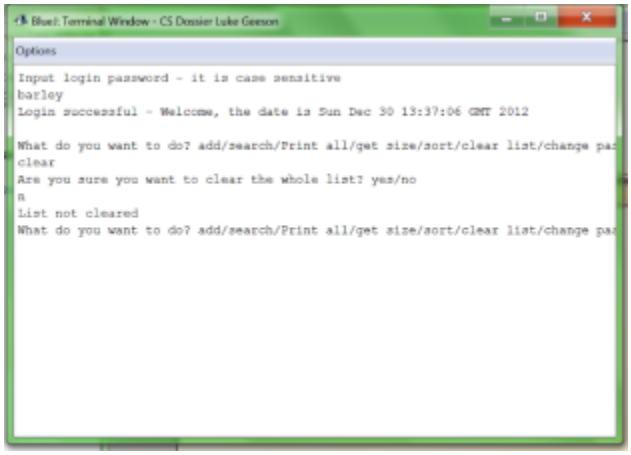


Figure 117: evidence that test 4 works: that "n" is an accepted answer

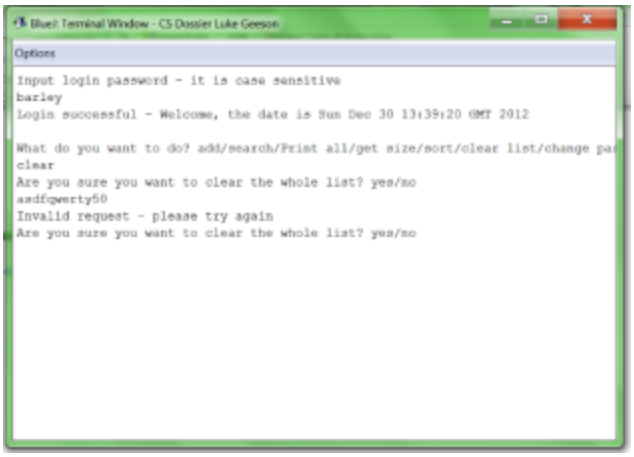


Figure 118: evidence that test 5 works: that "asdfqwerty50" is an invalid answer

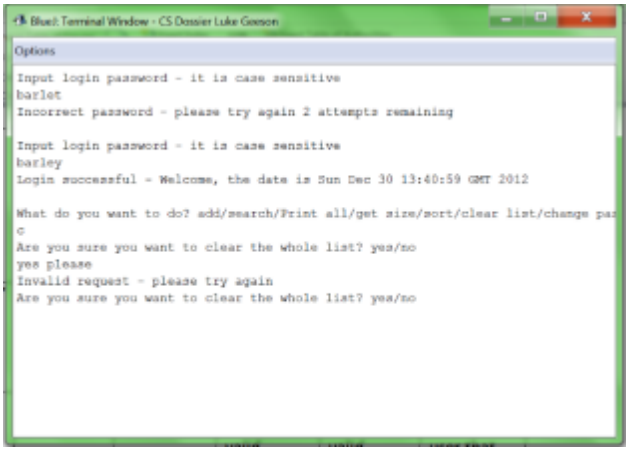


Figure 119: evidence that test 6 works: that "yes please" is an invalid input"

D1: Changing the password

This section is testing the ability to change the password once the user has access to the main menu. In a way this testing relates to requirement 9 as it allows the user to change their password and thus, reduce the chance of getting hacked by malicious users of the system. The password itself is serialised and stored in a separate file which means malicious viruses that directly access the program will not find the password for the system with the system data. It also partially satisfies

requirement 7 as the user must input the old password before they create a new one – if this is incorrect, then they must input commands to try again ("yes" or "no") which ensures the system does not crash.

For the following tests, we are assuming the default password is "barley"

Test for:	Input	Data type	Expected input	Expected result	Actual result	comments
1.Correct input of old password on request	"barley"	Valid	"barley"	Should accept old password and request new one	Accepts old password – requests new one	New password can be anything – no testing needed
2.Incorrect input of old password on request	"chewbacca"	Invalid	"barley"	Should reject password and ask the user if they would like to try again – the validation	rejects password – asks user to try again	Goes to validation screen
3.Respond to validation with "yes"	"yes"	Valid	"yes" or "Y"	should try old password input again	Tries old password input again	
4.Respond to validation with "y"	"y"	Valid	"yes" or "y"	Should try old password input again	Tries old password input again	
5.Respond to validation with "no"	"no"	Valid	"no" or "n"	Should return to main menu	Returns to main menu	Displays message "password not set"
6.Respond to validation with "n"	"n"	Valid	"no" or "n"	Should return to main menu	Returns to main menu	Displays message "password not set"
7.Respond to validation with invalid data	"guitar"	Invalid	"yes" or "y" or "no" or "n"	Should retry validation	Retries validation – loops until valid input Is found	

Evidence that the testing works:

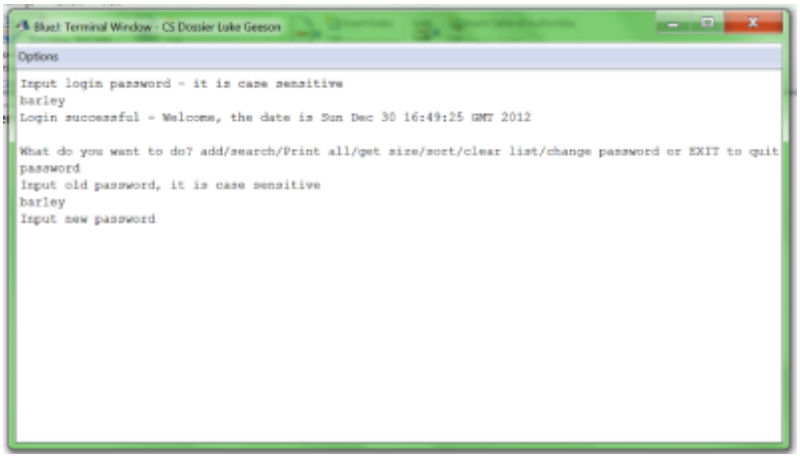


Figure 120: evidence that test 1 works: it has accepted the correct old password "barley" and now requests the user to input a new one

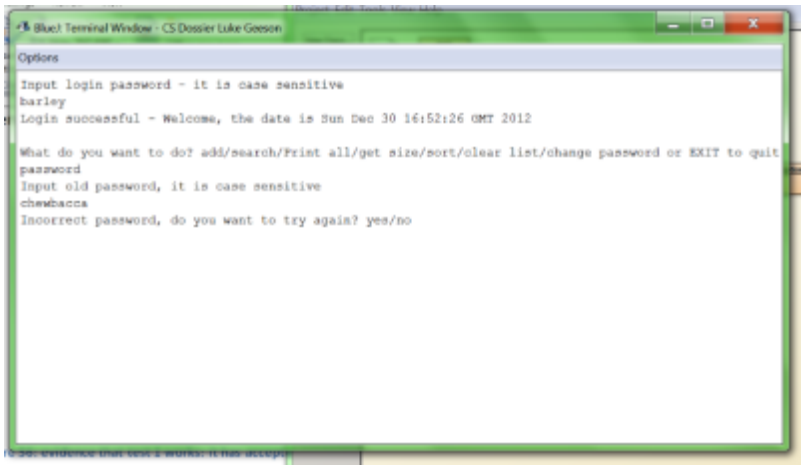


Figure 121: evidence that test 2 works: the incorrect password "chewbacca" has been input and the system has rejected it – showing the user the validation options ("try again? Yes/no") so they may try again or stop

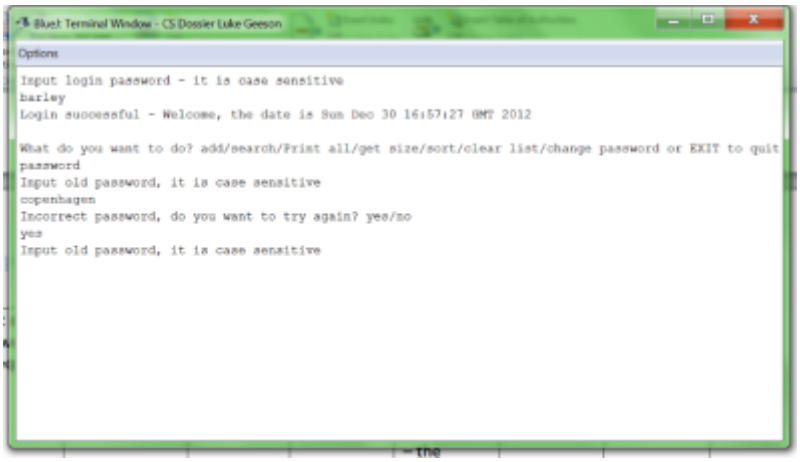


Figure 122: evidence that test 3 works: inputting "yes" for the validation has been accepted and the user has another attempt to input the old password (see tests 1 and 2). I have also input another incorrect password "Copenhagen" in the process of getting to the validation which further reinforces test 2

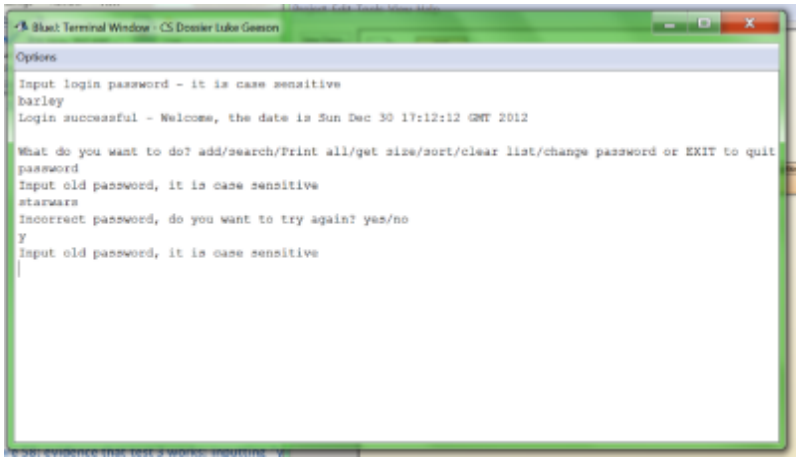


Figure 123: evidence that test 4 works: you can input "y" and get the same result on the validation screen as you would if you input "yes". You can also see that I have input the wrong password "starwars" to get to this stage which further reinforces the evidence for test 2.

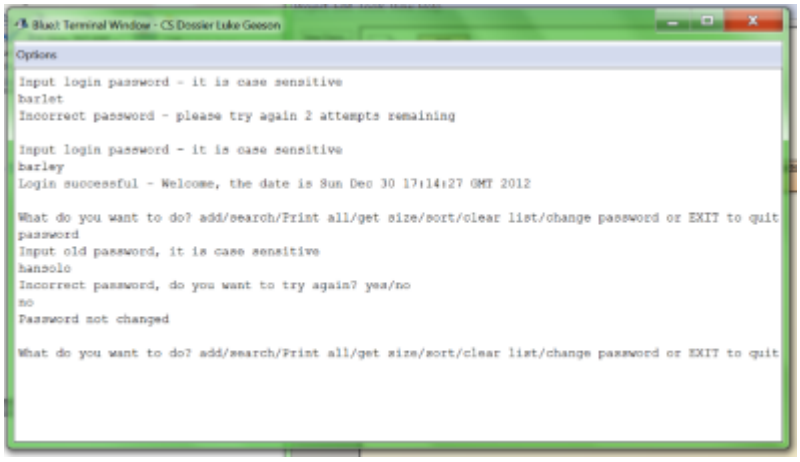


Figure 124: evidence that test 5 works: inputting "no" at the validation stage does exactly what you expect it to – it returns to the menu. You can also see that I have input the incorrect password "hansolo" to get to this stage which further reinforces test 2.

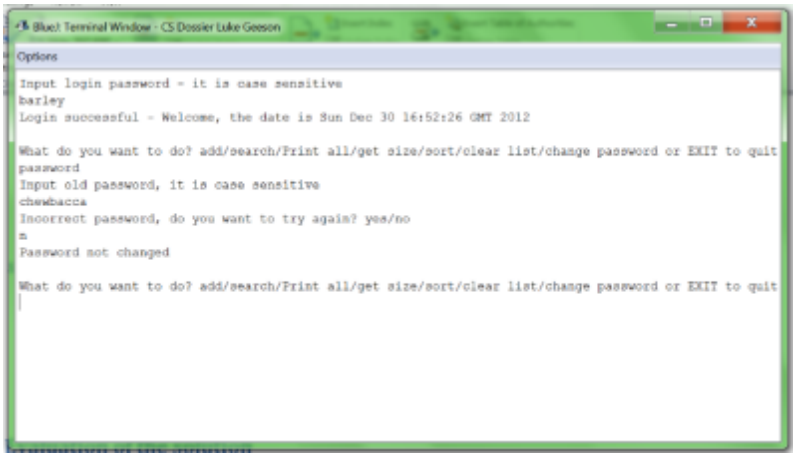


Figure 125: evidence that test 6 works: inputting "n" will yield the same result as inputting "no"

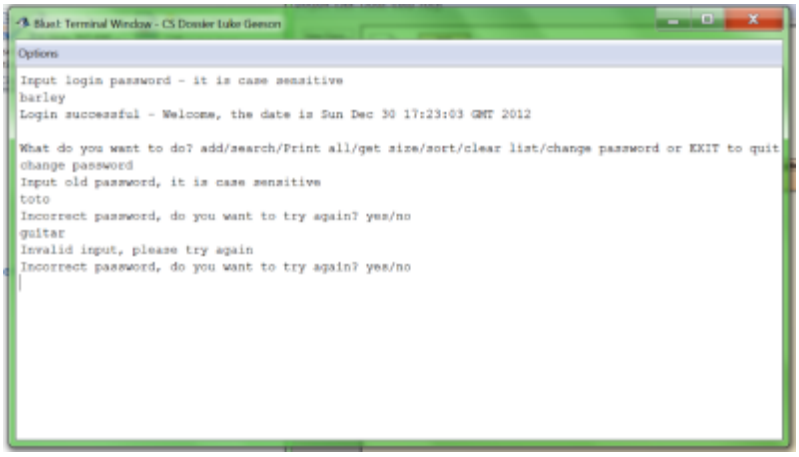


Figure 126: evidence that test 7 works: I have input the invalid data "guitar" and the system has dealt with it accordingly and continues the loop until valid commands are input

D1: Satisfying requirement 3

In order to show that I have satisfied requirement 3 (easy to back up), I will show some screen shots of me backing up the linked list data base using the 'drag and drop' method which will most likely be used by the user as it is the easiest way to back it up. I will be backing the data from a file location on my computer to a folder on an external drive

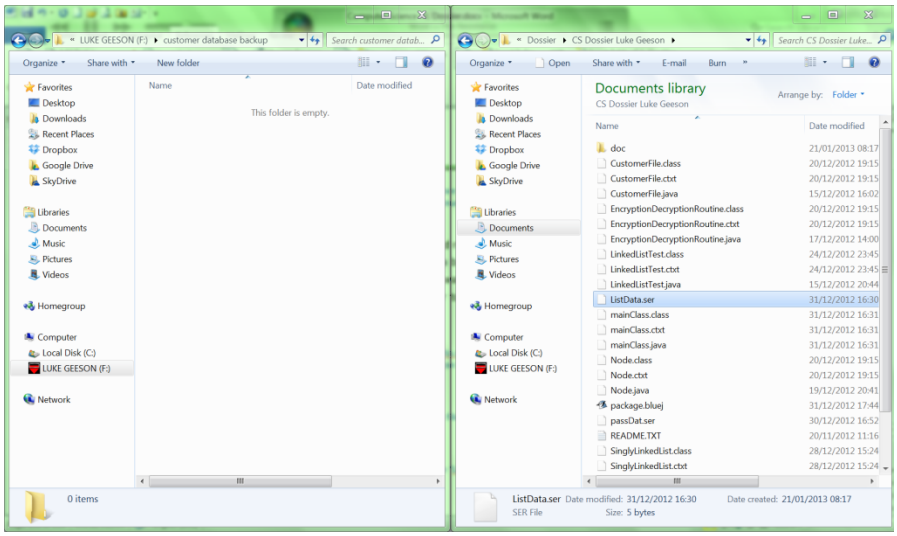


Figure 127: As you can see I have the 'ListData.ser' data file on the computer system and I have a 'customer database backup' folder on the flash drive; I can easily transfer the file the new location

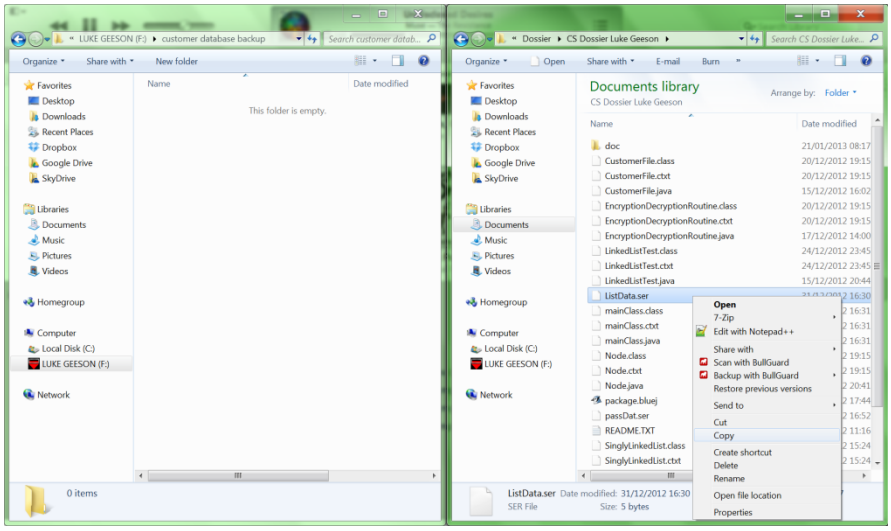


Figure 128: As you can see, I am copying the file to the external drive

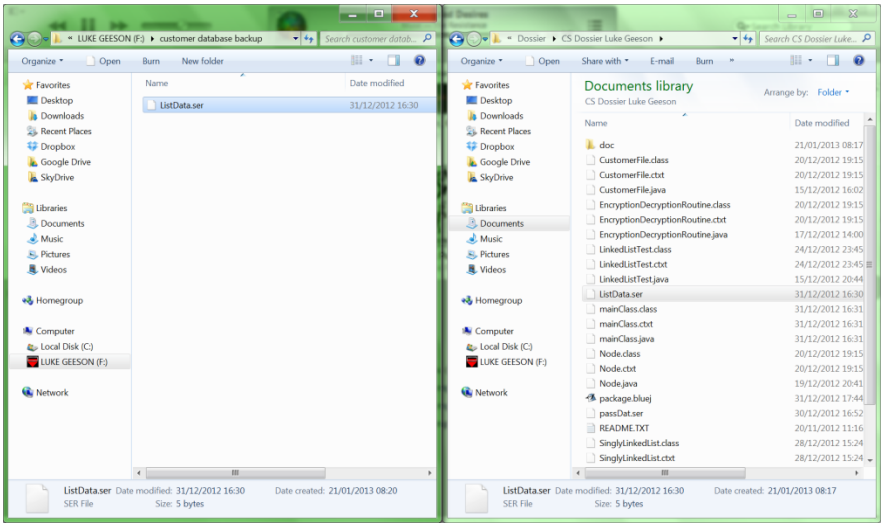


Figure 129: As you can see, I have copied the 'ListData.ser' to the new file location.

D2: Evaluation of the solution

D2: The Solution

D2: Did it work?

The program and the system does work, you can successfully login, add, remove, search, modify, clear the list, get the size and print the list with the knowledgeable use of commands of the command line interface. All of the commands and routines used have validation methods to ensure the system does not crash when data is input whether it is valid or not. The system cannot enter any infinite loops at the fault of the program or the user and will only operate based on the valid input from the user – which means it has a strict set of commands which work and any others are simply not accepted.

D2: Did it address the criteria for success in Stage A?

In short, yes the program and the system created did address all of the criteria of success and completed them sufficiently to provide a working system for the user. Unfortunately, some of the things done when making the system were not the most efficient (such as the use of a bubble sort, or the fact that the list is sequential when the user can only really deal with, and only needs to deal with, one record at a time) however, they do successfully satisfy the criteria for success fully and sufficiently for the current and the near future needs of the customer.

Please see:

C3: Success of the Program for evidence and descriptions of the criteria being satisfied.

D2: Does it work for some data sets, but not others?

The system was designed predominantly around the use strings as a data set because the data required to be used could feasibly be stored as string objects (such as the surname or email address) without much complication. This means that the system would ultimately have to compare the ASCII codes of the surnames which could run into problems: should the user input non-ASCII letters (such as Chinese on another keyboard) then the system may not accept this and may not be able to process this. Granted, this system is being designed for a company based in the UK and the range of available text typing is wide scale use is limited and mostly covered by ASCII – this factor is worth considering but not entirely relevant to the system as the user had stated that the business computer used a QWERTY keyboard which is fully catered for by ASCII.

When inputting invalid data to strings, the system did run into some validation issues. When the user would input an invalid, yet accepted string (such as 59simon) the system would accept it and would not crash but it would also not effectively deal with it as these have the relevant ASCII code equivalents and so the system simply compared these. With this said, a custom validation routine to filter out invalid data would have taken much longer to design and implement as the possibilities of data input are vast and unpredictable at best. This would not have been feasible with the time constraints of the dossier. There were, of course some basic validation routines which ensure the data input is correctly stored in the appropriate format (such as UK postcodes being capitalised, or the first character of a surname being capitalised); this ensured that the system was relatively flexible with the data input, as long as it is valid and can be formatted. This also means that the valid data was correctly stored and so it does not need to be changed later.

Doubles were not used in the system and did not need to be as none of the data stored had floating points, nor did the commands or the methods used in the system.

Ultimately, the list was sorted by ASCII codes, which means the use of numbers was limited to binary numbers which sorted the list and integers were only used in the **size** method which worked fine.

D2: The Efficiency of the Program

The list itself is a singly linked list and so it is sequential by nature. This means the big O efficiency of the singly linked list is O(n). This also means that all of the methods and routines associated with this are also sequential which means the big O efficiency of those is also O(n). This means that the use of the list will be quick initially; however the speed of the system will reduce as the size of the list grows: adding, searching, removing, and clearing the list will be directly proportional to the number of items in the list. This is not helped by the fact that the way that the list is programmed means that the whole list is copied into memory each time the system is accessed: this will take up more computer memory as records are added and slowly become slower over time as the list takes up more memory. Of course, with the hardware specification of the system this will not be a problem as the processor is very quick and there is a lot of RAM which means this will not be a problem until the business undergoes serious expansion (and with it, more customers to store records for). The secondary memory used to store the file is large enough to store all the data easily and will be helped by a regular defragmentation of the hard disks.

D2: Does the program in its current form have any limitations?

As aforementioned in the efficiency section, the use of a sequential file organisation means that the system itself will become slower as new records are added and this is a problem which will need to be addressed should the business take on more customers (and therefore have more records). Another problem with this is that the scenario at hand states that the user only ever needs one record at a time and the system is single access so that one user will only ever use the system at a time (a networked solution would work, where a central computer could host the master file and terminals could access and modify the master file with transaction files); with this system, the sequential nature means that many other records are accessed at a time (on average); and when the user searches for one, the very nature of the list means that the system must compare other records which is not necessary as the user only needs one.

Another considerable problem, as mentioned in the efficiency section is the fact that the entire list is loaded into memory each time it is required which will be difficult to manage and eventually be impossible to use because of the memory usage.

The command line interface was an interesting feature as it allowed me to become accustomed to how they function and ensured that I covered every possible problem that may occur from invalid input. However, it is the very range of valid input which makes this a problem - the user must know what to input in order to continue using the system. This requires training and experience with the system which decides whether the system is immediately accessible or not. The problem here is that the system is not entirely intuitive when compared to a much simpler (by appearance) graphical user interface and dynamic graphics which make each menu options much easier to use and more enjoyable as a result. Of course the program serves the purpose, but it would have been much better to have a system (like Microsoft access' interface) which allows for easy and clear input of data which allows the user to have an immediate idea of what they could possibly do. Developing this custom user interface would have been very time consuming much like the creation of a game and was not feasible to do in the time frame given but would have ultimately been better for the user.

While on the subject of usability, the search function is also not entirely intuitive. When using databases and search engines, the user expects it to be possible to input any field (such as name, surname or email) and receive the correct file. This database was limited to the use of just a surname as a key field to search (and sort) by; this is problematic as the user must know this, when it may not be given (such as over the phone) and will slow the process of business. Of course, this would be acceptable in a system whereby the unique key field is used which allows direct access straight to 1 file, but this system is sequential and should therefore be more flexible. This brings me onto my next point about the limitations of my search function – the very way it is programmed means the time taken is significantly longer than it would be if done with a binary search. My search function was a sequential and recursive one to traverse through the list to the expected record: the efficiency of this is O(n) (as are all sequential methods in my class) which is comparatively slower than the binary search counterpart which adopts a 'divide and conquer' method. This problem combined with the sequential nature of my system means the list will ultimately be slower – which is never good.

D2: Was the initial design appropriate?

The initial design was good in the sense that it allowed me to plan the core design of a possible solution for the user. This allowed me to have an idea of what is to be done with the program. However, the initial design itself was rather misleading to the user in the sense that it appeared to have a Graphical user interface and the user could select options from each menu by choosing boxes with a mouse. Unfortunately, my system does not have these features as I do not have the time or skill to program such things. This would have been misleading to a user who was shown a graphical user interface; especially so when we consider that most programs used today have a graphical user interface for the user. Therefore, the initial design was not entirely appropriate.

D2: What additional features could the program have? /possible future enhancements

- A GUI so that the expected initial design would have been appropriate and the system would be more intuitive to use with a mouse
- A storage system whereby the records are stored on the disk and only 1, or possibly 2 (when comparing) records would be in memory at a time to minimise the memory used by the system – or a networked system where the master file is on one system and it accessed by terminals.
- More commands – so that the user could change the list in many ways. There were commands for more things in the classes, such as the **appendLast** method, however this was only implemented so that a technical user could use this operation or make this operation available in the future.
- Possibly more fields - so that the customer may input values such as gender, card details or money owed/paid so that the program could be more closely integrated with the order system currently in use as mentioned in section A. Of course this is reliant on the user's opinion of the system and what they want.
- A more intuitive search engine that allows you to search by any field of data so the user is not limited to typing in a surname
- Redesigned functions which use the 'divide and conquer' method to edit the list – such as a binary search which would be much quicker to use than the current sequential functions
- More validation rules for data sets on strings – perhaps an intelligent validation system that reads the data input to ensure it is readable and valid
- If the sequential system becomes too slow, then a direct access system such as a fully indexed file may ensure that the system would have an efficiency of O(1), regardless of the file size.

D2: Alternative approaches to the solution

D2: An array list or treeset (or equivalent)

Other possible solutions I had thought of using included a few of Java's in built databases. The first consideration I had was that it could be possible to use an Array list as this would serve the very same function that a singly linked list would have, and the majority of the programming would already have been done (as an in built class of the Java library). This was a more appealing idea when I considered the fact that any technical user who has experience with Array lists would benefit from this as it would be a familiar concept to them and would therefore be easier to perfect/correct or adapt the system in the future. Of course, there was also the consideration that the list would then be sorted by its own **collections.sort()** routine which would not be specific to the user's

requirements. As a result, I also considered a tree map. This is because a tree map would sort the data in natural order (according to a pre-set rule by a Comparator object of the class which could be set to store the Surnames in alphabetical order. The use of a treeset seems more feasible now when I consider that the use of sets in general allows for the easy management of the union and intersection of sets as well as taking the median value (I could ensure 2 identical records where not present on in the list this way). As a result, it would be easier to implement the 'divide and conquer' theory in the functions which would make a quicker system, especially so when we have large amounts of data (which this system does and will do in the future).

D2: A fully indexed file

As explained above, the use of a sequential file organisation has its limitations which are amplified as more data is added (O(n)). An alternative solution to this problem is a system that uses direct access and a directory of indexes whereby indexes are allocated for each record. This is more feasible when we consider that the user only ever wants 1 record at a time and so the system shouldn't need to search through other records to find the correct one when it can simply 'jump' straight to the position in memory of the required file. Of course this would take more time to implement, especially as it is very complex. But it would remove the performance issues and limitations that this system presents when a lot of data is added - the system would have a consistent performance.

U1: User Manual

This is the text and illustrations used in the user documentation that allows the user to learn how to use the system and troubleshoot any problems. This allows for the satisfaction of requirement 8.

U1: Pete's pets database program

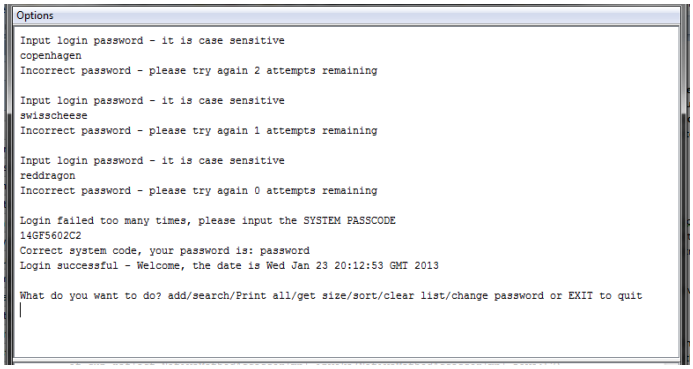
This program was made for Pete's pets in order to convert the previous paper-based system into a digital form whereby the users (employees of Pete's pets) can enter information about regular customers of the store and save this to a database which contains all of the previous data. With this, the user can search for a customer record, add a new customer record, remove an unwanted record, modify and old record as well as other features which enable the user to quickly access and manage the customer information so that service may be improved based on the improved speed at which the user can access this information and hence serve the customers quickly.

U1: First time using the system

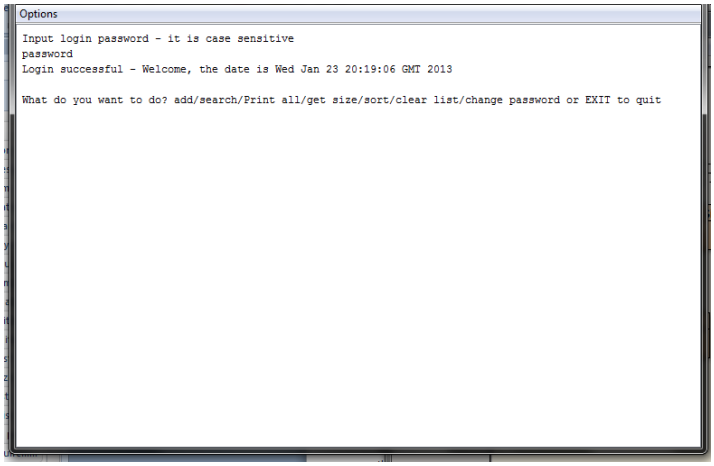
U1: Logging in to the system

This section will illustrate how to log in to the system. The system is **protected by password protection** that is **case sensitive** and enables a layer of security to protect customer data. To login the user must either login with the **system code** (see below) or use a **password that they had previously set**. The user is given **3 attempts to log in** (it is case sensitive) after which the system code must be used to log in

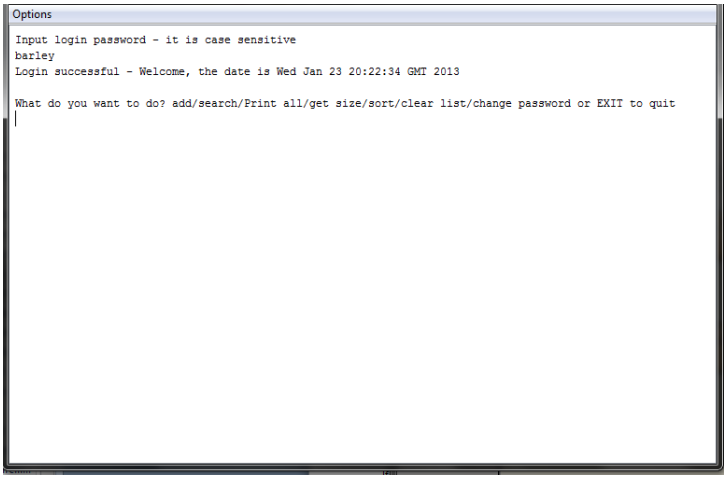
The system code is "14GF5602C2"



If this is the first time that the user is using the system, then they will need to login to the system with the **default password which is 'password'**. This will enable them to get to the main menu where they can change their password should they choose to do so. Here are illustrations that show how to log in to the system for the first time.



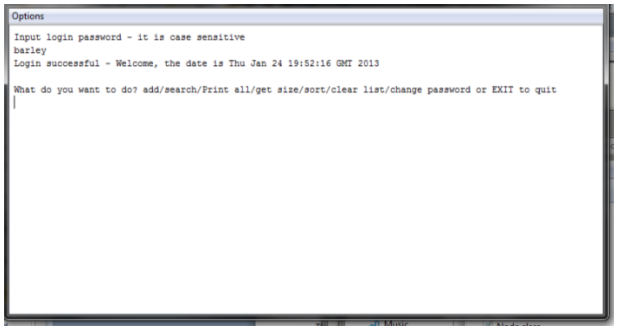
If this is not the first time using the system, input the password you had previously set (would still be the default password if you did not change it). The following example illustrates logging in with a pre-established password of "barley"



U1: Using the system

This program uses a **command line interface**. The user (you) inputs everything by typing in commands on the keyboard which are used by the program to do things with the database.

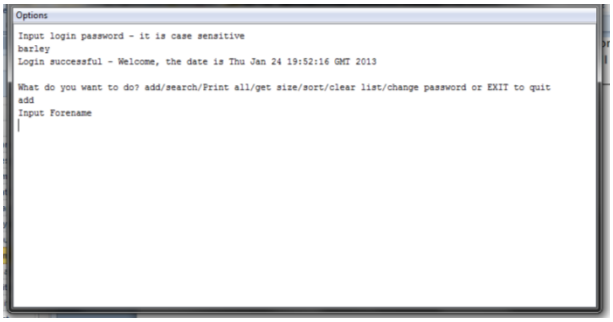
Once you have logged in, you should see the following screen:



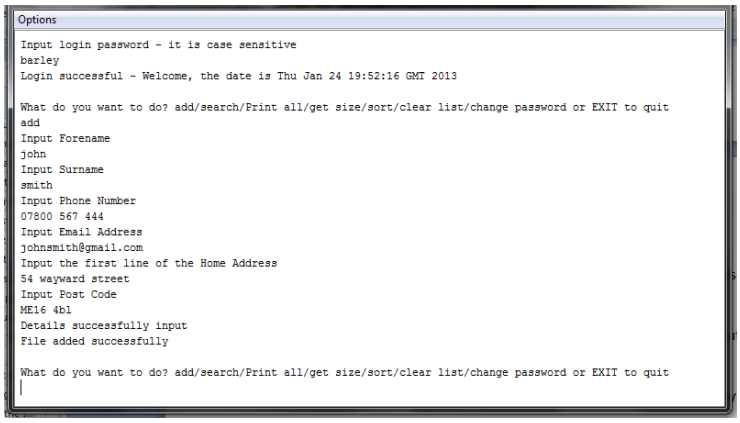
This is the **main menu** which you will return to with each consecutive action that is completed. This program is designed to act as a database where you can do a number of things; customer records for Pete's pets are stored in this database. There are a number of things you can do now: **you can add** a

customer record, **search for a customer record**, **display every record** in the database, **get the size** of the database, **clear the database** of every record or **change the password** used to login to the program.

For now, let's try **adding** a customer record to the list: **type "add"** on the keyboard and **press enter**

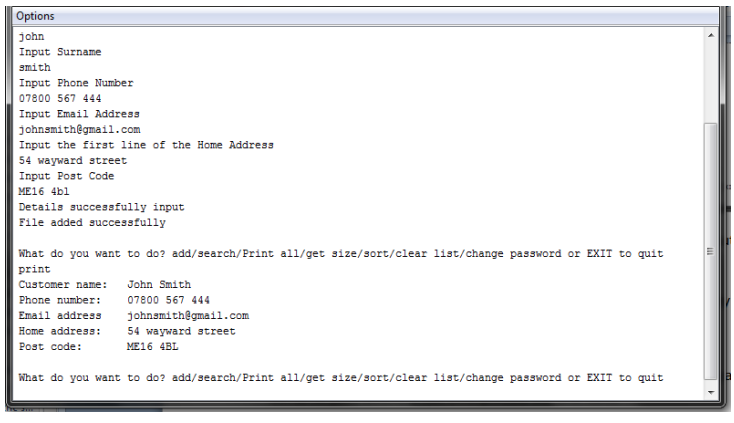


Then input the details of "john smith" (used for demonstration purposes), **input the details** of this fictional character with the keyboard **and press enter** to continue (note: to **input a piece of data, type the data into a keyboard and press enter**; this **must be done for each piece of data**)



As you can see the program has prompted that it has successfully input the details of this customer and returned to the main menu

Now try displaying the records in the database (**input 'print'** to the keyboard and **press enter**)



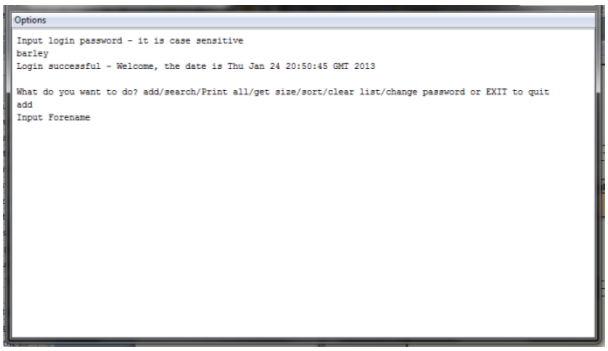
As you can see, it has displayed the items in the list; since we have only added one record (with the surname john smith) it has only displayed the one item (as you add more, more items will be printed).

This is the basics of how you would use this program; you i**nput commands using the keyboard and press enter**. In order to see all of the commands that can be input see the table of accepted commands (below but just before the troubleshooting section)

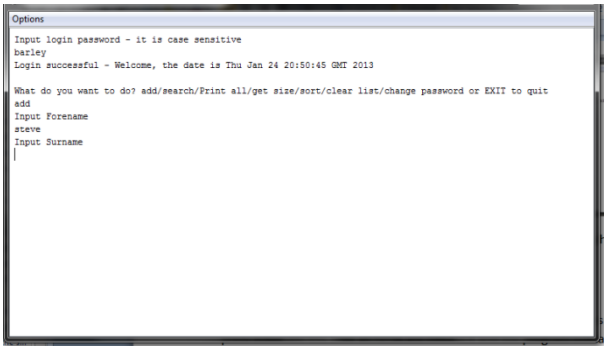
See the other sections of this guide to learn in detail how to use each function of the database.

U1: Adding a record

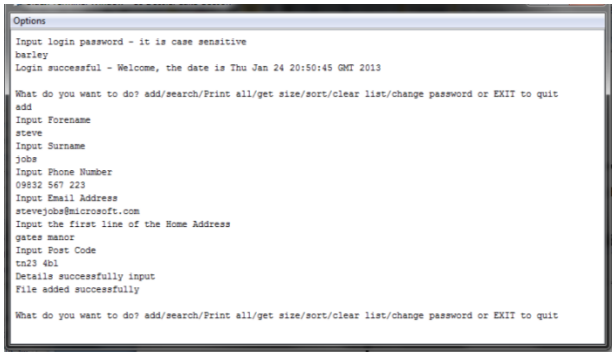
When on the main menu type **"add"** or **"a"** with the keyboard and **press enter**



You will then be asked to input the forename of the customer of which you are storing, **input the forename and press enter.**

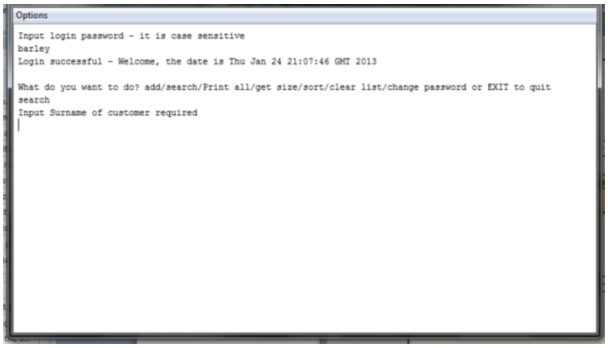


You will then be asked to input the surname, **input the details of the surname with keyboard and press enter**. Do this until **all fields** are filled out and the program indicates that it has added the file successfully. After this is done the system will return to the main menu, and you will have successfully added a file

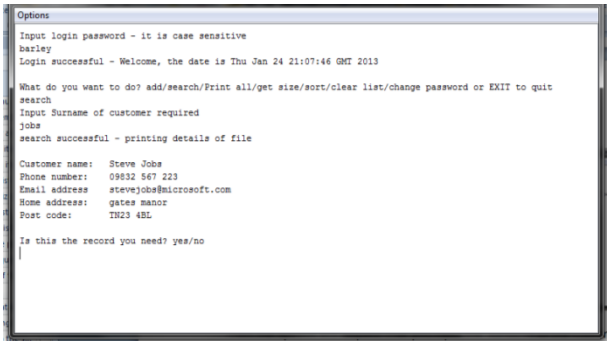


U1: Searching for a record

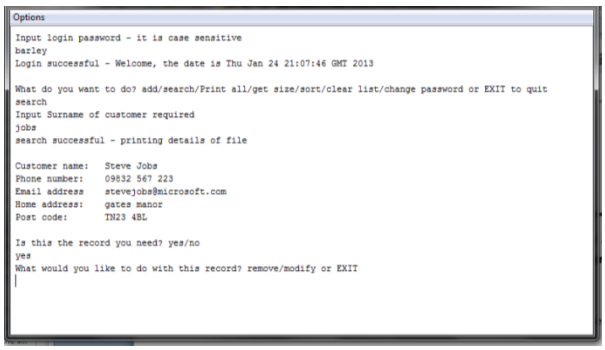
At the main menu, input **"search"** or **"s"** with the keyboard and **press enter.**



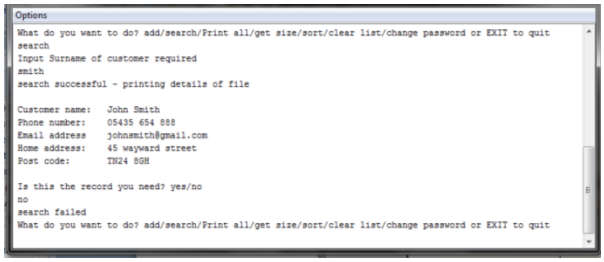
Then the system will request that you input the surname of the customer you are searching for, **input the surname with the keyboard and press enter.**



If the search is successful, the program will display the information of a customer. Indicate, when prompted whether this is the correct file by **inputting "yes" or "no" and press enter to input it**



If "yes" then the program will ask if you want to remove or modify the record

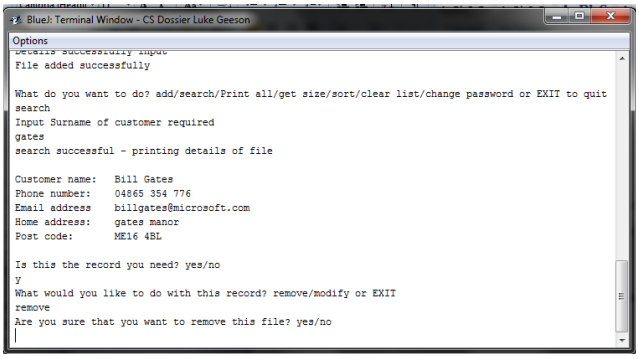


If “no” then the search will continue from that point and display any other matches (again asking for confirmation) or otherwise indicate the “search has failed” and will return to the main menu.

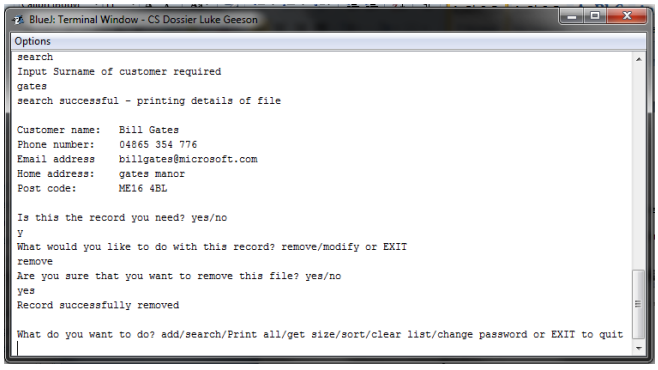
U1: Removing a record

In order to remove a record, you must first search for it and find it (see the previous section)

Once you have found the record and have been given the option to remove or modify the record, **input “remove” or “r” with the keyboard and press enter**



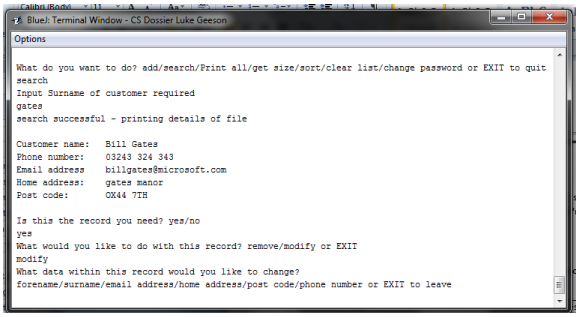
You will then be asked whether you want to remove the file or not, **input “yes” to the keyboard if you do and “no” if you do not, then press enter**



The file will or will not be removed, dependent on what you input

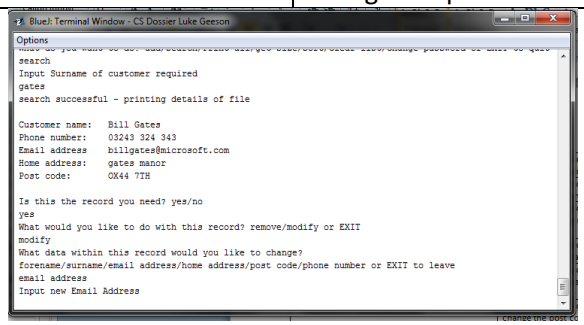
U1: Modifying a record

In order to modify an item, you must first search for it (see the search section). Once you have found the record, you will be given the option to remove or modify it; input **“modify”, “m”, “change” or “c”** and **press enter** to modify it

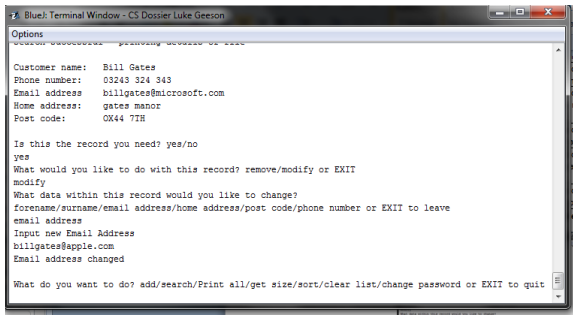


You will then be asked to input what field you want to edit, **input one of the following** for the desired result and **press enter to confirm**

Input to keyboard	Result/what it does
"forename", "first name", "firstname" (no space), "first" or "f"	Indicates to the program that you want to change the forename of the customer record
"surname", "second name", "secondname" (no space) or "s"	Indicates to the program that you want to change the surname of the customer record
"phone number", "phonenumber" (no space) or "p"	Indicates to the program that you want to change the phone number of the customer record
"email address", "emailaddress"(no space), "email" or "e"	Indicates to the program that you want to change the email address of the customer record
"home address", "homeaddress"(no space), "home" or "h"	Indicates to the program that you want to change the home address of the customer record
"post code" or "postcode" (no space)	Indicates to the program that you want to change the post code of the customer record



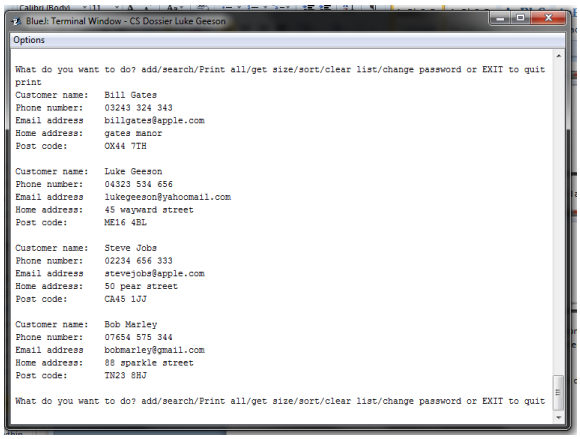
For example, I input **"email address"** with the keyboard and **press enter** – the program interprets my choice and allows me to input a new email address for the record I am dealing with.



Input the new data with the keyboard and **press enter.** Once this is done, the system will take you back to the main menu again so that another command can be accepted.

U1: Displaying all records in the list

On the main menu, input **"print all", "print" or "p"** and **press enter** to display all of the records in the database. This lists every item in the list

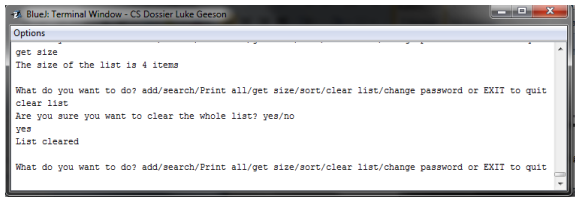


U1: Getting the size of the list

On the main menu, **input "get size", "size" or "gs"** and **press enter** to get the size of the list. I will display all the number of items in the list as an integer

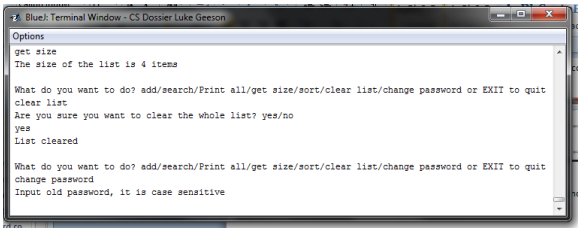
U1: Removing everything from the list

This function will clear the list of every item, on the main menu **input "clear list", "clear" or "c"** and **press enter** to empty the entire list. You will be asked for a prompt to confirm your choice as this clears every item in the list.

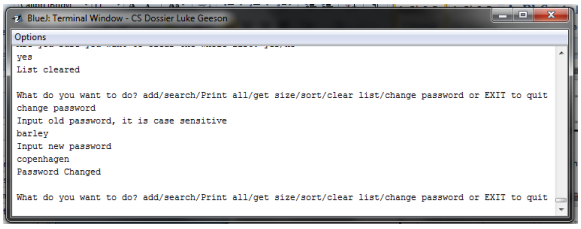


U1: Changing a password

On the main menu, **input "change password" or "changepassword"** (no space) and **press enter** to change the password used to log in to the system.

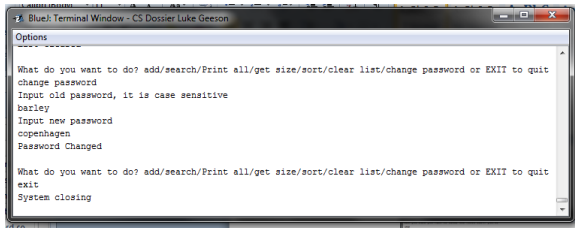


The program will ask for the previous password to be input before you can enter a new one, **input the old password** and **press enter**. If the password is incorrect, the system will not allow you to change the password until the new one is input. When the correct password is input, the system will accept it and request a new password to be input. When this happens, **input the new password** and **press enter**; the password will have been changed and you will be returned to the main menu.



U1: Closing the program

When you are done managing the database, you can close the program by **inputting "exit" or "e"** on the main menu and **pressing enter**. This will allow you to close the program.



U1: Table of accepted inputs

Here are the table of inputs that are accepted as commands on the main menu

Input	What it does
"add"	Add customer record command
"a"	Add customer record command
"search"	Search for customer record
"s"	Search for customer record
"print all"	Display all records in the list
"print"	Display all records in the list
"p"	Display all records in the list
"get size"	Gets the size of the list
"size"	Gets the size of the list
"gs"	Gets the size of the list
"sort"	Sorts the list
"clear list"	Clears the list of all items
"clear"	Clears the list of all items
"c"	Clears the list of all items
"exit"	Closes the program

“e”	Closes the program
“change password”	Allows user to change the password
“changepassword”	Allows user to change the password
“password”	Allows user to change the password

Remove commands – these are the possible commands that are accepted by the system when the user wants to remove a file

Input	What it does
“remove”	Used to request that the user wants to “remove” the file that has been found with the search function
“r”	Used to request that the user wants to “remove” the file that has been found with the search function

Modify commands – these are the possible inputs for the fields of input you may wish to change in a record

Input	What it does
“modify”	Allows the user to modify the record that has been found with the search function
“m”	Allows the user to modify the record that has been found with the search function
“change”	Allows the user to modify the record that has been found with the search function
“c”	Allows the user to modify the record that has been found with the search function
“forename”	Allows user to modify the forename of the file which has been found using the search function
“first name”	Allows user to modify the forename of the file which has been found using the search function
“firstname”	Allows user to modify the forename of the file which has been found using the search function
“first”	Allows user to modify the forename of the file which has been found using the search function
“f”	Allows user to modify the forename of the file which has been found using the search function
“surname”	Allows user to modify the surname of the file which has been found using the search function
“second name”	Allows user to modify the surname of the file which has been found using the search function
“secondname”	Allows user to modify the surname of the file which has been found using the search function
“s”	Allows user to modify the surname of the file which has been found using the search function
“phone number”	Allows user to modify the phone number of the file which has been found using the search function
“phonenumber”	Allows user to modify the phone number of the file which has been found using the search function

“p”	Allows user to modify the phone number of the file which has been found using the search function
“email address”	Allows user to modify the email address of the file which has been found using the search function
“emailaddress”	Allows user to modify the email address of the file which has been found using the search function
“email”	Allows user to modify the email address of the file which has been found using the search function
“e”	Allows user to modify the email address of the file which has been found using the search function
“home address”	Allows user to modify the home address of the file which has been found using the search function
“homeaddress”	Allows user to modify the home address of the file which has been found using the search function
“home”	Allows user to modify the home address of the file which has been found using the search function
“h”	Allows user to modify the home address of the file which has been found using the search function
“post code”	Allows user to modify the post code of the file which has been found using the search function
“postcode”	Allows user to modify the post code of the file which has been found using the search function

Other commands – these are not commands for the main menu but come up when selecting records, confirming record removes etc…

Input	What it does
“yes”	Continues with the action
“y”	Continues with the action
“no”	Does not continue with the action
“n”	Does not continue with the action

U1: Troubleshooting

Here are 10 questions you may have with the system:

1. I cannot log in, my password is wrong. It is possible that if many users are using this program then another user may have changed the password, if this is the case – ask them for the password or use the system code to log in, should you input the incorrect password 3 times.
2. I have input the wrong password too many times and now I cannot login. Use the system passcode to login (see logging in to the system), input this passcode and press enter on the login screen.

3. What I type in on the main menu is an "invalid" input, why? It is likely that the input you have used is not one of the accepted forms of input (see the table of accepted input)
4. Does it matter whether I use capital letters in my password? Yes, password input is case sensitive, so whatever you input must be identical to what is saved as the password
5. Are the commands of the main menu case sensitive? No, the commands you can input on the main menu are not case sensitive
6. I accidently typed in the wrong data for a customer file, how do I change it? Search for the file and then modify the field that is incorrect
7. I accidently typed the wrong data for the surname and now I cannot find the record to change it, how do I find and change it? Use the print function to display all items in the list, look for the record and find the desired incorrect record, use the surname stored for that record to search for the record in order to change it.
8. I have logged into the system with the system passcode and I want to change my password, but I don't know it, how do I change it? When you logged in with the system code, the program will have displayed the current password, find this point and then copy the password as it is displayed when you go to change the password
9. I cannot close the program, how do I do it? Navigate your way to the main menu and input "exit" or "e" with the keyboard. Press enter and the program should close
10. Inputting "add", "modify", "clear list" is a bit tedious and takes time, are there quicker ways to do it? See the table of accepted input for the program for the accepted abbreviations of input.

Appendices

Appendix A *Interviews*

During the analysis stage of the project, I used interviews to ask the employees about the paper based system that was being used in the business. This was done because the business is small (with 5 employees) and does not have a high level of customer presence at any one time. Therefore, an interview would be ideal as it can be dynamic and completed in a relatively quick time. Different sets of questions were asked to the employees and the customers.

Interviews conducted have been transcribed below

The questions asked to the employees were as follows:

- Name? forename and surname
- Occupation? Manager of the business or below
- Does the current system obstruct business? If so how? (can expand into technology, operation, economics, scheduling and law)
- Would you like a new database system? Yes/no
- If yes, what would you like in a new system?
- If no, would you like to change the current system? If so then what would you like to change?
- Are there any additional comments?

The following results were acquired from the employers

Employee 1

1. Name: Steve Jones
2. Occupation: Assistant Manager
3. Does the current System obstruct business? “Yes: when we receive a phone call from a customer who wishes to order goods, we have to leave the phone and the customer to go and search for their file; this has become more and more tedious as we have received more customers. When we compare our business to a rival down the road, who has an electronic system, the rival business can serve more customers quicker and also has a much larger customer base than us.“
4. Do you want a new system? “yes”
5. What would you like in a new system? “I would like a computer that I can quickly access customer files. With this I want to be able to add, remove, change and search for the information quickly; but I want it to be easy to use because I do not have the time when serving a customer to mess with lots of tedious options.”
6. Are there any additional comments? “Yes, I would like all of the options on the forms I fill out to be on the new one as I have gotten comfortable with using them.”

Employee 2

1. Name: Peter Smith
2. Occupation: Manager
3. Does the current system obstruct business? “I think so, the paper files are becoming less and less feasible to continue to use. It takes time to look for the forms and fill them out, also when we need to modify a form; we waste paper. This paper is expensive as its costs petrol, time and money to buy. Also the bin-men charge us an additional £50 a month to take away what they consider ‘industrial waste’. The current space we use to store the files safely is almost full I do not have the money to afford an expansion. We have a computer on our desk which is not being used effectively and 3 of my employees have qualifications in IT going to waste. This paper system is costing me money and a shop down the road is getting all the business as a result!”
4. Do you want a new system? “Yes please”
5. What would you like in a new system? “I want a system that can effectively replace the out-dated paper system and use the computer that is gathering dust here. I want it to run cheaply and be able to deal with all of our customer data that we have now. Also I want a cheap system that we can upgrade easily and cheaply in the future if we are to expand”
6. Are there any additional comments? “I want the system to be easy to use as I am not very good with computers.”

Employee 3

1. Name: Mary Kershaw
2. Occupation: Tills and checkout assistant
3. Does the current system obstruct business? “Yes it does. It takes a while to find any files and some of my customers have left or hung up from the phone call. Also when you want to change information it takes time as well. I find the job a bit tedious as I have better

technology in my hand (holds up mobile phone) and am sure that this can be done electronically."

4. Would you like a new database system? "if it helps"
5. What would you like in a new system? "I want to get the customer files quickly and easily. I am sure that the computer is capable of doing this but we need a digital copy to do this. "
6. Are there any additional comments? "No, I think that's everything."

Employee 4

1. Name: James Turner
2. Occupation: Tills and checkout assistant
3. Does the current system obstruct business? "Yeah it does, it takes ages to look for files and I find it a bit boring. Sometimes I have to go and find a customer file and it takes me so long that when I come back, they are gone. Also when I have to fill a form out, it takes a while to file away because we have so many files already."
4. Would you like a new database system? "Yes"
5. If yes, what would you like in a new system? "I want to be able to be able to fill out the information in a quick and easy way. A way in which I can store the file and find it again easily.
6. Are there any additional comments? "no"

Employee 5

1. Name: Jill Ramsey
2. Occupation: tills and checkout assistant and stock
3. Does the current system obstruct business? "Not for me. I spend most of my time doing deliveries for customers, preparation of customer files for waste and stocking the shelves for the shop. When it is busy; I have had to use the forms as customers were queuing up, it is a bit awkward and slow but I have not had any negative feedback from customers. When I have been told to prepare deliveries, I often find myself waiting for my colleagues to find customer files and complete order forms. This is a pain because some customers complain that their pets have gone without food and demand discounts: Which come out of my pay check!"
4. Would you like a new database system? "If it speeds up deliveries"
5. What would you like in a new system? "if would like system to be quicker to use so that the deliveries will be more efficient"
6. Are there any additional comments? "I don't think so"

Appendix B *Sample Data*

The hard copies of the sample data are attached to this page:

Appendix C *User response of prototype*

This appendix contains a report (transcribed from a recording) of the end-user response to the prototype as seen in A3: Initial design

Possible solutions were evaluated above and I feel that a custom built electronic system would be best for this company as it would ensure that their specific needs are catered for. The new system will have the following specifications:

It will be a text-based system so that it is easy for the user to use – using a command line interface.

The system will store customer information in a list which will be sorted at the input of a user command. The data will also be sorted alphabetically by surname every time the user adds/removes/modifies a customer file.

The system will have password protection that will dictate whether the user can use the system. This password will be changeable at a user command.

The system will use the computer in the shop as express interest was placed on this factor.

The system will use object oriented Java (5.0 or above) because it is my strongest programming language and it offers many features which are suitable for the requirements.

The system will have the same fields as the sample data.

The system will run windows 7 as it is the most recent (2012) and most reliable system for a user.

There will be specific commands to “add”, “remove”, “change”, search”, “sort” etc… so that the user can easily (and quickly as it is a computer) alter the database and its data.

A User-guide and documentation will be included so the users can learn how to use the system and fix problems with the system themselves. This will also ensure that technical staff can maintain to new system in the future.

The new system will have the capacity (hard-drive space) of 500 GB so that the user will have plenty of space to store many more customer files.

Custom classes will be used to create custom validation methods so that the system will not crash when there is an error.

User friendly operating system will be used so that the user can competently use the computer and this database. Anti- virus software will be used to ensure that the system itself is protected from cyber threats. Here is a diagram to illustrate this initial design (this serves as a basic model for the prototype where functions such as “sort” are removed for simplicity):

A3: Prototype of new system with aims. The prototype was shown to each user in confidence so that the integrity of what the user has said has remained as reliable as possible.

1. Employer: Peter Smith:
   Response to the prototype: 'I like this system, it is very clear and I'm sure my staff will be able to use it easily. The fact that it is text based is a bit simple but it will save us time so that we do not need to press a lot of buttons. I'm sure the company computer will be able to use this and it should be a lot quicker than our filing cabinet. I have one gripe however: surely it would take less time to just type in 'change' or 'remove' or 'C' or 'R' rather than having to navigate the mouse to press the buttons. Will I need to throw anything away or purchase new paper for this?' 'You will only need to purchase paper for order forms and ink to be able to them.' 'I like that. How can you ensure that the system cannot be hacked by viruses and such?' 'Measures can be put in place to ensure your computer has a secure network such as passwords and anti-virus software.' 'Ok then, I would be happy to use this system, with a little training.'
2. Employee: Steve Jones:
   'That is a very simple way of doing it; I can see myself on the phone and using this easily: the simplicity is good. I think with something like this, business will pick up I will be able to serve customers quicker. Can that little computer in our shop store all of our customer files in the cupboard?' 'Yes easily, and you can get storage upgrades as well' 'that's great! I would like to use this.'
3. Employee: Mary Kershaw:
   'Wow, that is really simple and easy on the eye, I could imagine it would a lot easier to use as well; this should help with the speed at which we serve customers. Will we still need to keep the old customer files?' 'Only until you have copied them to the new system; then you can dispose of them.' 'That's good; maybe we could use the cupboard as a staff area…with a fridge and coffee machine! Is it easy to keep safe?' 'There is the option to put a password on it so that no hackers can steal information and the whole system can be backed up onto an additional hard drive which the manager could keep safe.' 'That's good; I would be happy to use this.'
4. Employee: James Turner:
   'Is this system on the computer?' 'Yes, it will run on the computer so that the computer will store all the customer information' 'I'm guessing that it will be quicker to store and get customer information as a result. I like that. I sometimes forget information so will the information stay on the screen?' 'It will be a text based system, so yes, the screen will display as much information as it can. But you will need to search for the information first as it would not be able to show over 100 customer records at once.' 'Will we be able to search for customer stuff quicker than before?' 'You will be able to search as fast as you can type and the computer can process, so yes quite fast.'
5. Employee: Jill Ramsey:
   'I wouldn't know how to use this as I rarely used to work at the till, will I get training?' 'Yes, there will be a user guide with it so that you can read up on how to use it.' 'Ok, will it speed things up…For deliveries I mean?' 'You will be able to prepare order forms as quickly as you can search for and copy down or print the information and since you will be working at a computer, it will speed things up.' 'Cool, I haven't really used a computer since my O-levels

and so I'm a bit wary that I might do something wrong. Is it safe?' 'The shop should have the necessary fire precautions such as the fire extinguisher and the program itself will be designed so that it doesn't crash when you do something wrong so it should be safe as long as you look after it. You can also look at the user guide to look up problems you may have.' 'Ok then, I am satisfied with that.'

Appendix D *Hard copy of the program source code*

See the next page for the code (note it has been put on pages with a different orientation as some lines of code are quite long) total number of lines of code = 1334