Data Structures & Algorithms Report

# Introduction

In this report I will be comparing data structure approaches to see which is best suited for a data structure system of my choice. The system I have chosen to implement is for a video game pre-order. The system will incorporate three data structures, the first of these is an enhanced bubble sort algorithm to organise user data, the second is a hash table to hold the user data and the third is an operator overload for comparing specific data within a constructor, these structures will be justified later in the report. The report will also look at designing this system using requirements, actor’s lists and UML diagrams.

# System

The video game pre-order system will look to imitate the process of pre-ordering the game FIFA. FIFA has been chosen to imitate because there are a different variations of the game that the user can purchase, the variations include a standard edition, deluxe edition and super deluxe edition. Each pre-order account will be assigned an ID and will have a date, these variables will allow the system to use an enhanced bubble sort algorithm to prioritise the shipping process of the product. Other than recording the user’s edition, Id and date of the pre-order the system will look at recording their email and username and will insert this information into a hash table. The system will be modelled as a prototype this is so that it can demonstrate the pre-order process over a few seconds rather than months.

# Justification

In this section I will be justifying the decision to use specific data structures, also evaluating other data structures and analysing their relevance for the system. When pre-ordering a game a range of user data will need to be stored, this data then needs frequent access and to be sorted in priority order. Due to the functionality of the system I decided that a hash table was the most appropriate data structure to use, it stores data that requires frequent access and allows the system to locate the data. This is relevant as it allows the user to cancel their pre-order by removing their data from the hash table. Another benefit of using a hash table is that you can specify the amount of elements the table contains, this relates to the system as there will be a set limit of game copies available. Hash tables will benefit the performance of the system allowing for fast insertion and searching, this will improve the overall experience for the users of the system.

The shipping process is a core functionality of the system. The user’s product will be shipped based on the date the user pre-ordered the game and ID will be assigned based on the date the users account is created. The system will require a sorting algorithm to carry out the shipping functionality this is because a hash table does not have the ability to visit data in order. Looking at different types of sorting algorithms there are three main approaches that can be taken: selection sort, bubble sort and enhanced bubble sort. The first to look at is selection sort, it organises data by locating the smallest unprocessed element of an array and positioning it in its correct index position, this method would not be best for the system because it has the worst O(N^2) notation and also would require a small amount of elements to maintain a good performance. The second sorting algorithm to analyse is bubble sort, this algorithm involves making repeated passes through a list of items whilst exchanging adjacent items if necessary, for each pass the largest unsorted item is pushed to the correct position. This algorithm will be avoided because it has the worst O(N^2) notation and also isn’t the most efficient as the algorithm attempts to swap elements even when the data is organised. The final sorting algorithm to analyse is enhanced bubble sort, this algorithm works on the same basis as normal bubble sort but has an extra feature. This extra feature enables the algorithm to exit when no change has been made to the inner loop, this results in the algorithm becoming more efficient and allowing the operation to take the O(N^2) notation. In conclusion the best sorting algorithm to use in the system would be enhanced bubble sort, this is based on the good efficiency and performance of the algorithm compared to other methods.

When integrating bubble sort and a hash table another data structure will be needed. This data structure is an operator overload which is treated as a polymorphic function, it enables the system to manipulate the behaviour of an operator based on the current argument. The argument being anticipated is having to compare multiple variables of one account class against multiple variables of another account, the operator overload will allow the system to overcome this by only comparing one variable against another.

# Requirements

## Functional

|  |  |  |
| --- | --- | --- |
| **Number** | **Requirement** | **Use Case** |
| 1. | Store user account details in a data structure | Store Account Details |
| 2. | User can view account details | View Account Details |
| 3. | Sort user account details by edition order | Sort by Edition |
| 4. | Sort user account details by date order | Sort by Date |
| 5. | Allow the user to cancel their pre-order | Cancel Pre-Order |
| 6. | Create a pre-order time frame | Create Time Frame |
| 7. | Confirm shipping with the User | Confirm Shipping |

## Non-Functional

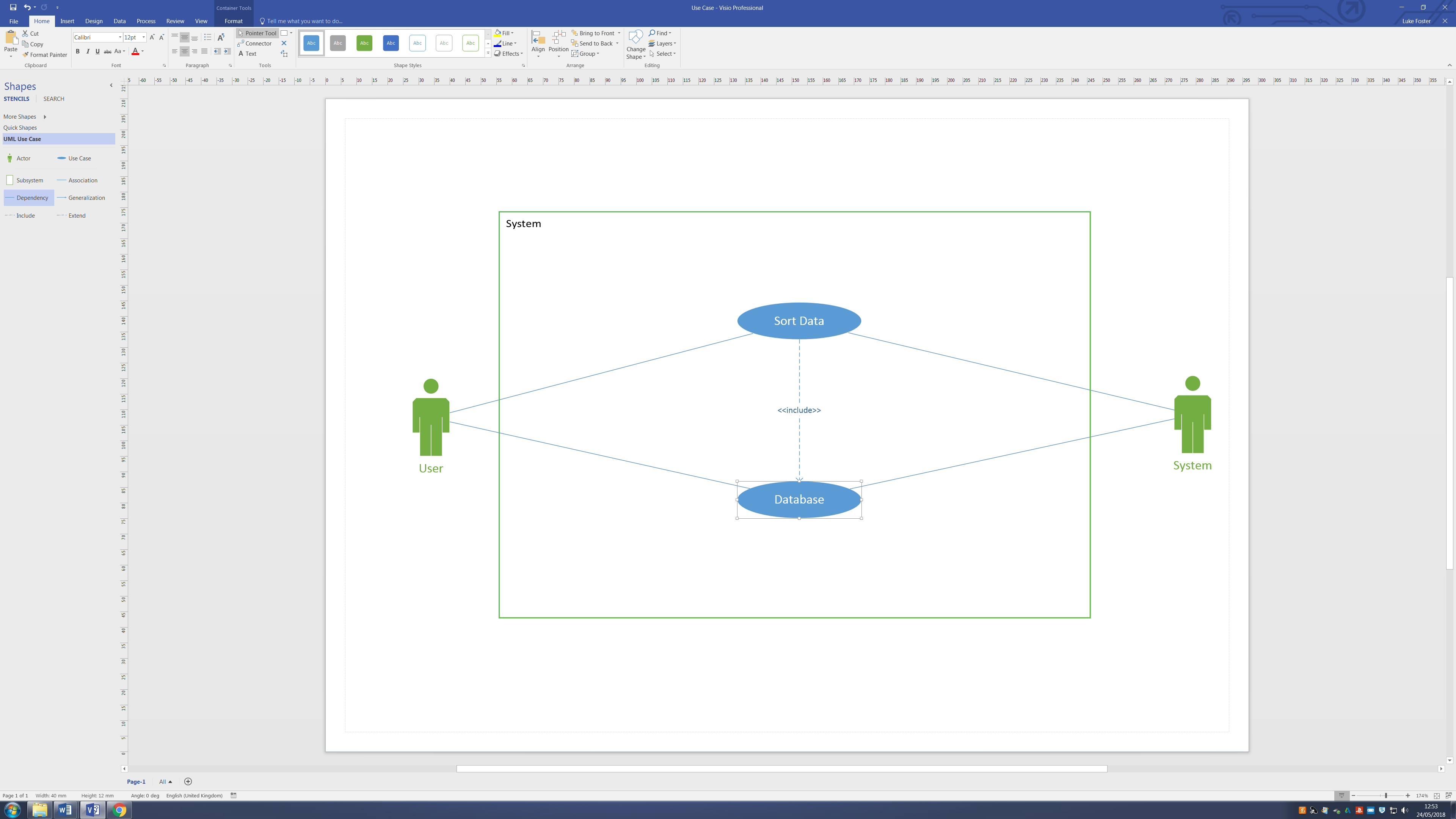
|  |  |  |
| --- | --- | --- |
| **Non-functional Requirements** | **Description** | **Use in system** |
| Readability | The system should be easily readable to the user | Use clear COUT messages giving the user as much information as possible |
| Usability | Allows a strong interaction with the user giving feedback when necessary | Inform the user if their order status has changed |
| Reliability | The system should be reliable not resulting in breaks or errors in the system | Implement error handling so that the system is robust |

## Actor List

|  |  |
| --- | --- |
| **Actor** | **Description** |
| User | A user must be able to enter their relevant details into the system. They must also be able to choose which edition of the game they would like to purchase. The user should be able to view the current status of their pre-order. The user must be able to cancel their per-order any time before shipment. The user should be notified when their order has been shipped. |
| System | The system must be able to take data from the user and insert it into a hash table. The system must organise the data so that it is prioritised in relation to edition and date. The system must enable frequent access to the data. |

# UML

## Use Case



## Use Case High Level Description

|  |  |
| --- | --- |
| **Use Case** | **Description** |
| Add Account | The system prompts the user to enter their account details and stores them within a database |
| View Accounts | The system displays a list of accounts to the user |
| Cancel Pre-Order | The system prompts the user whether they would like to cancel their order and responds accordingly in regards to the database |
| Shipment | The systems informs the user that their pre-order is on its way |

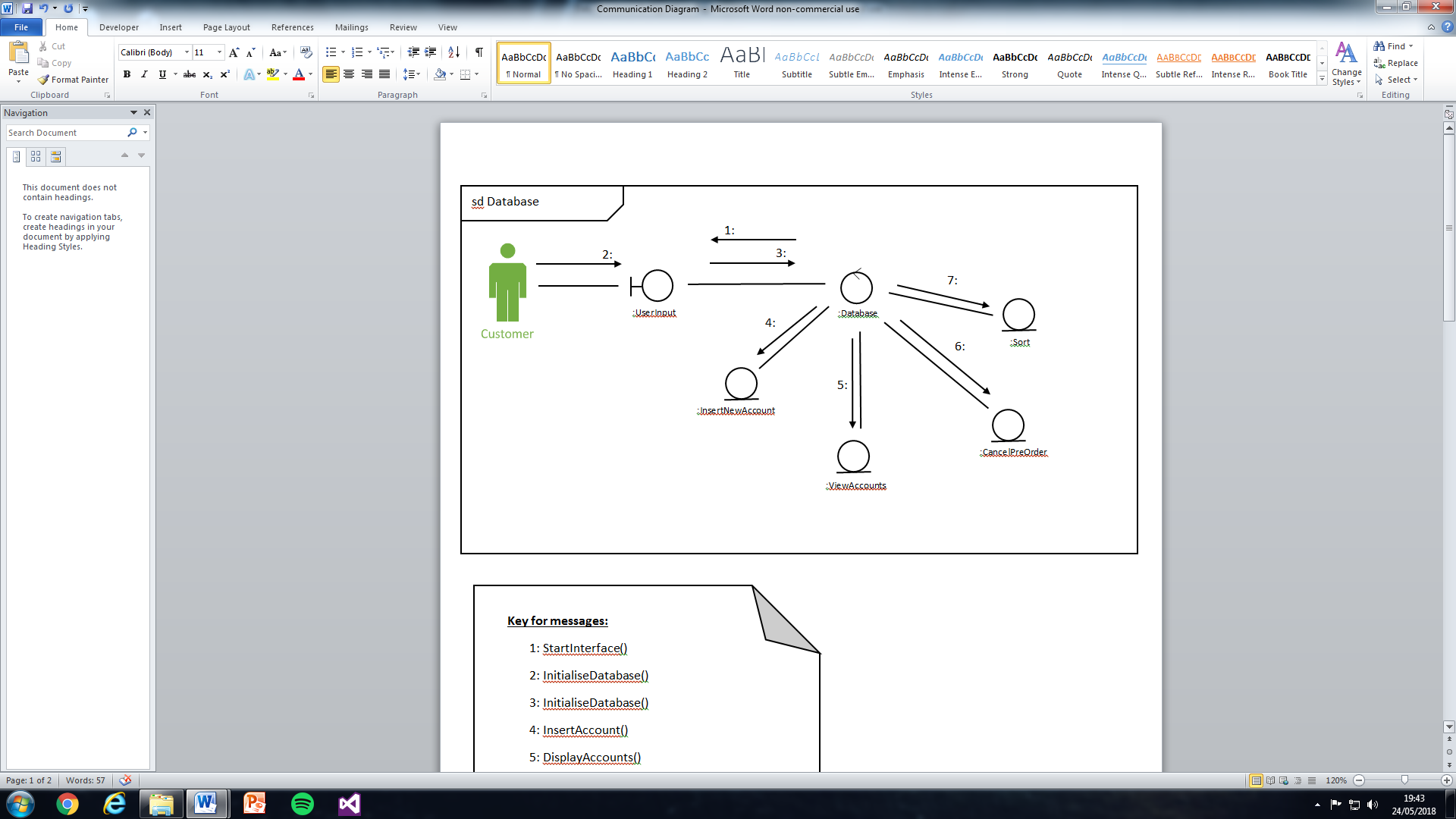
## Use Case Low Level Description

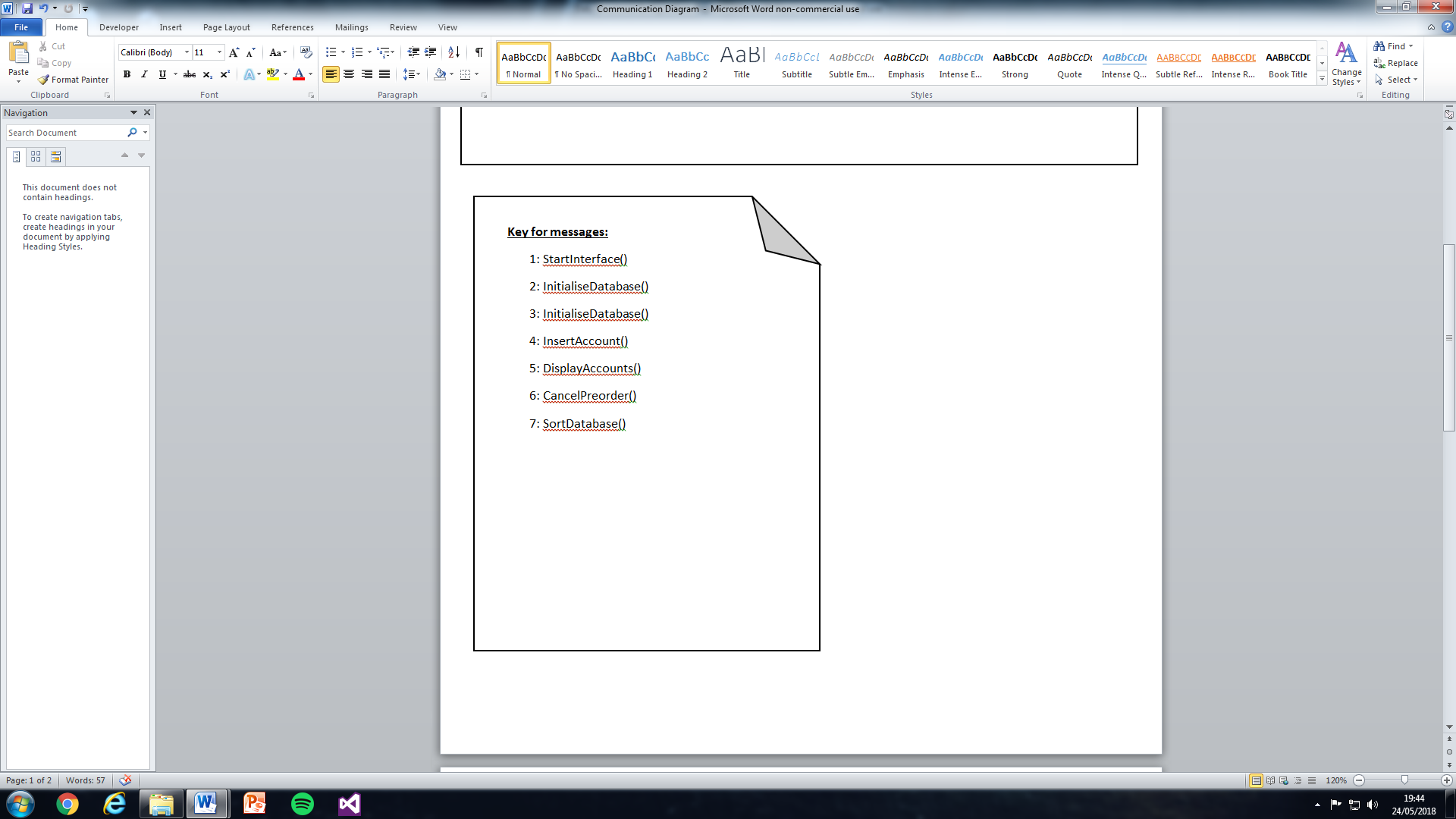
|  |  |
| --- | --- |
| **User Action** | **System Response** |
| 1. Input Account Details | 2. Initialise Database |
| 3. Nothing | 4. Insert New Account |
| 5. View Account | 6. Sort Database |
| 7. Cancel PreOrder (Y/N) | 8. Cancel PreOrder |

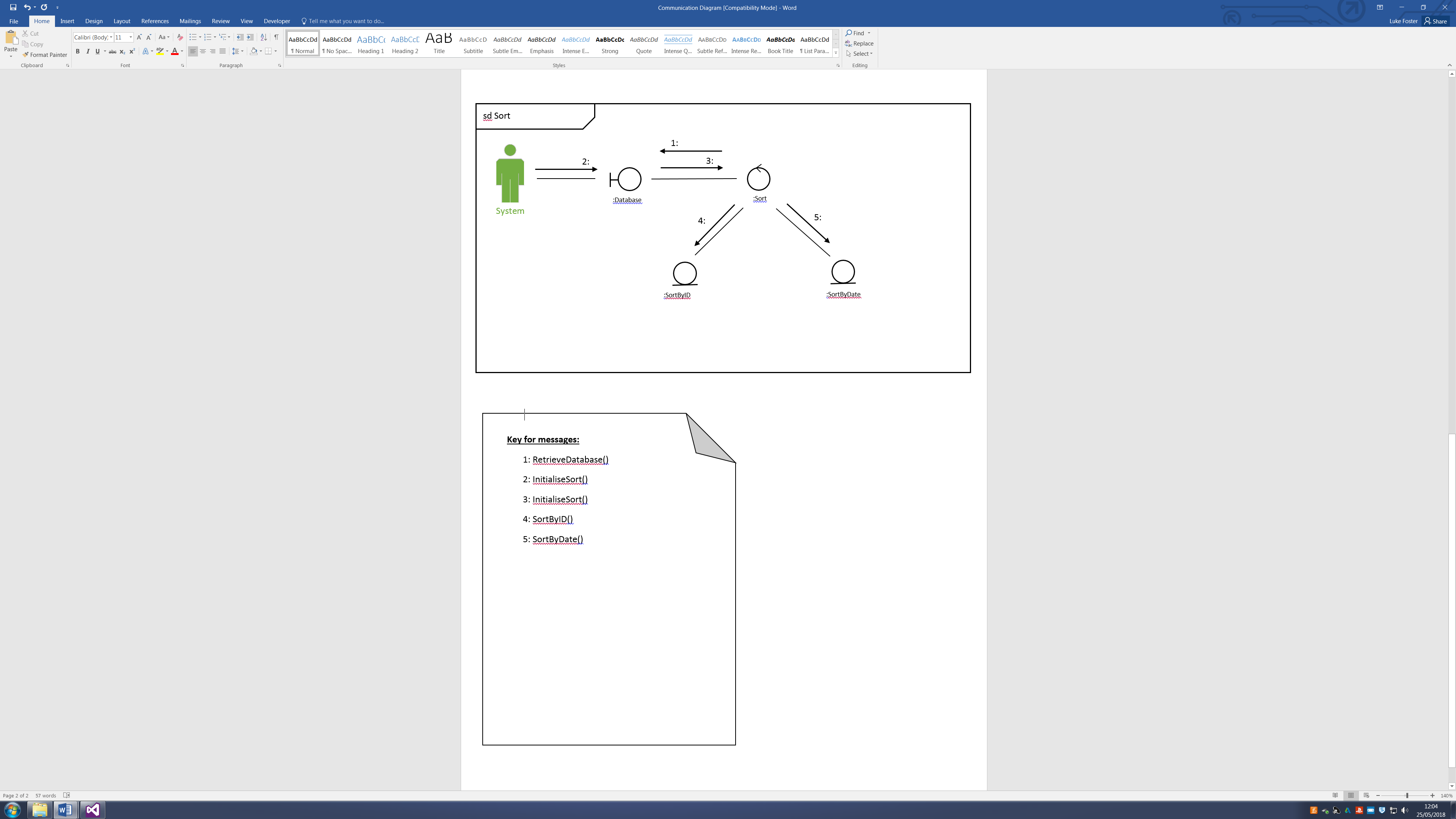
## Collaboration



## Communication







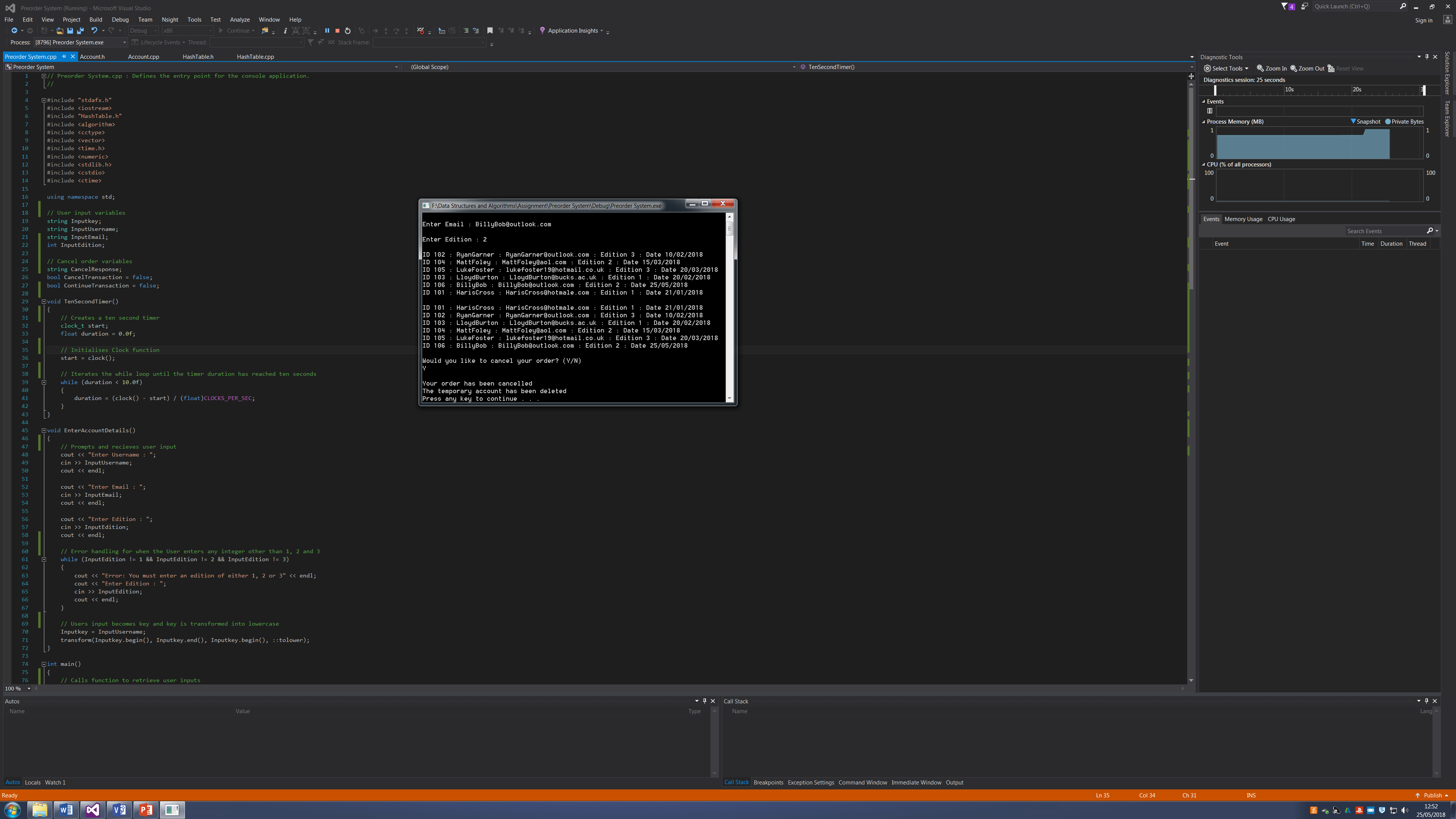
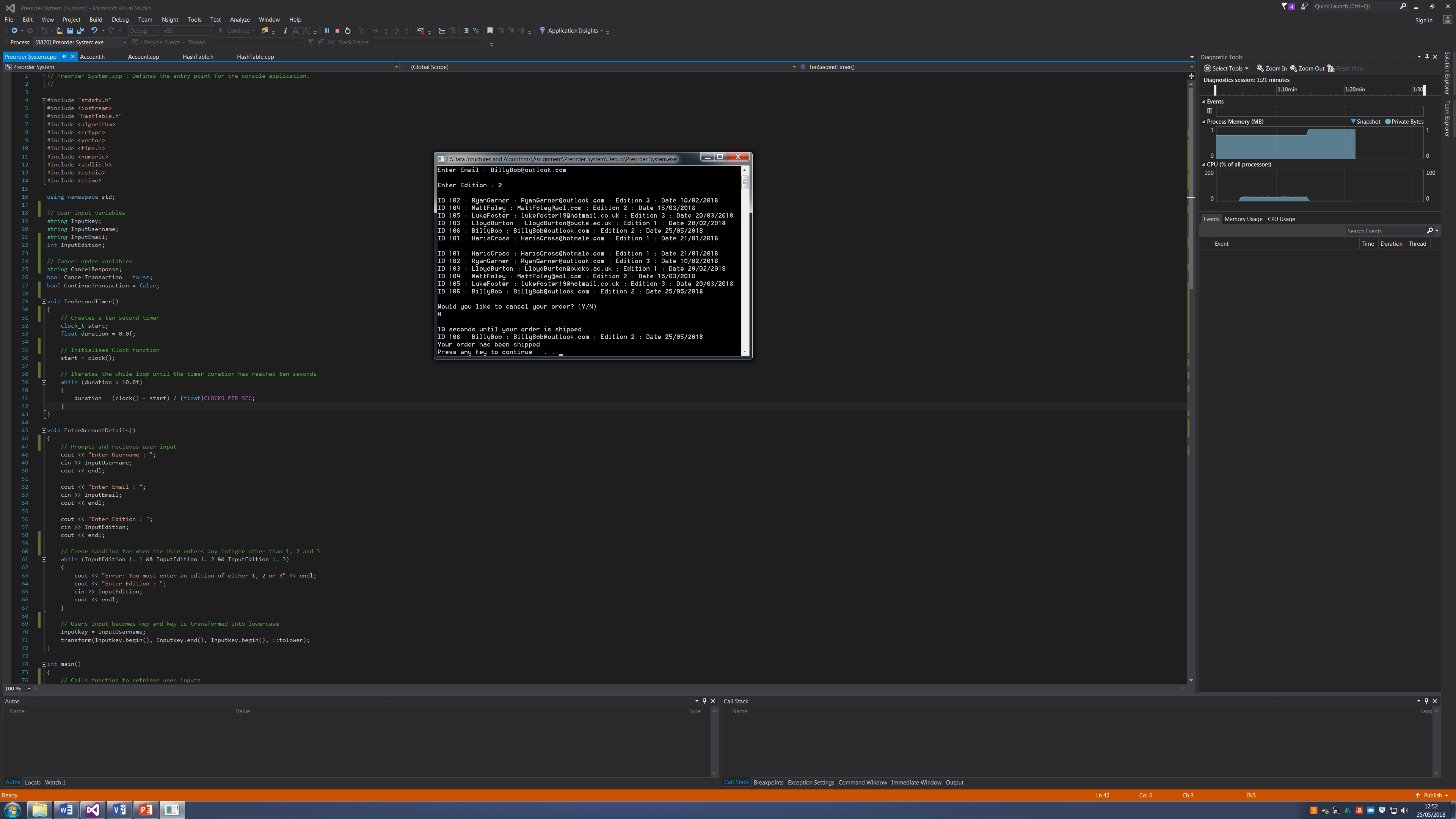
## Sub-Class

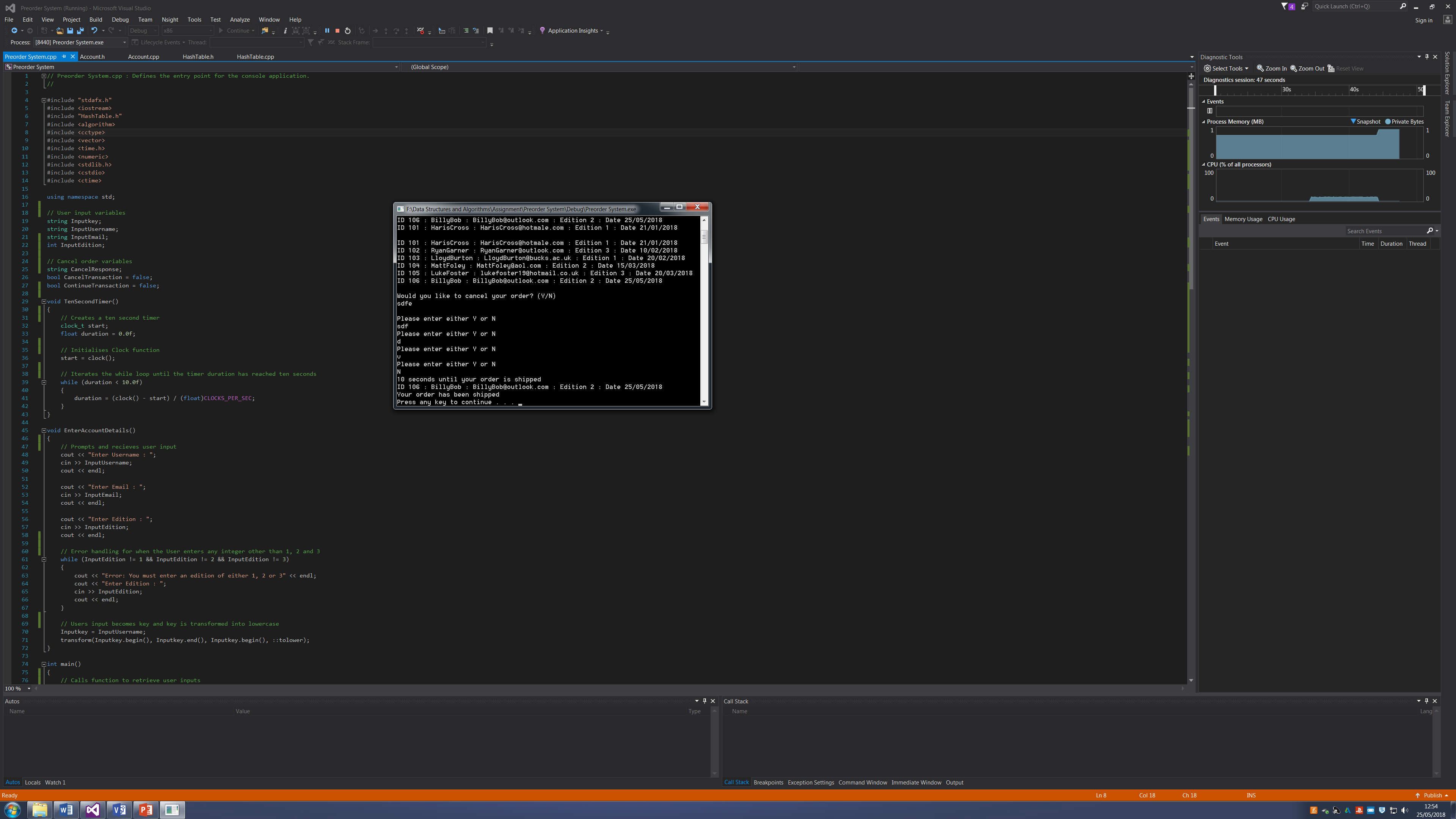
### Account Sub-Class

### Sort Sub-Class

## Class

## Screenshots





## Scripts

### Preorder System.cpp

// Preorder System.cpp : Defines the entry point for the console application.

//

#include "stdafx.h"

#include <iostream>

#include "HashTable.h"

#include <algorithm>

#include <cctype>

#include <vector>

#include <time.h>

#include <numeric>

#include <stdlib.h>

#include <cstdio>

#include <ctime>

using namespace std;

// User input variables

string Inputkey;

string InputUsername;

string InputEmail;

int InputEdition;

// Cancel order variables

string CancelResponse;

bool CancelTransaction = false;

bool ContinueTransaction = false;

void TenSecondTimer()

{

// Creates a ten second timer

clock\_t start;

float duration = 0.0f;

// Initialises Clock function

start = clock();

// Iterates the while loop until the timer duration has reached ten seconds

while (duration < 10.0f)

{

duration = (clock() - start) / (float)CLOCKS\_PER\_SEC;

}

}

void EnterAccountDetails()

{

// Prompts and recieves user input

cout << "Enter Username : ";

cin >> InputUsername;

cout << endl;

cout << "Enter Email : ";

cin >> InputEmail;

cout << endl;

cout << "Enter Edition : ";

cin >> InputEdition;

cout << endl;

// Error handling for when the User enters any integer other than 1, 2 and 3

while (InputEdition != 1 && InputEdition != 2 && InputEdition != 3)

{

cout << "Error: You must enter an edition of either 1, 2 or 3" << endl;

cout << "Enter Edition : ";

cin >> InputEdition;

cout << endl;

}

// Users input becomes key and key is transformed into lowercase

Inputkey = InputUsername;

transform(Inputkey.begin(), Inputkey.end(), Inputkey.begin(), ::tolower);

}

int main()

{

// Calls function to retrieve user inputs

EnterAccountDetails();

// Creates an instance of HashTable and assigns a maximum of 10 accounts

HashTable \* table = new HashTable(10);

// Predefined accounts are inserted into the hash table

table->Insert(new Account("hariscross", "HarisCross", 101, "HarisCross@hotmale.com", 1, 21, 1));

table->Insert(new Account("ryangarner", "RyanGarner", 102, "RyanGarner@outlook.com", 3, 10, 2));

table->Insert(new Account("lloydburton", "LloydBurton", 103, "LloydBurton@bucks.ac.uk", 1, 20, 2));

table->Insert(new Account("mattfoley", "MattFoley", 104, "MattFoley@aol.com", 2, 15, 3));

table->Insert(new Account("lukefoster", "LukeFoster", 105, "lukefoster19@hotmail.co.uk", 3, 20, 3));

// Users inputs are inserted into a new account

table->Insert(new Account(Inputkey, InputUsername, 106, InputEmail, InputEdition, 25, 05));

// All accounts are displayed in the natural hash table order

table->Display();

cout << endl;

// Calls the Sort function to enhance bubble sort the hash table

table->Sort();

// Displays the newly sorted hash table

table->Display();

cout << endl;

// Prompts and recieves users decision to cancel their order

cout << "Would you like to cancel your order? (Y/N)" << endl;

cin >> CancelResponse;

cout << endl;

// Checks Users response and will continously iterate for a new response if response is not y or n

while(CancelTransaction == false && ContinueTransaction == false)

{

if (CancelResponse == "n" || CancelResponse == "N")

{

// Breaks while loop

ContinueTransaction = true;

}

else if (CancelResponse == "y" || CancelResponse == "Y")

{

// Breaks while loop

CancelTransaction = true;

}

else

{

cout << "Please enter either Y or N" << endl;

cin >> CancelResponse;

}

}

if (ContinueTransaction == true)

{

cout << "10 seconds until your order is shipped" << endl;

// Calls for a ten second timer

TenSecondTimer();

// Finds and displays the users order from within hash table

table->Find(Inputkey);

cout << "Your order has been shipped" << endl;

}

if (CancelTransaction == true)

{

cout << "Your order has been cancelled" << endl;

// Deletes the users account from the hash table

table->Delete(Inputkey);

cout << "The temporary account has been deleted" << endl;

}

system("PAUSE");

return 0;

}

### Account.h

#pragma once

#include "string"

using namespace std;

class Account

{

public:

string key;

string Username;

int ID;

string Email;

int Edition;

int Day;

int Month;

// Account constructor that defines the Account variables

Account(string key, string Username, int ID, string Email, int Edition, int Day, int Month)

{

this->key = key;

this->Username = Username;

this->ID = ID;

this->Email = Email;

this->Edition = Edition;

this->Day = Day;

this->Month = Month;

}

};

### Account.cpp

#include "stdafx.h"

#include "Account.h"

### HashTable.h

#pragma once

#include <iostream>

#include <cstring>

#include <string>

#include "Account.h"

using namespace std;

class HashTable

{

private:

// Variables

int size;

Account \*\* data;

public:

// Functions

HashTable(int size);

int HashFunction(string key);

Account \* Find(string key);

Account \* Delete(string key);

void Insert(Account \*item);

void Display();

void Sort();

};

### HashTable.cpp

#include "stdafx.h"

#include "HashTable.h"

// Operator overloading > so that the sort function sorts by Account ID's

bool operator>(const Account& left, const Account right)

{

return left.ID > right.ID;

}

// Operator overloading < so that the sort function sorts by Account Date in Months

bool operator<(const Account& left, const Account right)

{

return left.Month < right.Month;

}

HashTable::HashTable(int size)

{

this->size = size;

// Defines data as a new set of accounts

data = new Account\*[size];

for (int n = 0; n < size; n++)

{

data[n] = NULL;

}

}

int HashTable::HashFunction(string key)

{

int sum = 0;

for (int n = 0; n < key.length(); n++)

{

// Increments sum based on key

sum += (int)key[n];

}

// Returns the size of the hash table

return sum % size;

}

Account \* HashTable::Find(string key)

{

int hash = HashFunction(key.c\_str());

// Iterates comparing Accounts to the key until no more Accounts remain

while (data[hash] != NULL)

{

// Compares an Account key with key passed to the Find function

if (data[hash]->key == key)

{

// Outputs the Account data found

cout << "ID " << data[hash]->ID << " : " << data[hash]->Username << " : " << data[hash]->Email << " : Edition " << data[hash]->Edition << " : Date " << data[hash]->Day << "/0" << data[hash]->Month << "/2018"<< endl;

return data[hash];

}

// Moves onto next Account

hash++;

hash %= size;

}

return NULL;

}

void HashTable::Insert(Account \* item)

{

// Assigns key to new Account

string key = item->key;

// Calls to the hash function

int hash = HashFunction(key.c\_str());

while (data[hash] != NULL)

{

++hash;

hash %= size;

}

// Assigns new item to the correct data hash location

data[hash] = item;

}

Account \* HashTable::Delete(string key)

{

// Calls to hash function

int hash = HashFunction(key.c\_str());

while (data[hash] != NULL)

{

// Compares to see if key passed to function mataches a key in the hash table

if (data[hash]->key == key)

{

Account \* temp = data[hash];

// Deletes account with matching key

data[hash] = NULL;

// Returns a temp account to use to inform the user of the deleted account

return temp;

}

// Moves onto next account to compare

++hash;

hash %= size;

}

return NULL;

}

void HashTable::Display()

{

for (int n = 0; n < size; n++)

{

if (data[n] != NULL)

{

// Displays the hash table

cout << "ID " << data[n]->ID << " : " << data[n]->Username << " : " << data[n]->Email << " : Edition " << data[n]->Edition << " : Date " << data[n]->Day << "/0" << data[n]->Month << "/2018" << endl;

}

}

}

void HashTable::Sort()

{

bool sorted = false;

// Iterates comparing each account until they have all been compared

for (int i = 0; i < size; i++)

{

// Sorted modifys the algorithm to exit when no change needs to be made in the inner loop

if (sorted);

sorted = true;

for (int j = 0; j < (size - 1); j++)

{

// Swaps to sort ID using the > operator overloader

if (data[j] > data[j + 1])

{

Account\* temp = data[j + 1];

data[j + 1] = data[j];

data[j] = temp;

sorted = false;

}

// Swaps to sort Date using the < operator overloader

if (data[j + 1] < data[j])

{

Account\* temp = data[j + 1];

data[j + 1] = data[j];

data[j] = temp;

sorted = false;

}

}

}

}