

CMP7032 - Systems Development

CWRK001 - Assessment Report

Assessment report for MSc Computing,
Post Graduate Systems Development
module.

S12763849

Table of Contents

TABLE OF FIGURES	3
FUNCTIONS OF PROGRAM	4
READING FROM FILE USING STREAMREADER	4
SEPARATING OUT DISTINCT LINES FROM DATA FILE	4
STORING DATA READ IN ARRAYLIST	5
PASSING DATA TO METHODS	5
CALCULATIONS	6
<i>Calculate the hours of normal pay</i>	6
<i>Calculate the hours of overtime pay</i>	7
<i>Calculate the basic pay earned</i>	7
<i>Calculate the overtime pay earned</i>	8
<i>Calculate the Gross Pay Earned</i>	8
<i>Calculate the amount of pay taxed at the higher rate</i>	9
<i>Calculate the amount of pay taxed at the lower rate</i>	9
<i>Calculate Higher Rate Tax</i>	9
<i>Calculate Lower Rate Tax</i>	10
<i>Calculate the gross tax payable</i>	10
<i>Calculate the net pay for each employee</i>	10
OUTPUT THE FINAL WAGES FOR EACH INDIVIDUAL INCLUDING, GROSS PAY, TAX PAYABLE AND NET PAY	10
SEPARATE OUT EACH INDIVIDUAL BASED ON THEIR DEPARTMENT (FROM DEPARTMENT CODE)	11
OUTPUT THE REQUIRED FOLLOWING DATA TO SEPARATE AREAS ON THE SAME DISPLAY FOR THE FOLLOWING;	11
CALCULATE THE CUMULATIVE NET PAY FOR EACH DISCRETE DEPARTMENT	11
CALCULATE THE CUMULATIVE NET PAY AND TAX PAYABLE FOR THE COMPANY AS A WHOLE	12
GET VARIABLES AND VALUES PRODUCED FROM METHODS TO OUTPUT RECORDS	12
CRITICAL ANALYSIS OF DATA STRUCTURES	13
ABSTRACT	13
ARRAYS	13
LISTS	14
LINKED LISTS	14
SELECTION AND JUSTIFICATION	14
ANNOTATION OF COMPLETED PROGRAM	15
PROGRAM.CS	15
DEPARTMENT.CS	16
EMPLOYEE.CS	21
CRITICAL ANALYSIS OF TESTING APPROACHES	27
BLACK BOX TESTING	27
WHITE BOX TESTING	27
WHITE BOX TESTING WITH TEST RESULTS	29
FLYEASY PAYROLL PROGRAM TESTING	29
<i>Test Data</i>	29
<i>Test Output for Preliminary Test</i>	30

<i>Single Line Data Tests (White Box)</i>	31
<i>Test Data 1</i>	31
<i>Test Outputs for Test 1</i>	32
<i>Test Table for all Data Sets</i>	39
<i>Test Data 2</i>	39
<i>Selected Test Outputs for Test Data 2</i>	39
<i>Test Data 3</i>	40
<i>Selected Test Outputs for Test Data 3</i>	41
<i>Test Data 4</i>	43
<i>Selected Test Outputs for Test Data 4</i>	43
<i>Test Data 5</i>	44
<i>Selected Test Outputs for Test Data 5</i>	44
<i>Test Data 6</i>	46
<i>Selected Test Outputs for Test Data 6</i>	46
BIBLIOGRAPHY	48

Table of Figures

Figure 1: Set of ten employees all formatted correctly using capital letters where necessary.....	29
Figure 2: Full, successful test result for ten employees from full test data file.....	30
Figure 3: Single employee from the Sales department with codes in capital letters.	31
Figure 4: Testing If/Else Statement.....	32
Figure 5: Testing Entry into correct If Statement	32
Figure 6: Testing Gross Pay Call	33
Figure 7: Testing Correct Values in Call	34
Figure 8: Testing Correct Values in Call	35
Figure 9: Testing Correct Values Out of Call	36
Figure 10: Testing Correct Values Out of Call	36
Figure 11: Testing Correct Values from Call.....	37
Figure 12: Testing Correct Path to If/Else Statement	37
Figure 13: Testing Correct Decision into If/Else Statement	38
Figure 14: Single employee from the Operations department with codes on capital letters.	39
Figure 15: Testing Correct Values Out of Call	39
Figure 16: Testing Correct Entrance into If/Else Statement	40
Figure 17: Testing Correct Values out of Call.....	40
Figure 18: Single employee from the Marketing department with codes in capital letters.....	40
Figure 19: Testing Correct Entrance to If/Else Statement	41
Figure 20: Testing Correct Values Out of Call	41
Figure 21: Testing Correct Values Out of Call	42
Figure 22: Single employee from the Sales department with codes in lower case.	43
Figure 23: Testing Correct Values Out of Call	43
Figure 24: Testing Correct Values Out of Call	43
Figure 25: Single employee from the Marketing department with codes in lower case.....	44
Figure 26: Testing Correct Values Out of Call	44
Figure 27: Testing Correct Entrance into If/Else Statement	44
Figure 28: Testing Correct Entrance to If/Else Statement	45
Figure 29: Single employee from the Operations department with codes in lower case.	46
Figure 30: Testing Correct Values Out of Call	46
Figure 31: Testing Correct Value Out of Call.....	47

Functions of Program

Reading from File using StreamReader

The first process the program carries out is to open a stream reader and read the identified text file. It does this with the use of a while loop, and while there are readable lines of data within the file it reads them, once there are no further lines of readable data in the file it stops reading the designated file and via the `tr.Close` command closes the stream.

If there are any reasons why the file cannot be read, such as an incorrect file name or location the try look will catch this and output an exception message, stating that an exception has been caught.

```
//-----  
// Reading Data from file  
//-----  
public void readEmployeeDataFile(string fileName)  
{  
    string line;  
    int lineCount = 0;  
  
    try  
    {  
        // Create a reader and open the file  
        TextReader tr = new StreamReader(fileName);  
        while ((line = tr.ReadLine()) != null)  
        {  
            lineCount = lineCount + 1;  
            // separates out the required parts of the data file  
            string id = line.Substring(0, 4);  
            string surname = line.Substring(4, 20);  
            string forename = line.Substring(24, 20);  
            string deptCode = line.Substring(44, 1);  
            string grade = line.Substring(45, 1);  
            string hoursWorked = line.Substring(47, 2);  
  
            // passes data for each separated out individual into  
            Array List  
            Employee newEmployee = new Employee(id, surname,  
forename, deptCode, grade, Convert.ToDouble(hoursWorked));  
            Employee.Add(newEmployee);  
        }  
  
        // Close the Stream  
        tr.Close();  
    }  
    // Exception in case there are problems reading the file name  
    catch (Exception e)  
    {  
        Console.WriteLine("Exception Caught While Reading File");  
        Console.WriteLine(e.Message);  
    }  
}
```

Separating out distinct lines from data file

Whilst being read by the stream reader, the program uses the substring command to separate out each like of data into its composite parts. These are defined by the starting position within each line and the amount of characters in this section; i.e. (0, 4) means starting at the 0 character in the line

with a character length of 4 characters, or (44, 1) means starting at the 44th character in the line with a character length of 1 character, and so on.

```
// separates out the required parts of the data file
string id = line.Substring(0, 4);
string surname = line.Substring(4, 20);
string forename = line.Substring(24, 20);
string deptCode = line.Substring(44, 1);
string grade = line.Substring(45, 1);
string hoursWorked = line.Substring(47, 2);
```

Storing Data Read in ArrayList

The data read from the designated file is then stored in an Array List, for future use by the methods within the program. It is stored as a new Employee within the Array of this name

```
// passes data for each separated out individual into Array List
Employee newEmployee = new Employee(id, surname,
forename, deptCode, grade, Convert.ToDouble(hoursWorked));
Employee.Add(newEmployee);
```

AND

```
// The array to store employee deatails
private ArrayList Employee = new ArrayList();
```

Passing Data to methods

Once the required data is stored in the Array List, it is then called into the Employee class, where it is actioned upon as detailed later via various calculations methods. However in order for this to be possible, it needs to be turned into class member variables. The program does this by means of a constructor which takes the variables read by the stream reader and stored in the Array List and instigated new class member variables.

```
//-----
// Variables read from file
//-----

string id;
string surname;
string forename;
string deptCode;
string grade;
double hoursWorked;

//-----
// Default Constructor
//-----

public Employee ()
{
    id = "";
    surname = "";
    forename = "";
    deptCode = "";
    grade = "";
    hoursWorked = 0;
}

//-----
// Normal Constructor
```

```
//-----
public Employee(string id, string surname, string forename, string
deptCode, string grade, double hoursWorked)
{
    // Loading Parameters to match class memeber variables
    this.id = id;
    this.surname = surname;
    this.forename = forename;
    this.deptCode = deptCode;
    this.grade = grade;
    this.hoursWorked = Convert.ToDouble(hoursWorked);
}
}
```

Calculations

The class member variables once created can be used in the following methods to perform the calculations, some returning integer results and others returning new variables which are be used in further calculations to produce outputs. Please note, all methods that return accounting values, i.e. doubles, use the `Math.Round` calculation within them to return the value to two decimal places only.

Calculate the hours of normal pay

Firstly by using an if/else statement, the program has a method that will calculate the hours worked that can be classed as 'normal hours' which will be used to calculate the amount of pay that is paid at the normal basic pay rate. It also uses the `Math.Min` command to calculate this based on a minimum required working hours.

```
//-----
// Method to return normal hours
//-----
public double calcNormalHours()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Min(hoursWorked, 45);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Min(hoursWorked, 45);
    }
    else if (grade == "C" || grade == "c")
    {
        return Math.Min(hoursWorked, 40);
    }
    else if (grade == "D" || grade == "d")
    {
        return Math.Min(hoursWorked, 40);
    }
    else if (grade == "E" || grade == "e")
    {
        return Math.Min(hoursWorked, 37);
    }
    else
    {
        return 0;
    }
}
}
```

Calculate the hours of overtime pay

The program will then use a similar combination of if/else statements and the `Math.Max` command to calculate the hours worked that are over the required basic rate hours. This method will then return the amount of hours worked that are payable at the overtime pay rate.

```
//-----  
// Method to return Overtime Hours  
//-----  
public double calcOvertimeHours()  
{  
    if (grade == "A" || grade == "a")  
    {  
        return Math.Max(hoursWorked - 45, 0);  
    }  
    else if (grade == "B" || grade == "b")  
    {  
        return Math.Max(hoursWorked - 45, 0);  
    }  
    else if (grade == "C" || grade == "c")  
    {  
        return Math.Max(hoursWorked - 40, 0);  
    }  
    else if (grade == "D" || grade == "d")  
    {  
        return Math.Max(hoursWorked - 40, 0);  
    }  
    else if (grade == "E" || grade == "e")  
    {  
        return Math.Max(hoursWorked - 37, 0);  
    }  
    else  
    {  
        return 0;  
    }  
}
```

Calculate the basic pay earned

The program will then take the hours returned by the previous method as normal rate hours, to calculate the basic pay of each employee. Again this method uses if/else statements to differentiate the pay rates based on the grade of each employee that has been passed to it in the class member variables.

```
//-----  
// Method to return basic Pay  
//-----  
public double calcBasicPay()  
{  
    if (grade == "A" || grade == "a")  
    {  
        return Math.Round(calcNormalHours() * 3.00, 2);  
    }  
    else if (grade == "B" || grade == "b")  
    {  
        return Math.Round(calcNormalHours() * 3.50, 2);  
    }  
    else if (grade == "C" || grade == "c")  
    {  
        return Math.Round(calcNormalHours() * 4.00, 2);  
    }  
    else if (grade == "D" || grade == "d")
```



```

    {
        return Math.Round(calcNormalHours() * 4.00, 2);
    }
    else if (grade == "E" || grade == "e")
    {
        return Math.Round(calcNormalHours() * 4.50, 2);
    }
    else
    {
        return 0;
    }
}

```

Calculate the overtime pay earned

This method again is nearly identical to that to work out the basic pay amounts by using a series of if/else statements and the values passed to it from the previous overtime hours' method.

```

//-----
// Method to return overtime
//-----
public double calcOvertimePay()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Round(calcOvertimeHours() * 1.10, 2);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Round(calcOvertimeHours() * 1.25, 2);
    }
    else if (grade == "C" || grade == "c")
    {
        return Math.Round(calcOvertimeHours() * 1.25, 2);
    }
    else if (grade == "D" || grade == "d")
    {
        return Math.Round(calcOvertimeHours() * 1.50, 2);
    }
    else if (grade == "E" || grade == "e")
    {
        return Math.Round(calcOvertimeHours() * 1.50, 2);
    }
    else
    {
        return 0;
    }
}

```

Calculate the Gross Pay Earned

The program then uses a method to calculate the gross pay for each employee based on the values passed through to it from the previous methods.

```

//-----
// Method to Gross Pay
//-----
public double calcGrossPay()
{
    return Math.Round(calcBasicPay() - calcOvertimePay(), 2);
}

```

Calculate the amount of pay taxed at the higher rate

Once the gross pay is calculated a new method takes the value returned to work out the amount of this gross pay that is taxable at the higher rate. It does this via another if else statement, asking if the gross pay is above £100 and then returning a value to state how much of the gross pay can be taxed at the higher rate. Note this does not return a value, merely a new variable that can be then passed to a further method to calculate the accounting value itself.

```
//-----  
// Method to return Ammount taxable at Higher Rate  
//-----  
public double taxableHigherRate()  
{  
    if (calcGrossPay() > 100)  
    {  
        return Math.Round(calcGrossPay() - 100, 2);  
    }  
    else  
    {  
        return 0;  
    }  
}
```

Calculate the amount of pay taxed at the lower rate

Similarly the taxable amount variable for the lower rate of tax is created through this method and a series of if/else statements.

```
//-----  
// Method to return Ammount taxable at Lower Rate  
//-----  
public double taxableLowerRate()  
{  
    if (calcGrossPay() > 100)  
    {  
        return 80;  
    }  
    else if (calcGrossPay() <= 100)  
    {  
        return Math.Round(calcGrossPay() - 20, 2);  
    }  
    else  
    {  
        return 0;  
    }  
}
```

Calculate Higher Rate Tax

Once the variable for the tax payable at the higher rate is returned, this method calculates the actual accounting value for each employee's higher rate of tax; where applicable.

```
//-----  
// Method to return Higher Rate Tax  
//-----  
public double higherRateTax()  
{  
    return Math.Round(taxableHigherRate() * 0.15, 2);  
}
```

Calculate Lower Rate Tax

Similarly from the variable derived for the amount of lower rate tax to be paid, this method then returns the accounting value for each employee's lower rate of tax to be paid.

```
//-----  
// Method to return Lower Rate Tax  
//-----  
public double lowerRateTax()  
{  
    return Math.Round(taxableLowerRate() * 0.1, 2);  
}
```

Calculate the gross tax payable

From the two values produces in the above two methods, this method in the program then calculates the actual amount of gross tax to be paid.

```
//-----  
// Method to return Gross Tax Payable  
//-----  
public double calcGrossTaxPayable()  
{  
    return Math.Round(higherRateTax() + lowerRateTax(), 2);  
}
```

Calculate the net pay for each employee

The program then follows on from the calculation method to calculate gross pay, and along with the method that produces tax payable. This method within the program then calculates the net pay for each employee by a simple subtraction calculation.

```
//-----  
// Method to return Net Pay  
//-----  
public double calcNetPay()  
{  
    return Math.Round(calcGrossPay() - calcGrossTaxPayable(), 2);  
}
```

Output the final wages for each individual including, gross pay, tax payable and net pay

Returning to the Department class within the program, there are a series of methods which get information from the calculations performed within the employee class, to return values for each employee and department, before they are passed to the Program class to be output to the screen.

As shown below, in the Marketing example, there is a method to produce the final wages for each department. This method returns values to the variable that is then called from the Program class. It does this with the use of a for loop. The program will step through each record in the Array List of variables that have been passed to the Employee methods earlier. The program class method then calls the results for the calculations, to return all of the required fields for the report.

```
public string showEmployeePayMarketing()  
{  
    string id = "";  
    string surname = "";  
    string forename = "";  
    string deptCode = "";  
    string grade = "";
```

```

double hoursWorked = 0;
double grossPay = 0.0;
double taxPayable = 0.0;
double netPay = 0.0;

// step through each employee
for (int i = 0; i < Employee.Count; i++)
{
    Employee nextEmployee = (Employee)Employee[i];
    id = nextEmployee.returnId();
    surname = nextEmployee.returnSurname();
    forename = nextEmployee.returnForename();
    deptCode = nextEmployee.returnDept();
    grade = nextEmployee.returnGrade();
    hoursWorked = nextEmployee.returnHoursWorked();
    grossPay = nextEmployee.calcGrossPay();
    netPay = nextEmployee.calcNetPay();
    taxPayable = nextEmployee.calcGrossTaxPayable();
}

```

Separate out each individual based on their department (from Department Code)

These methods then use an if/else statement to isolate each record based on its department code.

```

// Write the outputs to screen
    if (deptCode == "M" || deptCode == "m")
    {
        Console.WriteLine(id + "\t" + surname + "\t" + forename
+ "\t" + grade + "\t" + hoursWorked + "\t" + grossPay + "\t" + taxPayable +
"\t" + netPay);
    }

```

Output the required following data to separate areas on the same display for the following;

These above methods then produce the values required, i.e. ID, Surname, Forename, Grade, Hours Worked, Gross Pay, Tax Payable and Net Pay. This can be seen above.

Calculate the cumulative Net Pay for each discrete department

The next method in the Department Class then uses a for loop to loop through all employees in the Array List to output a departmental total for each employee with an if/else statement used to isolate each employee dependant on their department code.

```

public void showMarketingTotal ()
{
    double marketingNetPay = 0.0;
    string deptCode = "";

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        // get the next employee
        Employee nextEmployee = (Employee)Employee[i];

        // find the department code value
        deptCode = nextEmployee.returnDept();
        // add the next net pay item to the departmental total (for
each matching variable)
        if (deptCode == "M" || deptCode == "m")

```

```

        {
            marketingNetPay = marketingNetPay +
nextEmployee.calcNetPay();
        }
    }

    // write the total net pay and tax payable for the department
    Console.WriteLine("\t" + "\t" + "\t" + "\t" + "\t" + "\t" +
"\t" + "\t" + "\t" + "\t" + "\t" + "-----");
    Console.WriteLine("Total for: Marketing" + "\t" + "\t" + "\t" +
"\t" + "\t" + "\t" + "\t" + "\t" + " £" + marketingNetPay);
    Console.WriteLine();
}

```

Calculate the cumulative Net Pay and Tax Payable for the company as a whole.

The final method in this class then calculates the total net pay for the whole company, again with the use of a for loop to loop through all of the employee details stored in the array list.

```

//-----
// Method To calculate the Company's Total Net Pay & Tax Payable
//-----

public void showTotalPay()
{
    double taxPayable = 0.0;
    double netPay = 0.0;

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        // get the next employee
        Employee nextEmployee = (Employee)Employee[i];
        // add the next net pay item to the departmental total (for
each variable)
        netPay = netPay + nextEmployee.calcNetPay();
        taxPayable = taxPayable +
nextEmployee.calcGrossTaxPayable();
    }

    // write the total net pay and tax payable for the company

    Console.WriteLine("Total Net Pay:" + "\t" + "\t" + "£" +
netPay);
    Console.WriteLine("Total Tax Payable:" + "\t" + "£" +
taxPayable);
    Console.WriteLine();
}

```

Get Variables and Values produced from methods to output records

Finally the Program Class gets all of the variables required to produce the required output. It does this with a series of calls from the other methods to produce the outputs. This can be seen in the full annotated program, shown later.

Critical Analysis of Data Structures

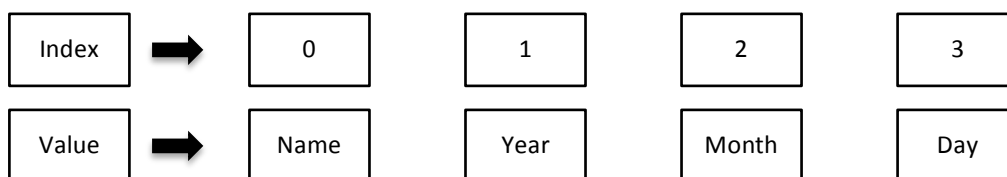
Abstract

The C# programming language uses a vast library of data structures in order to make a program complete and valid. These range from Arrays, ArrayLists and various *collection classes* to Stacks and Queues. For the purpose of this piece we will focus on three of these data structures, namely; Arrays and ArrayLists and Lists and LinkedLists.

Arrays

An Array as described by McMillan (2007) is “a collection of elements with the same data type.” Arrays and other data structures are used when the program requires multiple related values that cannot be simply stored as variables. Arrays can be either static or dynamic. We use arrays to store data that we wish to refer to or call from at a later date (or point) within a program’s running thus overcoming the problems that arise when using separate variables. The English computer science definition of an Array being; “An arrangement of memory elements in one or more planes.”
(Unsupported source type (InternetSite) for source Far12.)

Arrays can be simple one dimensional or two dimensional instances or much more complex three dimensional instances. Each item within an Array is accessed using a number, or index. This points the program to the specific cell within an Array where the data it required is located and stored as shown below.



Arrays can be used easily within a programs creations, but do have their limitations. A static Array is fixed in length and therefore the size cannot change during the running of a program after creation. This will cause problems in programs such as the Payroll Program for any business where employee numbers can change rapidly and over time. Arrays also cannot perform operations themselves and operations cannot be performed on the entirety of the Array, therefore loops are often needed.

To this end other data structures can be used, such as the ArrayList. While an ArrayList itself also has some limitations; some of those associated with Arrays, an ArrayList is a dynamic form of Array, or as has been described, “a non generic collection that behaves like an array on steroids (Miller 2012).

An Array List, is a dynamic Array, which is hugely useful when the size of the array required is not known in the first place. As an array will not be able to resize itself when it runs out of size, by definition an ArrayList will be able to perform this task. When Array lists are first initialised they set a default property parameter as 16. However if this size parameter is met then they will automatically add a further 16 elements to their storage space, thus overcoming the problem of statically sized arrays.

Lists

A list is an alternative to an array. Whilst using a List is very similar to an Array the significant difference between an Array and a List is that a List does not have a specific number of elements that can be stored inside it. Lists can grow and shrink automatically when values are added or removed. As Miller (2012) states, Lists provide a wide assortment of methods that let you manipulate them in many ways. Lists do still use indexes in the same way as an Array to access the data stored within it but are limited only to the amount of free memory available when running the program.

Linked Lists

A Linked list is a collection of objects called nodes (Mc Millan 2007). Each of these nodes is linked to its successor node using a reference or link. The benefits of Linked Lists are that adding or removing items within the list's internal structure is very easy to implement, thus making them much more flexible than both Arrays (including ArrayLists) and Lists. However as Mc Millan, (2007) goes on to say there are still problems associated with Linked Lists, most notably that the program cannot refer to two positions within the Linked List at the same time. When this problem is encountered, further code lines are required as an iterator class will need to be created and utilised.

Selection and Justification

While a List or Linked list could be seen from the analysis above to be suitable data structures to use for the Fly Eazy Payroll program due to the number of employees not being known at the start of the program, it is seen that a Static Array will not be suitable.

Furthermore due to the amount of free memory available not being known it can be assumed that a List itself would equally not be a suitable data structure.

So in conclusion as the Fly Eazy Payroll Program requires the flexibility of a List, with the employee count being an unknown variable, but the relative simplicity of an Array an ArrayList would be the most appropriate data structure to use in its creation.

To this end an ArrayList will be used as the primary data structure used in the Fly Eazy Payroll Program.

Annotation of completed program

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace FlyEazyRWFinal
{
    class Program
    {
        static void Main(string[] args)
        {
            Department record = new Department();
            // "G:\~ " Will need altering for different file locations
            record.readEmployeeDataFile(@"G:\University\MSc\CMP7032 -
Systems
Development\Assessment\FlyEazyV2\FlyEazyV2\bin\Debug\EmployeeData.txt");

            // Main Header with line return afterwards
            Console.WriteLine("FlyEazy Flytes Payroll Report");
            Console.WriteLine();

            // Header for Marketing
            Console.WriteLine("Department: Marketing");
            // Sub Header
            Console.WriteLine("ID" + "\t" + "Surname" + "\t" + "\t" + "\t"
+ "Forename" + "\t" + "\t" + "Grade" + "\t" + "Hrs" + "\t" + "Gross" + "\t"
+ "Tax" + "\t" + "Net Pay");
            // Show records of Employee pay
            record.showEmployeePayMarketing();
            record.showMarketingTotal();
            Console.WriteLine();

            // Header for Sales
            Console.WriteLine("Department: Sales");
            // Sub Header
            Console.WriteLine("ID" + "\t" + "Surname" + "\t" + "\t" + "\t"
+ "Forename" + "\t" + "\t" + "Grade" + "\t" + "Hrs" + "\t" + "Gross" + "\t"
+ "Tax" + "\t" + "Net Pay");
            // Show records of Employee pay
            record.showEmployeePaySales();
            record.showSalesTotal();
            Console.WriteLine();

            // Header for Operations
            Console.WriteLine("Department: Operations");
            // Sub Header
            Console.WriteLine("ID" + "\t" + "Surname" + "\t" + "\t" + "\t"
+ "Forename" + "\t" + "\t" + "Grade" + "\t" + "Hrs" + "\t" + "Gross" + "\t"
+ "Tax" + "\t" + "Net Pay");
            // Show records of Employee pay
            record.showEmployeePayOperations();
            record.showOperationsTotal();
            Console.WriteLine();

            // Header for Whole Company Totals
```



```

        Console.WriteLine("Totals for whole company");
        Console.WriteLine();
        // Show records for whole company
        record.showTotalPay();
    }
}

```

Department.cs

```

using System;
using System.Collections;
using System.IO;

namespace FlyEazyRWFinal
{
    public class Department
    {
        // The array to store employee deatails
        private ArrayList Employee = new ArrayList();

        //-----
        // Reading Data from file
        //-----
        public void readEmployeeDataFile(string fileName)
        {
            string line;
            int lineCount = 0;

            try
            {
                // Create a reader and open the file
                TextReader tr = new StreamReader(fileName);
                while ((line = tr.ReadLine()) != null)
                {
                    lineCount = lineCount + 1;
                    // separates out the required parts of the data file
                    string id = line.Substring(0, 4);
                    string surname = line.Substring(4, 20);
                    string forename = line.Substring(24, 20);
                    string deptCode = line.Substring(44, 1);
                    string grade = line.Substring(45, 1);
                    string hoursWorked = line.Substring(47, 2);

                    // passes data for each separated out individual into
                    Array List
                    Employee newEmployee = new Employee(id, surname,
                    forename, deptCode, grade, Convert.ToDouble(hoursWorked));
                    Employee.Add(newEmployee);
                }

                // Close the Stream
                tr.Close();
            }
            // Exception in case there are problems reading the file name
            catch (Exception e)
            {
                Console.WriteLine("Exception Caught While Reading File");
            }
        }
    }
}

```

```

        Console.WriteLine(e.Message);
    }
}

//-----
// Method To calculate the Company's Total Net Pay & Tax Payable
//-----
public void showTotalPay()
{
    double taxPayable = 0.0;
    double netPay = 0.0;

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        // get the next employee
        Employee nextEmployee = (Employee)Employee[i];
        // add the next net pay item to the departmental total (for
each variable)
        netPay = netPay + nextEmployee.calcNetPay();
        taxPayable = taxPayable +
nextEmployee.calcGrossTaxPayable();
    }

    // write the total net pay and tax payable for the company

    Console.WriteLine("Total Net Pay:" + "\t" + "\t" + "£" +
netPay);
    Console.WriteLine("Total Tax Payable:" + "\t" + "£" +
taxPayable);
    Console.WriteLine();
}

//-----
// Wages Method To calculate each individual's full wages breakdown for
Marketing
//-----
public string showEmployeePayMarketing()
{
    string id = "";
    string surname = "";
    string forename = "";
    string deptCode = "";
    string grade = "";
    double hoursWorked = 0;
    double grossPay = 0.0;
    double taxPayable = 0.0;
    double netPay = 0.0;

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        Employee nextEmployee = (Employee)Employee[i];
        id = nextEmployee.returnId();
        surname = nextEmployee.returnSurname();
        forename = nextEmployee.returnForename();
        deptCode = nextEmployee.returnDept();
        grade = nextEmployee.returnGrade();
        hoursWorked = nextEmployee.returnHoursWorked();
    }
}

```

```

        grossPay = nextEmployee.calcGrossPay();
        netPay = nextEmployee.calcNetPay();
        taxPayable = nextEmployee.calcGrossTaxPayable();

        // Write the outputs to screen
        if (deptCode == "M" || deptCode == "m")
        {
            Console.WriteLine(id + "\t" + surname + "\t" + forename
+ "\t" + grade + "\t" + hoursWorked + "\t" + grossPay + "\t" + taxPayable +
"\t" + netPay);
        }
    }
    // No return as not all code paths return a value
    return Convert.ToString(0);
}

//-----
// Wages Method To calculate each individual's full wages breakdown for
Sales
//-----
public string showEmployeePaySales()
{
    string id = "";
    string surname = "";
    string forename = "";
    string deptCode = "";
    string grade = "";
    double hoursWorked = 0;
    double grossPay = 0.0;
    double taxPayable = 0.0;
    double netPay = 0.0;

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        Employee nextEmployee = (Employee)Employee[i];
        id = nextEmployee.returnId();
        surname = nextEmployee.returnSurname();
        forename = nextEmployee.returnForename();
        deptCode = nextEmployee.returnDept();
        hoursWorked = nextEmployee.returnHoursWorked();
        grade = nextEmployee.returnGrade();
        grossPay = nextEmployee.calcGrossPay();
        netPay = nextEmployee.calcNetPay();
        taxPayable = nextEmployee.calcGrossTaxPayable();

        // Write the outputs to screen
        if (deptCode == "S" || deptCode == "s")
        {
            Console.WriteLine(id + "\t" + surname + "\t" + forename
+ "\t" + grade + "\t" + hoursWorked + "\t" + grossPay + "\t" + taxPayable +
"\t" + netPay);
        }
    }

    // No return as not all code paths return a value
    return Convert.ToString(0);
}

//-----

```

```

// Wages Method To calculate each individual's full wages breakdown for
Operations
//-----
public string showEmployeePayOperations()
{
    string id = "";
    string surname = "";
    string forename = "";
    string deptCode = "";
    string grade = "";
    double hoursWorked = 0;
    double grossPay = 0.0;
    double taxPayable = 0.0;
    double netPay = 0.0;

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        Employee nextEmployee = (Employee)Employee[i];
        id = nextEmployee.returnId();
        surname = nextEmployee.returnSurname();
        forename = nextEmployee.returnForename();
        deptCode = nextEmployee.returnDept();
        grade = nextEmployee.returnGrade();
        hoursWorked = nextEmployee.returnHoursWorked();
        grossPay = nextEmployee.calcGrossPay();
        netPay = nextEmployee.calcNetPay();
        taxPayable = nextEmployee.calcGrossTaxPayable();

        // Write the outputs to screen
        if (deptCode == "0" || deptCode == "o")
        {
            Console.WriteLine(id + "\t" + surname + "\t" + forename
+ "\t" + grade + "\t" + hoursWorked + "\t" + grossPay + "\t" + taxPayable +
"\t" + netPay);
        }
    }

    // No return as not all code paths return a value
    return Convert.ToString(0);
}

//-----
// Method To calculate the departmentas Total Net Pay for Marketing
//-----
public void showMarketingTotal()
{
    double marketingNetPay = 0.0;
    string deptCode = "";

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        // get the next employee
        Employee nextEmployee = (Employee)Employee[i];

        // find the department code value
        deptCode = nextEmployee.returnDept();
        // add the next net pay item to the departmental total (for
each matching variable)
    }
}

```

```

        if (deptCode == "M" || deptCode == "m")
        {
            marketingNetPay = marketingNetPay +
nextEmployee.calcNetPay();
        }
    }

    // write the total net pay and tax payable for the department
    Console.WriteLine("\t" + "\t" + "\t" + "\t" + "\t" + "\t" +
"\t" + "\t" + "\t" + "\t" + "\t" + "-----");
    Console.WriteLine("Total for: Marketing" + "\t" + "\t" + "\t" +
"\t" + "\t" + "\t" + "\t" + "\t" + "\t" + " £" + marketingNetPay);
    Console.WriteLine();
}

//-----
// Method To calculate the departmentas Total Net Pay for Sales
//-----
public void showSalesTotal()
{
    double salesNetPay = 0.0;
    string deptCode = "";

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        // get the next employee
        Employee nextEmployee = (Employee)Employee[i];

        // find the department code value
        deptCode = nextEmployee.returnDept();
        // add the next net pay item to the departmental total (for
each matching variable)
        if (deptCode == "S" || deptCode == "s")
        {
            salesNetPay = salesNetPay + nextEmployee.calcNetPay();
        }
    }

    // write the total net pay and tax payable for the department
    Console.WriteLine("\t" + "\t" + "\t" + "\t" + "\t" + "\t" +
"\t" + "\t" + "\t" + "\t" + "\t" + "-----");
    Console.WriteLine("Total for: Sales" + "\t" + "\t" + "\t" +
"\t" + "\t" + "\t" + "\t" + "\t" + "\t" + " £" + salesNetPay);
    Console.WriteLine();
}

//-----
// Method To calculate the departmentas Total Net Pay for Operations
//-----
public void showOperationsTotal()
{
    double operationsNetPay = 0.0;
    string deptCode = "";

    // step through each employee
    for (int i = 0; i < Employee.Count; i++)
    {
        // get the next employee
        Employee nextEmployee = (Employee)Employee[i];

```



```

//-----
// Normal Constructor
//-----
    public Employee(string id, string surname, string forename, string
deptCode, string grade, double hoursWorked)
    {
        // Loading Parameters to match class memeber variables
        this.id = id;
        this.surname = surname;
        this.forename = forename;
        this.deptCode = deptCode;
        this.grade = grade;
        this.hoursWorked = Convert.ToDouble(hoursWorked);
    }

//-----
// Method to return Employee ID
//-----
    public string returnId()
    {
        return id;
    }

//-----
// Method to return Surname
//-----
    public string returnSurname ()
    {
        return surname;
    }

//-----
// Method to return Forename
//-----
    public string returnForename ()
    {
        return forename;
    }

//-----
// Method to return Grade
//-----
    public string returnGrade ()
    {
        return grade;
    }

//-----
// Method to return Department
//-----
    public string returnDept ()
    {
        return deptCode;
    }

//-----
// Method to return Hours Worked
//-----
    public double returnHoursWorked ()

```

```

    {
        return hoursWorked;
    }

//-----
// Method to return normal hours
//-----
public double calcNormalHours()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Min(hoursWorked, 45);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Min(hoursWorked, 45);
    }
    else if (grade == "C" || grade == "c")
    {
        return Math.Min(hoursWorked, 40);
    }
    else if (grade == "D" || grade == "d")
    {
        return Math.Min(hoursWorked, 40);
    }
    else if (grade == "E" || grade == "e")
    {
        return Math.Min(hoursWorked, 37);
    }
    else
    {
        return 0;
    }
}

//-----
// Method to return Overtime Hours
//-----
public double calcOvertimeHours()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Max(hoursWorked - 45, 0);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Max(hoursWorked - 45, 0);
    }
    else if (grade == "C" || grade == "c")
    {
        return Math.Max(hoursWorked - 40, 0);
    }
    else if (grade == "D" || grade == "d")
    {
        return Math.Max(hoursWorked - 40, 0);
    }
    else if (grade == "E" || grade == "e")
    {
        return Math.Max(hoursWorked - 37, 0);
    }
}

```



```

    }
    else
    {
        return 0;
    }
}

//-----
// Method to return basic Pay
//-----
public double calcBasicPay()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Round(calcNormalHours() * 3.00, 2);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Round(calcNormalHours() * 3.50, 2);
    }
    else if (grade == "C" || grade == "c")
    {
        return Math.Round(calcNormalHours() * 4.00, 2);
    }
    else if (grade == "D" || grade == "d")
    {
        return Math.Round(calcNormalHours() * 4.00, 2);
    }
    else if (grade == "E" || grade == "e")
    {
        return Math.Round(calcNormalHours() * 4.50, 2);
    }
    else
    {
        return 0;
    }
}

//-----
// Method to return overtime
//-----
public double calcOvertimePay()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Round(calcOvertimeHours() * 1.10, 2);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Round(calcOvertimeHours() * 1.25, 2);
    }
    else if (grade == "C" || grade == "c")
    {
        return Math.Round(calcOvertimeHours() * 1.25, 2);
    }
    else if (grade == "D" || grade == "d")
    {
        return Math.Round(calcOvertimeHours() * 1.50, 2);
    }
}

```

```

else if (grade == "E" || grade == "e")
{
    return Math.Round(calcOvertimeHours() * 1.50, 2);
}
else
{
    return 0;
}
}

//-----
// Method to Gross Pay
//-----
public double calcGrossPay()
{
    return Math.Round(calcBasicPay() - calcOvertimePay(), 2);
}

//-----
// Method to return Ammount taxable at Higher Rate
//-----
public double taxableHigherRate()
{
    if (calcGrossPay() > 100)
    {
        return Math.Round(calcGrossPay() - 100, 2);
    }
    else
    {
        return 0;
    }
}

//-----
// Method to return Ammount taxable at Lower Rate
//-----
public double taxableLowerRate()
{
    if (calcGrossPay() > 100)
    {
        return 80;
    }
    else if (calcGrossPay() <= 100)
    {
        return Math.Round(calcGrossPay() - 20, 2);
    }
    else
    {
        return 0;
    }
}

//-----
// Method to return Higher Rate Tax
//-----
public double higherRateTax()

```

```

    {
        return Math.Round(taxableHigherRate() * 0.15, 2);
    }

//-----
// Method to return Lower Rate Tax
//-----
    public double lowerRateTax()
    {
        return Math.Round(taxableLowerRate() * 0.1, 2);
    }

//-----
// Method to return Gross Tax Payable
//-----
    public double calcGrossTaxPayable()
    {
        return Math.Round(higherRateTax() + lowerRateTax(), 2);
    }

//-----
// Method to return Net Pay
//-----
    public double calcNetPay()
    {
        return Math.Round(calcGrossPay() - calcGrossTaxPayable(), 2);
    }
}

```

Critical analysis of testing approaches

According to the IEEE, *“Software testing is the process of analysing a software item to detect the differences between existing and required conditions (that is bugs) and to evaluate the features of the software item.”* The purpose of testing is to find out if the program is both working correctly, both at compile time and run time, and producing both the correct outputs and the required outputs. The IEEE also describes testing as one of verification and validation software practices.

Boehm (1981) describes verification and validation as such:

“Through verification we make sure the product behaves the way we want it to. [And] Through validation we check to make sure that somewhere in the process a mistake hasn’t been made such that the product build is not what the customer asked for.”

Black Box Testing

Black Box testing, also known as data-driven testing is an input/output testing methodology. In Black Box testing the program is viewed as if it were a metaphorical ‘black box’. The person doing the testing is unable to see inside the program or black box and therefore is unable to test any of the code used in creating the program. The tester using a Black Box testing methodology is therefore only able to test inputs and outputs exhaustively, to check all possible cases and see if the program behaves as it is supposed to and gives correct outputs for all possible variations of input. One main benefit of Black Box testing is that the program creator and the tester are independent of each other therefore there is less chance of subconsciously avoiding and ignoring errors. Another benefit to Black Box testing is that it is usually the customer that undertakes either directly or indirectly the testing process and therefore it is with their needs in focus that the testing is undertaken.

However there are also disadvantages to Black Box testing. Black Box testing for even small programs can include many, many permutations, even in the case of the Fly Eazy Payroll Program where there are theoretically a huge number of permutations of grade, department, and hours worked (assuming hours worked are uncapped). This amount of permutations means that Black Box Testing can be a very time consuming and therefore expensive process. The danger of this is to cut time and money on Black Box testing a smaller number of permutations can be chosen and used, which by definition means that another amount of permutations go ignored and without testing them it is unknown as to whether these permutations could cause errors.

White Box Testing

White Box Testing, conversely to Black Box testing, is a methodology where the program is seen as if it were a metaphorical ‘white box’ or a box that can be seen into; hence it is also often referred to as *structural testing, clear box testing and Glass Box testing* (Beizer, 1995).

White Box testing is also a verification and validation methodology, and adheres to the same IEEE standards as Black Bow testing as detailed above. However with the nature of White Box testing, it can be controlled and performed by the programmer and creator of the program to be tested.

White Box testing, as it is undertaken inside the program, has full view of all of the code and paths taken within a program and can therefore be used to test more of how the program works, and not merely the inputs and outputs. To this end it can spot potential errors which would otherwise be missed with just Black Box testing alone. Albeit for full rigorous testing a combination of both Black and White Box testing methodologies would be recommended.

White Box testing takes each single component of the program and runs tests on these to see if the correct answer to the given test is achieved, to this end it can be simpler and less time consuming than Black Box testing, albeit this is not always the case. This can however also lead to correcting mistakes within the code lines easier to make as they can be made in situ. Whereas if these incorrect results are noticed purely in Black Box testing then the program must return to the programmer to correct mistakes (if it can be found) before passing it back to the tester to continue with Black Box testing.

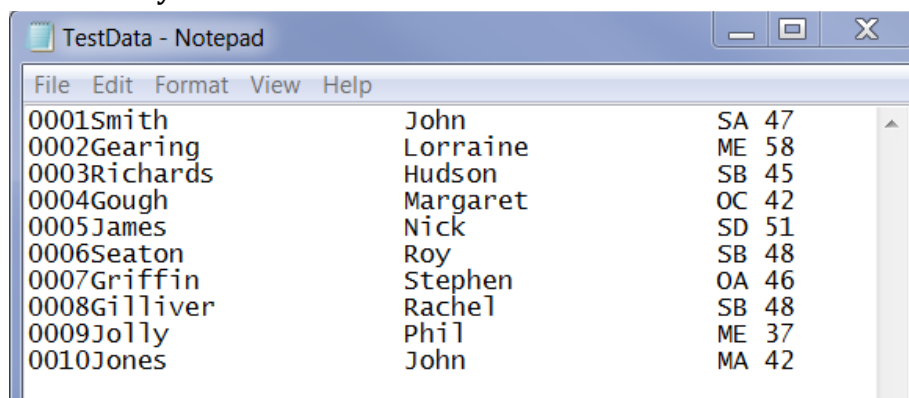
White Box Testing with Test Results

FlyEazy Payroll Program Testing

Using a 'White Box' testing approach and the debugger facility of Microsoft Visual Studio, various breakpoints will be set and run on the FlyEazy Flytes Payroll Program to test and verify the correct working of the program.

The testing process will include using different data sets to check both individual working of all aspects of the program and testing with a complete set off correctly formatted data to test a final output solution. It is impossible to test all potential permutations of data in the time allowed, as discussed previously, so only a small, selected, random set of data will be used in this case.

Preliminary Test Data



0001	Smith	John	SA	47
0002	Gearing	Lorraine	ME	58
0003	Richards	Hudson	SB	45
0004	Gough	Margaret	OC	42
0005	James	Nick	SD	51
0006	Seaton	Roy	SB	48
0007	Griffin	Stephen	OA	46
0008	Gilliver	Rachel	SB	48
0009	Jolly	Phil	ME	37
0010	Jones	John	MA	42

Figure 1: Set of ten employees all formatted correctly using capital letters where necessary.

In this preliminary test scenario a full list of ten employees with all correct data and formatting is inputted to the program and an output obtained. The calculations in this data have all been tested using the mathematical formulae required to produce answers for Gross Pay, Tax and Net Pay per employee along with totals per department and for the company as a whole. These calculations have been done dependant on grade, department and hours worked and it can be confirmed that with these preliminary tests, the output produced was as desired and correct, as shown overleaf.

Test Output for Preliminary Test

```
C:\Windows\system32\cmd.exe
FlyEazy Flytes Payroll Report

Department: Marketing
ID      Surname      Forename      Grade  Hrs   Gross  Tax    Net Pay
0002    Gearing      Lorraine      E      58    135    13.25  121.75
0009    Jolly        Phil          E      37    166.5  17.98  148.52
0010    Jones        John          A      42    126    11.9   114.1
-----
Total for: Marketing                                £384.37

Department: Sales
ID      Surname      Forename      Grade  Hrs   Gross  Tax    Net Pay
0001    Smith        John          A      47    132.8  12.92  119.88
0003    Richards     Hudson        B      45    157.5  16.62  140.88
0005    James        Nick          D      51    143.5  14.52  128.98
0006    Seaton       Roy           B      48    153.75 16.06  137.69
0008    Gilliver     Rachel        B      48    153.75 16.06  137.69
-----
Total for: Sales                                  £665.12

Department: Operations
ID      Surname      Forename      Grade  Hrs   Gross  Tax    Net Pay
0004    Gough        Margaret      C      42    157.5  16.62  140.88
0007    Griffin      Stephen       A      46    133.9  13.08  120.82
-----
Total for: Operations                            £261.7

Totals for whole company
Total Net Pay:      £1311.19
Total Tax Payable:  £149.01

Press any key to continue . . .
```

Figure 2: Full, successful test result for ten employees from full test data file.

Single Line Data Tests (White Box)

In this test, a breakpoint has been added at the *public double calcNormalHours* to test that the program is accessing the if/else loop correctly. It can be seen from the output that this indeed is happening as expected correctly when the grade in this case is A the program enters the loop at the if statement for a grade A (or a) employee. When the breakpoint is stepped into and the debugging continued throughout the program, it can also be seen that the program enters all subsequent loops at the correct grade for this grade A employee.

As we move on with the test for this set of data (TestData1) we can see that when the breakpoint arrives at final output calculations for Gross and Net Pay, the debugger correctly changes the local values stored on MS Visual Studio to the newly created local variables as shown in the associated screen prints.

Finally the final outputs for Gross Pay, Tax Payable and Net Pay were tested to make sure that these outputs were being called to the correct locations in the program, testing the if/else statements and loops to do this. It can be seen again that these are working correctly and shown in the output screen shots.

These same tests were carried out with further test data (see attached screen shots) to test a wide range or permutations for different options of departments and grades, along with the effects of lower case letters on the program.

An attached testing table, shows that all of these data sets passed the test conditions within the program. However for brevity not all screen grabs are shown within this report and as previously stated to test all potential permutations of test data will not be possible within the time and scope allowed.

Test Data 1

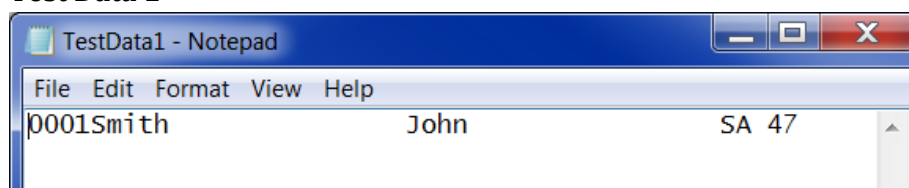


Figure 3: Single employee from the Sales department with codes in capital letters.

Test Outputs for Test 1

```

//-----
// Method to return normal hours
//-----
public double calcNormalHours()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Min(hoursWorked, 45);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Min(hoursWorked, 45);
    }
    else if (grade == "C" || grade == "c")
    {

```

Name	Value	Type
this	{FlyEazyRWFinal.Employee}	FlyEazyf
deptCode	"S"	string
forename	"John"	string
grade	"A"	string
hoursWorked	47.0	double
id	"0001"	string
surname	"Smith"	string

Figure 4: Testing If/Else Statement

```

//-----
// Method to return basic Pay
//-----
public double calcBasicPay()
{
    if (grade == "A" || grade == "a")
    {
        return Math.Round(calcNormalHours() * 3.00, 2);
    }
    else if (grade == "B" || grade == "b")
    {
        return Math.Round(calcNormalHours() * 3.50, 2);
    }
    else if (grade == "C" || grade == "c")
    {
        return Math.Round(calcNormalHours() * 4.00, 2);
    }
    else if (grade == "D" || grade == "d")
    {

```

Name	Value	Type
this	{FlyEazyRWFinal.Employee}	FlyEazyf
deptCode	"S"	string
forename	"John"	string
grade	"A"	string
hoursWorked	47.0	double
id	"0001"	string
surname	"Smith"	string

Figure 5: Testing Entry into correct If Statement

```

        grade = nextEmployee.returnGrade();
        hoursWorked = nextEmployee.returnHoursWorked();
        grossPay = nextEmployee.calcGrossPay();
        netPay = nextEmployee.calcNetPay();
        taxPayable = nextEmployee.calcGrossTaxPayable();

        // Write the outputs to screen
        if (deptCode == "M" || deptCode == "m")
        {
            Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" + hoursWorked + "\t" + grossPay + "\t" + netPay + "\t" + taxPayable);
        }
        // No return as not all code paths return a value
        return Convert.ToString(0);
    }

    //-----
    // Wages Method To calculate each individual's full wages breakdown for Sales
    //-----
    public string showEmployeePaySales()

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"S"	string
grade	"A"	string
hoursWorked	47.0	double
grossPay	0.0	double
taxPayable	0.0	double
netPay	0.0	double

Figure 6: Testing Gross Pay Call

```

deptCode = nextEmployee.returnDept();
grade = nextEmployee.returnGrade();
hoursWorked = nextEmployee.returnHoursWorked();
grossPay = nextEmployee.calcGrossPay();
netPay = nextEmployee.calcNetPay();
taxPayable = nextEmployee.calcGrossTaxPayable();

// Write the outputs to screen
if (deptCode == "M" || deptCode == "m")
{
    Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" + grossPay + "\t" + taxPayable + "\t" + netPay);
}
// No return as not all code paths return a value
return Convert.ToString(0);
}

//-----
// Wages Method To calculate each individual's full wages breakdown for Sales
//-----
public string showEmployeePaySales()

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyRWFinal.Department
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyRWFinal.Employee
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"S"	string
grade	"A"	string
hoursWorked	47.0	double
grossPay	132.8	double
taxPayable	0.0	double
netPay	0.0	double

Figure 7: Testing Correct Values in Call

```

// step through each employee
for (int i = 0; i < Employee.Count; i++)
{
    Employee nextEmployee = (Employee)Employee[i];
    id = nextEmployee.returnId();
    surname = nextEmployee.returnSurname();
    forename = nextEmployee.returnForename();
    deptCode = nextEmployee.returnDept();
    grade = nextEmployee.returnGrade();
    hoursWorked = nextEmployee.returnHoursWorked();
    grossPay = nextEmployee.calcGrossPay();
    netPay = nextEmployee.calcNetPay();
    taxPayable = nextEmployee.calcGrossTaxPayable();

    // Write the outputs to screen
    if (deptCode == "M" || deptCode == "m")
    {
        Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" + hoursWorked + "\t" + grossPay + "\t" + netPay + "\t" + taxPayable);
    }
}
// No return as not all code paths return a value
return Convert.ToString(0);

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"S"	string
grade	"A"	string
hoursWorked	47.0	double
grossPay	132.8	double
taxPayable	0.0	double
netPay	0.0	double

Figure 8: Testing Correct Values in Call

```

// step through each employee
for (int i = 0; i < Employee.Count; i++)
{
    Employee nextEmployee = (Employee)Employee[i];
    id = nextEmployee.returnId();
    surname = nextEmployee.returnSurname();
    forename = nextEmployee.returnForename();
    deptCode = nextEmployee.returnDept();
    grade = nextEmployee.returnGrade();
    hoursWorked = nextEmployee.returnHoursWorked();
    grossPay = nextEmployee.calcGrossPay();
    netPay = nextEmployee.calcNetPay();
    taxPayable = nextEmployee.calcGrossTaxPayable();

    // Write the outputs to screen
    if (deptCode == "M" || deptCode == "m")
    {
        Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" + netPay);
    }
}
// No return as not all code paths return a value
return Convert.ToString(0);

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"S"	string
grade	"A"	string
hoursWorked	47.0	double
grossPay	132.8	double
taxPayable	0.0	double
netPay	119.88	double

Figure 9: Testing Correct Values Out of Call

```

grade = nextEmployee.returnGrade();
hoursWorked = nextEmployee.returnHoursWorked();
grossPay = nextEmployee.calcGrossPay();
netPay = nextEmployee.calcNetPay();
taxPayable = nextEmployee.calcGrossTaxPayable();

// Write the outputs to screen
if (deptCode == "M" || deptCode == "m")
{
    Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" + netPay);
}
// No return as not all code paths return a value
return Convert.ToString(0);

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"S"	string
grade	"A"	string
hoursWorked	47.0	double
grossPay	132.8	double
taxPayable	0.0	double
netPay	119.88	double

Figure 10: Testing Correct Values Out of Call


```

taxPayable = nextEmployee.calcGrossTaxPayable();

// Write the outputs to screen
if (deptCode == "S" || deptCode == "s")
{
    Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "
}

// No return as not all code paths return a value
return Convert.ToString(0);
}
//

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyRWFinal.Department
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyRWFinal.Employee
deptCode	"S"	string
forename	"John"	string
grade	"A"	string
hoursWorked	47.0	double
id	"0001"	string
surname	"Smith"	string
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"S"	string
grade	"A"	string
hoursWorked	47.0	double
grossPay	132.8	double
taxPayable	12.92	double
netPay	119.88	double

Figure 13: Testing Correct Decision into If/Else Statement

Test Table for all Data Sets

Data Set	If/Else	Loops	Gross Pay	Tax Payable	Net Pay	Grade Locations	Department Location	Totals
TestData1	✓	✓	✓	✓	✓	✓	✓	✓
TestData2	✓	✓	✓	✓	✓	✓	✓	✓
TestData3	✓	✓	✓	✓	✓	✓	✓	✓
TestData4	✓	✓	✓	✓	✓	✓	✓	✓
TestData5	✓	✓	✓	✓	✓	✓	✓	✓
TestData5	✓	✓	✓	✓	✓	✓	✓	✓

Test Data 2

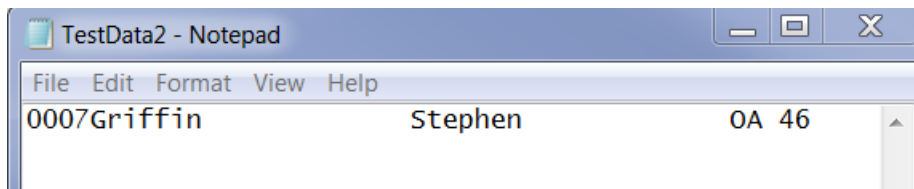


Figure 14: Single employee from the Operations department with codes on capital letters.

Selected Test Outputs for Test Data 2

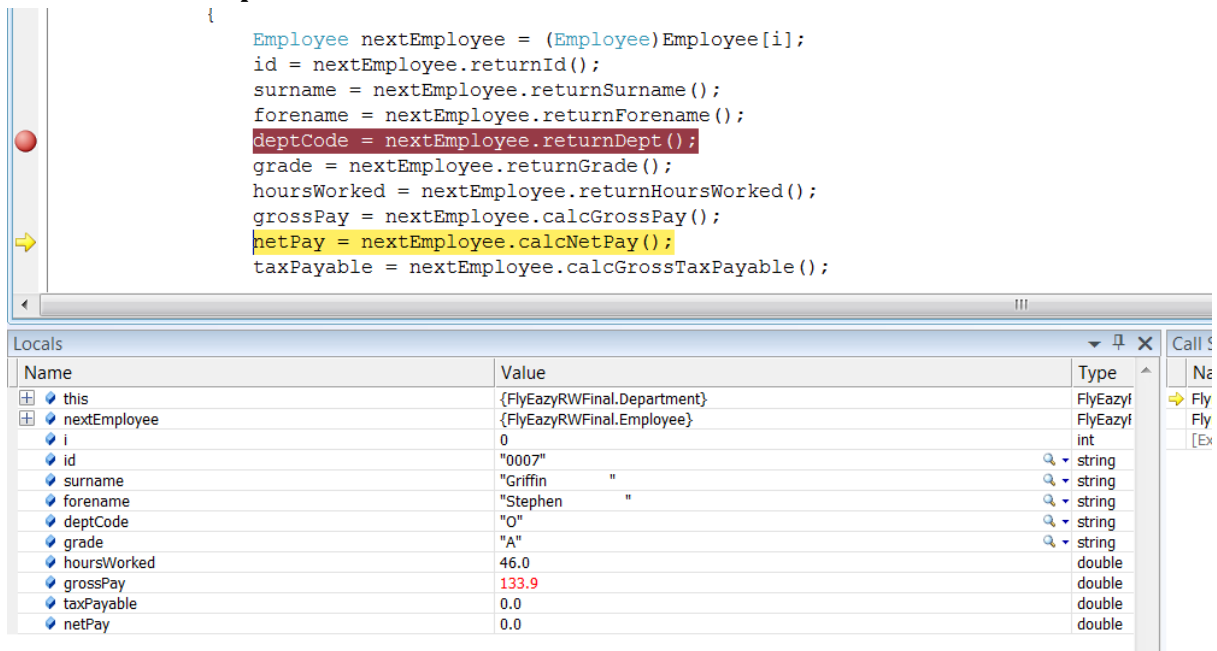


Figure 15: Testing Correct Values Out of Call

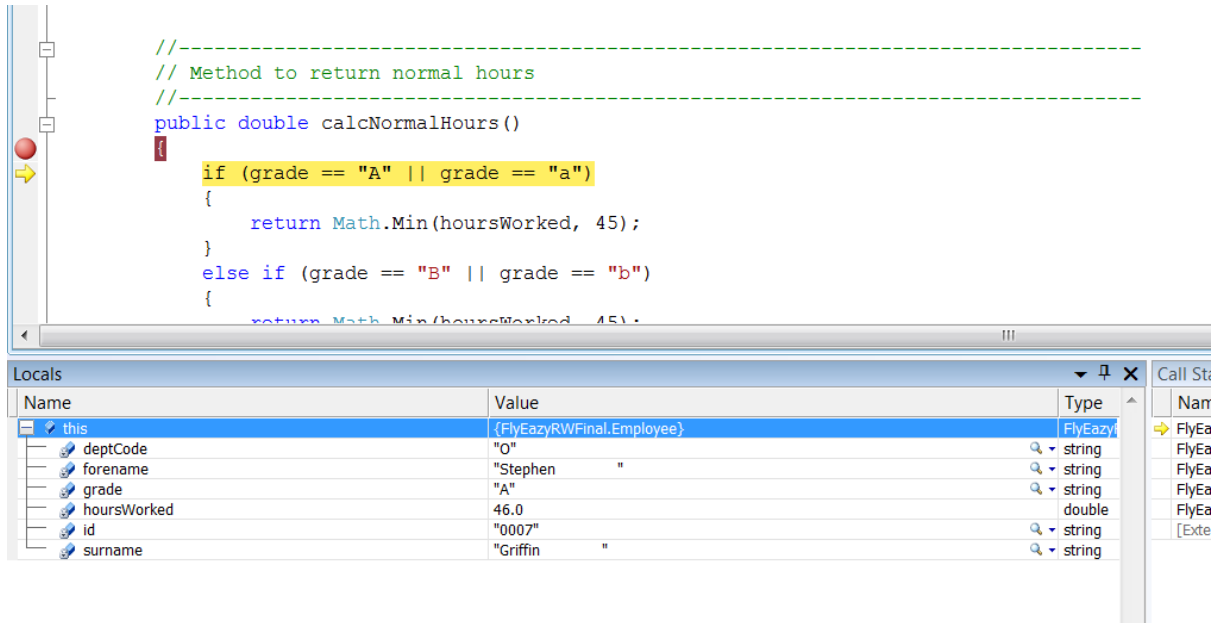


Figure 16: Testing Correct Entrance into If/Else Statement

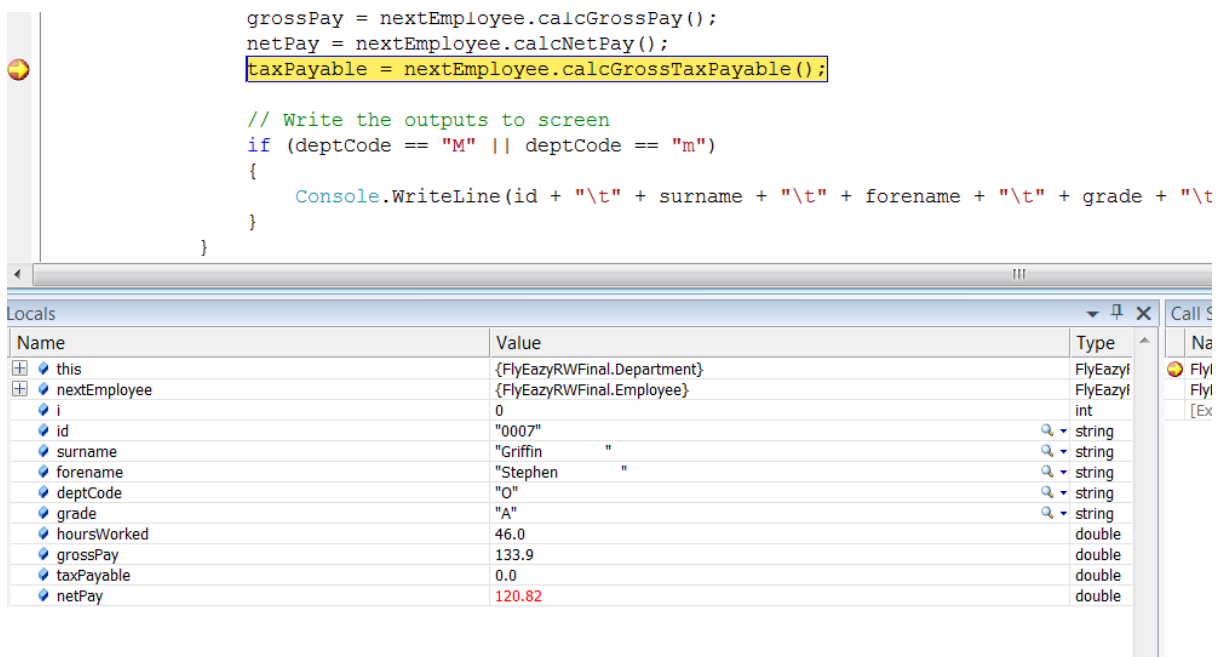


Figure 17: Testing Correct Values out of Call

Test Data 3

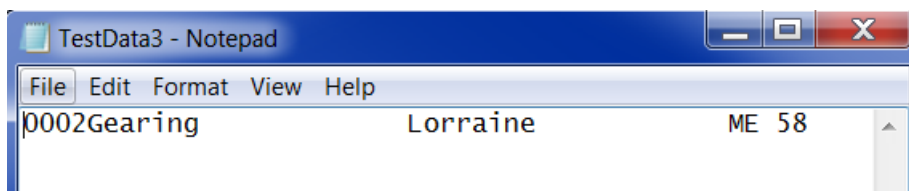


Figure 18: Single employee from the Marketing department with codes in capital letters.

Selected Test Outputs for Test Data 3

```

    }
    else if (grade == "E" || grade == "e")
    {
        return Math.Min(hoursWorked, 37);
    }
    else
    {
        return 0;
    }
}

```

Name	Value	Type
this	{FlyEazyRWFinal.Employee}	FlyEazyf
deptCode	"M"	string
forename	"Lorraine "	string
grade	"E"	string
hoursWorked	58.0	double
id	"0002"	string
surname	"Gearing "	string

Figure 19: Testing Correct Entrance to If/Else Statement

```

hoursWorked = nextEmployee.returnHoursWorked();
grossPay = nextEmployee.calcGrossPay();
netPay = nextEmployee.calcNetPay();
taxPayable = nextEmployee.calcGrossTaxPayable();

// Write the outputs to screen
if (deptCode == "M" || deptCode == "m")
{
    Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t");
}
// No return as not all code paths return a value

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0002"	string
surname	"Gearing "	string
forename	"Lorraine "	string
deptCode	"M"	string
grade	"E"	string
hoursWorked	58.0	double
grossPay	135.0	double
taxPayable	0.0	double
netPay	0.0	double

Figure 20: Testing Correct Values Out of Call

```

netPay = nextEmployee.calcNetPay();
taxPayable = nextEmployee.calcGrossTaxPayable();

// Write the outputs to screen
if (deptCode == "M" || deptCode == "m")
{
    Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" +
}
}
// No return as not all code paths return a value

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyi
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyi
i	0	int
id	"0002"	string
surname	"Gearing"	string
forename	"Lorraine"	string
deptCode	"M"	string
grade	"E"	string
hoursWorked	58.0	double
grossPay	135.0	double
taxPayable	0.0	double
netPay	121.75	double

Figure 21: Testing Correct Values Out of Call

Test Data 4

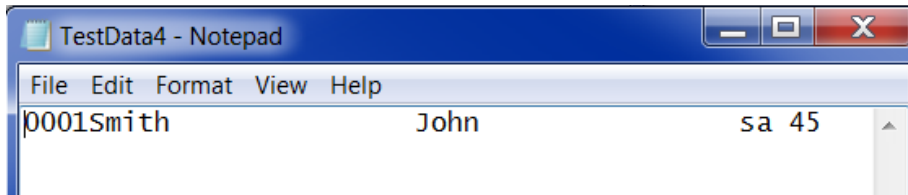


Figure 22: Single employee from the Sales department with codes in lower case.

Selected Test Outputs for Test Data 4

The screenshot shows a code editor with the following code:


```

    grade = nextEmployee.returnGrade();
    hoursWorked = nextEmployee.returnHoursWorked();
    grossPay = nextEmployee.calcGrossPay();
    netPay = nextEmployee.calcNetPay();
    taxPayable = nextEmployee.calcGrossTaxPayable();

    // Write the outputs to screen
    if (deptCode == "M" || deptCode == "m")
    {
        Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" + h
    }
    }
    // No return as not all code paths return a value
    
```

 The Locals window below the code shows the following variables and values:

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"s"	string
grade	"a"	string
hoursWorked	45.0	double
grossPay	135.0	double
taxPayable	0.0	double
netPay	0.0	double

Figure 23: Testing Correct Values Out of Call

The screenshot shows a code editor with the following code:


```

    grossPay = nextEmployee.calcGrossPay();
    netPay = nextEmployee.calcNetPay();
    taxPayable = nextEmployee.calcGrossTaxPayable();

    // Write the outputs to screen
    if (deptCode == "M" || deptCode == "m")
    {
        Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" +
    }
    }
    // No return as not all code paths return a value
    
```

 The Locals window below the code shows the following variables and values:

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0001"	string
surname	"Smith"	string
forename	"John"	string
deptCode	"s"	string
grade	"a"	string
hoursWorked	45.0	double
grossPay	135.0	double
taxPayable	0.0	double
netPay	121.75	double

Figure 24: Testing Correct Values Out of Call

Test Data 5

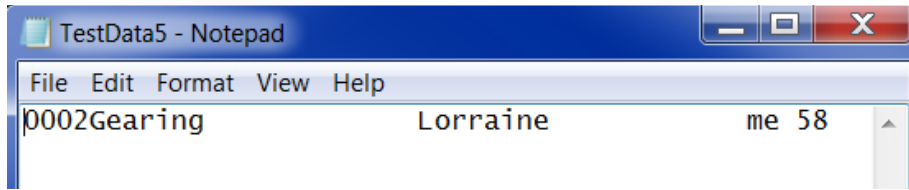


Figure 25: Single employee from the Marketing department with codes in lower case.

Selected Test Outputs for Test Data 5

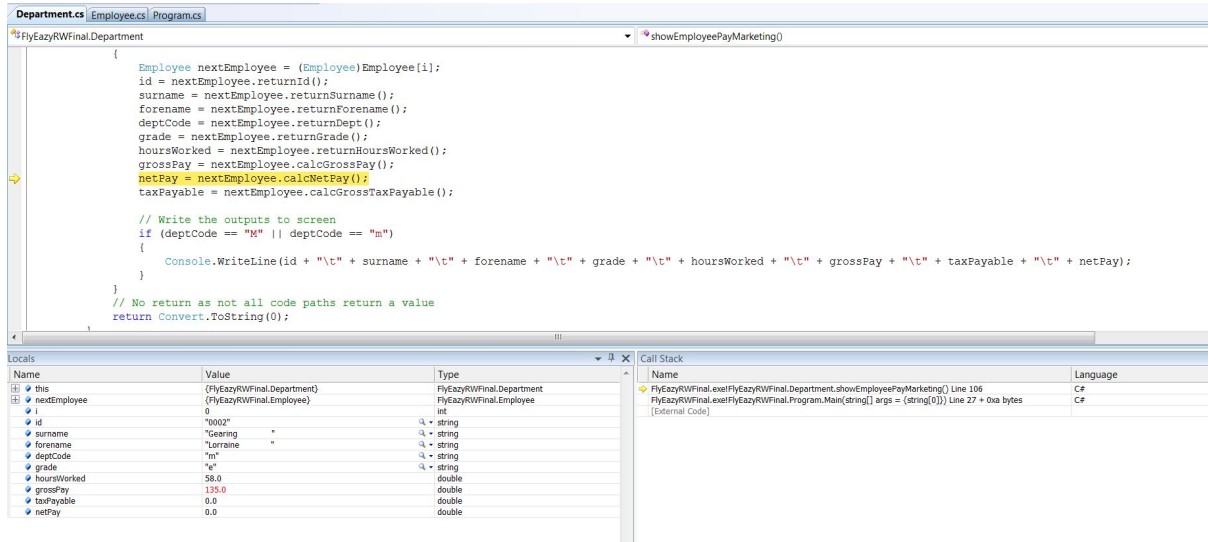


Figure 26: Testing Correct Values Out of Call

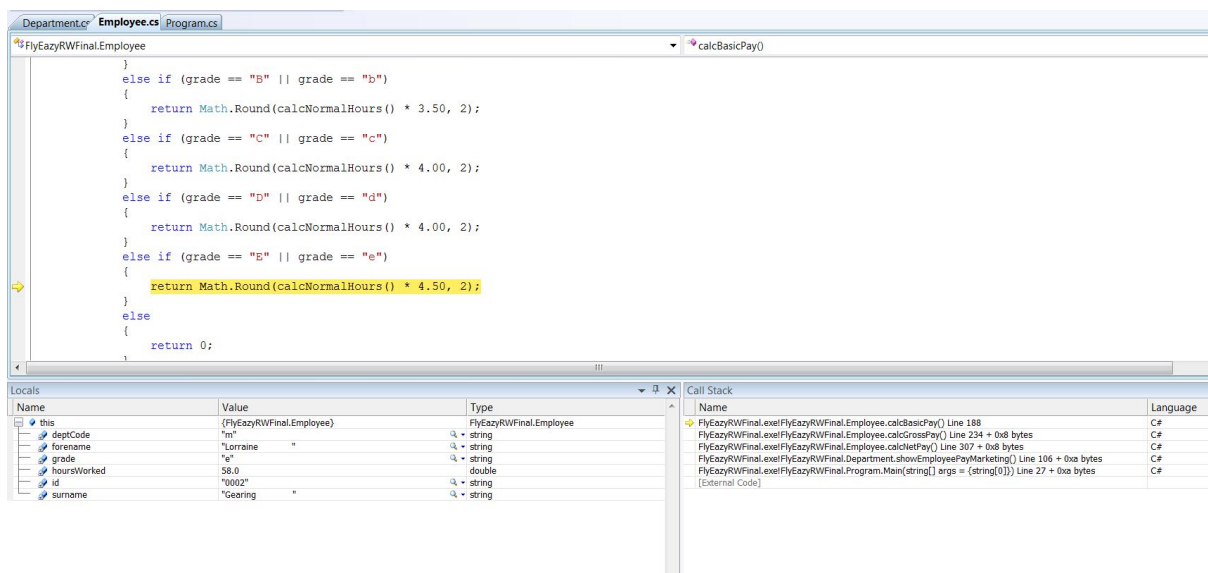


Figure 27: Testing Correct Entrance into If/Else Statement

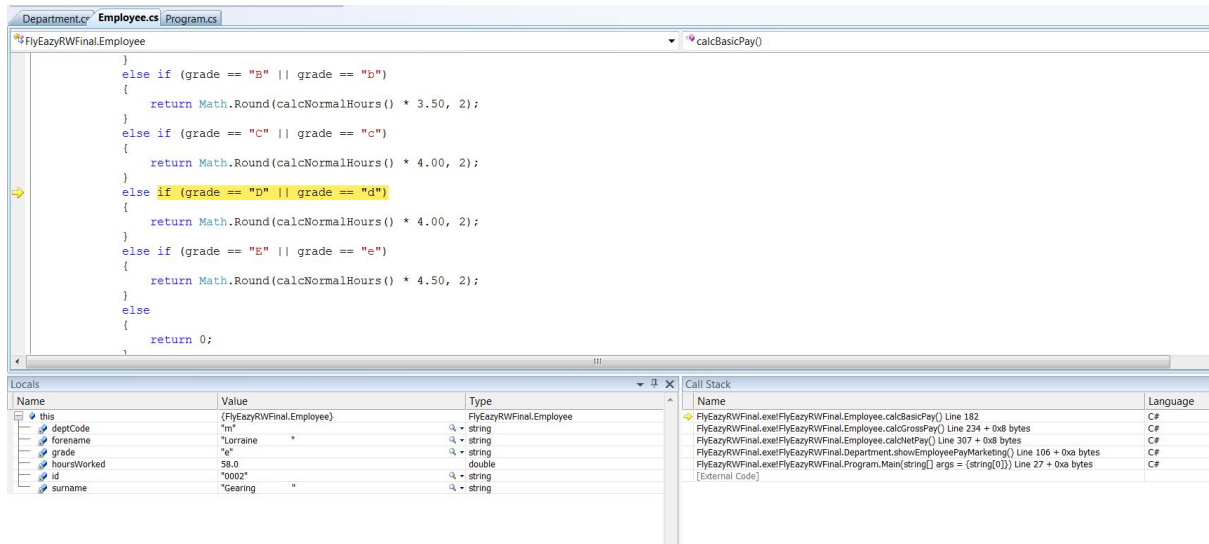


Figure 28: Testing Correct Entrance to If/Else Statement

Test Data 6

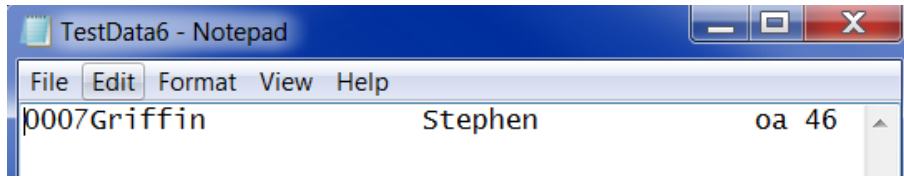
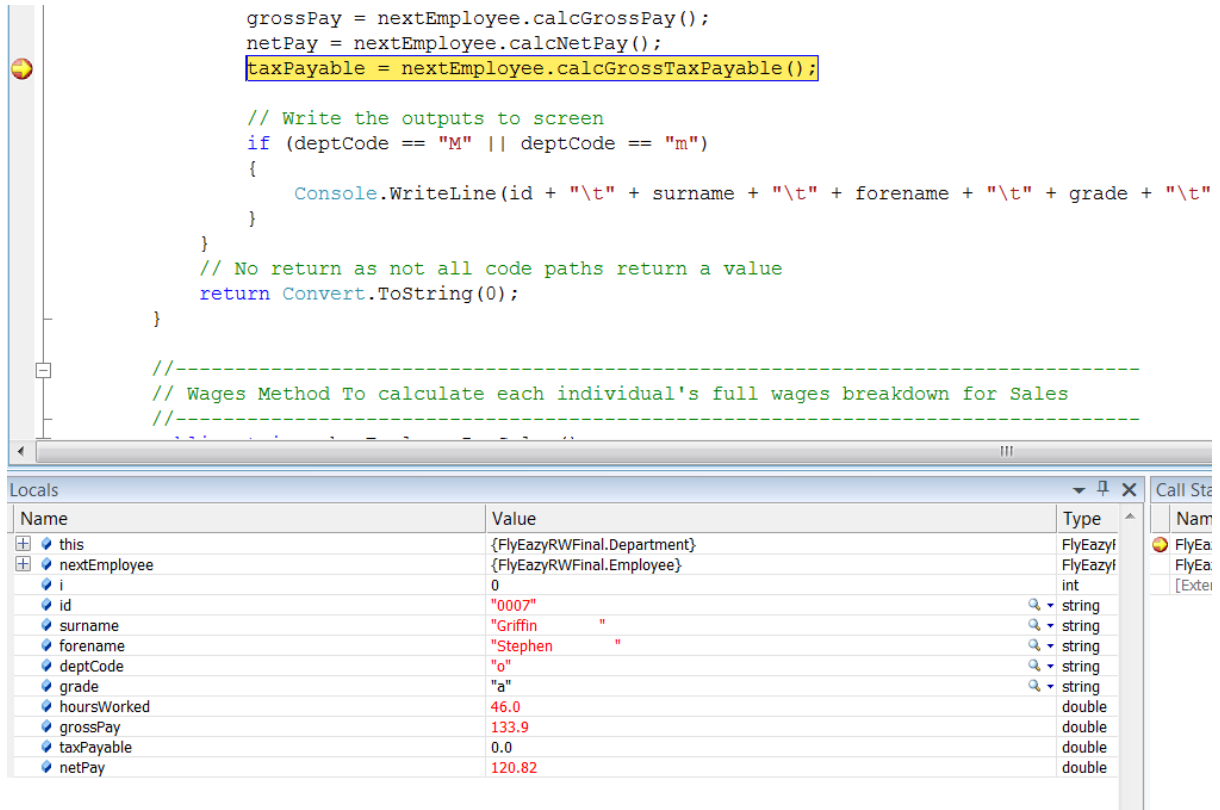


Figure 29: Single employee from the Operations department with codes in lower case.

Selected Test Outputs for Test Data 6



```
grossPay = nextEmployee.calcGrossPay();
netPay = nextEmployee.calcNetPay();
taxPayable = nextEmployee.calcGrossTaxPayable();

// Write the outputs to screen
if (deptCode == "M" || deptCode == "m")
{
    Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t"
}
}
// No return as not all code paths return a value
return Convert.ToString(0);
}

//-----
// Wages Method To calculate each individual's full wages breakdown for Sales
//-----
```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0007"	string
surname	"Griffin"	string
forename	"Stephen"	string
deptCode	"o"	string
grade	"a"	string
hoursWorked	46.0	double
grossPay	133.9	double
taxPayable	0.0	double
netPay	120.82	double

Figure 30: Testing Correct Values Out of Call

```

grossPay = nextEmployee.calcGrossPay();
netPay = nextEmployee.calcNetPay();
taxPayable = nextEmployee.calcGrossTaxPayable();

// Write the outputs to screen
if (deptCode == "M" || deptCode == "m")
{
    Console.WriteLine(id + "\t" + surname + "\t" + forename + "\t" + grade + "\t" + h
}
}
// No return as not all code paths return a value
return Convert.ToString(0);
}

//-----
// Wages Method To calculate each individual's full wages breakdown for Sales
//-----

```

Name	Value	Type
this	{FlyEazyRWFinal.Department}	FlyEazyf
nextEmployee	{FlyEazyRWFinal.Employee}	FlyEazyf
i	0	int
id	"0007"	string
surname	"Griffin"	string
forename	"Stephen"	string
deptCode	"o"	string
grade	"a"	string
hoursWorked	46.0	double
grossPay	133.9	double
taxPayable	13.08	double
netPay	120.82	double

Name
FlyEazyRWF
FlyEazyRWF
[External Co

Figure 31: Testing Correct Value Out of Call

Bibliography

Beizer, B. 1995. *Black Box Testing*. New York: John Wiley and Sons

Boehm, B.W., 1981. *Software Engineering Economics*. New Jersey: Prentice-Hall.

Farlex, 2012. *The Free Dictionary*. [Online] Available at: "<http://www.thefreedictionary.com/Array>" [Accessed 21 December 2012].

IEEE. 2011. "ANSI/IEEE Standard 1008-1987, IEEE Standard for Software Unit Testing," no., 1986.

McMillan, M., 2007. *Data Structures and Algorithms Using C#*. Cambridge: Cambridge University Press.

Miller, R., 2008. *C# For Artists*. Pulp Free Press.

Miller, R., 2012. *C# Collections: A Detailed Presentation*. Pulp Free Press.

Webopedia. 2012. *Black Box Testing* [Online] Available at: http://www.webopedia.com/TERM/B/Black_Box_Testing.html [Accessed, 21 December 2012]

Webopedia. 2012. *White Box Testing* [Online] Available at: http://www.webopedia.com/TERM/W/White_Box_Testing.html [Accessed, 21 December 2012]