# Table of Contents

[Table of Contents 1](#_Toc5007288)

[Introduction 2](#_Toc5007289)

[Scenario 3](#_Toc5007290)

[Scope 3](#_Toc5007291)

[In Scope 3](#_Toc5007292)

[ The program calculates and displays a gym member's basal metabolic rate (BMR) 3](#_Toc5007293)

[ Calculates and display the gym member's body mass index (BMI) 3](#_Toc5007294)

[ The number of kilocalories to maintain their current weight. 3](#_Toc5007295)

[ The BMR calculation is given to 2 decimal places. 3](#_Toc5007296)

[ The BMI calculation is given to 1 decimal place. 3](#_Toc5007297)

[ The kilocalorie requirement output is shown rounded to a whole number. 3](#_Toc5007298)

[ The program needs a "login" screen to access the application. 3](#_Toc5007299)

[ A screen to collect customer details, if they're an existing gym member, name, address, email and telephone. 3](#_Toc5007300)

[Out of scope 3](#_Toc5007301)

[Non-Functional Requirements 4](#_Toc5007302)

[Must be Windows Form application written in C# or Java 4](#_Toc5007303)

[Form design 4](#_Toc5007304)

[First iteration 4](#_Toc5007305)

[Customer Details Form 4](#_Toc5007306)

[Login Form 5](#_Toc5007307)

[Gym Form 5](#_Toc5007308)

[Class Diagram 7](#_Toc5007309)

[Classes 7](#_Toc5007310)

[CalculatorEngine.cs 7](#_Toc5007311)

[Customer.cs 8](#_Toc5007312)

[EnteringDetails.cs 9](#_Toc5007313)

[ESCGym.cs 10](#_Toc5007314)

[Login.cs 11](#_Toc5007315)

[Utility.cs 12](#_Toc5007316)

[Validator.cs 13](#_Toc5007317)

[Properties 14](#_Toc5007318)

[Adding properties 14](#_Toc5007319)

[CalculatorEngine properties 14](#_Toc5007320)

[Customer properties 15](#_Toc5007321)

[Methods 16](#_Toc5007322)

[RoundValue : double 16](#_Toc5007323)

[GetBMICategory : string 17](#_Toc5007324)

[BMR\_HB : double 17](#_Toc5007325)

[Testing 18](#_Toc5007326)

[NUnit - Unit Tests 18](#_Toc5007327)

[CalculatorEngineTest.cs 19](#_Toc5007328)

[CustomerTest.cs 19](#_Toc5007329)

[UtilityTest.cs 20](#_Toc5007330)

[ValidatorTest.cs 21](#_Toc5007331)

[Review 22](#_Toc5007332)

[Mentor Review 22](#_Toc5007333)

[Development 22](#_Toc5007334)

[QA Testing 23](#_Toc5007335)

[Debugging 23](#_Toc5007336)

[Documentation 23](#_Toc5007337)

# Introduction

This is a report of my assignment “East Surrey College GymApp” that I worked on at Infor.

ESC GymApp is a Windows Forms application written in C# using Visual Studio 2017. I was tasked to produce a piece of software using the contextual scenario below and follow all professional computing standards and good practice procedures set by Infor. The finished application consists of the frontend (GUI) and the back-end (code).

# Scenario

You are asked to use your software design, development, testing and evaluation understanding and skills to produce a program that meets the client’s requirements.

East Surrey College gym has commissioned you as a software developer to write a program that will assess a gym member’s requirement to maintain their current weight accurately.

You need to create a program that will give the learners gym member information about:

• Their current basal metabolic rate (BMR)

• Their current body mass index (BMI)

• Their target BMI

• The number of kilocalories to maintain their current weight.

# Scope

## In Scope

## The program calculates and displays a gym member's basal metabolic rate (BMR)

## Calculates and display the gym member's body mass index (BMI)

## The number of kilocalories to maintain their current weight.

## The BMR calculation is given to 2 decimal places.

## The BMI calculation is given to 1 decimal place.

## The kilocalorie requirement output is shown rounded to a whole number.

## The program needs a "login" screen to access the application.

## A screen to collect customer details, if they're an existing gym member, name, address, email and telephone.

## Out of scope

* Create a user account
* Print information
* View help

## Non-Functional Requirements

## Must be Windows Form application written in C# or Java

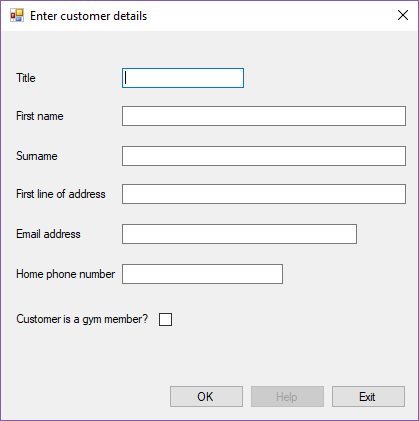
# Form design

## First iteration

The first iteration of form designs was slightly different then they are displayed below, when work on the first iteration of the user interface was complete, I had it reviewed by my mentor at Infor. He told me about ‘*The Magical Number Seven, Plus or Minus Two*’ article which should be used as a guideline for buttons on the ‘*Gym Form*’ form. Since then I’ve made sure to keep buttons to a minimum and to place them in a layout where they didn’t hinder the user experience.

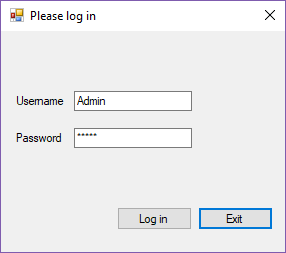
## Customer Details Form

This form is made up of multiple labels, text boxes and buttons. It also has a single checkbox that can be ticked or unticked to determine if the customer is already a gym member.



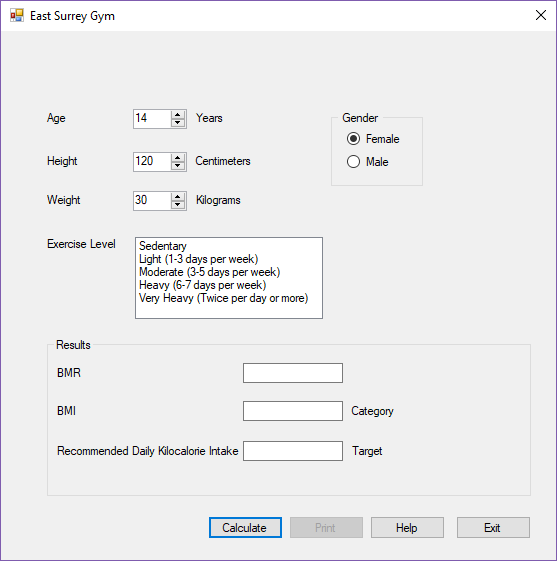
## Login Form

The login form is made up of multiple labels, text boxes and buttons. The Password text box masks all input into asterisks to hide the password. The window itself is small since it doesn’t contain many controls.

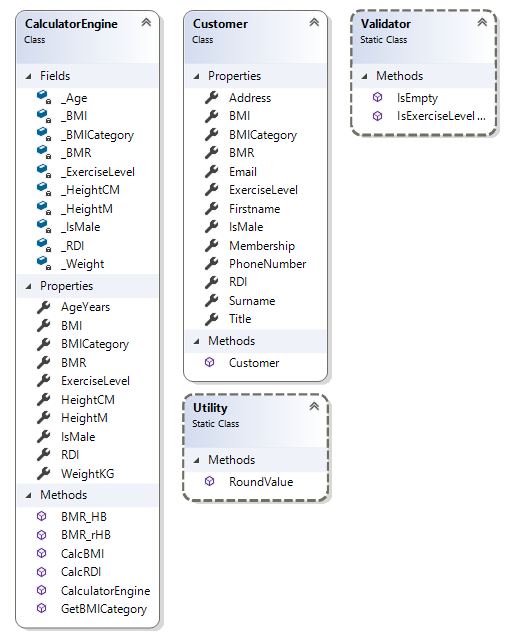


## Gym Form

As this is the main form of the application I prioritized to make sure that this form’s user interface was as user friendly as possible. For Age, Height and Weight, I used ‘NumericUpDown’ control as this allowed the user to quickly scroll or enter their information while also keeping it validated without having to use a validation method in the backend. The exercise level uses a ListBox as the user can only click on a set value that will be used in the ‘*Harris-Benedict equation*’ Radio buttons are used for the user to select their gender. Finally, text boxes are used for the application to display the results back to the user.



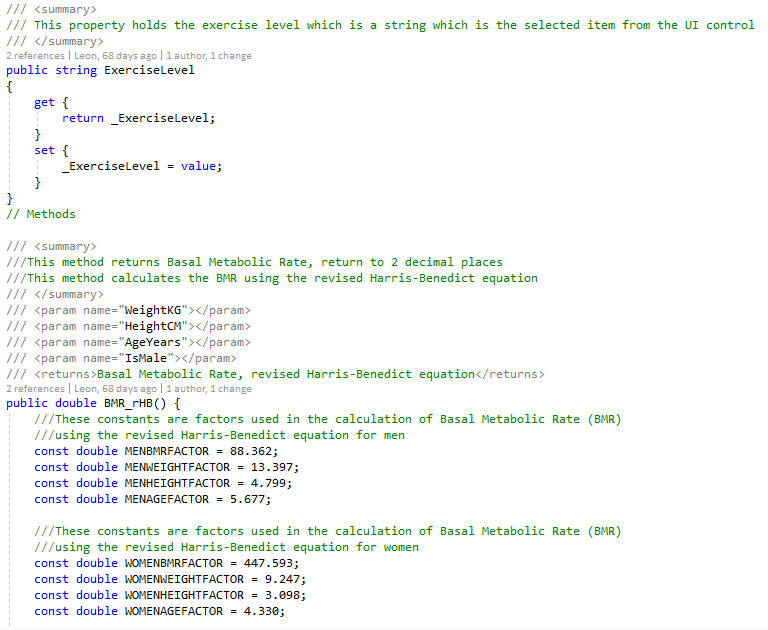
# Class Diagram



# Classes

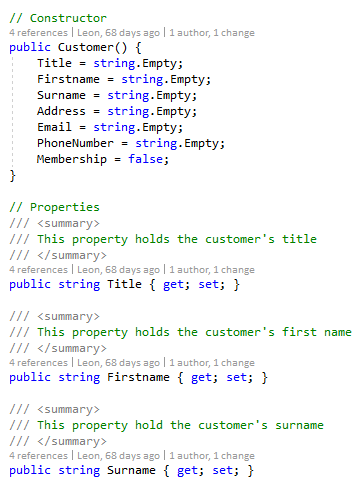
## CalculatorEngine.cs

This class is used to create a Calculator Engine object, there is ever only one instance of this class. The class is instantiated multiple times throughout the application, it stores customer’s health information like their weight and age, then calculates what their BMI & BMR are. Since the application’s primary purpose is to calculate a user’s health information, most methods are stored in this class.



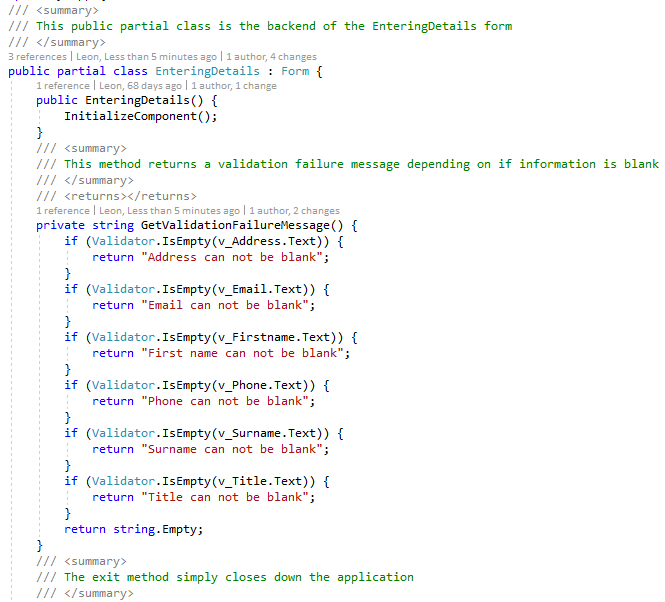
## Customer.cs

This class is used to store Customer information like their name, address and whether they’re already a member of the gym. Its primary use is to create a Customer object to be used in the Calculator Engine.



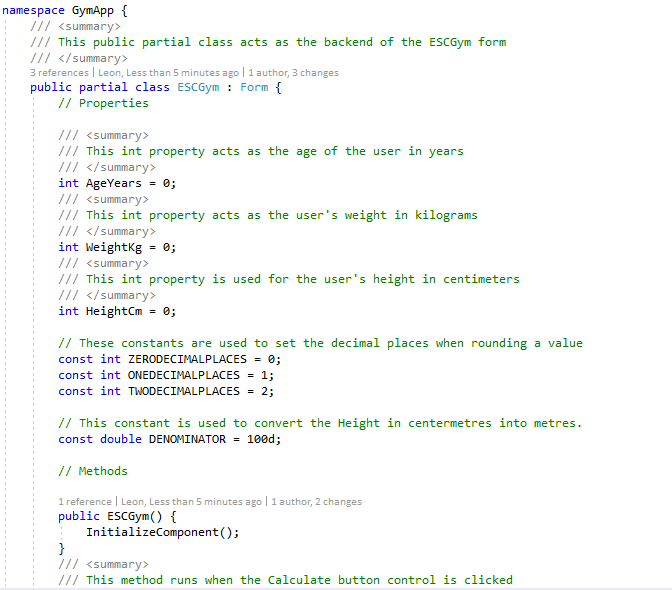
## EnteringDetails.cs

This is the back-end of the “EnteringDetails” form. It is used for validation, Designing the class this way has allowed the application to first make a customer object, then validate it before performing any calculations. Now the calculations will have valid customers before interacting with them.



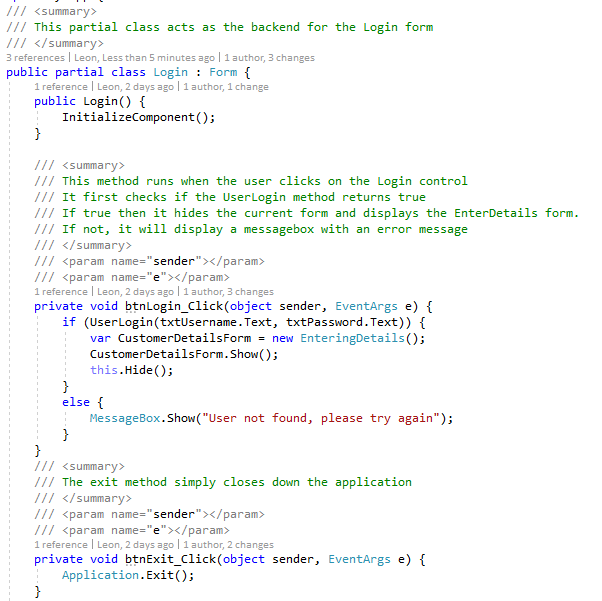
## ESCGym.cs

This is the back-end of the ‘*ESCGym’* form. It holds methods for events on the form controls. The calculate button when clicked will calculate the BMI, BMR & daily kilocalorie intake. The exit button will close the application. The main method in this class is used to gather values from the form controls like age and weight and then validate them using a try catch statement and assigning them to the CalculatorEngine class. These are then assigned as parameters for a method in the Utility class, these values are rounded to a value set by another parameter, this in turn are used in the CalculatorEngine to calculate the user’s health. It then sets the form controls values to display the calculated information for the user to view.



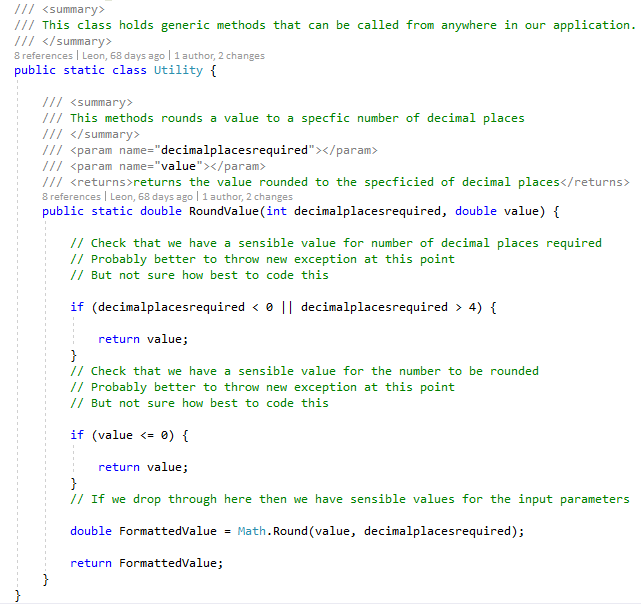
## Login.cs

This is the backend for the “Login” form. This class allows the user to access the ‘*ESCGym*’ form once they successfully log in with the correct credentials. I originally wanted to allow the user to register multiple accounts, but this was descoped due to higher priorities. The class is structured in a way that should allow adding other accounts easy as well as moving it to a database.



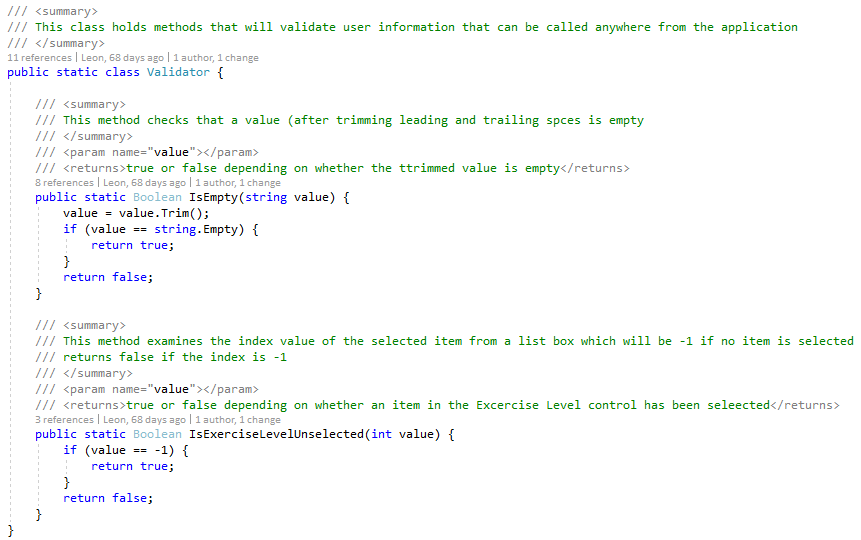
## Utility.cs

The utility class is used for methods that are generic and can be called from anywhere. It currently only has a single method to round a value to a significant figure but can be easily changed to support more methods.



## Validator.cs

Like the utility class, the validator class holds generic methods that validate customer information and can be called from anywhere. This class can also be easily modified to validate other information like calculations or for inserting data into a model layer.



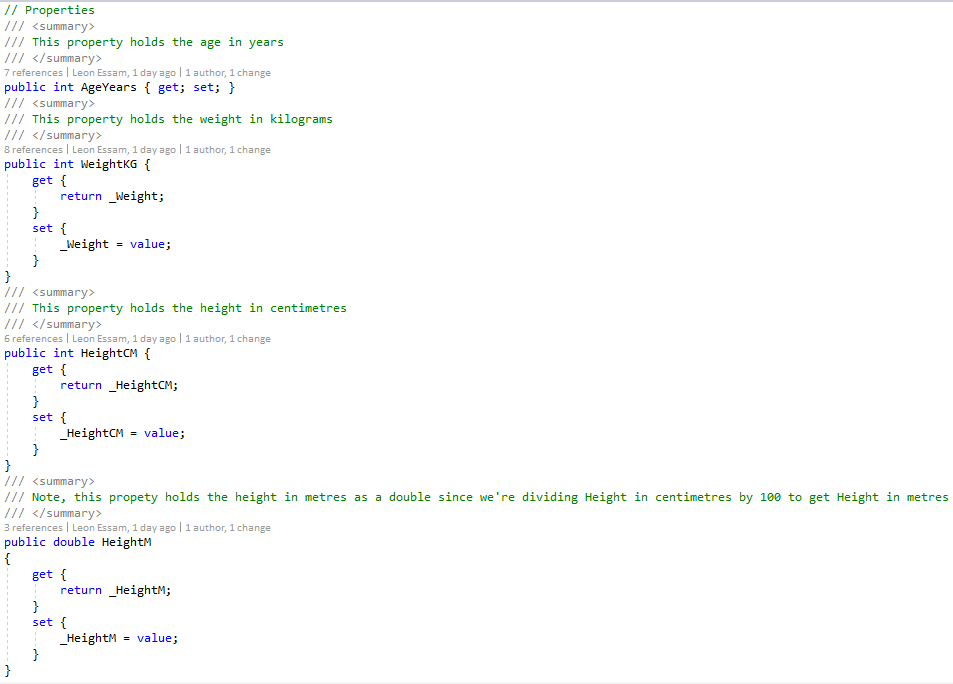
# Properties

## Adding properties

The first initial application didn't have any properties, instead it relied on public static variables in different classes to store and access information. The revised application however uses properties and fields in classes that must be instantiated. This was a better approach because now multiple customers can be instantiated with different information. It is also useful for further development like printing out a list of customers. It also allows further development to work with a model layer. This taught me the importance of using classes, objects and properties with proper access modifiers for proper encapsulation.

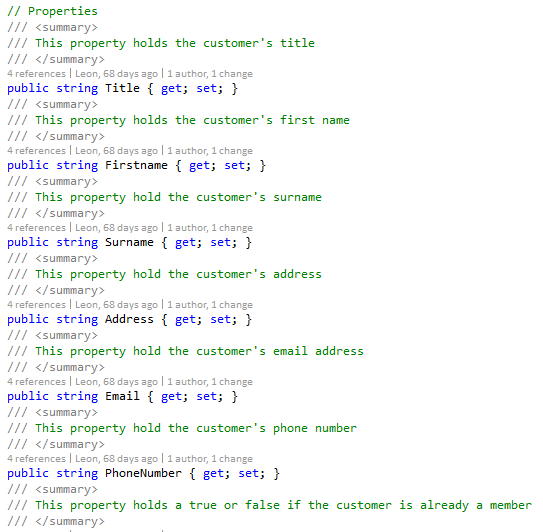
## CalculatorEngine properties

These properties are used in calculations of BMI, BMR and other information. Customer information is passed to this class through getters, setters and parameters which in turn is used in multiple methods and then assigned to new properties like BMI and BMR, which are used in different classes. An example of CalculatorEngine properties are ‘private double \_HeightM;‘ which is a private double used by the getters and setters in this class. This variable must be a double because it gets divided by a 100 to convert it to meters.



## Customer properties

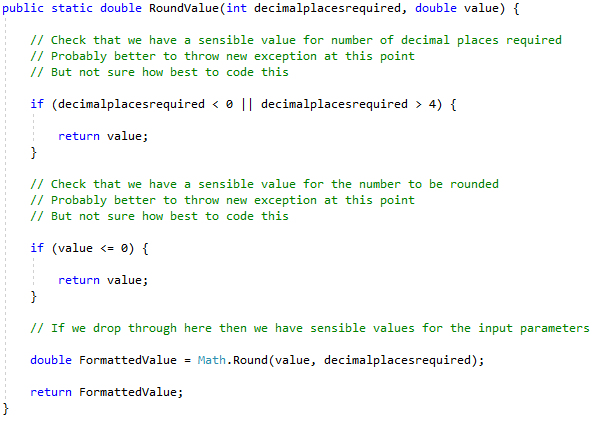
Customer properties include personal information like name, address and email. The class is also used to store BMI and other information calculated from ‘*ESCGym*’. I have designed it this way to create a Customer object which in turn is used in the Calculator Engine class. The structure of the class allows this to be easily added to a database.



# Methods

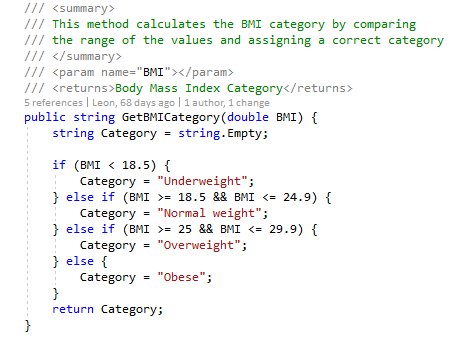
## RoundValue : double

This is method is in the Utility class, it takes two parameters; the number of decimal places to round to and the value. I’ve done it this way as it makes maintaining easier and avoids hardcoding values.



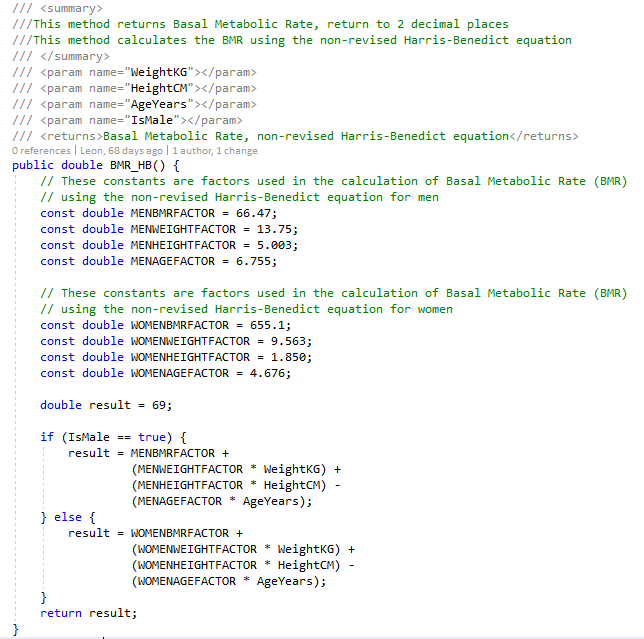
## GetBMICategory : string

This method checks the value of the BMI and returns a string accordingly for their current weight category. This method is easy to maintain since we know the BMI requirements from the scenario, we can use hardcoded values to return the correct BMI category. It can also be easily changed for using other equations.



## BMR\_HB : double

This method is similar for calculating the BMR, this method however uses the non-revised Harris-Benedict equation, so this allows the gym staff to now choose one of two equations. Since it’s already used for another equation, this could be changed to support a different equation.



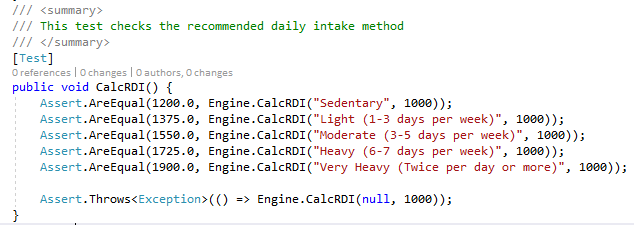
# Testing

## NUnit - Unit Tests

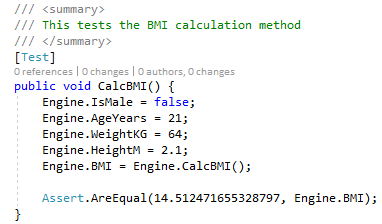
While working on my application, I started working with Unit Tests to make sure that the methods, properties and classes were all working as intended. This helped save time as I didn't have to worry about launching the program, navigate to what I wanted to check and see if it worked. A new class for unit tests that related to a single class, for example one class would be made for unit tests that all tested one class, then another class for a different set tests for a different class.

## CalculatorEngineTest.cs

CalcRDI is a test that makes sure that the 'CalcRDI' method functions correctly by returning the correct BMI category. First it checks all categories are returned with correct BMI values. Then it checks to see if the application throws an exception if the BMI is invalid.

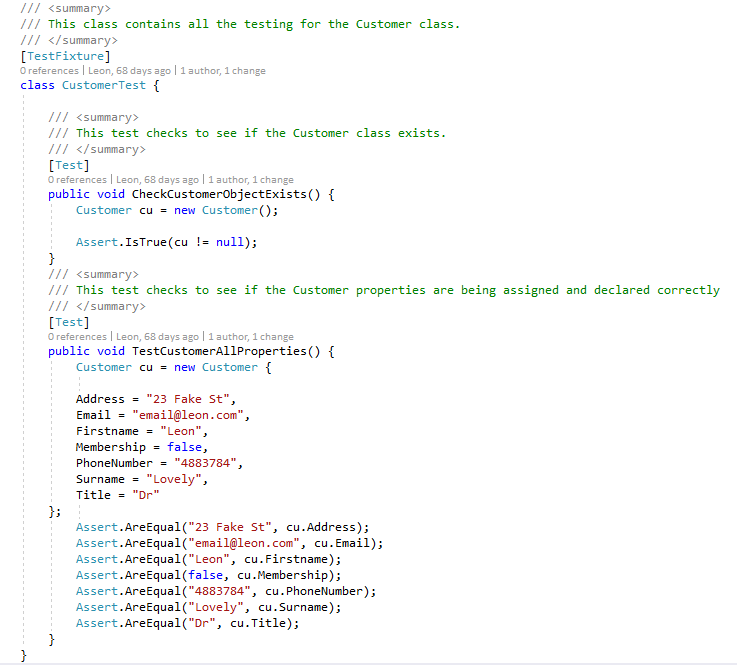


CalcBMI is a similar test in that it checks if the method 'CalcBMI' method functions correctly by returning by the correct BMI value. The test first passes the CalculatorEngine class basic information then expects a set value that was calculated outside to find the correct value.



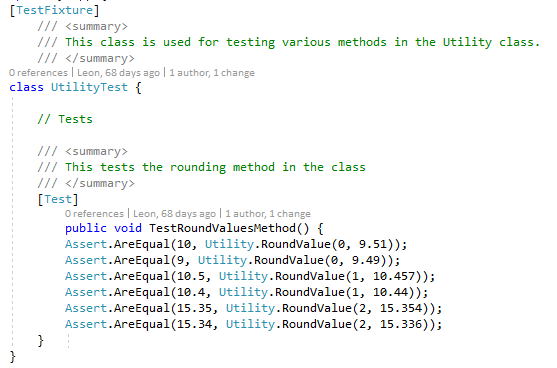
## CustomerTest.cs

This class contains two tests for the customer class, first it checks if the class is being instantiated correctly. The second test ensures that the properties of the class are working by assigning them values then checking the values.



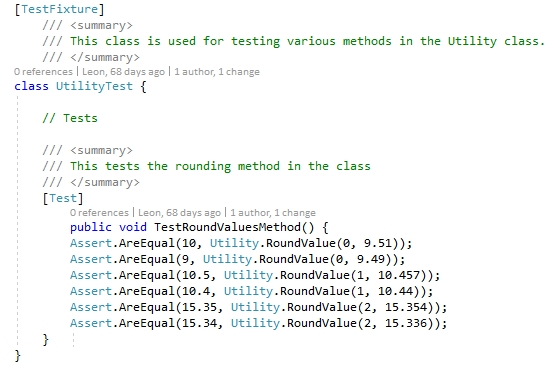
## UtilityTest.cs

Since the utility class only contains a single method, this test class only has one test method. The test is used to ensure that the calculation is correctly rounding values to the right number.



## ValidatorTest.cs

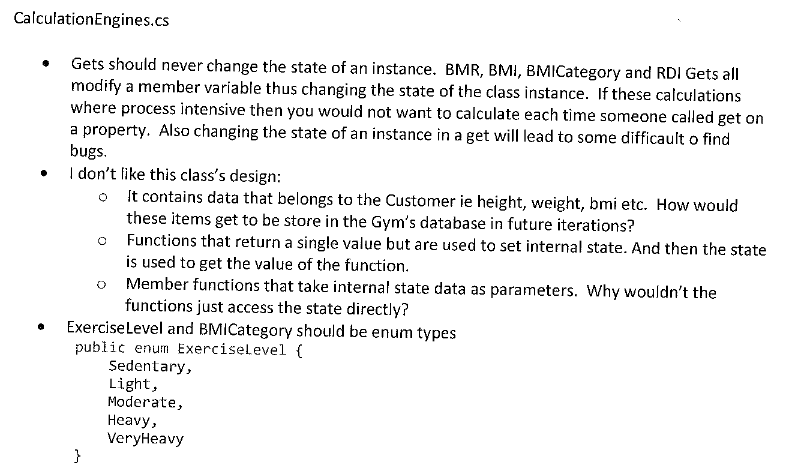
This class holds two tests for the Validator class. 'IsEmptyTestMethod' checks to see if the method outputs a correct true or false depending on if the string is empty. 'IsExerciseLevelUnselectedTestMethod' does something similar by checking that the number is in range.



# Review

## Mentor Review

My initial solution was reviewed by one of my mentors when I had completed it. The comments below illustrate the feedback I received. I changed it accordingly in my second iteration.



## Development

I worked closely with my mentor during development of this program. This was good because we worked to build the program as more of a product delivered by a junior software engineer than as an assignment. I used professional OOP techniques to create objects that each have their own appropriate methods and properties. This allowed me to have a much easier to read and maintain program instead of having everything in a main class.

I also used constructors, getters and setters for objects and classes, this made using the Calculator Engine much easier as well as being able to manipulate customer objects.

Object orientation was better implemented in my second iteration as it initially was still a concept I was not confident in. During the second iteration, I started to plan everything as objects, this allowed me to create separate classes to hold objects that were made up of variables instead of static lists. One example of this was methods in different classes depending on their use. For example, the calculator class has a BMI calculator method while the utility class has a value rounding method.

Having my program reviewed allowed me to see how a professional developer would tackle the same problems that I encounter or a different method of achieving the same goal.

## QA Testing

This was the first time I used unit testing in my application. I had a small exercise from my C# mentor about how to use NUnit and write them as a professional developer, I found it very easy to implement into the gym application and now all methods, classes and properties are tested in their respective test classes.

## Debugging

During my first iteration of development, I used the Visual Studio 2017 built-in debugger to set breakpoints on methods and to view properties to ensure that the correct data was being set and correct data was being returned. I also used this during the second iteration as it was more useful for checking problems the user interface might have had as these kind of problems weren’t able to be tested by unit testing.

As an example, there were no values displayed on the form controls when clicking on the calculate button, all the unit tests were passing but I wouldn’t have noticed the issue if I didn’t run the program myself and noticed it. I was able to fix this using the debugger to find out that the rounding method was not returning any data. This taught me the importance of using various ways of testing my application instead of relying on one.

## Documentation

This was my first-time writing documentation (‘///’) for the application, I was already using comments (‘//’) but I learnt the importance of writing documentation within the application. This allows me to describe what each class, property and method do, the inputs, how they interact with other aspects of the program. It was also good to picture it as me explaining to someone else next to me or describe to myself what it did and how it worked.

This was also good for me so that I can quickly refamiliarize myself with the application if I needed to come back to it later.