Josh Mkhari

Student number: 20104681

PATHWAY: PROG6212

Lecturer: Michael Mapundu

19 September 2021

Programming 2B Task One

TABLE OF CONTENTS

TABLES. DIAGRAMS AND FIGURES

1. INTRODUCTION

Applications exist to automate a set of tasks and for this task one of the POE, the creation of an app that performs time management tasks was required. The following points explain how the app fulfils every requirement of the scenario.

2. Adding multiple modules

Module Adder

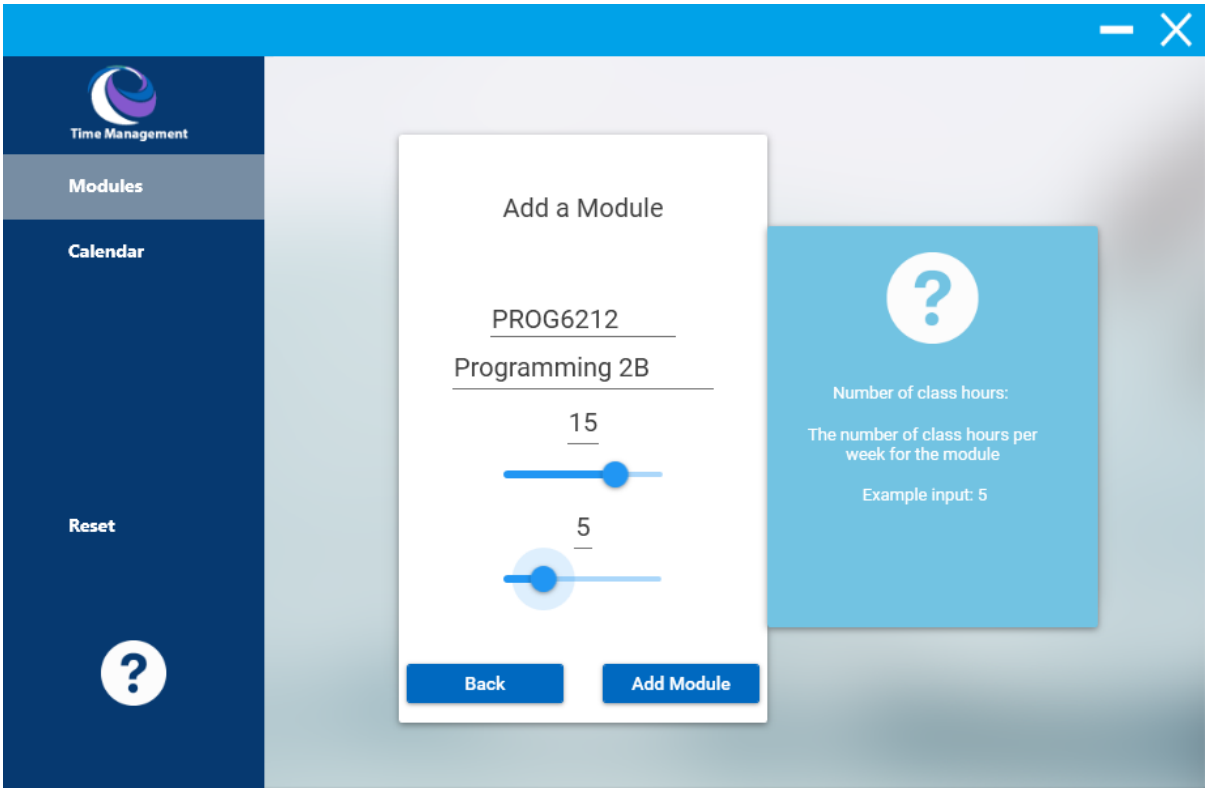

Figure 1 (ModuleAdder view) (EN, 2012), (imagecolorpicker, n.d.) (Price, 2015) (apc, 2014)

A user is capable of adding a module using the **ModuleAdder**. The user is able to specify the module code, module name, number of credits and class hours. The code behind adding a module is in figure 2.

```
1 reference
public static void AddModule(int current, String moduleCode, String moduleName, int moduleCredits, int moduleHours) //To add modules
{
    Calculations c1 = new Calculations();//Using CLassLibrary for Calculations
    c1.setSelfStudyHours(moduleCredits, Convert.ToInt32(StartModel.semesterWeeks), moduleHours);//Sets self study hours
    int modSelfHours;
    if (c1.getSelfStudyHours() < 1)//Ensuring that selfstudy hours are always displayed as a number that is greater than -1
    {
        modSelfHours = 0;
    }
    else
        modSelfHours = c1.getSelfStudyHours();

    moduleList.Add(new Module()//Populating moduleList
    {
        moduleID = current,
        codes = moduleCode,
        names = moduleName,
        credits = moduleCredits,
        hours = moduleHours,
        selfHours = modSelfHours,
        chosenHours = 0,
        remainHours = 0
    });
    stored++;
}
```

Figure 2: AddModule Method

The **moduleList** is a list of type Module (The module class only contains: moduleId, codes, names, credits, hours, selfHours).

- ModuleID: used to store the ID of the module and is later used to find specific modules.
- Codes: Refers to the module Code.
- Names: Refers to the module name.
- Credits: Refers to the module Credits.
- Hours: Refers to the class hours the module has per week.
- SelfHours: Refers to the calculation done for the module which represents the amount of self-study hours needed per week for the module.

Each element within the **moduleList** will store the above objects.

Self-study hours calculation.

```
public void setSelfStudyHours(int credits, int weeks, int hours)
{
    this.selfStudyHours = ((credits * 10) / weeks) - hours;
}
```

Figure 3: Self Study Hours Calculation.

3. Number of weeks and semester start date

Semester Period

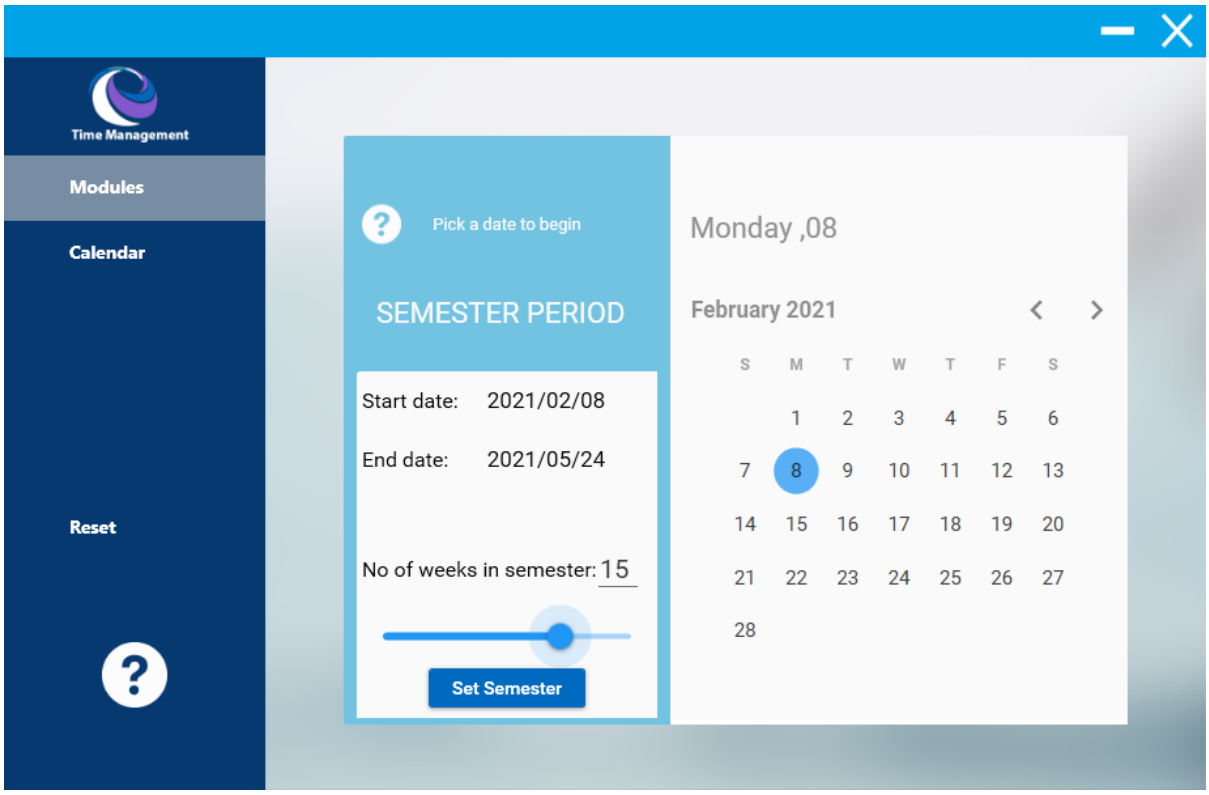

Figure 4: SemesterPeriod View (rj, n.d.), (lab, 2021)

The user on this view would first select a date for the semester start period (grapecity, 2021). Once this is selected, the user is capable of using the slider to set the number of weeks within the semester. The end date is also displayed here and changes according the number of weeks within the semester and start date.

```
private void startCalendar_SelectedDatesChanged(object sender, SelectionChangedEventArgs e)//Whenever a date is selected
{
    if (TSemesterStartDate != null)
    {
        //Calculate the semester end day
        DateTime currDay = (DateTime)startCalendar.SelectedDate;
        DateTime endDay = (DateTime)startCalendar.SelectedDate;
        DateTime answer = endDay.AddDays(NumWeeks.Value * 7);
        StartModel.semesterEndDate = answer;

        //Returning only date and removing time aspect
        TSemesterStartDate.Text = startCalendar.SelectedDate.ToString().Substring(0, 10);
        TSemesterEndDate.Text = StartModel.semesterEndDate.ToString().Substring(0, 10);
        TDayOfWeek.Text = currDay.ToString("dddd") + " ," + currDay.ToString("dd");
    }

}
```

Figure 5: startCalendar Date change() (tricks, 2020)

```
//Calculate the semester end day
DateTime answer = endDay.AddDays(NumWeeks.Value * 7);
```

Figure 6: end date calculation

The program is capable of setting the semester end date (Vudatha, 2017)by first using the users selected number of weeks within the semester and multiplying it by 7 as there are 7 days in a week. This value is then added to the semester start date. Now the program knows the exact date the semester ends on.

4. List of added modules

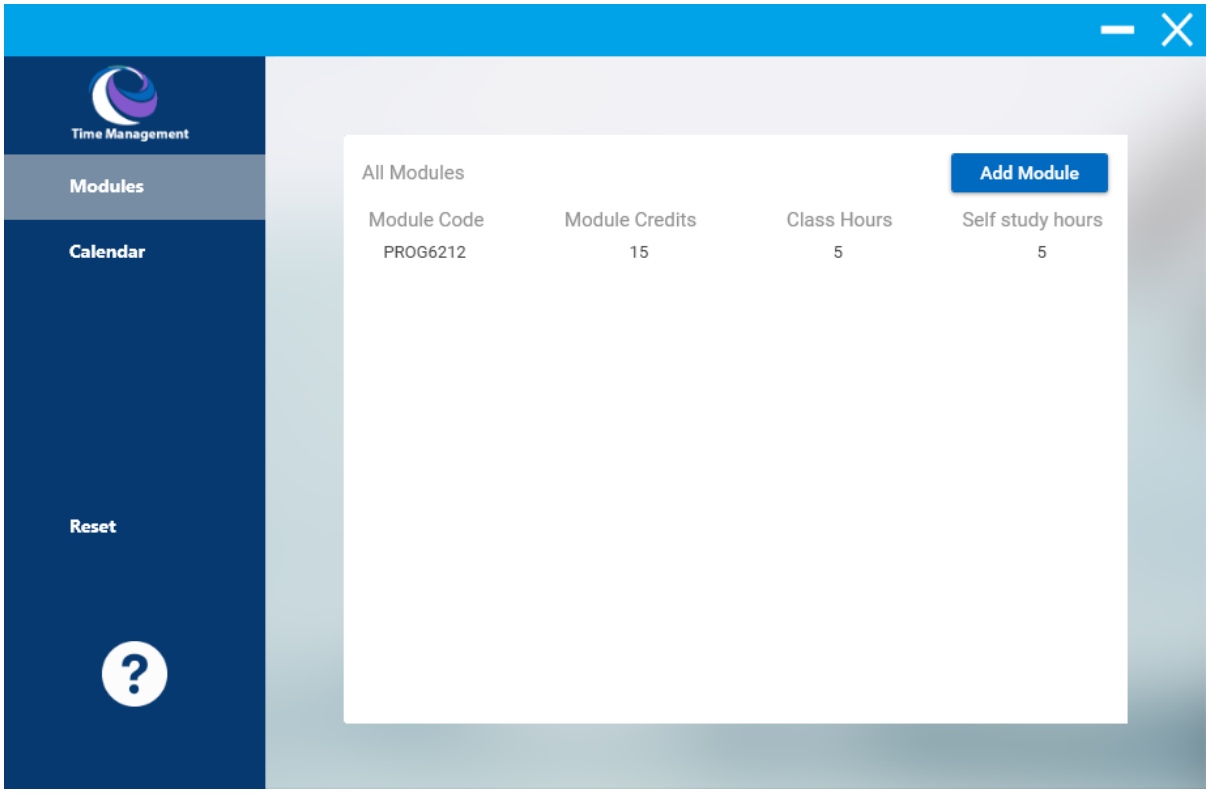

Figure 7: ModulesView (Advance, n.d.) (Jallepalli, 2019)

Once a user has added a module, the view in Figure 7 is displayed. It is a list of the modules the user has added including the number of self-study hours that are required for each week.

```
public void display()//used to display every module stored within module list
{
    String fakeTab = "          ";
    for (int i = 0; i < Program.stored; i++)
    {
        if (!Program.chosenIDs.Contains(i))//repeat for the length of stored modules, checking ID
        {
            var currentModule = from m in Program.moduleList
                                where m.moduleID == i
                                select new { m.codes, m.credits, m.selfHours, m.hours };//Retrieving relevant data for ID

            foreach (var item in currentModule)//Displaying the data from ID
                LDisplayModules.Items.Add(item.codes + "\t\t" + fakeTab + item.credits + "\t\t\t" + fakeTab + item.hours + "\t\t\t" + fakeTab + item.selfHours);
        }
    }
}
```

Figure 8:display method within ModuleView (Teacher, n.d.)

Using LINQ, the program can easily search the **moduleList** for a moduleID, once an ID is found, the variable **currentModule** is set to the relevant data that corresponds to the found moduleID. The program will then display the found module, resulting in Figure 7

5. Calendar View

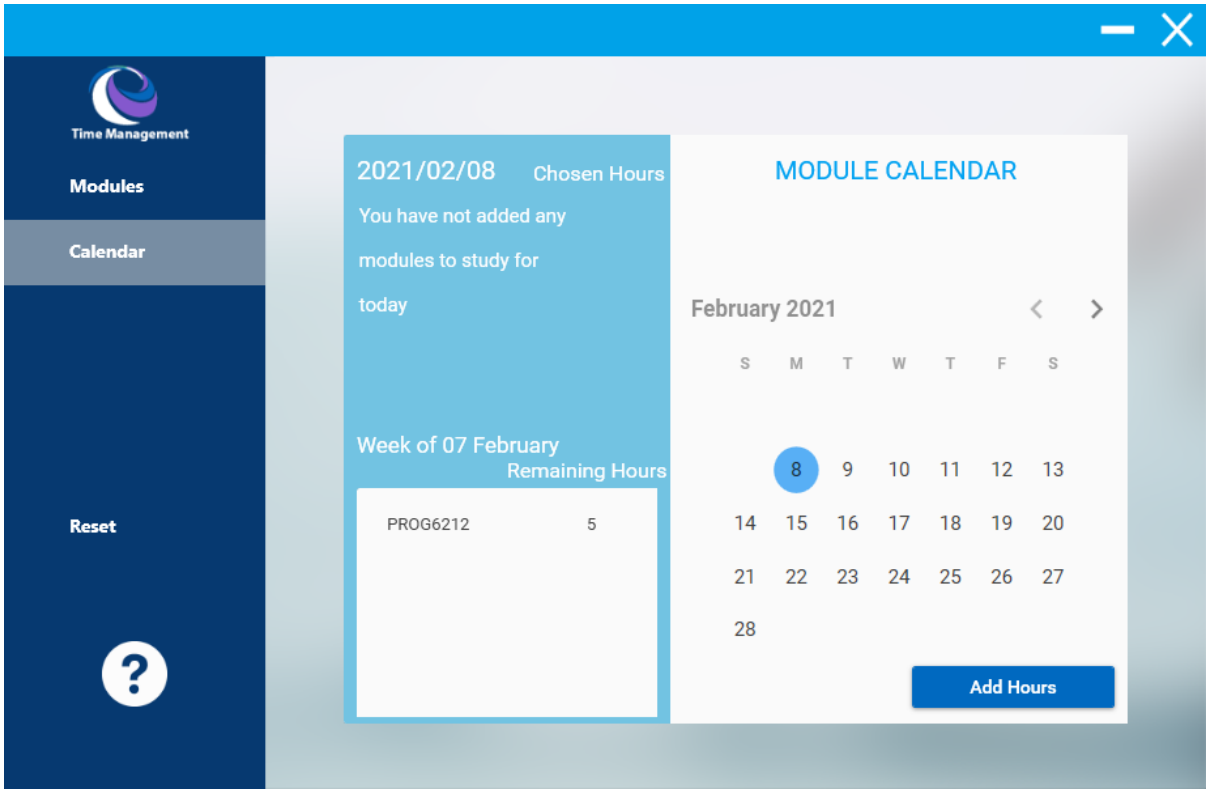

Figure 9: Calendar View, no chosen hours (tutorialsEU, 2021) (foson, 2011)

Figure 9 above is the default view the user will be met with once they have added a module to the **moduleList.** From this view the user is able to select a date and the program will display the chosen hours for that date. To choose hours, the use must select the **Add Hours** button.

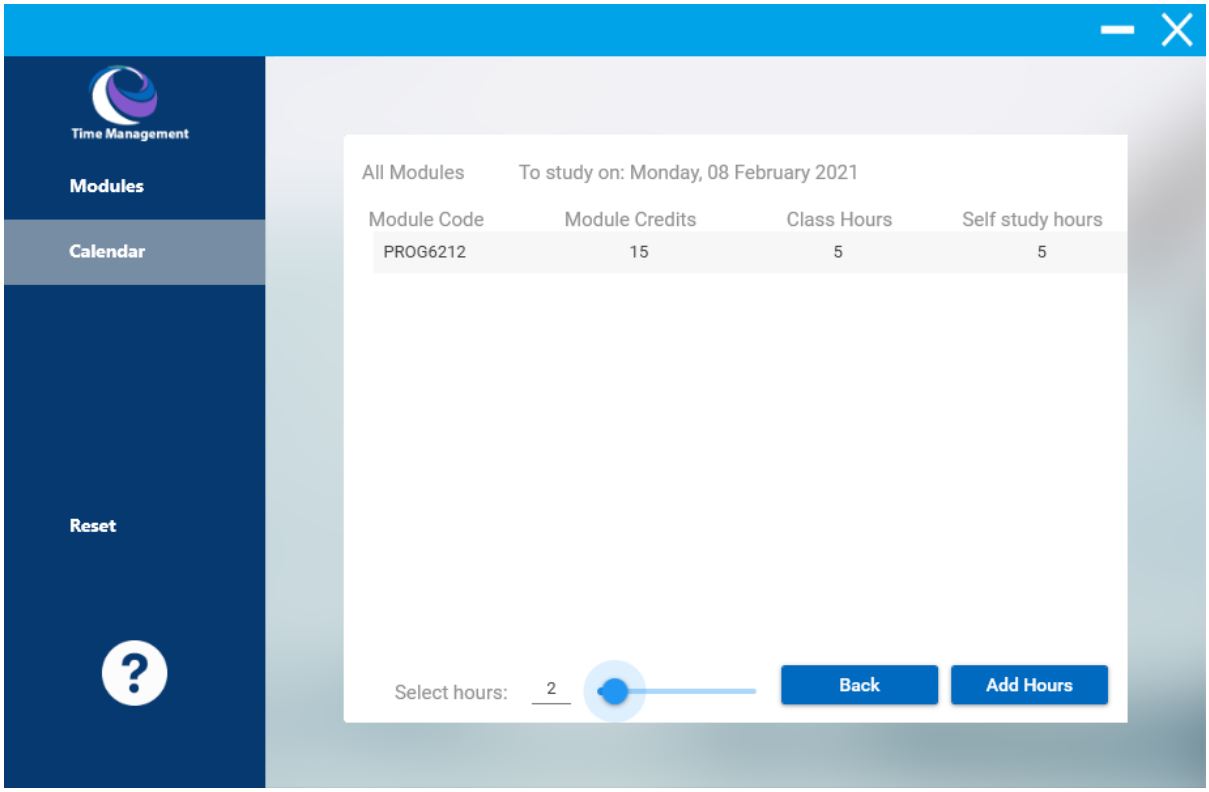

Figure 10: Select Hours view Monday 8 February (BinaryTox1n, 2011) (Davipb, 215)

On the view shown above in figure 10, the user is able to select a specific module to study on the date they had selected on the previous view (Figure 9) and also record the number of hours they will spend on the module.

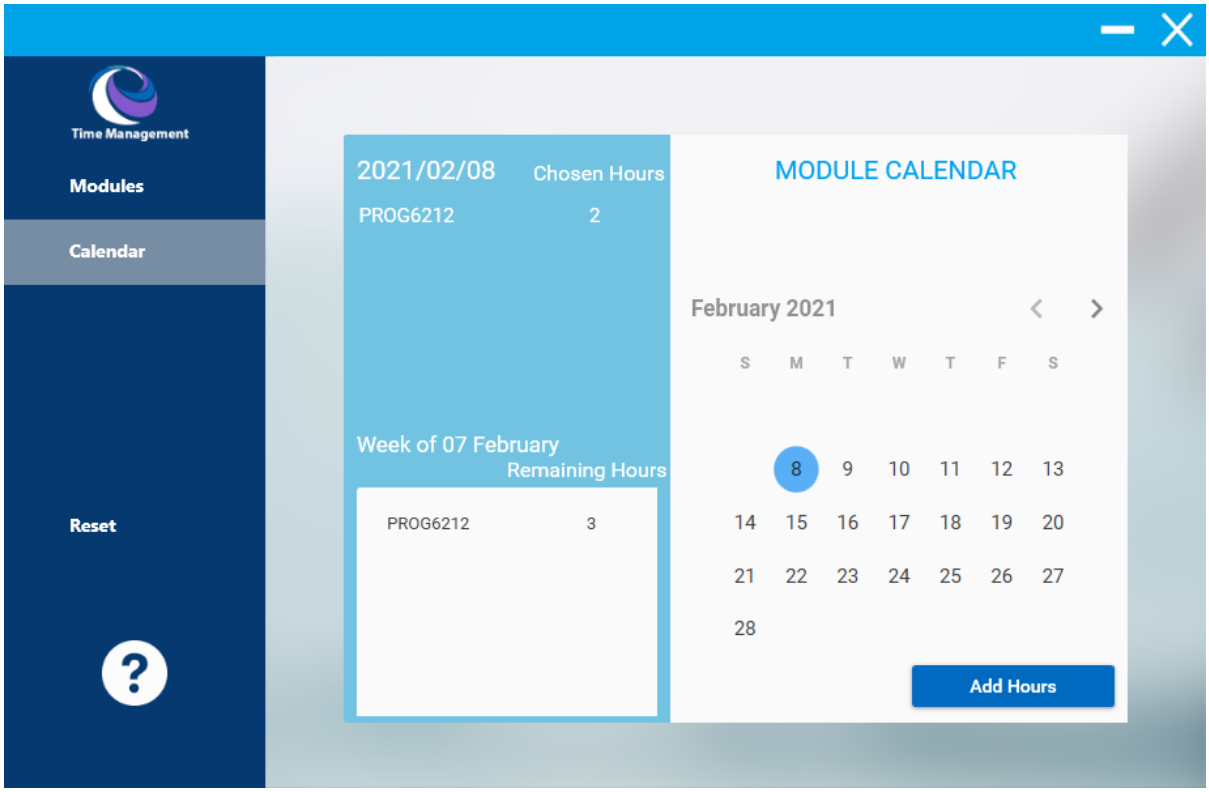

Figure 11: Calendar view with chosen hours

As the user chose to study 2 hours on the day, the program records it and displays it on the top left, it also subtracts the selected hours from the self-study hours for the week. If the user decides to study another 2 hours on the 10th this is the series of views they will see.

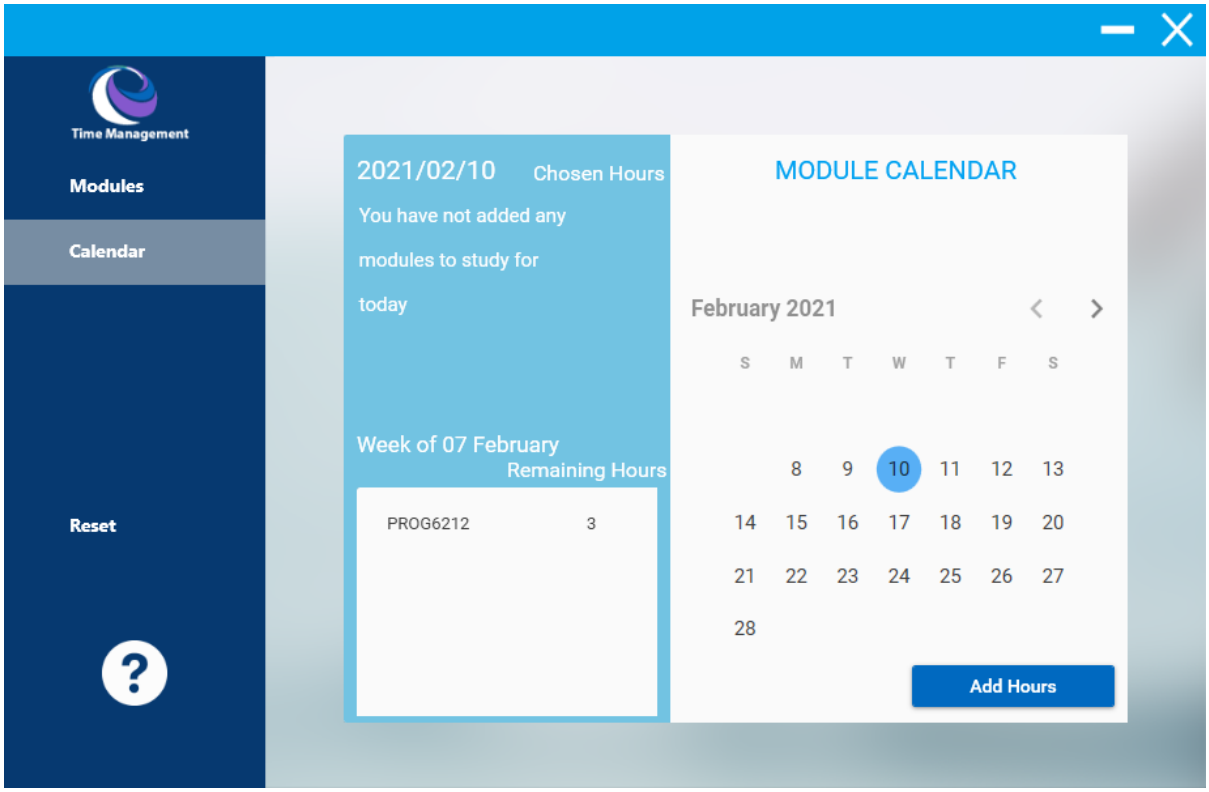

Figure 12: Calendar View, A different day

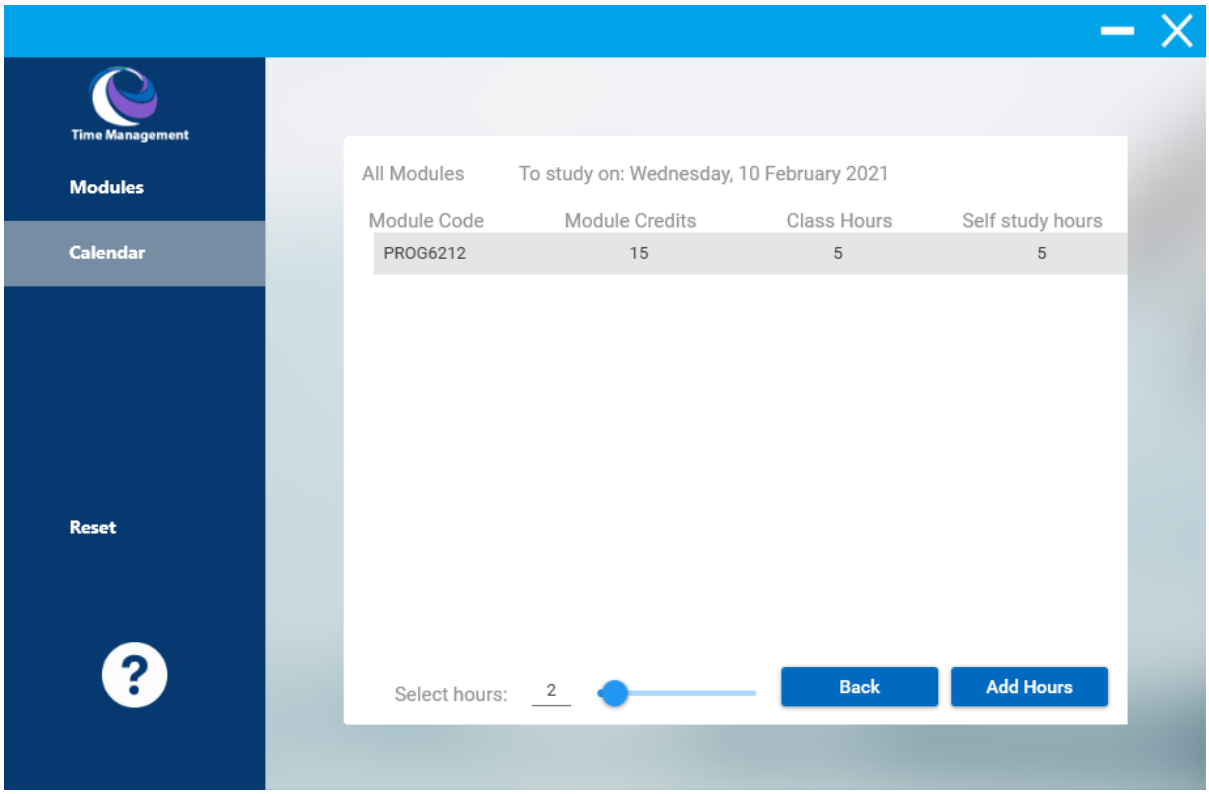

Figure 13: Selected Hours view: Wednesday 10 February (Chand, 2019)

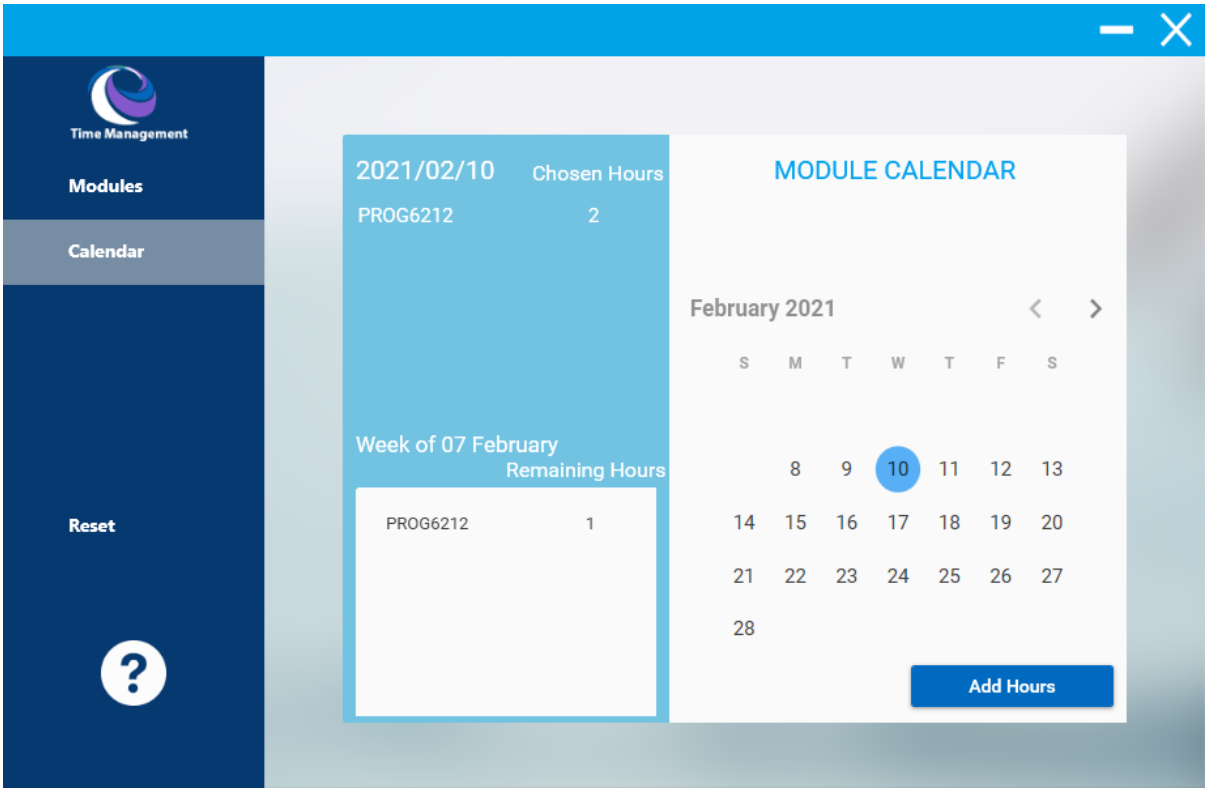

Figure 14: Calendar view, 10 February (wpf-tutorial, n.d.)

As the user added another 2 days to the week of 7 february, the program would subtract the 2 hours from the previous total of 3. This current week now only has one hour remaining.

StoredDates

In order for the program to know which dates have modules to study and which week has how many dates with modules to study, the program uses a list of **StoredDates** objects.

```
11 references
public class PlannedModule....
{
    /*
     * Class summary
     *
     * Used to store the modules planned for a specific day
     */
    public string codes;
    public int hours;
}

7 references
public class StoredDates
{   /*
     * Class summary
     *
     * Used to store the date along side the list of oolanned modules for the date
     */
    public string storedDate;
    public IList<PlannedModule> plannedList;//stores all modules
}
```

Figure 15: Class definitions for PlannedModule and StoredDates (TutorialsTeacher, 2021)

```
public static IList<StoredDates> datesList = new List<StoredDates>();//stores all modules for a specific day and the date
```

Figure 16: datesList, (Teacher, n.d.)

PlannedModule is a class which contains two variables **codes** and **hours.**

- **Codes:** Will store a module code such as, **PROG6212**
- **Hours:** Will store the number of hours a user would like to study for that module such as, **2**

StoredDates is a class which contains a **storedDate** string and **plannedList**

- **storedDate:** Will store the date a user decides to study on, eg: 08/02/2021
- **plannedList:** Will store a list of objects of **PlannedModule** type.

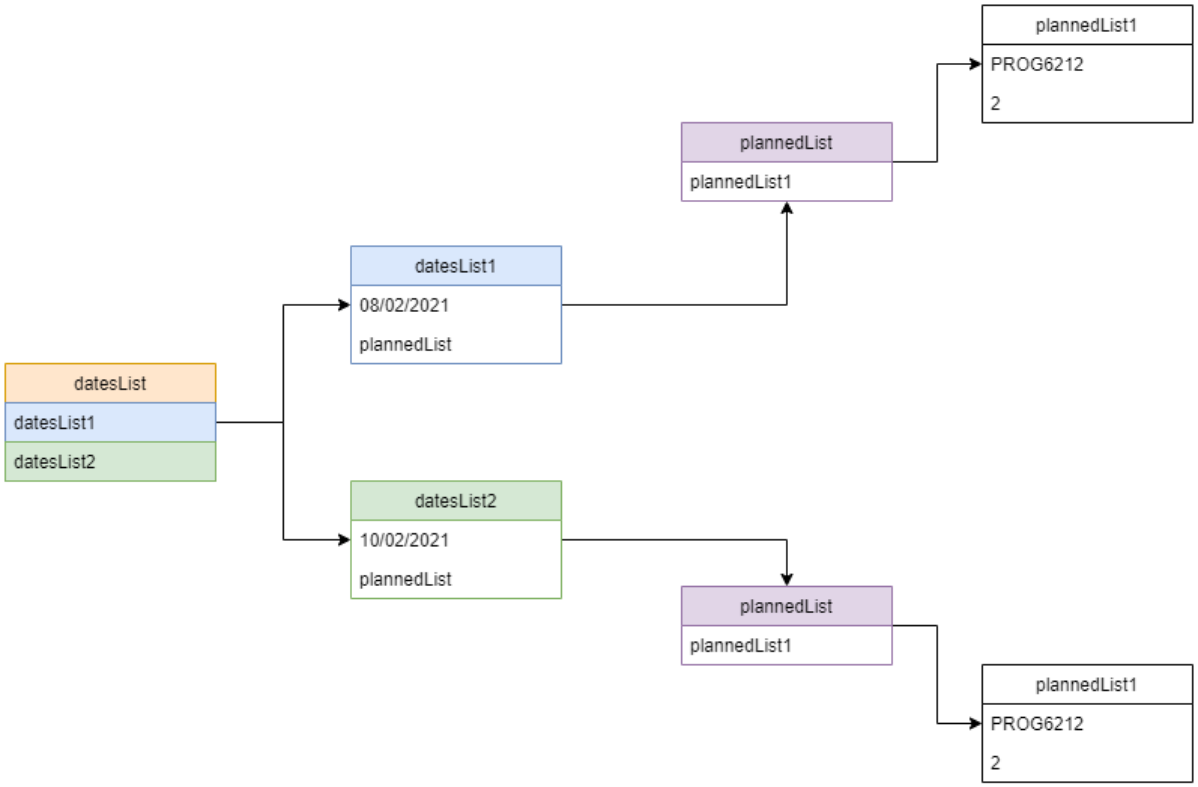

Figure 17: dateList structure according to Figure 14 example (Teacher, n.d.), (Teacher, 2021)

For the programmer to efficiently use the datesList, I wrote the following code. A clearer image can be found in the screenshots folder.

```
1 reference
public void dateStorer()//Used to add a date and the modules for the date
{
    IList<PlannedModule> istOfModules = new List<PlannedModule>();//Used to keep a temporary copy of the module codes and hours already stored within the dates list for a specific da

    List<string> foundCodes = new List<string>();//Stores a list of module codes that already exist for a specific day

    Boolean modFound = false;//used to determine if the current module being stored already exists within the curent dates list of modules
    Boolean datFound = false;//used to determine if the current date being stored already exists within the date list
    int datefound = 0;//Stores the location of the found date from the dates list, used to remove it later as it is replaced with an update for the specific day
    for (int i = 0; i < datesList.Count; i++)//repeat for the current length of the dates list
    {
        var currDate = datesList.ElementAt(i);//store the current element within the dateslist

        if (currDate.storedDate.Equals(SelectedDate))//we found a date that already exists
        {
            datFound = true;
            datefound = i;
            int repeat = currDate.plannedList.Count;//stores the count of modules within the current date

            for (int s = 0; s < repeat; s++)//now we are going to look at each module in found date
            {
                var curplan = currDate.plannedList.ElementAt(s);
                if (curplan.codes.Equals(ModuleCode))//if a module is the same as our current module to add
                {

                    istOfModules.Add(new PlannedModule()//add that module code and then add its old hours with our new hours
                    {
                        codes = curplan.codes,
                        hours = curplan.hours + ModuleHours
                    });
                    modFound = true;
                }
                else
                {
                    if (!foundCodes.Contains(curplan.codes))// check if our found codes does not contain the current code from the current date
                    {
                        istOfModules.Add(new PlannedModule()//add that module and the add its code
                        {
                            codes = curplan.codes,
                            hours = curplan.hours
                        });
                        foundCodes.Add(curplan.codes);
                    }

                }
            }
```

Figure 18 dateStorer() part 1 (Teacher, n.d.)

```
            if (!modFound)//If the day did not have the module we are trying to add
            {
                //we want every module stored on said day
                istOfModules = currDate.plannedList;// make a copy of the current fays planned modules
                istOfModules.Add(new PlannedModule()//add our new module and its code
                {
                    codes = ModuleCode,
                    hours = ModuleHours
                });
                datesList.RemoveAt(i);
                datesList.Add(new StoredDates()
                {
                    storedDate = SelectedDate,
                    plannedList = istOfModules
                });
            }
        }
    }
    if (!datFound)//If we did not find the current day within our dates list

    {
        //First add the module and code we want for this current day
        istOfModules.Add(new PlannedModule()
        {
            codes = ModuleCode,
            hours = ModuleHours
        });

        //add the current day to the dates list
        datesList.Add(new StoredDates()
        {
            storedDate = SelectedDate,
            plannedList = istOfModules
        });
    }

    if (datFound)//if the date was found
    {
        //first remove the previous version of the current day
        datesList.RemoveAt(datefound);

        //now add the current day and the updated list of modules which includes the old modules and our new addition
        datesList.Add(new StoredDates()
        {
            storedDate = SelectedDate,
            plannedList = istOfModules
        });
    }
    duplicateChecker();//check if we have any duplicates, just in case
}
```

Figure 19 dateStorer() part 2

For this example we will use the data in figure 17 and will have the user store the following values
ModuleCode = PROG6212
ModuleHours = 1
SelectedDate = 08/02/2021

The program will first store the **ModuleCode**, **ModuleHours** and **SelectedDate** for the selected module (this code is not shown within the dateStorer()). The first for loop ensures the loop will run for each element stored within **datesList** (for the ongoing example there are 2 elements). The variable **currDate** will select the first element stored within the **datesList.**
Next an if statement will determine if the **currDate.StoredDate** is equal to **SelectedDate?** (the currDate.StoredDate would be 08/02/2021 which matches our

SelectedDate). and retrieve the first **plannedList** (figure 17 shows it to be plannedList1).
The variable **datFound** is set true and the location of the found date within the datesList is stored within **dateFound**(They will both be useful later).

The second for loop will repeat for the length of modules within our plannedList (figure 17 shows only one object within the plannedList).
The variable **currplan** is set to the very first item within our plannedList as the for loop is currently in its first run. (TutorialsTeacher, n.d.)

The program now checks if the **code** within the **currplan** matches our **ModuleCode**. This happens to be true(refer to figure 17). As the if statement is true, we will create a list of type **PlannedModule** and store our **ModuleCode/currplan.codes** (as they are the same) and we will increment the hours already stored for that module with our **ModuleHours**(in this example the hours were 2, we are adding 1 for a total of 3). The program then sets **modFound** to true.(Will be useful later).
The else statement exists to check if the module we were trying to add exists within our list of found codes as it does not match the code we are trying to add. This ensures that any modules that were already stored for the day but do not match the module we are adding are not removed from the day.

Once the second forloop ends we have an if statement that checks if the module was ever found for the day. If it was not, the program will now add our **ModuleCode** and **ModuleHours**, it will then remove the current item in the datesList and add a new item(we are replacing the old 08/02/2021 with our new one).

Once the first forloop has either been broken or completed its loops, an if statement will check if the code ever found the date we were trying to add to the datesList, if not the code will add our **ModuleCode** and **ModuleHours** and the **SelectedDate**

The last if statement exists for when the date was found, we remove the old version of the date and replace it with our new one.

Determining Current Week

To determine the current week we need to first understand how **day.DayOfWeek** and **day.AddDays** work. The **DayOfWeek** (Microsoft, n.d.) method returns an int value for the current day of week, for example a Wednesday would be an int value of 3 (refer to table 1). **AddDays** (Microsoft, n.d.)method will add an int value of days to a day you give and return the date for it. For example if you add 2 days to the date 08/02/2021 you would get 10/02/2021. Now figure 20 represents the use of these 2 methods with a combination of string manipulation to return the date without the year bit at the end. Monday, 8 February 2021 = 8 February. (adegeo, 2017)

Day	Index
Sunday	0
Monday	1
Tuesday	2
Wednesday	3
Thursday	4
Friday	5
Saturday	6

Table 1: DayOfWeek index

```
void calculateCurrentWeek(DateTime day)//Figuring out which week we are in currently
{

    //String manipulation to the max
    int currentDay = Convert.ToInt32(day.DayOfWeek); //5

    String edit = day.AddDays(-currentDay).ToString("D");// Monday, 15 June 2009
    Boolean spaceMissing = true;
    int index = 0;

    //Finding the first empty string character
    do
    {

        if (edit.Substring(index, 1).Equals(" "))
        {
            spaceMissing = false;
        }
        index++;
    }
    while (spaceMissing);

    /*
     * Eg: String is "Monday, 15 June 2009"
     *
     * THe code below will extract only the "15 June " part
     */
    int start = index;
    int spacesCount = 0;
    edit = edit.Substring(start);
    index = 0;
    do
    {
        if (edit.Substring(index, 1).Equals(" "))
        {
            spacesCount++;
        }
        index++;
    }
    while (spacesCount != 2);

    string edited = edit.Substring(0, index);
    TWeek.Text = "Week of " + edited;
}
```

Figure 20 calculateCurrentWeek() (tutorialspoint, n.d.)

The calculateCurrentWeek() method allows the program to take a date such as the 11th of February 20201, convert that into the day of the week which is 4 and finally subtract those days -4 to that date to retrieve the 7th of February, which allows the program to know which date would be the start of a particular week.

```
void displayCurrentWeekModule(DateTime day)
{
    List<string> currentWeekCode = new List<string>();
    List<int> currentWeekHours = new List<int>();
    LCurrentWeekModules.Items.Clear();
    int currentModuleSelfHours = 0;
    int currentDayInt = Convert.ToInt32(day.DayOfWeek); //5

    for (int i = 0; i < 7; i++)// Repeats for each day of the week
    {
        String currentDay = day.AddDays(-currentDayInt + i).ToString().Substring(0, 10);
        for (int s = 0; s < CalendarModel.datesList.Count(); s++) //Repeats for each item in dateslist
        {
            var currentDate = CalendarModel.datesList.ElementAt(s);
            if (currentDate.storedDate.Equals(currentDay)) //check if the currentdate is our current day
            {
                for (int t = 0; t < currentDate.plannedList.Count(); t++) //now we are going to extract every planned module object for the day
                {
                    var currentList = currentDate.plannedList.ElementAt(t);
                    if (currentWeekCode.Contains(currentList.codes)) //check if the list we currently has this module
                    {
                        for (int b = 0; b < currentWeekCode.Count(); b++) //repeat for the length of added modules in our list
                        {
                            if (currentWeekCode.ElementAt(b).Equals(currentList.codes))//if we find the right module code
                            {
                                int currTotal = currentWeekHours.ElementAt(b);//8
                                //currentWeekHours is alwayys remaining hours
                                // now we should subtract this total from self hours

                                for (int v = 0; v < Program.moduleList.Count(); v++)//let us retrieve the self hours for this module
                                {
                                    var currentProgram = Program.moduleList.ElementAt(v);
                                    if (currentProgram.codes.Equals(currentList.codes))
                                    {
                                        currentModuleSelfHours = currentProgram.selfHours;//9
                                        break;//Stop searching as we found what we needed
                                    }
                                }
                                currTotal = currentModuleSelfHours - currTotal;//9-8 = 1
                                int newTotal = currTotal + currentList.hours; //retrieving the current total for week hours
                                int remainingHours = currentModuleSelfHours - newTotal; //this is the remaining hours
                                if (remainingHours < 1)
                                {
                                    remainingHours = 0;
                                }
                                currentWeekCode.RemoveAt(b);
                                currentWeekHours.RemoveAt(b);

                                currentWeekCode.Add(currentList.codes);
                                currentWeekHours.Add(remainingHours);
                                break;//stop searching
                            }
```

Figure 21 displayCurrentWeekModule(), (Jon, 2012)

```
                    else
                    {
                        currentWeekCode.Add(currentList.codes);
                        for (int v = 0; v < Program.moduleList.Count(); v++)//let us retrieve the self hours for this module
                        {
                            var currentProgram = Program.moduleList.ElementAt(v);
                            if (currentProgram.codes.Equals(currentList.codes))
                            {
                                currentModuleSelfHours = currentProgram.selfHours;
                                break;
                            }
                        }
                        int remainingHours = currentModuleSelfHours - currentList.hours;
                        if (remainingHours < 1)
                        {
                            remainingHours = 0;
                        }
                        currentWeekHours.Add(remainingHours);
                    }
                }
            }
        }
    }
    //run through module list, check if elemet at 1 is in current week, if not add it to current week with its self study hours xD
    for (int i = 0; i < Program.moduleList.Count; i++)
    {
        var currentModule = Program.moduleList.ElementAt(i);
        if (!currentWeekCode.Contains(currentModule.codes))// if our list of current week modules has the
        {
            currentWeekCode.Add(currentModule.codes);
            currentWeekHours.Add(currentModule.selfHours);
        }
    }
    if (currentWeekCode.Count==0)
    {
        LCurrentWeekModules.Items.Add("No modules to display");
        LCurrentWeekModules.Items.Add("Add a module");

    }
    else
    {
        for (int i = 0; i < currentWeekCode.Count; i++)
        {

            LCurrentWeekModules.Items.Add(" " + currentWeekCode.ElementAt(i) + "\t\t" + currentWeekHours.ElementAt(i));
        }
    }
}
```

Figure 21 displayCurerntWeekModule part 2 (tutorialspoint, n.d.)

currentWeekCode and **currentWeekHours** are parallel lists with **currentWeekCode** storing the module code and **currentWeekHours** storing the respective total hours for that module.

currentModuleSelfHours will keep track of how many hours have been recorded for the current module.

currentDayInt will keep track of which day of the week the for loop is currently on.

The first For loop will repeat for 7 days (as there are 7 days in a week). **currentDay** will store the date format for the currentDay of the loop (2021/02/07) as this is used to match other dates within the **datesList**.

The second forloop will repeat for the length of the **datesList**. **currentDate** is set to the loop element of **datesList** , Now an if statement will check if the storedDate within the **currentDate** is equal to the **currentDay** (Checking if the date within our dates list is the loop day of the current week).

The third for loop will start if the the **datesList** current element is within the **currentweek**, this loop will repeat based on how many **plannedList** objects exist for the specified day. **currentList** is set to the loop element of **plannedList.** Now the program will check if we already have the module code found at this element within our **currentWeekCode** list, if so we loop through the list till we find it. If we do find the current module within the **currentWeekCode** list, we store the hours found for it in **currTotal** as they are the stored remaining hours of that module. The code then retrieves the selfstudy hours for the module and sets **currentModuleSelfHours** to this value. The program then updates the **currTotal** value by subtracting the accumulated remaining hours for the current module from the just found remaining hours for the current module(the value stoed in **currTotal** is all the chosen self study hours). **newTotal** then becomes the addition of all previously chosen hours (**currTotal**) and the hours stored within **currentList. remainingHours** is checked in case it is below 0. The old **currentWeekCode** and **currentWeekHours** is removed and updated.

The else statement refers to the if comparing **currentWeekCode.Contains(currentList.codes)**, in the event that the code being looked at has not been found before, it is simply added to the list of **curentWeekCode**, the program then retrieves the value of **currentModuleSelfHours** in order to determine the remaining hours for this module.

Now a usability forloop is used to check whcich modules the user has not planned for the week, if it finds any it will then add them to **currentWeekCode** and the selfStudyHours for that module as well (this will show a module and their remaining hours), if there are no elements within **currentWeekCode** the program will then alert the user to add a module.

6. LINQ

```
public void display()//used to display every module stored within module list
{
    String fakeTab = "          ";
    for (int i = 0; i < Program.stored; i++)
    {
        if (!Program.chosenIDs.Contains(i))//repeat for the length of stored modules, checking ID
        {
            var currentModule = from m in Program.moduleList
                                where m.moduleID == i
                                select new { m.codes, m.credits, m.selfHours, m.hours };//Retrieving relevant data for ID

            foreach (var item in currentModule)//Displaying the data from ID
                LDisplayModules.Items.Add(item.codes + "\t\t" + fakeTab + item.credits + "\t\t\t" + fakeTab + item.hours + "\t\t\t" + fakeTab + item.selfHours);
        }
    }
}
```

Figure 22 display method using LINQ (Teacher, n.d.), (Teacher, n.d.)

Using LINQ my program is able to not only create variable methods as displayed in figures 22 and figure 23 but it also allows the program to use query like code blocks.

```
var currentModuloe = Program.moduleList.ElementAt(i);
```

Figure 23 var objects using ElementAt (Teacher, n.d.)

```
char[] chars = input.ToCharArray();
for (int i = 0; i < 4; i++)
{
    if (!Alphabet.Contains(chars[i]))
    {
        passed = false;
    }
}
```

Figure 24 indexing char array (geeksforgeeks, 2019)

7. Custom Class Library

Figure 25 shows the structure of the class library (tdykstra, 2021), with the **Calculations** class being used to perform necessary calculations and the **Module** class being used to store the module information. Figure 26 represents the using directive with Figure 27 displaying the creation of a List object of type **Module.**

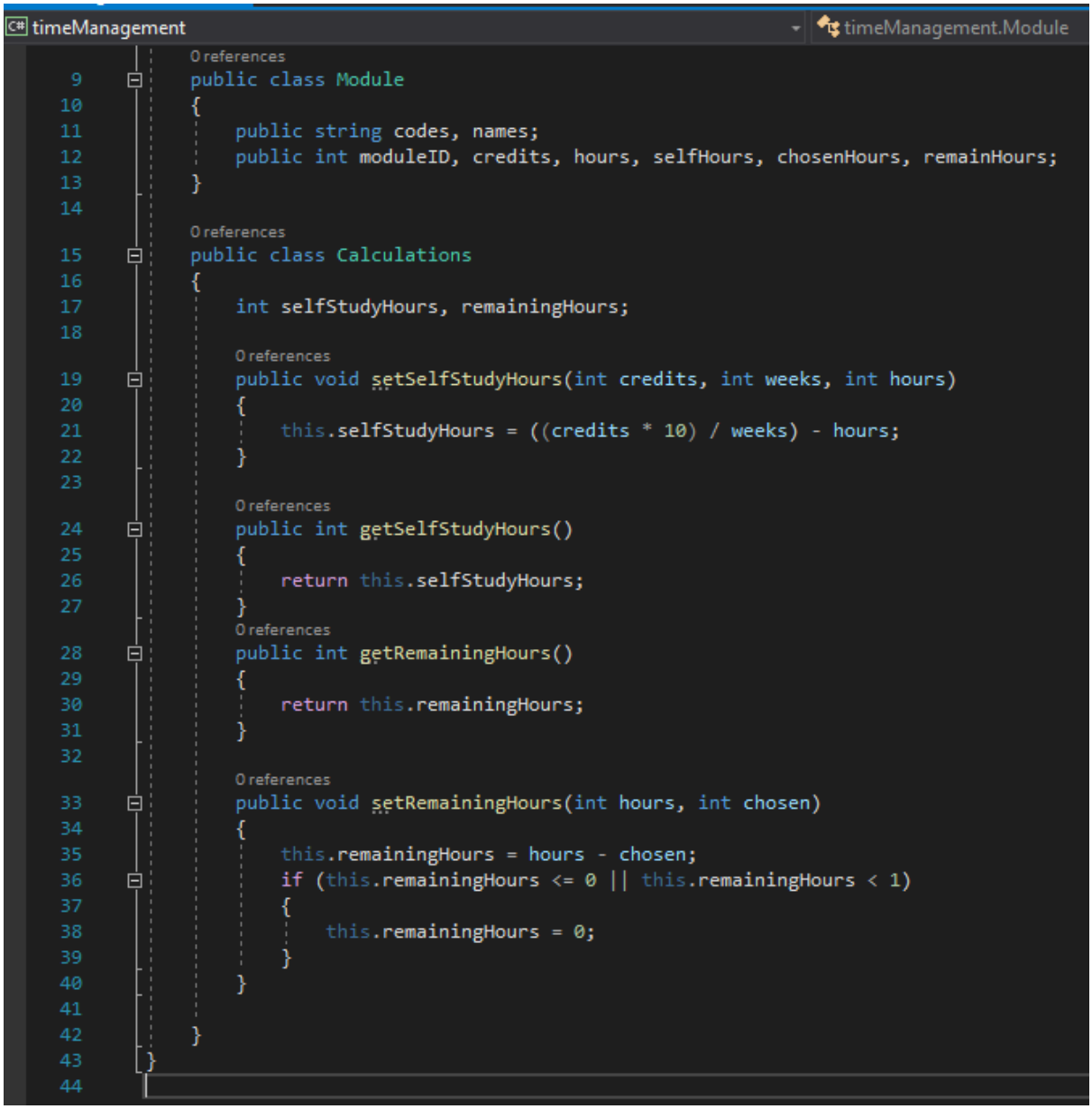

Figure 25 Time management classlibrary

```
using _20104681PROG6212JoshMkhari.MVVM.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using timeManagement;
```

Figure 26 Using timemanagement class library

```
//Custom Class Library
public static IList<Module> moduleList = new List<Module>();//stores all modules
```

Figure 27 ModuleList from class Library

8. Conclusion

It may never be possible to have a fully secure system that is immune to any form of attack. As long as a system needs to be logged into a backdoor, phishing technique or even malware can be invented to combat its defences. It is in every security analyst's best interest to safeguard their systems to a standard so high it automatically discourages bad actors from attempting to hack their systems.

REFERENCE LIST

adegeo, 2017. *How to: Extract the Day of the Week from a Specific Date.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/standard/base-types/how-to-extract-the-day-of-the-week-from-a-specific-date
[Accessed 17 September 2021].

adegeo, 2018. *Parse date and time strings in .NET.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/standard/base-types/parsing-datetime
[Accessed 17 September 2021].

Advance, R. C., n.d. *Modern UI, MultiColor Random Themes, Highlight Button-Active Form, WinForm, C #, VB.NET.* [Online]
Available at: https://rjcodeadvance.com/iu-moderno-temas-multicolor-aleatorio-resaltar-boton-form-activo-winform-c/
[Accessed 16 September 2021].

apc, 2014. *Watermark / hint / placeholder text in TextBox?.* [Online]
Available at: https://stackoverflow.com/questions/833943/watermark-hint-placeholder-text-in-textbox
[Accessed 17 September 2021].

BinaryTox1n, 2011. *making textblock readonly.* [Online]
Available at: https://stackoverflow.com/questions/5073244/making-textblock-readonly
[Accessed 17 September 2021].

Chand, M., 2019. *Listboc In WPF.* [Online]
Available at: https://www.c-sharpcorner.com/UploadFile/mahesh/listbox-in-wpf/
[Accessed 17 Se[tember 2021].

Davipb, 215. *Button Styles.* [Online]
Available at:
https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit/wiki/Button-Styles#materialdesignraisedlightbutton
[Accessed 17 September 2021].

EN, R. C. A., 2012. *Final Modern UI - Aero Snap Window, Resizing, Sliding Menu - C#, WinForms.* [Online]
Available at: https://www.youtube.com/watch?v=N5oZnV3cA64
[Accessed 16 September 2021].

foson, 2011. *How to make overlay control above all other controls?.* [Online]
Available at: https://stackoverflow.com/questions/5450985/how-to-make-overlay-control-above-all-other-controls
[Accessed 17 September 2021].

geeksforgeeks, 2019. *C# | ToCharArray() Method.* [Online]
Available at: https://www.geeksforgeeks.org/c-sharp-tochararray-method/
[Accessed 15 September 2021].

grapecity, 2021. *Setting the Calendar Start and End Date.* [Online]
Available at: https://www.grapecity.com/componentone/docs/wpf/online-datetimeeditors/Setting_the_Calendar_Start_and_End_Date.html
[Accessed 15 September 2021].

imagecolorpicker, n.d. *...pick your color online.* [Online]
Available at: https://imagecolorpicker.com/
[Accessed 17 September 2021].

Jallepalli, K., 2019. *MessageBox.Show Method in C#.* [Online]
Available at: https://www.c-sharpcorner.com/UploadFile/736bf5/messagebox-show/
[Accessed 17 September 2021].

Jon, 2012. *Converting a double to an int in C#.* [Online]
Available at: https://stackoverflow.com/questions/10754251/converting-a-double-to-an-int-in-c-sharp
[Accessed 16 September 2021].

lab, J. c., 2021. *WPF C# | How to customize Calendar Control in WPF? | UI Design in Wpf C# (Jd's Code Lab).* [Online]
Available at: https://www.youtube.com/watch?v=t5gMrygW05M&list=PLaOQFU4T-KW3S1CWzyYh1ekLOt5tUJadN&index=6
[Accessed 17 September 2021].

MaterialDesignInXamlToolkit, 2021. *ControlStyleList.* [Online]
Available at: https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit/wiki/ControlStyleList
[Accessed 17 September 2021].

Microsoft, n.d. *DateTime.Add(TimeSpan) Method.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.datetime.add?view=net-5.0
[Accessed 16 September 2021].

Microsoft, n.d. *DateTime.AddDays(Double) Method.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.datetime.adddays?view=net-5.0
[Accessed 17 Se[tember 2021].

Microsoft, n.d. *DateTime.DayOfWeek Property.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.datetime.dayofweek?view=net-5.0
[Accessed 17 September 2021].

Microsoft, n.d. *DateTime.Today Property.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.datetime.today?view=net-5.0
[Accessed 16 September 2021].

Oleson, C., 2021. *Super Quick Start.* [Online]
Available at:
https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkit/wiki/Super-Quick-Start
[Accessed 17 September 2021].

Price, C., 2015. *WPF set Textbox Border color from C# code.* [Online]
Available at: https://stackoverflow.com/questions/34168662/wpf-set-textbox-border-color-from-c-sharp-code
[Accessed 17 September 2021].

rj, p., n.d. *Final Modern UI – Aero Snap Window, Resizing, Sliding Menu – C#, WinForms.* [Online]
Available at: https://rjcodeadvance.com/final-modern-ui-aero-snap-window-resizing-sliding-menu-c-winforms/
[Accessed 15 September 2021].

tdykstra, 2021. *Tutorial: Create a .NET class library using Visual Studio.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/core/tutorials/library-with-visual-studio?pivots=dotnet-5-0
[Accessed 16 September 2021].

Teacher, T., 2021. *C# - List<T>.* [Online]
Available at: https://www.tutorialsteacher.com/csharp/csharp-list
[Accessed 16 September 2021].

Teacher, T., n.d. *Anatomy of the Lambda Expression.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-lambda-expression
[Accessed 15 September 2021].

Teacher, T., n.d. *LINQ API in .NET.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-api
[Accessed 15 September 2021].

Teacher, T., n.d. *LINQ Method Syntax.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-method-syntax
[Accessed 15 September 2021].

Teacher, T., n.d. *LINQ Query Syntax.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-query-syntax
[Accessed 15 September 2021].

Teacher, T., n.d. *Standard Query Operators.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-standard-query-operators
[Accessed 15 2021sEPTEMBER].

Teacher, T., n.d. *What is LINQ?.* [Online]
Available at: https://www.tutorialsteacher.com/linq/what-is-linq
[Accessed 15 September 2021].

Teacher, T., n.d. *Why LINQ?.* [Online]
Available at: https://www.tutorialsteacher.com/linq/why-linq
[Accessed 15 September 2021].

tricks, t., 2020. *WPF Controls | 10-Calendar | HD.* [Online]
Available at: https://www.youtube.com/watch?v=gvyWHB3i930&list=PLaOQFU4T-KW3S1CWzyYh1ekLOt5tUJadN&index=4
[Accessed 16 September 2021].

tutorialsEU, 2021. *WPF Tutorial for Beginners - Control Calendar.* [Online]
Available at: https://www.youtube.com/watch?v=0GEb57UXr-s&list=PLaOQFU4T-KW3S1CWzyYh1ekLOt5tUJadN&index=5
[Accessed 16 September 2021].

tutorialspoint, 2021. *C# int.TryParse Method.* [Online]
Available at: https://www.tutorialspoint.com/chash-int-tryparse-method
[Accessed 15 September 2021].

tutorialspoint, n.d. *C# Substring() Method.* [Online]
Available at: https://www.tutorialspoint.com/chash-substring-method
[Accessed 15 September 2021].

tutorialspoint, n.d. *WPF - Listbox.* [Online]
Available at: https://www.tutorialspoint.com/wpf/wpf_listbox.htm
[Accessed 17 September 2021].

TutorialsTeacher, 2021. *Element Operators: Single & SingleOrDefault.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-element-operator-single-singleordefault
[Accessed 15 September 2021].

TutorialsTeacher, n.d. *Element Operators : Last & LastOrDefault.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-element-operator-last-lastordefault
[Accessed 16 September 2021].

TutorialsTeacher, n.d. *Filtering Operator - Where.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-filtering-operators-where
[Accessed 15 September 2021].

Vudatha, P., 2017. *how do I set calendardaterange end to yesterday in xaml wpf c# [closed].* [Online]
Available at: https://stackoverflow.com/questions/45262624/how-do-i-set-calendardaterange-end-to-yesterday-in-xaml-wpf-c-sharp
[Accessed 17 September 2021].

Wagner, B., 2021. *How to convert a string to a number (C# Programming Guide).* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/how-to-convert-a-string-to-a-number
[Accessed 15 September 2021].

wpf-tutorial, n.d. *https://www.wpf-tutorial.com/list-controls/listbox-control/.* [Online] Available at: https://www.wpf-tutorial.com/list-controls/listbox-control/ [Accessed 17 September 2021].