Josh Mkhari

Student number: 20104681

PATHWAY: PROG6212

Lecturer: Michael Mapundu

26 October 2021

Programming 2B Task Two

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. Feedback

No changes were made in these sections (Points 1-6) as it makes no sense.

The rubric has a mark of 90% (which I still don't agree with) instead of the 89% displayed on VC portal.

Below I will write out points from the rubric in bold and make a case against the mark I received in those sections.

My points of dispute.

1) **Advanced Feature section**: 3/5

I believe I must be missing marks here; I will list features that are at least worth 3 marks each below.

- Material Design. (Used throughout the GUI) (Not taught in class or in module)
- Update View Command. (Used to change the views) (Not taught in class or in module)
- Observable Object Class (Used in conjunction with Update View Command) (Not taught in class or in module)
- Relay Command integration Class (A command class that benefits the Observable Object Class) (Not taught in class or in module)
- Use of updater slider within each view. (To allow program to determine view commands)
- **CalendarModel** class **DateStorer**(), **datesList** (This one on its own is so detailed and complex I am certain it wasn't even seen by you)
- **ViewCalendar xaml** displayCurrentWeekModule(DateTime day) (Another requirement that I am aware most of the entire course did not have running in their programs but it works in mine.)
- Reset function (Not taught in class or in module)
- Minimize function (Not taught in class or in module)
- Close function. (Not taught in class or in module)

The above list is 10 examples of concepts not taught in class or even within the scope of the PROG6212 module.

2) **App Functionality**: The Remaining hours for the week is correctly calculated and displayed to the user: I got 10/10

The comment says NO function to calculate.

Even though I did receive all 10 marks here the function does exist it is named **displayCurrentWeekModule(DateTime day)**

It is found within a folder called **View** and the class file of the **Calendar.xaml** which is also one of the advanced features listed above in the **Advanced Feature Section.**

3) **Coding Standards**: The code is well structured and documented: I got 3/5

When it comes to code structure I do understand that this one would be subjective to the eyes of the person reading through the code.

The comment I received here is Work on commenting.

Now considering how every line of every complex method has a comment on it explaining what is occurring and an entire section within the Task 1 documentation I submitted. (A 30-page document explaining in better detail wherever the comments could not as the document has diagrams, images and tables explaining concepts) If the comment I received here was related to the coding structure I would instead be asking what could be done but the comment I received is in relation to the comments placed within the code which is more than sufficient.

4) **Application Structure:** The application makes use of LINQ: I got 4/5

In the context of my program as It is built with task 2 in mind as this is something we are meant to be building to, there is no more efficient way to use LINQ statements within my program. The task went through exactly 9 iterations with different implementations of LINQ until the most efficient form was found. Efficiency meaning the statement does what it needs to without using any more lines or system memory. If there is a more efficient way, I am open to any suggestion.

Or is the problem lying within the quantity of 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 1 display method using LINQ (Teacher, n.d.), (Teacher, n.d.)

5) **App functionality:** The list of modules is displayed to the user. I got 9/10

At this point I believe my program was ran for less than 30 seconds, as this feature is the **first view** the user is met with after setting a semester period. The user is capable of adding a module and the module is displayed accordingly the moment it is added. The user is capable of accessing this list at any point in time. Once again if my Task 1 documentation was read, it also contains an example of the program running with screenshots of the list of modules showing as well.

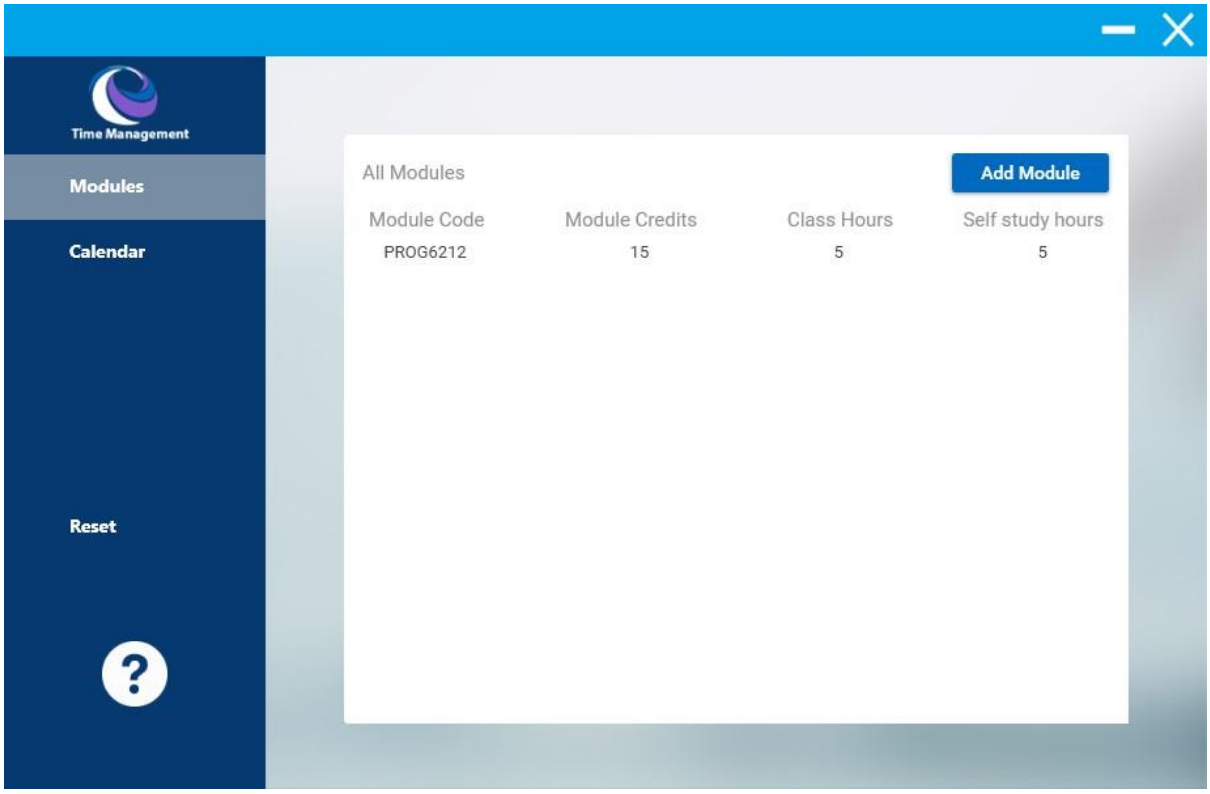


Figure 2: ModulesView (Advance, n.d.) (Jallepalli, 2019)

6) **Usability:** the user interface is easy to use: I got 8/10

Comment: Add more self-describing buttons.

Note: The interface was tested by 6 people without any complaints thanks to the help button.

Clean user interfaces are the direction programs are moving to, as such my program has a few buttons but to compensate for this a **help button** is visible on every single view. No matter where the user may find themselves, they can use the help button to learn how to navigate through the application.

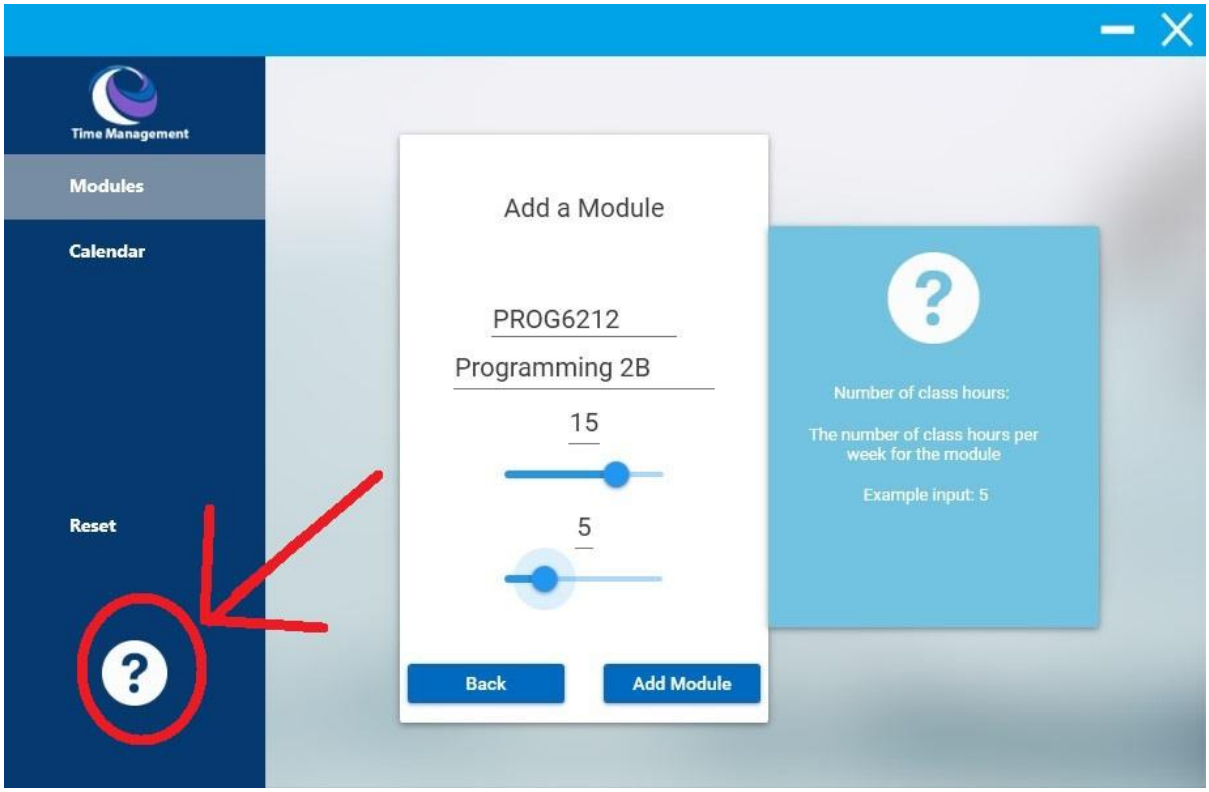


Figure 3 (ModuleAdder view) (EN, 2012), (imagecolorpicker, n.d.) (Price, 2015) (apc, 2014)

Feedback Remark

My biggest problem with receiving the mark 90% is that my program completely performs all the functionality requirements set out in Task 1 including the calendar requirement. As the class rep for group 1 I am aware of the challenges my classmates were facing in implementing the calendar parts of the assignment. Reading the comments on my rubric makes it seem as if my program was not read through extensively, the word document was mostly ignored and the program itself was not stress tested as it should have been. I do accept any decision you come will decide on but if not favourable I would just like to know what more can I do?

7) UML Class diagram

Comment: Included but cluttered everywhere

I completely agree, it was terrible. An update has been made.

Module Adder

3. Calendar View (An Advanced Feature)

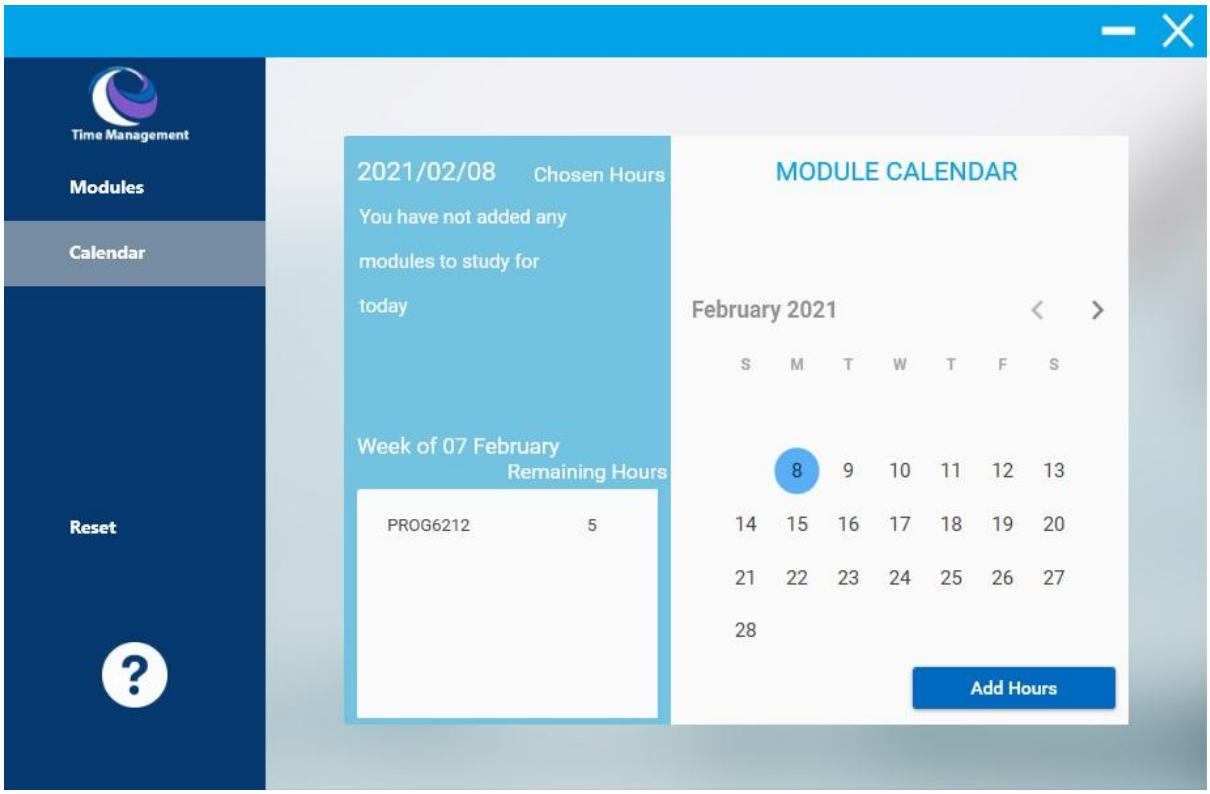


Figure 4: Calendar View, no chosen hours (tutorialsEU, 2021) (foson, 2011)

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.

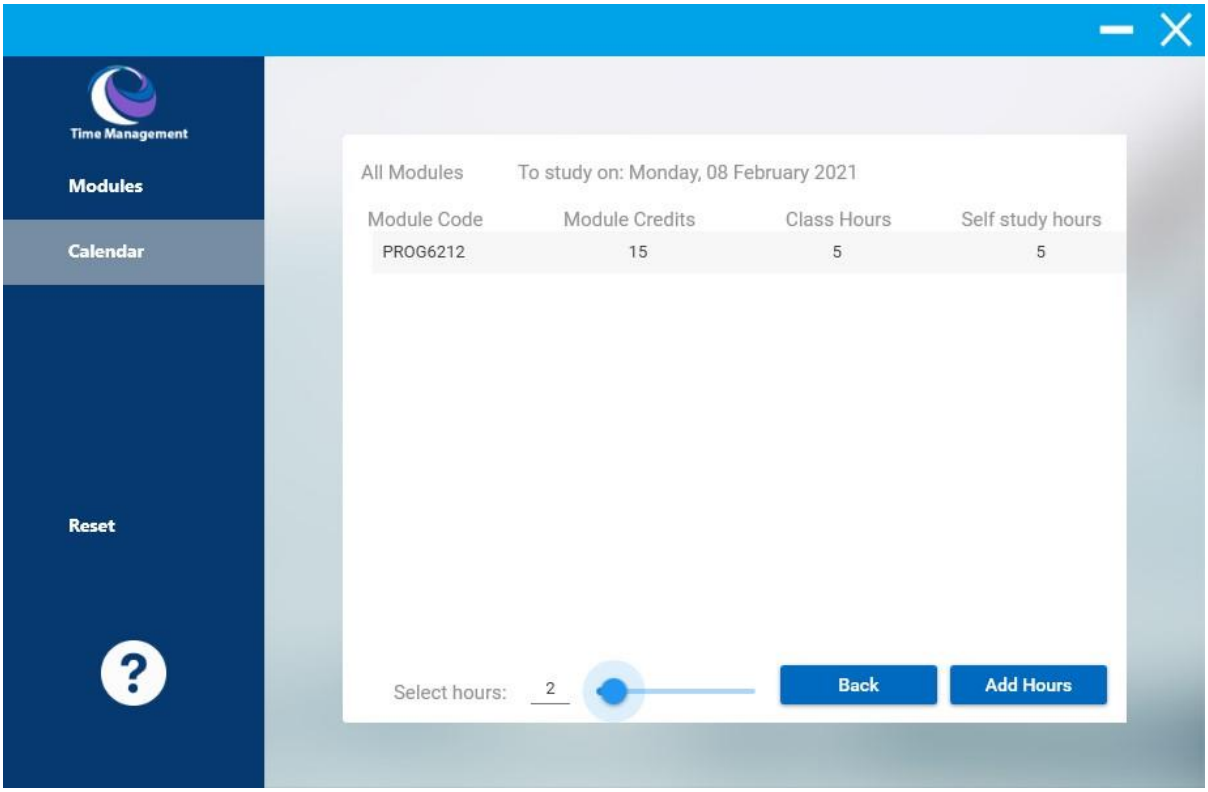


Figure 5: Select Hours view Monday 8 February (BinaryTox1n, 2011) (Davipb, 215)

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.

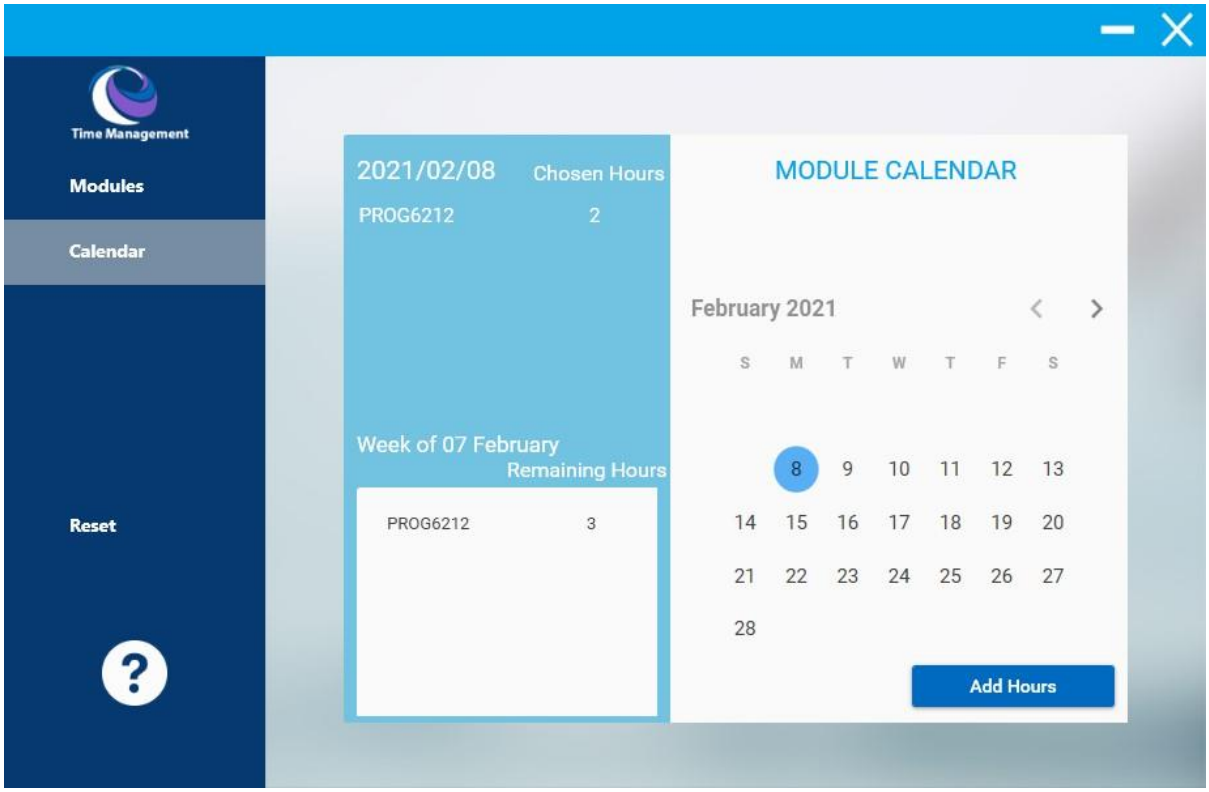


Figure 6: Calendar view with chosen hours

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.

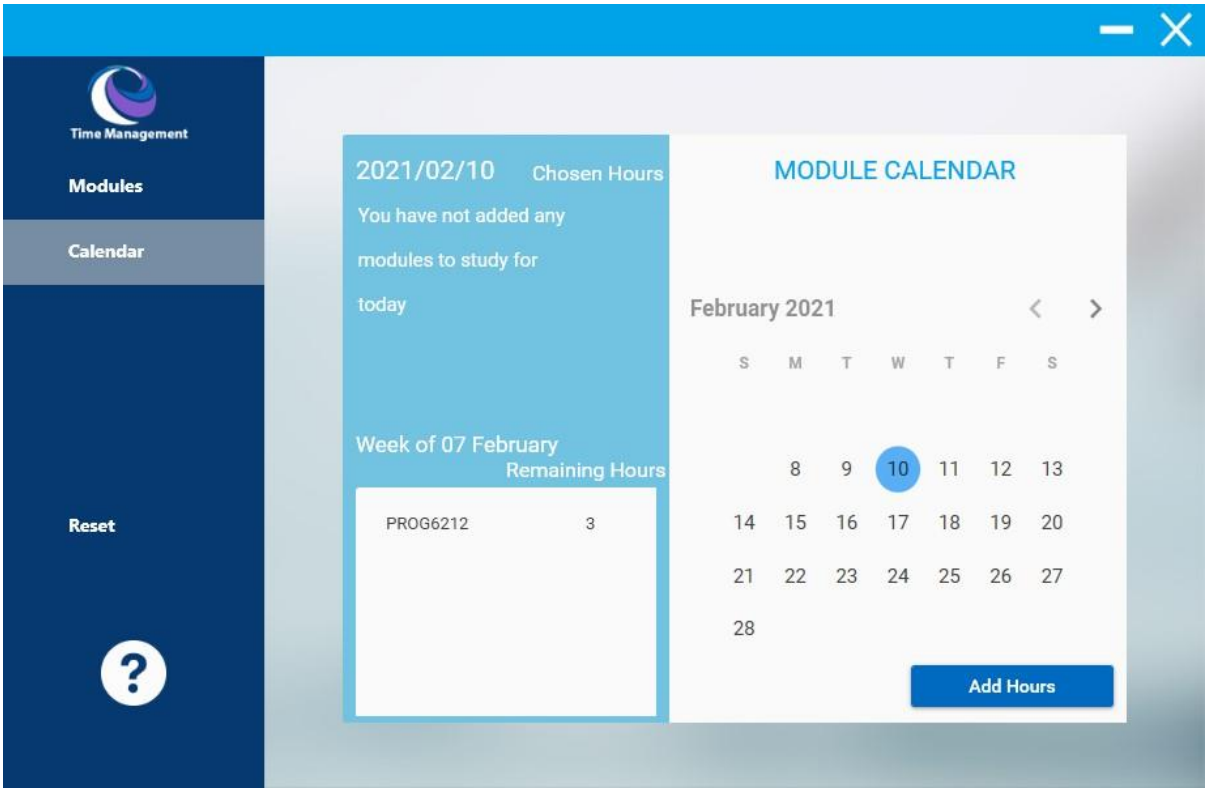


Figure 7: Calendar View, A different day

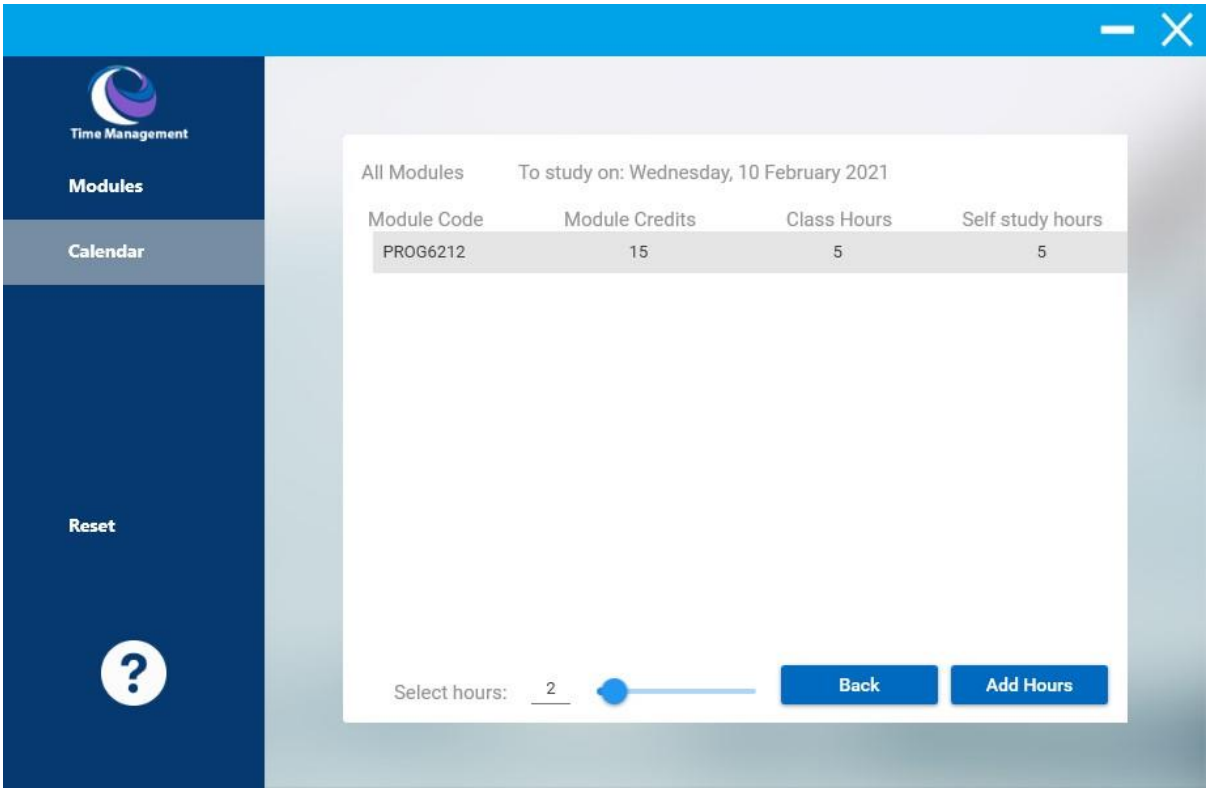


Figure 8: Selected Hours view: Wednesday 10 February (Chand, 2019)

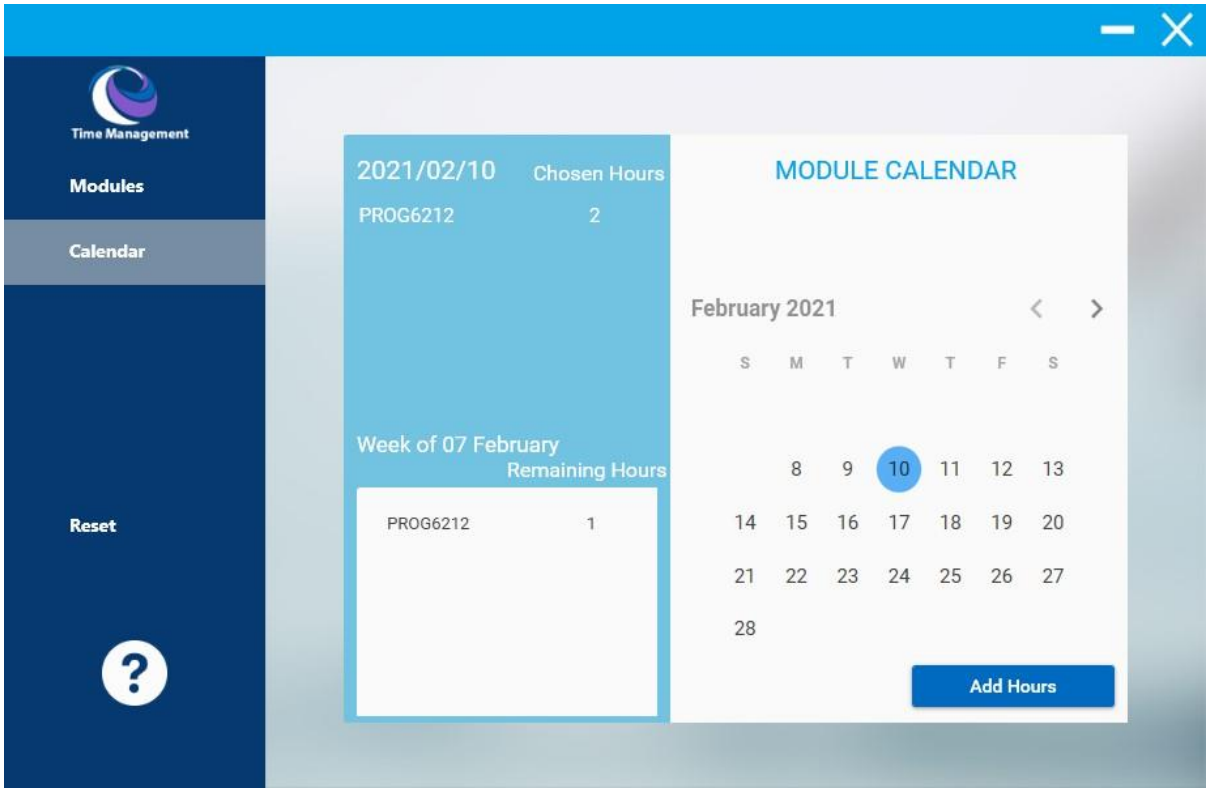
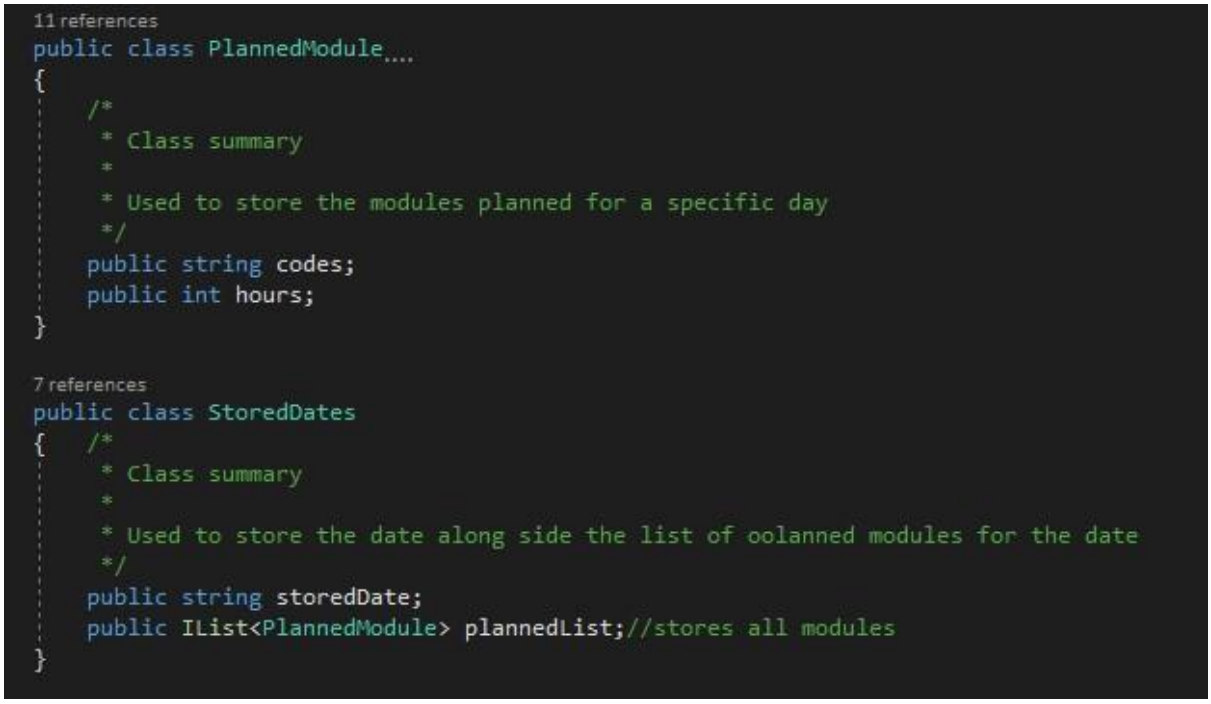


Figure 9: Calendar view, 10 February (wpf-tutorial, n.d.)

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

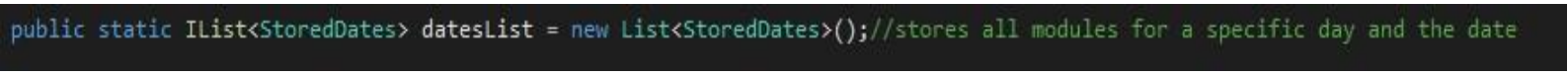
In order for the program to know which dates have modules to study and which week has how many dates with modules to study, the program uses a list of **StoredDates** objects.



```
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 10: 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 11: 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.

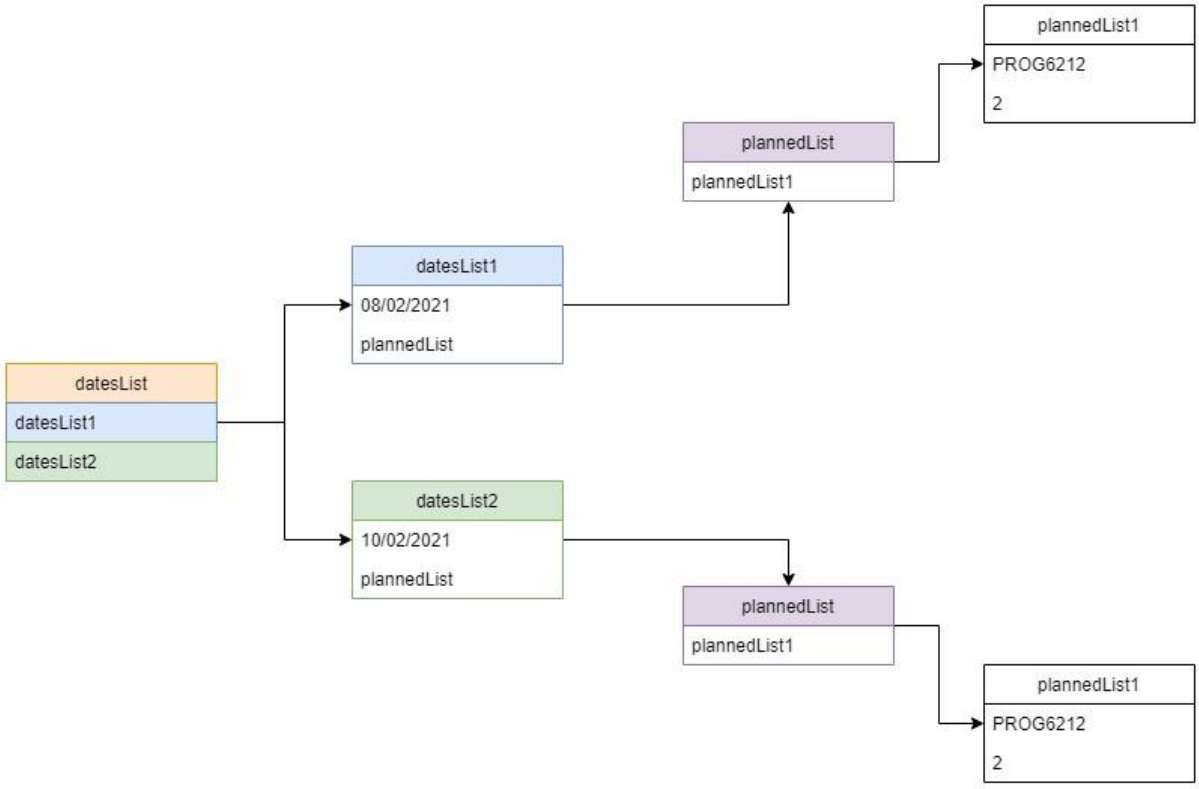


Figure 12: dateList structure according to Figure 14 example (Teacher, n.d.), (Teacher, 2021)

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

Figure 13 dateStorer() part 1 (Teacher, n.d.)

```
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);
                    }

                }
            }
```

```
                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 14 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 15 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.

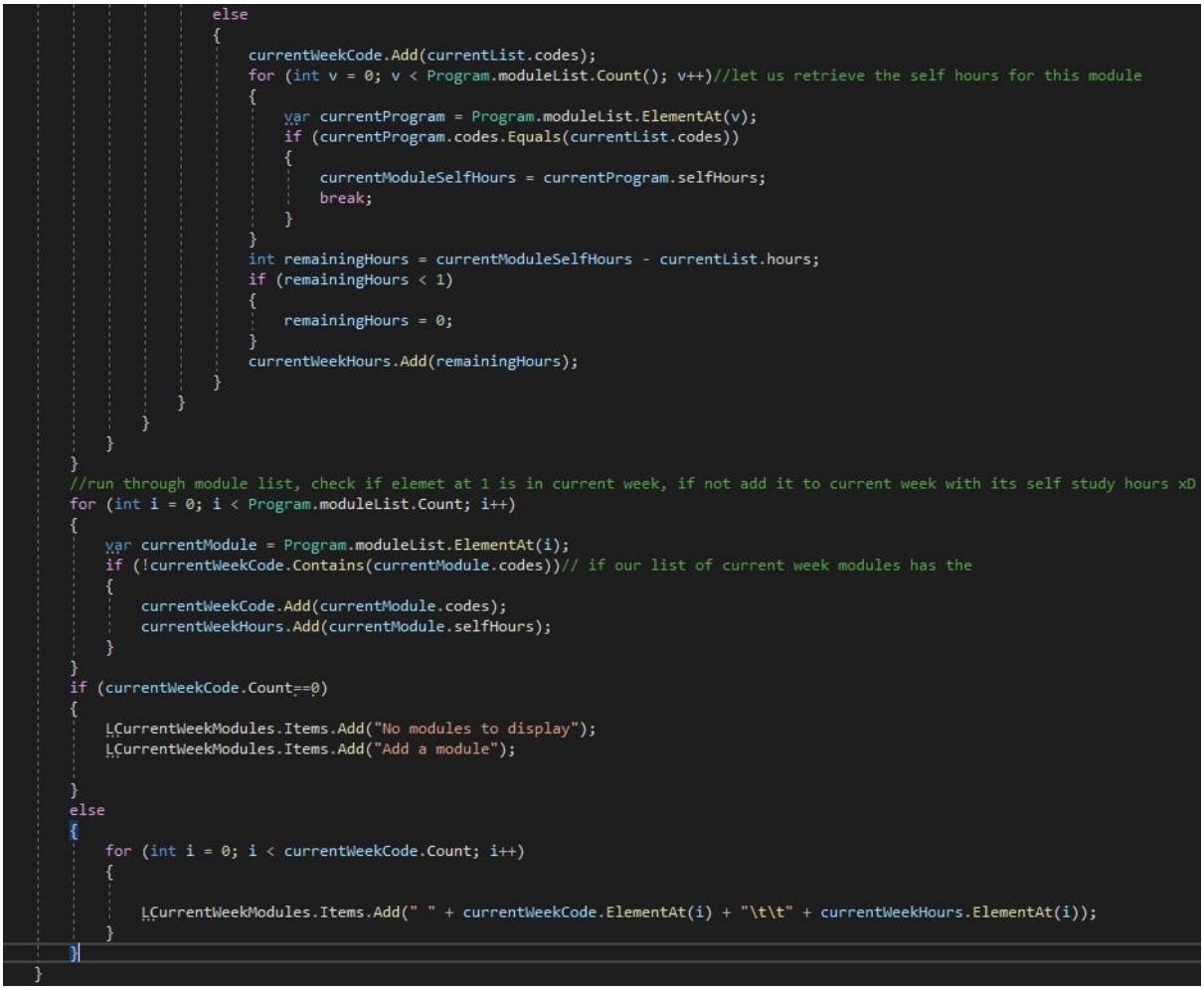
Figure 16 displayCurrentWeekModule(), (Jon, 2012)

```
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
                            }
```



```
                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 17 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.

4. DATABASE ERD

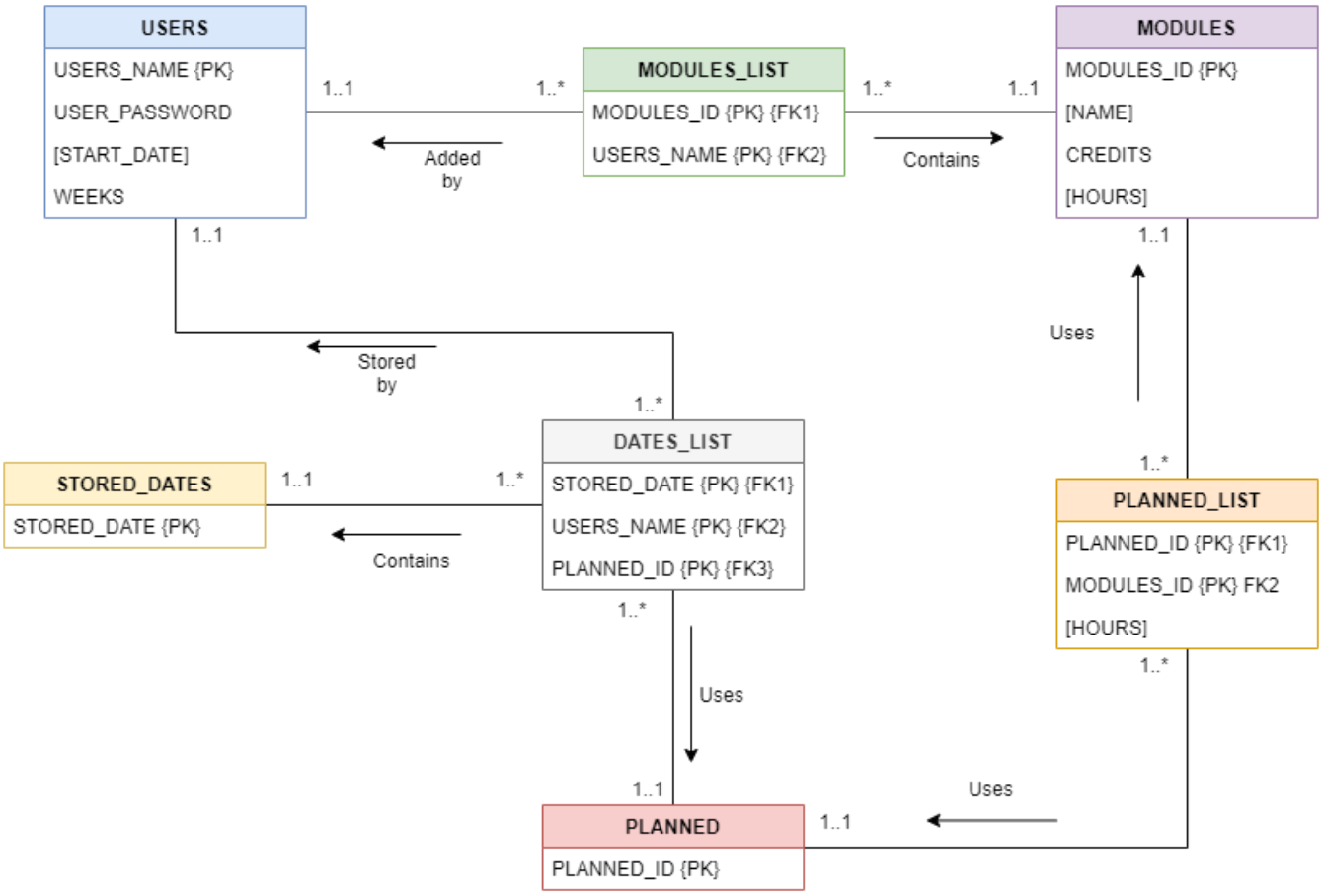


Figure 18 database ERD

5. Data saved to SQL database

Process of Adding a user

```
1 reference
public static void SaveDetails()
{
    UserModel use = new UserModel();
    use.User_Name = StartModel.Users[0];
    use.User_Password = StartModel.Users[1];
    use.Start_Date = StartModel.semesterStartDate;
    use.Weeks = Convert.ToInt32(StartModel.semesterWeeks);

    ProgramDAL pal = new ProgramDAL();
    pal.AddUser(use);
    CalendarModel.addingToDatabase = true;
}
```

Figure 19 SaveDetails()

```
1 reference
public void AddUser(UserModel use)
{
    using (SqlConnection con = new SqlConnection(connectionStringDEV))
    {
        SqlCommand cmd = new SqlCommand("SP_InsertUser", con);
        cmd.CommandType = System.Data.CommandType.StoredProcedure;

        cmd.Parameters.AddWithValue("@USERS_NAME", use.User_Name);
        cmd.Parameters.AddWithValue("@USER_PASSWORD", use.User_Password);
        cmd.Parameters.AddWithValue("@START_DATE", use.Start_Date);
        cmd.Parameters.AddWithValue("@WEEKS", use.Weeks);

        con.Open();
        cmd.ExecuteNonQuery();
        con.Close();
    }
}
```

Figure 20 sqlConnection for SP_InsertUser

```
CREATE PROCEDURE SP_InsertUser
(
 @USERS_NAME VARCHAR (50) = ' ',
 @USER_PASSWORD VARCHAR (100) = ' ',
 @START_DATE DATE =' ',
 @WEEKS INT = 0
)
AS
BEGIN
 INSERT INTO USERS (USERS_NAME,[START_DATE],WEEKS,USER_PASSWORD)
 VALUES (@USERS_NAME,@START_DATE,@WEEKS,@USER_PASSWORD)
END
```

Figure 21 Stored Procedure InsertUser

A similar setup is made for each SQL database function needed. The following Stored Procedures each have a method within the class **ProgDal.**

- SP_GetAllUsers
- SP_InsertModule
- SP_InsertModuleList
- SP_GetAModule
- SP_GetAllModulesList
- SP_InsertStoredDates
- SP_InsertPlanned
- SP_GetLastPlannedID
- SP_InsertDatesList
- SP_InsertPlannedList
- SP_UpdatePlannedList
- SP_GetPlannedFromDatesList

- SP_GetAllDatesList
- SP_GetAllStoreDates
- SP_GetModuleHours
- SP_Reset

6. Multi-Threading

```
if (UserModel.checkLogin(TUserName.Text, TPassword.Text))
{
    ModuleAdderModel mad = new ModuleAdderModel();
    Thread retModules = new Thread(new ThreadStart(mad.retrieveModules));
    retModules.Start();
    CalendarModel cm = new CalendarModel();
    Thread popDates = new Thread(new ThreadStart(cm.populateFromDatabase));
    popDates.Start();
    updater.Value = 5;
    StartModel.Users[0] = TUserName.Text;
    UserModel.loggedIn = true;
    Thread.Sleep(400);
}
```

Figure 22 Threading

Using Thread sleep in the main thread as accessing the database takes too long while the program moves ahead without it.

7. 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-toextract-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/parsingdatetime [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-aleatorioresaltar-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-hintplaceholder-text-in-textbox [Accessed 17 September 2021].

BinaryTox1n, 2011. *making textblock readonly.* [Online] Available at: https://stackoverflow.com/questions/5073244/making-textblockreadonly [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-overlaycontrol-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/onlinedatetimeeditors/S

etting_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-toan-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/enus/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/enus/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/enus/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/enus/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-bordercolor-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-resizingsliding-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-visualstudio?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-singlesingleordefault
[Accessed 15 September 2021].

TutorialsTeacher, n.d. *Element Operators : Last & LastOrDefault.* [Online]
Available at: https://www.tutorialsteacher.com/linq/linq-element-operator-lastlastordefault
[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-setcalendardaterange-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/programmingguide/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].