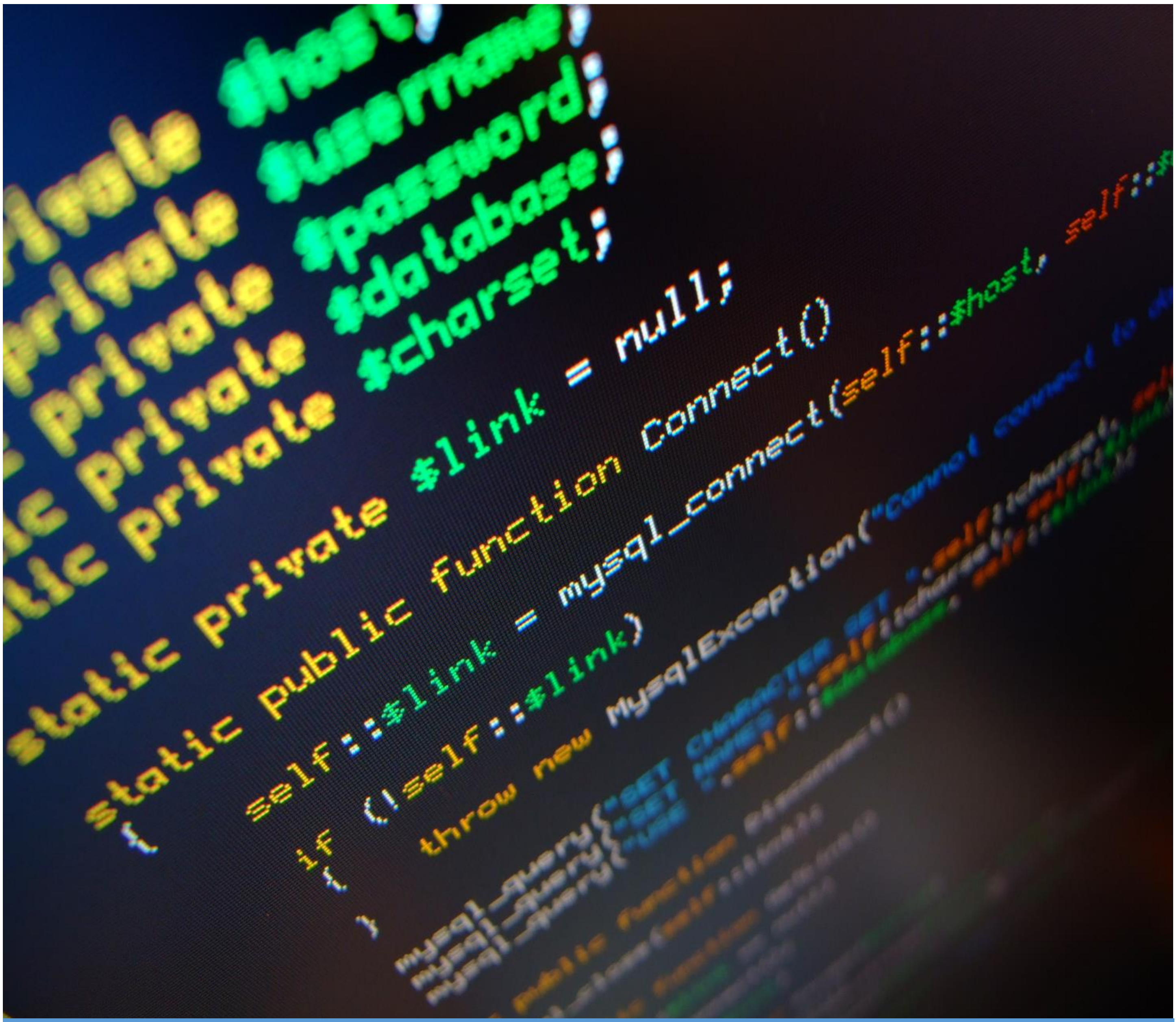


Digital Signage

Information Processing Technology

The creation and deployment of an interactive, fluid display in the ever changing digital landscape

Lachlan Stevens St. Patrick's College Mrs. Agius

Contents

Digital signage programs are used all within the industry world to implement and use real time signage within their businesses. So what is digital signage? Digital signage is where a TV or display monitor is plugged into a computing device to then be controlled with real time information on the current status of various pieces of information (be it flight information, etc.) During the course of this project a list of very specific step processes will be followed (this commonly referred to as the 'Software Development Cycle).

Identification

For this task, the chosen client was the head of students of the local college, St. Patrick's Mackay; Mr. Geoghegan. This client was after a digital sign to be implemented within the under croft, so that all students were able to be constantly up to date with the latest news from all over the world, enlisting a variety of resources. This display was envisioned to contain information such as the current and next lines of each year level (depending on the time of the day), constant updating with daily notices and meeting reminders for anything that is occurring during the lunchtime. Alongside this, most of the screen-space will be taken up by large dynamic images and text highlighting the latest news from all over the world. In essence, this program will provide the gateway between students' ability to understand and appreciate world problems while also offering a quick way to become educated while glancing at their next class.

To gauge an accurate understanding of how all of these elements were designed to work together a simple storyboard was constructed on grid paper (Appendix 1), this storyboard showed a basic layout in which these elements could be fitted and it allowed the client to state their thoughts and feedback, to allow changes to make the product more suited to the clients wants. Once this was shown to the client they verbally agreed on the design and continued explaining how it was envisioned inside of their head.

Once design had been settled upon, it was time to analyse the market to see what worked and what didn't work within these specific types of programs. Following from the research, various ideas became increasingly standard; the first of which being the implementation of a client and server model with the roles reversed for each. This means that the server (the interface the user interacts with) will send messages to the client (the digital sign) instead of the client normally sending messages to the server [this computational paradigm was replaced by a more traditional approach due to it ending up being more efficient in the long term availability of the program]. Alongside this, the program was going to need to have various clocks that refreshed at certain intervals, therefore thread affinity would become a concern that would need to be addressed.

For this project, the programming language C# (C Sharp) was utilised. Originally, nodeJS was going to be the foundation of this project although after some quick research C# having all of the .net library as its backend suggested a faster more streamlined development process. This came at its own cost, as the program will hence be only recommended to run on Windows Machines, whereas nodeJS would have allowed flexibility; running the exact same on all of the different types of platforms. This minor lack of features can be resolved through the use of "Mono" which acts as a 'compiler' for C# that works on any platform.

For this project the main simplifications will be the whole 'server-client' idea, with the main focus being on making sure the RSS Feeds are as interactive and appealing as possible. This is due to that being what draws the users in; making that easily the biggest section of this project that needs to perform optimally. Another simplification will the lack of scalability within the project, obviously due to the finite amount of space, it will be difficult to account for any future renovations that allow for a bigger screen, this means that each size the programs form becomes, each element will be placed based on static locations instead of dynamically changing to fit the given space. Another simplification of the program will the general lack of usability, most of the more ambitious ideas for this project (as discussed later on) were not able to be implemented due to the time constraints on this project.

Moving beyond simplifications, assumptions will be created when needing to actually implement the software ideas into the program. This is purely to allow for changes to be made during the development process which reflect on the overall growth of the project. Overall, this task is manageable when enough effort is placed to allow each part to be created with the degrees of performance. The degree of performance is a measure that allows the developer decide what impacts the user the most and how to prioritise each problem that arises.

The main software needed is Visual Studio IDE (integrated development environment) this tool allows C# programs be created. This software on top of Adobe Photoshop encompasses all of the software tools needed for this task. In terms of hardware, all is needed is to meet the minimum requirements of the previously stated programs. The target audience will be generally students at the ages of 15-18, this means that only news links that can be relevant to this age group should be implemented; although this is subjective as the whole point is to allow all kinds of stories from all over the world be found on this display. Knowing this target audience will help below when the testing phase comes along and the overall success of this project is needing to be known. The user of the client interface will be within the ages of 30 – 60 which means the interface needs to be very simplistic and not be convoluted with options.

Specification

During the development of this project, the overarching goal is to provide students an easy way to stay up to date with the world they live whilst also providing an easy way of finding out what is happening on around their local campus. To accomplish this task a simple inverse client server will be needed alongside various threads to communicate with the RSS server in order to keep the form continuously updated. Alongside this, delegates will need to be implemented into the program in order for each separate threads to have the ability to communicate with the main display form.

The only real 'prompts' to the user will be in the form of the localised windows message boxes. These are inbuilt to the Windows API and offer a quick, simple way to alert the user without disrupting the overall 'flow' of their experience.

This program takes inputs from a variety amount of sources, be it the mouse or keyboard through the use of textboxes, combo boxes or buttons. These inputs are send through the client interface which then talks to the server to get the interface across. From here the outputs are shown either through the client interface itself (in the form of a message box) or through the digital display server program (on the TV). Another output of this program is through the use of the RSS feeds where it polls the server looking for the latest news and shows the users the scraped feed.

Finally, the user interface of the program brings together all of the inputs and outputs and provides the gateway between the two in a perfect synergy. Without having a well-designed, easy user interface none of the other inputs and outputs would be able to work together. To ensure the interface is successful there are strict guidelines to follow; as will be discussed later.

During the course of this project whenever changes are made to the report or project they will be saved to both a project USB and be synced up with Google Drive. This is to ensure that in the event of a technological failure, a backup exists.

Formulisation

Find attached Appendices (1 – 4) for the original sketches of the proposed layouts with Appendices (5 – 8) containing a tree diagram of the envisioned code and the Nassi Schneiderman. Refer below for pseudo code.

Server

As can be noted on Appendix 1, the server is a simple interface with each section fairly well divided. On the right hand side of the screen there is three separations, the first enlisting the 'current line' if a more in-depth storyboard is needed refer to Appendix 9, where the revised storyboard can be seen when shown to the client. This was due to this being too vague and the client was revisited to allow for a clearer idea of what was wanted. As can be noted in Appendix 9, again it is a fairly grid based layout with 80% of the screen space devoted to the RSS feeds and 20% devoted to the school information.

The gradient will be added over the image to allow for the text to be readable no matter what the image is. On the side there is a multitude of colours being incorporated; this is to allow a brighter interface and attracts more attention to onlookers. Another thing to note is that the "Special Arrangements" was removed and replaced with the two separate grades lines. This was to allow for less cramming of information and let it be spread out and easier to read. Overall simple design and client really liked it.

Client

Moving onto the client interface (Appendix 2). This interface is very simple and only contains what is needed to avoid confusion. As can be noted there is a very clear succinct title along the top of the form to make it very clear to the user what form has been selected, with three very large buttons proceeding deeper into the program. At the top of the page a simple status label tells the user the current status of the server (if they are connected to one) and allows them to connect to one below by entering the IP address of the server. Normally, in a production environment this IP will be static and will be entered automatically through a configuration file.

Once connected the other buttons will become available for pressing to then proceed further into the functioning of the program. Beginning with the notices button (Appendix 3), this opens up the notices form which clearly shows the user what format to enter the data into the textboxes. It can be noted that the other two labels Meeting Reminders and Special events will be centred vertically along the centre of the class Changes and the proceeding label underneath. From here there is a simple submit button down the bottom which will connect to the server and process the information entered above.

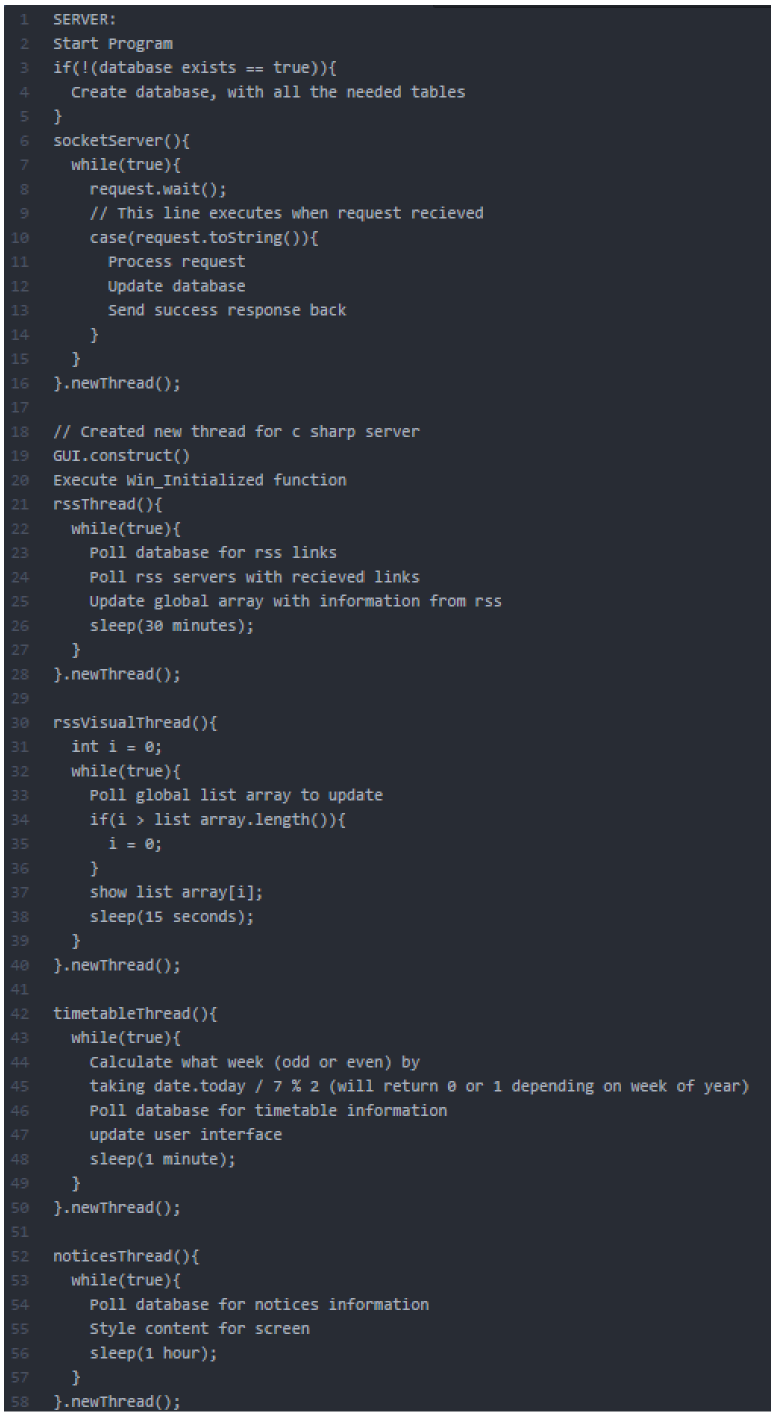
Next is the RSS Page (see bottom of Appendix 3). This page is simple by only displaying 5 textboxes and a simple submit button sending the information entered. Designing the page like this allows the user quick and easy setting of the sources used within the Server program and allows for quick error fix if the user types in a mistake. The submit program will show a message box if something goes awry, allowing seamless transition between the user's interaction with the program and the knowledge that something unexpected as occurred.

Finally, there is the timetable page (Appendix 4). This page looks convoluted but once it is being interfaced with by the client it will work out to be quite a quick process. The way this interaction is justified is through the fact that this form will be used very rarely, as updating the timetable only happens once a year. As for the data that is being sent over this form, this layout makes a lot of sense to fill out as there isn't much thinking involved, it is just click the entry from the drop down list and then repeat. This form is the concluded with a simple submit button which takes all of the information and sends it off to the server.

Across all of the forms used within this project, the font will be "Futura Std. Book" as this is a clean and easy to read font that works well no matter what the text. This font was recommended by the client for being very readable and didn't need to strain to read it. Alongside this, this font scales very well on bigger screens (which is exactly what is needed in this case).

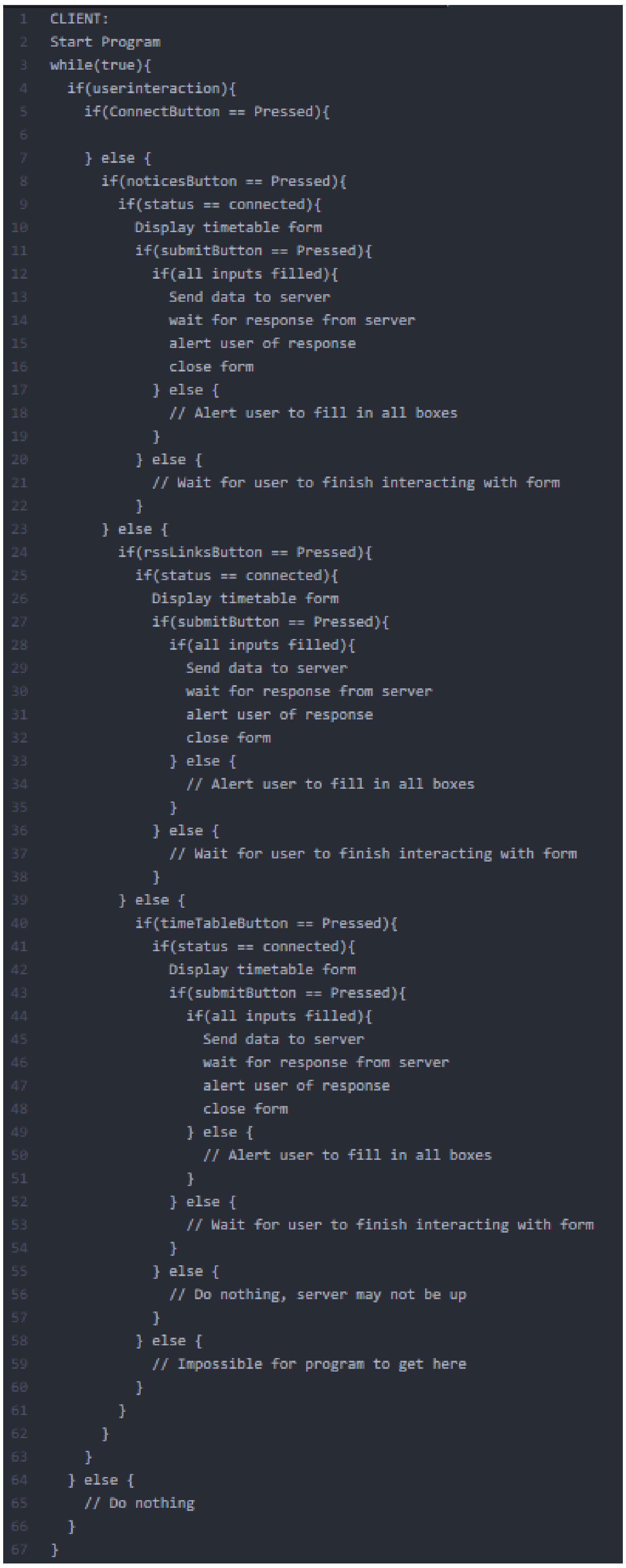
Pseudocode

Server



```
SERVER:
Start Program
if(!(database exists == true)){
  Create database, with all the needed tables
}
socketServer(){
  while(true){
    request.wait();
    // This line executes when request recieved
    case(request.toString()){
      Process request
      Update database
      Send success response back
    }
  }
}.newThread();

// Created new thread for c sharp server
GUI.construct()
Execute Win_Initialized function
rssThread(){
  while(true){
    Poll database for rss links
    Poll rss servers with recieved links
    Update global array with information from rss
    sleep(30 minutes);
  }
}.newThread();

rssVisualThread(){
  int i = 0;
  while(true){
    Poll global list array to update
    if(i > list array.length()){
      i = 0;
    }
    show list array[i];
    sleep(15 seconds);
  }
}.newThread();

timetableThread(){
  while(true){
    Calculate what week (odd or even) by
    taking date.today / 7 % 2 (will return 0 or 1 depending on week of year)
    Poll database for timetable information
    update user interface
    sleep(1 minute);
  }
}.newThread();

noticesThread(){
  while(true){
    Poll database for notices information
    Style content for screen
    sleep(1 hour);
  }
}.newThread();
```

Client



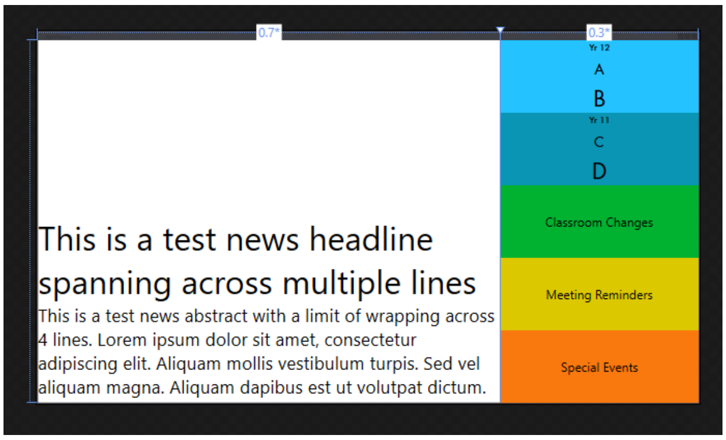
```
CLIENT:
Start Program
while(true){
  if(userinteraction){
    if(ConnectButton == Pressed){

    } else {
      if(noticesButton == Pressed){
        if(status == connected){
          Display timetable form
          if(submitButton == Pressed){
            if(all inputs filled){
              Send data to server
              wait for response from server
              alert user of response
              close form
            } else {
              // Alert user to fill in all boxes
            }
          } else {
            // Wait for user to finish interacting with form
          }
      } else {
        if(rssLinksButton == Pressed){
          if(status == connected){
            Display timetable form
            if(submitButton == Pressed){
              if(all inputs filled){
                Send data to server
                wait for response from server
                alert user of response
                close form
              } else {
                // Alert user to fill in all boxes
              }
            } else {
              // Wait for user to finish interacting with form
            }
        } else {
          if(timeTableButton == Pressed){
            if(status == connected){
              Display timetable form
              if(submitButton == Pressed){
                if(all inputs filled){
                  Send data to server
                  wait for response from server
                  alert user of response
                  close form
                } else {
                  // Alert user to fill in all boxes
                }
              } else {
                // Wait for user to finish interacting with form
              }
            } else {
              // Do nothing, server may not be up
            }
          } else {
            // Impossible for program to get here
          }
        }
      }
    }
  } else {
    // Do nothing
  }
}
```

Implementation

Server

Graphical Interface

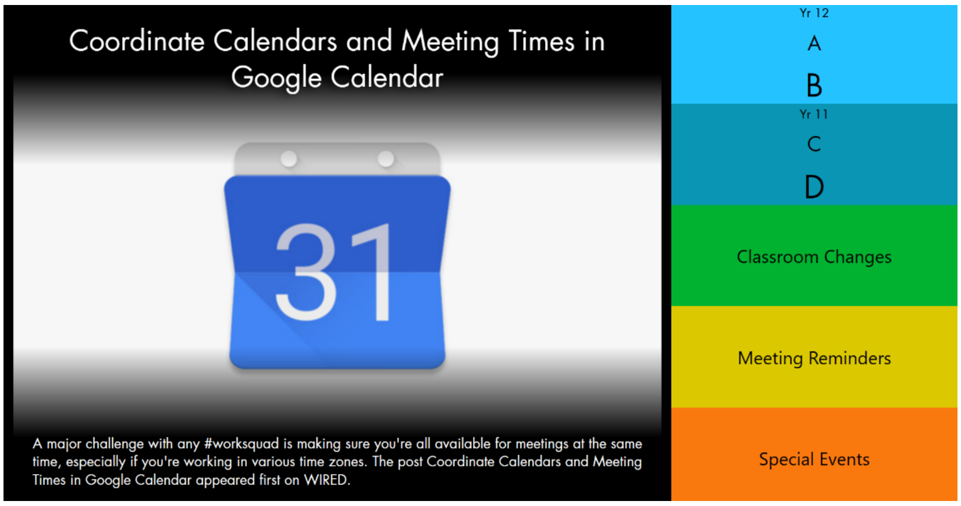
Once storyboard had been constructed (refer to Appendix 1) it was ready to transform this project from paper to product. To begin this transformation as simple methodology was devised, mimic the design (wireframe) in the visual program and see how well the elements worked together; from an application point of view. Inside of a new WPF (Windows Presentation Framework) form the mock-up was able to be achieved quickly and easily through the use of advanced XAML code which allowed for the XML construction of a mathematically perfect interface. As can be seen below in the first placement of the elements in their respective grids the RSS news section didn't look as attention drawing as it could have been. This is what caused a redesign of the interface, to achieve this the client was contacted and asked to offer input. After collaboratively brainstorming the end result was what can be noted in Appendix 9. This storyboard was much more in-depth and allowed more refinement on the final product.


This is a test news headline spanning across multiple lines

This is a test news abstract with a limit of wrapping across 4 lines. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam mollis vestibulum turpis. Sed vel aliquam magna. Aliquam dapibus est ut volutpat dictum.

First revision of Display program

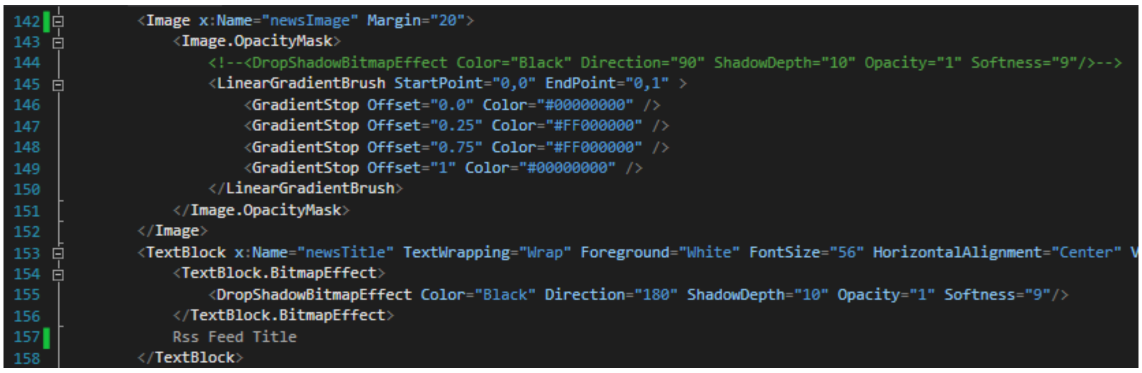
As can be noted below in the mock-up of Appendix 9, the gradient behind the text; originally this wasn't planned until it looked as though the white didn't 'pop out' of the display as it was supposed to. To achieve this gradient a linear overlay was added to the image element.


Coordinate Calendars and Meeting Times in Google Calendar

A major challenge with any #worksquad is making sure you're all available for meetings at the same time, especially if you're working in various time zones. The post Coordinate Calendars and Meeting Times in Google Calendar appeared first on WIRED.

Second revision of Display program

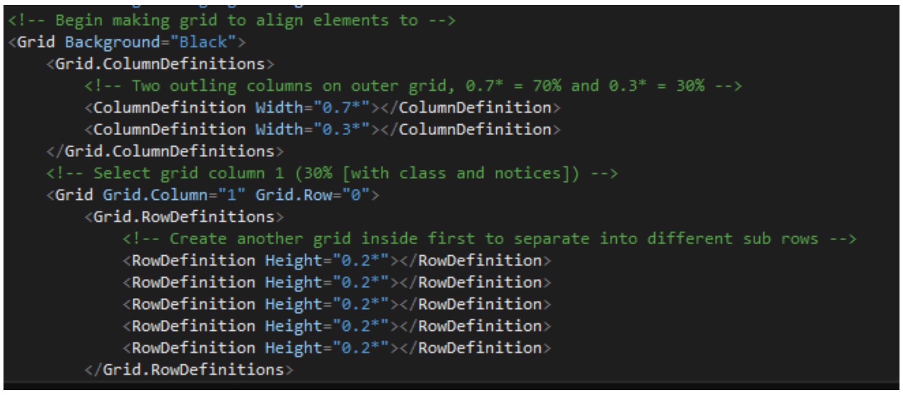
Below the process used to accomplish this can be seen, first the image is rendered and then a gradient brush was added, this brush began at the top of the image and then rendered down to the middle where its offset reflected it vertically, this producing the dark at both vertical extremes but light in the middle effect. Refer to Microsoft's XAML documentation for complete documentation of gradient manipulation[1].



```xml
<Image x:Name="newsImage" Margin="20">
    <Image.OpacityMask>
        <!--<DropShadowBitmapEffect Color="Black" Direction="90" ShadowDepth="10" Opacity="1" Softness="9"/>-->
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1" >
            <GradientStop Offset="0.0" Color="#00000000" />
            <GradientStop Offset="0.25" Color="#FF000000" />
            <GradientStop Offset="0.75" Color="#FF000000" />
            <GradientStop Offset="1" Color="#00000000" />
        </LinearGradientBrush>
    </Image.OpacityMask>
</Image>
<TextBlock x:Name="newsTitle" TextWrapping="Wrap" Foreground="White" FontSize="56" HorizontalAlignment="Center" V
    <TextBlock.BitmapEffect>
        <DropShadowBitmapEffect Color="Black" Direction="180" ShadowDepth="10" Opacity="1" Softness="9"/>
    </TextBlock.BitmapEffect>
    Rss Feed Title
</TextBlock>
```

Overlay underneath image

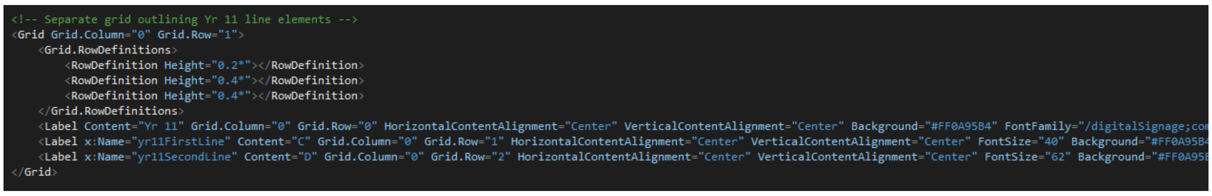
All of the elements used without the forms were dictated by the grid placing them exactly where they needed to be, below it can be noted the process used by defining the columns and rows respectively.



```xml
<!-- Begin making grid to align elements to -->
<Grid Background="Black">
    <Grid.ColumnDefinitions>
        <!-- Two outling columns on outer grid, 0.7* = 70% and 0.3* = 30% -->
        <ColumnDefinition Width="0.7*"></ColumnDefinition>
        <ColumnDefinition Width="0.3*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <!-- Select grid column 1 (30% [with class and notices]) -->
    <Grid Grid.Column="1" Grid.Row="0">
        <Grid.RowDefinitions>
            <!-- Create another grid inside first to separate into different sub rows -->
            <RowDefinition Height="0.2*"></RowDefinition>
            <RowDefinition Height="0.2*"></RowDefinition>
            <RowDefinition Height="0.2*"></RowDefinition>
            <RowDefinition Height="0.2*"></RowDefinition>
            <RowDefinition Height="0.2*"></RowDefinition>
        </Grid.RowDefinitions>
```

Grid element definition

Another thing to note is that while XAML made it very easy to implement a graphical interface, it became very fiddly when this interface was trying to become 'dynamic'. Most of the way it works is that elements are statically configured to go to a respective spot; although for this program to work the way it needed to, everything needed to be as dynamic as possible. Eventually this was achieved by restricting each element to a grid and merely defining the sizes of grids using percentages. Refer to the image below for an example of this. As can be noted, the labels are forced to obey the parent grid.

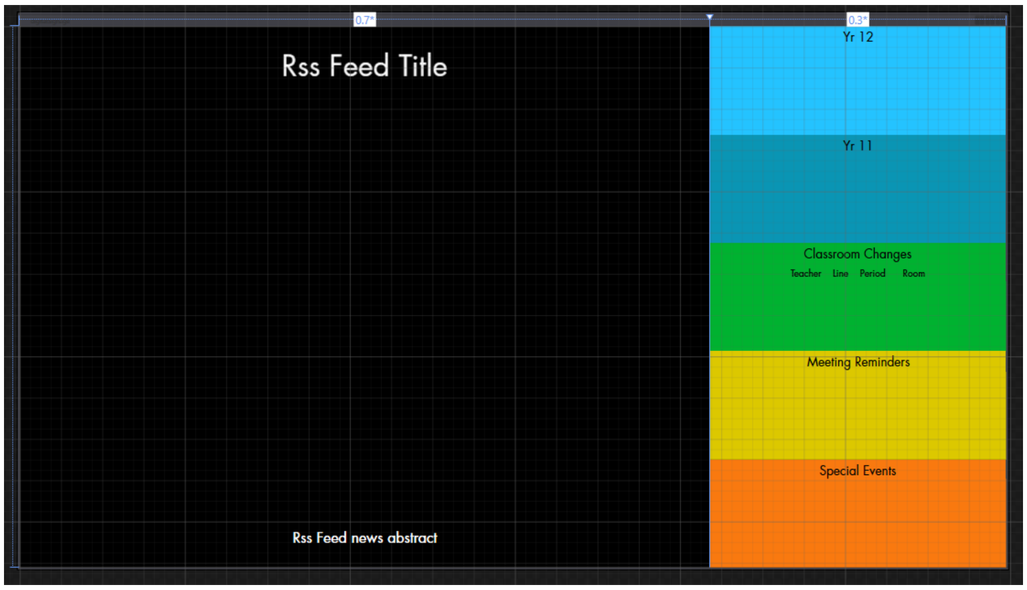


```xml
<!-- Separate grid outlining Yr 11 line elements -->
<Grid Grid.Column="0" Grid.Row="1">
    <Grid.RowDefinitions>
        <RowDefinition Height="0.2*"></RowDefinition>
        <RowDefinition Height="0.4*"></RowDefinition>
        <RowDefinition Height="0.4*"></RowDefinition>
    </Grid.RowDefinitions>
    <Label Content="Yr 11" Grid.Column="0" Grid.Row="0" HorizontalContentAlignment="Center" VerticalContentAlignment="Center" Background="#FF0A95B4" FontFamily="/digitalSignage;com
    <Label x:Name="yr11FirstLine" Content="C" Grid.Column="0" Grid.Row="1" HorizontalContentAlignment="Center" VerticalContentAlignment="Center" FontSize="40" Background="#FF0A95B4
    <Label x:Name="yr11SecondLine" Content="D" Grid.Column="0" Grid.Row="2" HorizontalContentAlignment="Center" VerticalContentAlignment="Center" FontSize="62" Background="#FF0A95E
</Grid>
```

Grid forcing elements

[1] https://msdn.microsoft.com/en-us/library/system.windows.media.lineargradientbrush(v=vs.110).aspx

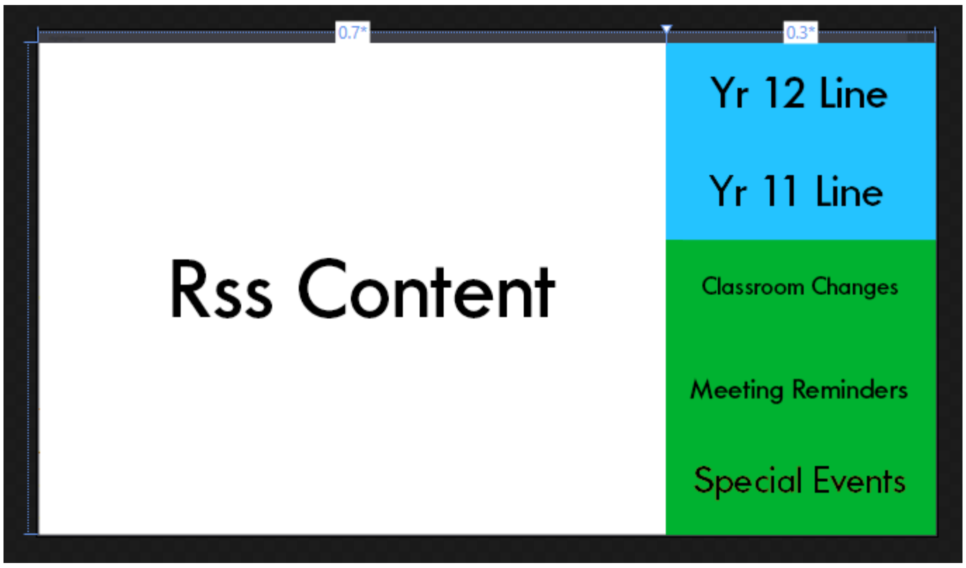
This isn't necessarily a bad thing, as a fluid display allowed for a lot more flexibility in the future for improvements to the program. Once all elements had their respective places it was time to start adding the logic to the program, see below for the final view of the completed interface (from the design view)



Completed Server interface

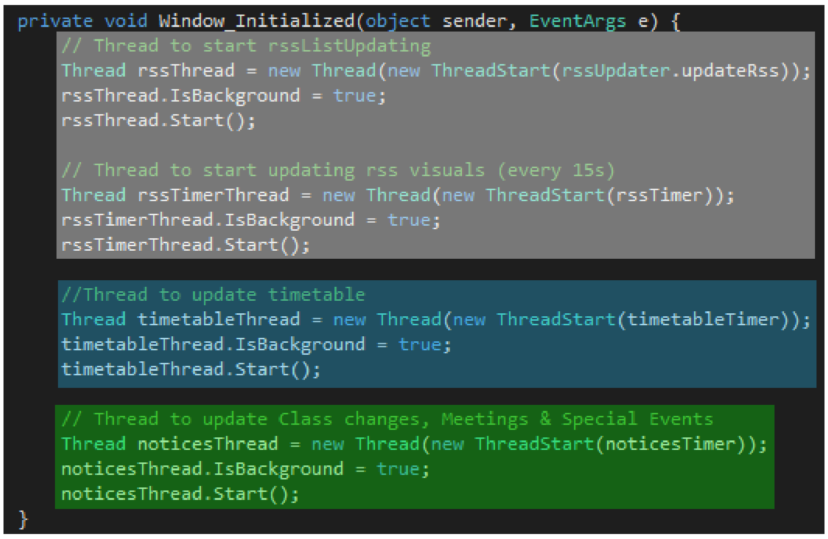
Logic behind Interface

Following on from the more graphical emphasis placed so far, the code behind the program will be explained using a series of diagrams. To begin, each UI element will have a certain colour to refer to its function; for a guideline of these colours please refer to image below.



Server guideline

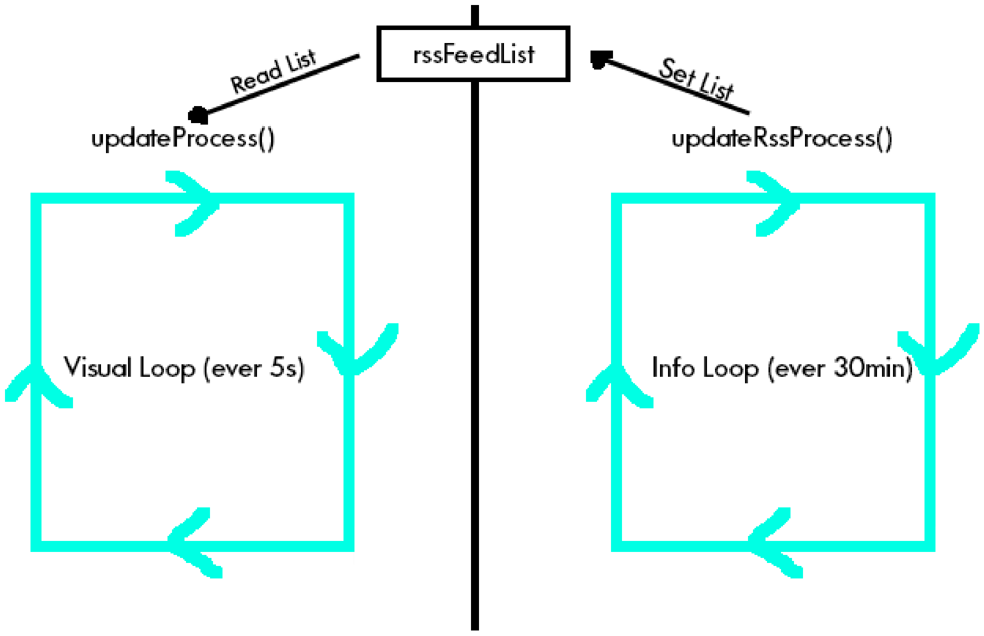
This guideline will be used to highlight the various parts of the program and their roles. When the program is first run, several threads are created, refer to image below for what these threads look like; these threads deal with all of the updating on the main interface.



```
private void Window_Initialized(object sender, EventArgs e) {
    // Thread to start rssListUpdating
    Thread rssThread = new Thread(new ThreadStart(rssUpdater.updateRss));
    rssThread.IsBackground = true;
    rssThread.Start();

    // Thread to start updating rss visuals (every 15s)
    Thread rssTimerThread = new Thread(new ThreadStart(rssTimer));
    rssTimerThread.IsBackground = true;
    rssTimerThread.Start();

    //Thread to update timetable
    Thread timetableThread = new Thread(new ThreadStart(timetableTimer));
    timetableThread.IsBackground = true;
    timetableThread.Start();

    // Thread to update Class changes, Meetings & Special Events
    Thread noticesThread = new Thread(new ThreadStart(noticesTimer));
    noticesThread.IsBackground = true;
    noticesThread.Start();
}
```

All of the threads within server application

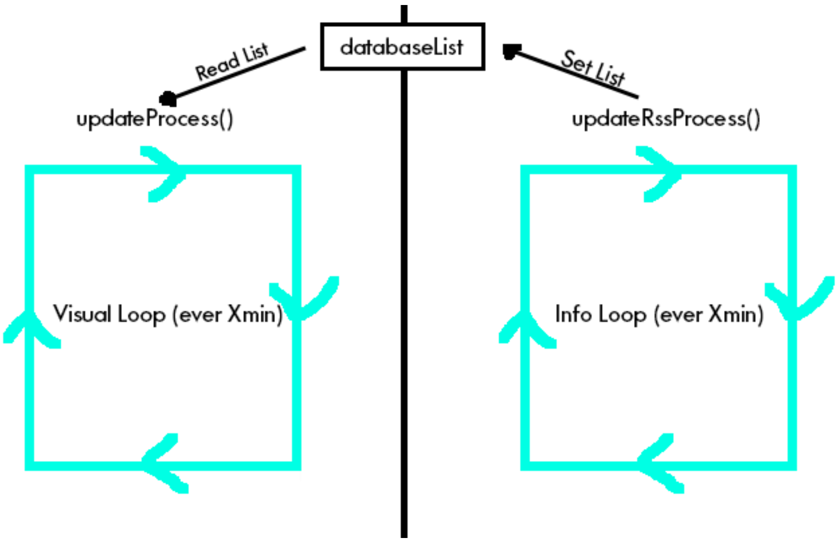
The first thread, rssThread loops every 30 minutes and polls the database and the RSS server for information on the latest information from the feeds. This information is then added to a global list (as can be seen below) which is access every 5 seconds by the other RSS thread iterating through each object in the list Due to this process happening consecutively and the updateProcess simply iterating through the list, whenever the list is updated through its respective delegate it doesn't change any timings from a user's point of view (this suggesting that the user will experience no lag).



RSS feed paradigm

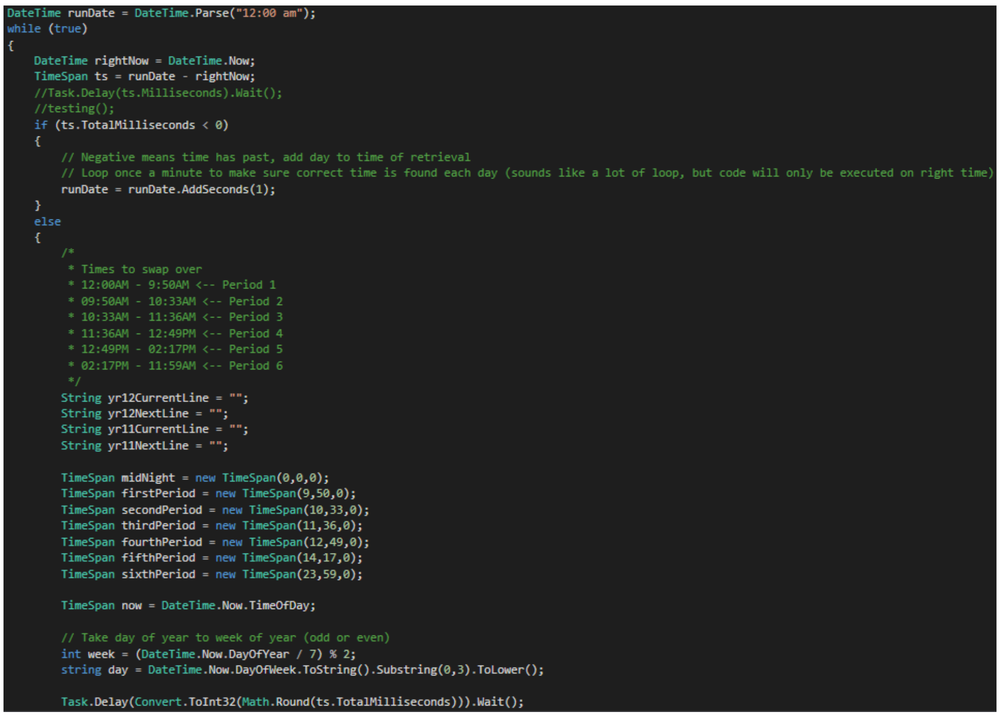
The archetype that was implemented works extremely well due to it being a simple solution to a problem. The only real improvements that could possibly be made to this set up would be a way to make sure two objects aren't trying to access the same list at once. Although this isn't a problem, over time the probability of that happening gets higher.

Since this program was designed to be very modular, the following three colours (Blue, Yellow and Orange or Year 12/11, Classroom, Meeting Reminders/Spec Events) all pertain to having the same architectural software design. Refer below for an example of this design. As can be seen, it is the exact same idea as the RSS feed; except the replacement of pulling from a database instead of a server. Alongside this, the X represents an arbitrary time used to update said values.



RSS Feed paradigm extrapolated to other
parts of program

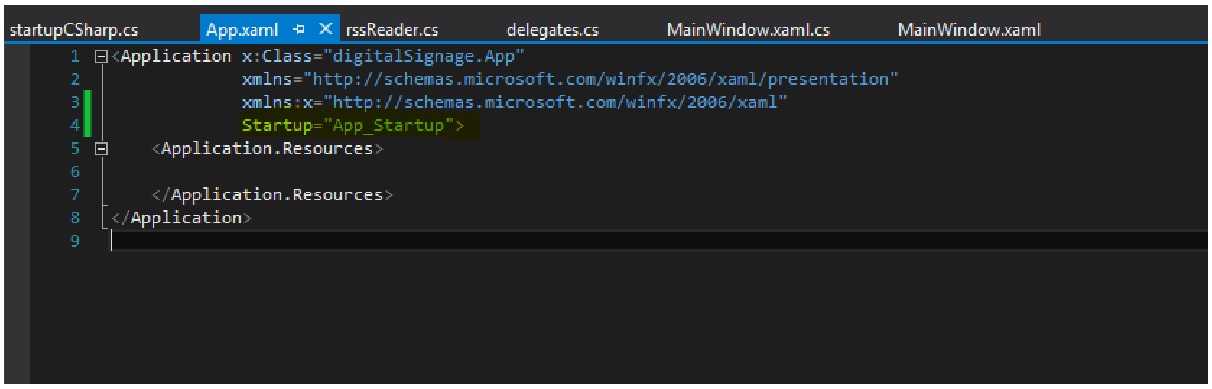
The way the timers function is very interesting, the process can be seen below within an extract from the calculation of the current line. As can be noted, it begins by finding the current Date Time (this being 12am) the thread then retreats into an infinite loop finding the difference between right now and the original set Date Time (12am). It is then checking if the total milliseconds between now and then is negative, if it is negative add interval of pause to Date time. Effectively looping every X interval.



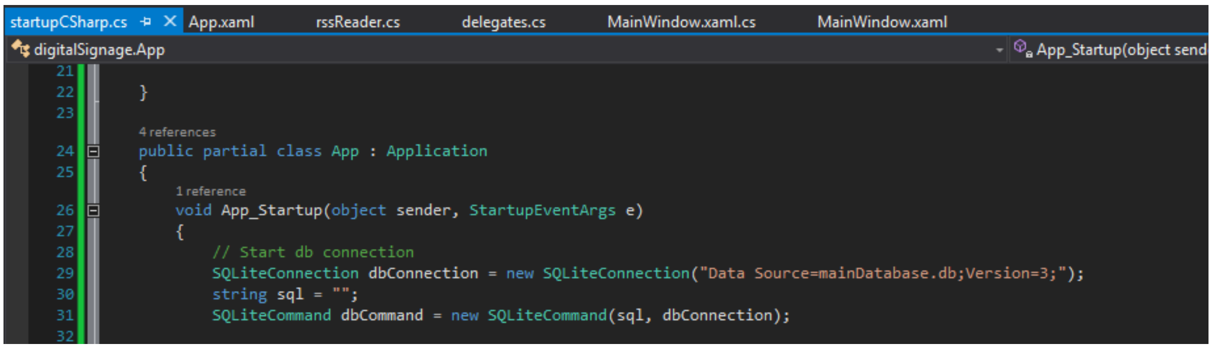
```
DateTime runDate = DateTime.Parse("12:00 am");
while (true)
{
    DateTime rightNow = DateTime.Now;
    TimeSpan ts = runDate - rightNow;
    //Task.Delay(ts.Milliseconds).Wait();
    //testing();
    if (ts.TotalMilliseconds < 0)
    {
        // Negative means time has past, add day to time of retrieval
        // Loop once a minute to make sure correct time is found each day (sounds like a lot of loop, but code will only be executed on right time)
        runDate = runDate.AddSeconds(1);
    }
    else
    {
        /*
         * Times to swap over
         * 12:00AM - 9:50AM <-- Period 1
         * 09:50AM - 10:33AM <-- Period 2
         * 10:33AM - 11:36AM <-- Period 3
         * 11:36AM - 12:49PM <-- Period 4
         * 12:49PM - 02:17PM <-- Period 5
         * 02:17PM - 11:59AM <-- Period 6
         */
        String yr12CurrentLine = "";
        String yr12NextLine = "";
        String yr11CurrentLine = "";
        String yr11NextLine = "";

        TimeSpan midNight = new TimeSpan(0,0,0);
        TimeSpan firstPeriod = new TimeSpan(9,50,0);
        TimeSpan secondPeriod = new TimeSpan(10,33,0);
        TimeSpan thirdPeriod = new TimeSpan(11,36,0);
        TimeSpan fourthPeriod = new TimeSpan(12,49,0);
        TimeSpan fifthPeriod = new TimeSpan(14,17,0);
        TimeSpan sixthPeriod = new TimeSpan(23,59,0);

        TimeSpan now = DateTime.Now.TimeOfDay;

        // Take day of year to week of year (odd or even)
        int week = (DateTime.Now.DayOfYear / 7) % 2;
        string day = DateTime.Now.DayOfWeek.ToString().Substring(0,3).ToLower();

        Task.Delay(Convert.ToInt32(Math.Round(ts.TotalMilliseconds))).Wait();
```

Timer example

Once each dynamic element within the interface could be added from their own separated threads, through the use of delegates it was time to integrate the database into the program, alongside the webserver for the client to connect up to. In order to accomplish this; the manifest of the program was going to need to be manipulated to allow the execution of a small custom class which created the database before anything else within the program executed. See below for the addition of this extra class in the 'App.xaml' manifest file.



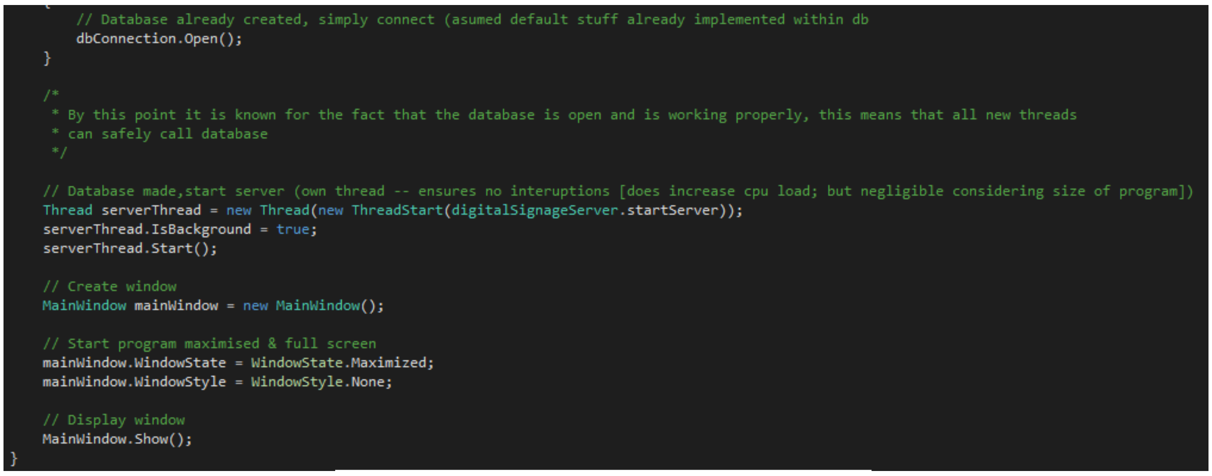
```
<Application x:Class="digitalSignage.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             Startup="App_Startup">
    <Application.Resources>

    </Application.Resources>
</Application>
```



```
    }

    public partial class App : Application
    {
        void App_Startup(object sender, StartupEventArgs e)
        {
            // Start db connection
            SQLiteConnection dbConnection = new SQLiteConnection("Data Source=mainDatabase.db;Version=3;");
            string sql = "";
            SQLiteCommand dbCommand = new SQLiteCommand(sql, dbConnection);
```

Creation of entry point to application

The manifest is a file which contains certain 'pre-processor' instructions, which once bypassed with a custom one allows the developer the ability to add their own classes which start up before anything else. From this, it was simply a process of adding the instructions to run a 'server' class before the main window showed up. During this point in time; since the server had to talk to a database, the Sqlite plugin was implemented. This plugin enabled quick and easy access to a custom database placed in the WinPE directory.



```
            // Database already created, simply connect (asumed default stuff already implemented within db
            dbConnection.Open();
        }

        /*
         * By this point it is known for the fact that the database is open and is working properly, this means that all new threads
         * can safely call database
         */

        // Database made,start server (own thread -- ensures no interuptions [does increase cpu load; but negligible considering size of program])
        Thread serverThread = new Thread(new ThreadStart(digitalSignageServer.startServer));
        serverThread.IsBackground = true;
        serverThread.Start();

        // Create window
        MainWindow mainWindow = new MainWindow();

        // Start program maximised & full screen
        mainWindow.WindowState = WindowState.Maximized;
        mainWindow.WindowStyle = WindowStyle.None;

        // Display window
        MainWindow.Show();
    }
```

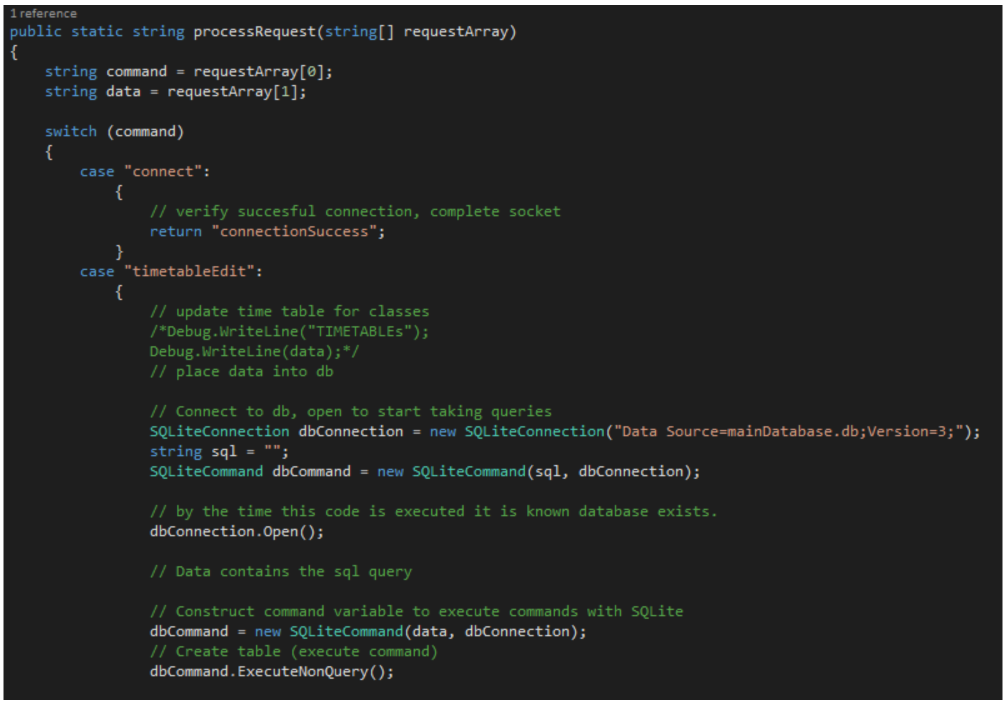
End of entry point of application

As can be seen above, the entry point creates a new thread and places the server in it, which will bind to a socket and wait for requests.

The logic of the server start up went as follows

```
If (can connect to database) {
        // Good, do nothing
} else {
        // Can't connect do nothing [create db.]
        // Code to create db.
}
Turn on socket networking and wait for request
```

Once the code was at the point where it was waiting for a request it would simply pause the thread it was in and wait, again not affecting what the user sees. From here the thread would deal with the request by placing it into a case statement where verified key words are used by the server to authenticate what the client software is asking. The request is places into an array and split into string arrays based on the delimiter used (~). From here the switch case statement will do various things depending on what request the client has sent to the server. See below for the case of 'connect' and 'timetableEdit'. Which send a handshake of 'connectionSuccess' and update the SQL of the timetable table respectively.



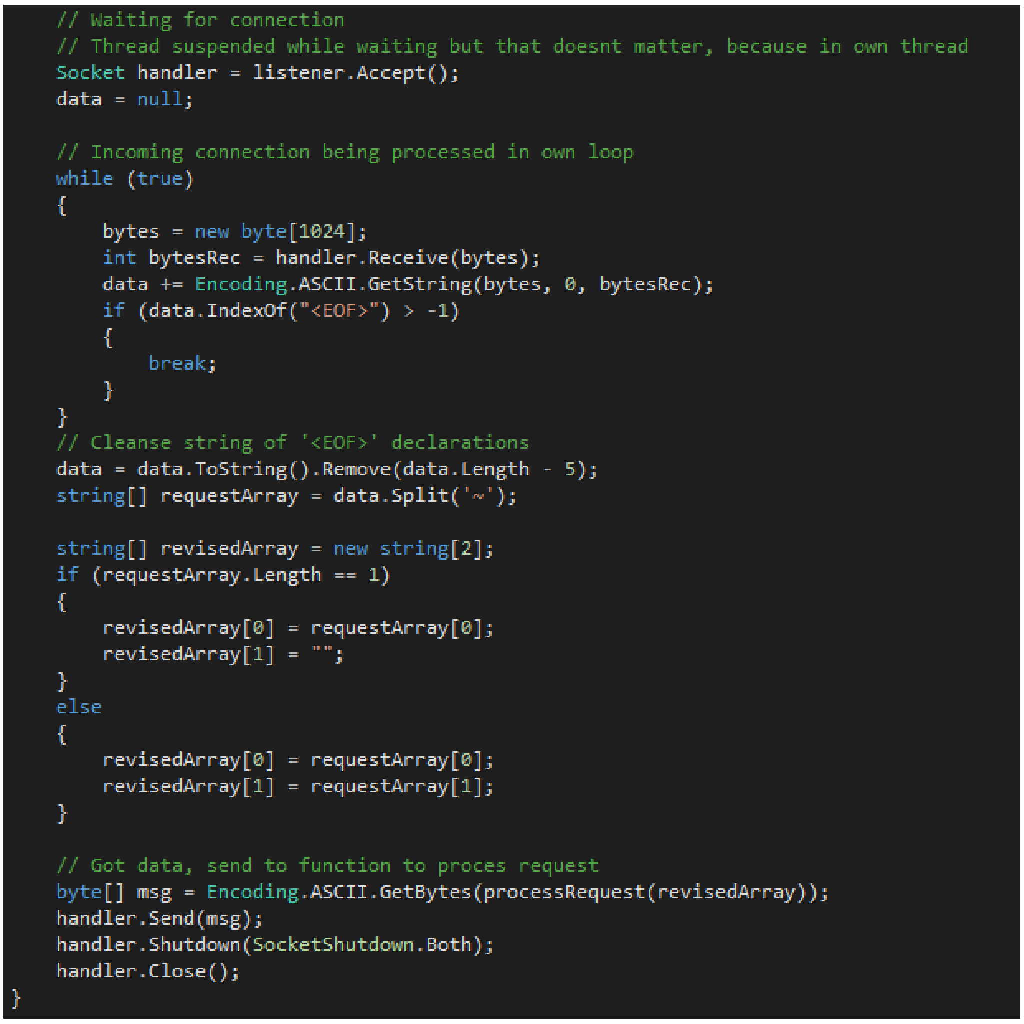
```
1 reference
public static string processRequest(string[] requestArray)
{
    string command = requestArray[0];
    string data = requestArray[1];

    switch (command)
    {
        case "connect":
            {
                // verify succesful connection, complete socket
                return "connectionSuccess";
            }
        case "timetableEdit":
            {
                // update time table for classes
                /*Debug.WriteLine("TIMETABLEs");
                Debug.WriteLine(data);*/
                // place data into db

                // Connect to db, open to start taking queries
                SQLiteConnection dbConnection = new SQLiteConnection("Data Source=mainDatabase.db;Version=3;");
                string sql = "";
                SQLiteCommand dbCommand = new SQLiteCommand(sql, dbConnection);

                // by the time this code is executed it is known database exists.
                dbConnection.Open();

                // Data contains the sql query

                // Construct command variable to execute commands with SQLite
                dbCommand = new SQLiteCommand(data, dbConnection);
                // Create table (execute command)
                dbCommand.ExecuteNonQuery();
```

Processing Request with Server

It should be explained at this point at how data is transferred over a medium. Usually with computational communications information is send over a binary (1001010) connection, each of those individual ‘1’s or ‘0’s represents a bit, 8 of those bits represent a byte. For simplistic analogy each of those 8 bits represents a number which corresponds to a letter in the alphabet. This code comparison is known as ASCII (American Standard Code for Information Interchange) and it is this index which tells the computer how to ‘convert’ between those bytes to get letters.

Extending from this data of any time (images, music, videos, etc.) can be sent using the same system you just need to define how to encode (convert the data to binary) and then decode (convert binary back to the data). For this there exists MIME types (Media Type), these MIME types are what defines how to interpret different forms of data.

Following on from this information, inside of the server program there is an <EOF> declaration, this is added to the end of all strings back and forth from the client and server as it allows the server to know when the <EOF> (End of file) is so it can stop reading the data. This can be seen by the iteration through the byte array to then decode the message from it.



```
    // Waiting for connection
    // Thread suspended while waiting but that doesnt matter, because in own thread
    Socket handler = listener.Accept();
    data = null;

    // Incoming connection being processed in own loop
    while (true)
    {
        bytes = new byte[1024];
        int bytesRec = handler.Receive(bytes);
        data += Encoding.ASCII.GetString(bytes, 0, bytesRec);
        if (data.IndexOf("<EOF>") > -1)
        {
            break;
        }
    }
    // Cleanse string of '<EOF>' declarations
    data = data.ToString().Remove(data.Length - 5);
    string[] requestArray = data.Split('~');

    string[] revisedArray = new string[2];
    if (requestArray.Length == 1)
    {
        revisedArray[0] = requestArray[0];
        revisedArray[1] = "";
    }
    else
    {
        revisedArray[0] = requestArray[0];
        revisedArray[1] = requestArray[1];
    }

    // Got data, send to function to proces request
    byte[] msg = Encoding.ASCII.GetBytes(processRequest(revisedArray));
    handler.Send(msg);
    handler.Shutdown(SocketShutdown.Both);
    handler.Close();
}
```

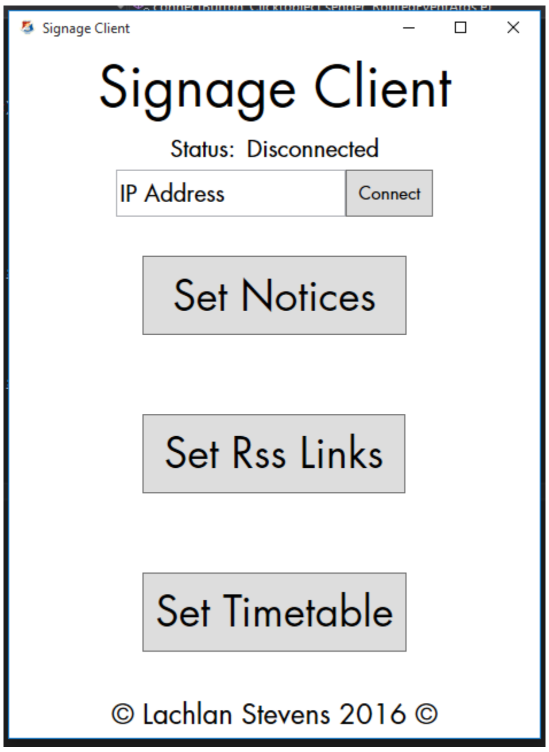
Processing Request with Server

Once the server has processed and read the request it closes that thread and silently waits for another request. A quick note has to be made that the server makes its threads asynchronously (meaning no two threads can exist at the same time), this means that only one client can connect to the server at a time; which is needed as it would make it impossible to actively edit the database with two people.

Client

Graphical Interface

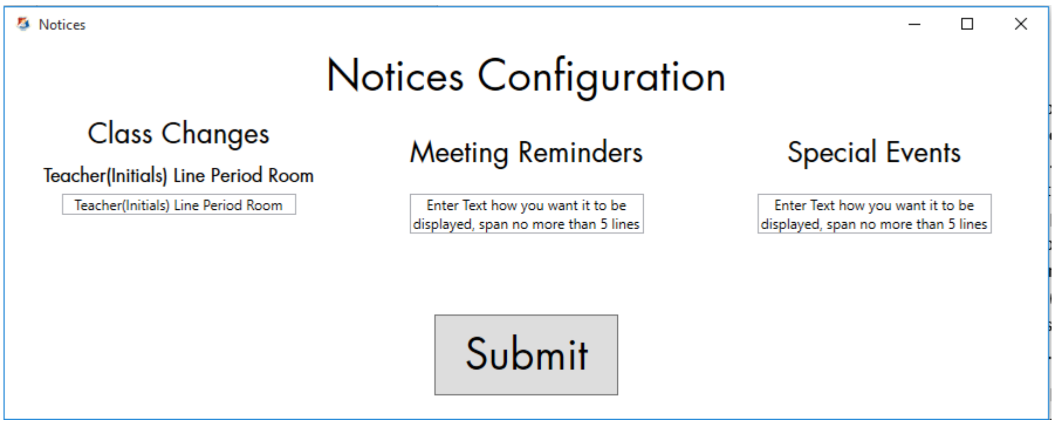
The client program was designed after the server in order to have a foundation to connect to, to ensure it works. To begin a basic interface was devised, only placing elements where they needed to be in order to function. As can be noted below, this design emulates basically an exact copy of the storyboard devised earlier. This is good, as it suggests a solid design that is simple and works. To begin, the client enters the servers IP address and selects connect to connect to the digital signage server. Once connected the user will have 3 options, to set the notices for the day, setting the RSS links and finally setting the timetable. Whenever each respective button is pressed a new window will open hosting all of the new options to do with the given button that was pressed.



Signage Client Interface

The XAML for this window was fairly straight forward, again it was extrapolated into each of their respective subparts to then mathematically deduce where on the page it should be. To do this effectively, the Title at the top was placed at a fixed height with everything below it depending on that fixed height. This might sound like bad practice but it was circumvented by allowing the values to change if the window was resized.

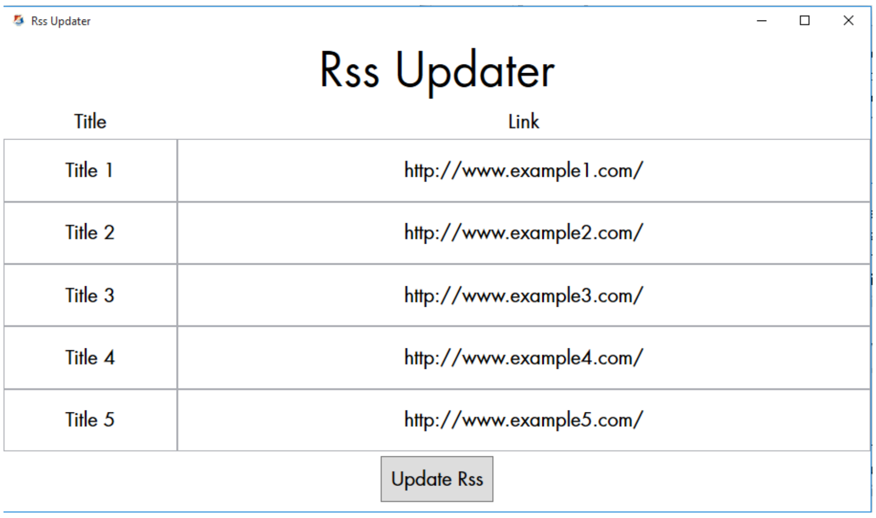
When the notices button is selected it brings up the form to edit what is displayed for the notices subsection of the display. This design is very simplistic and mimics the storyboard perfectly for what was needed, this is mainly due to the story board being specific enough to implement and use.


Notices
Notices Configuration
Class Changes
Teacher(Initials) Line Period Room
Teacher(Initials) Line Period Room
Meeting Reminders
Enter Text how you want it to be displayed, span no more than 5 lines
Special Events
Enter Text how you want it to be displayed, span no more than 5 lines
Submit

Notices configuration form

The way the Meeting reminders label was aligned to the centre of the two class changes labels was by adding up the heights of each, dividing by two to find the centre point and then finally dividing the height of the resulting label by 2 to then subtract from the height given. This resulting in a vertically centred label.

Following onto the RSS Links page, this is the easiest form to navigate as it only has a series of textboxes that need to be filled out. There can only be at maximum 5 RSS feeds due to mathematically pulling any more than 10 from each will cause an overflow in the amount of images that can be pulled off of an image server in the future, this simplification allows for future advancement in image location (See evaluation for more detail).

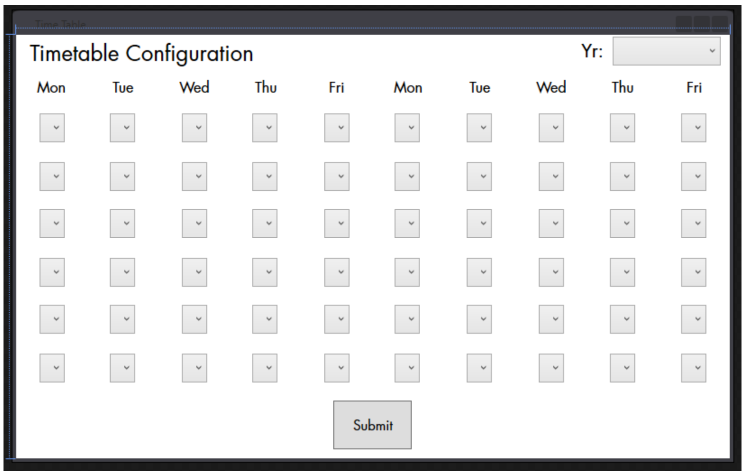


Title	Link
Title 1	http://www.example1.com/
Title 2	http://www.example2.com/
Title 3	http://www.example3.com/
Title 4	http://www.example4.com/
Title 5	http://www.example5.com/

RSS Updater form

This form was very simple in regards to design as there is only several textboxes aligned with a title and a link. Construction of this form was made easy through WPF as it only took one grid element to align all of the elements evenly. Improvements that could be made to this form would be the inclusion of on load having it poll the SQL Server for the current status of RSS links; having this would allow the user quick and easy updating of the information within the program, but is beyond the scope of this project.

Finally, is the timetable form, this form is what allows the user the ability to navigate and set the time tables for the line segments to show up. The only problem with this chosen design is it is very convoluted and looks messy; although it does work very well for what it is supposed to do. Originally it is a slow interface to use, although after some practice it becomes a quick job to fill out each entry. The justification for having such a convoluted user interaction is that it is fast to get a lot of information out of and the user will not have to use this form very often. The time tables are only updated once or twice a year at most, which means that the client was need of something that was tedious but got the job done well.

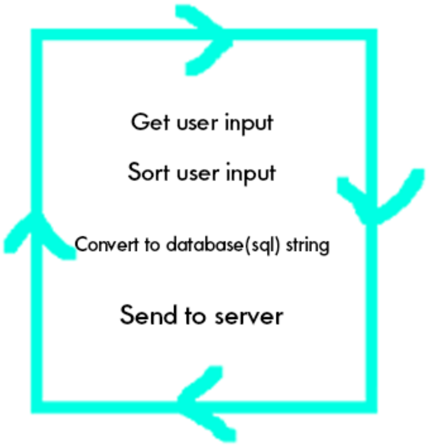


Timetable Configuration Form

The most difficult thing within this form is the amount of combo boxes, having this many combo boxes needing to be read from and sanitized meant that there were a lot of typing mistakes and syntax errors. Aside from this, once it was all working properly and each combo box was in place it was a quick, responsive form.

Logic behind Interface

Behind all of the client interfaces there is one software paradigm being repeated over and over (excluding the timetable) this paradigm is demonstrated below, refer to it for the explanation. This diagram demonstrates the repetitive nature of each feature added as it is just the old feature with a different data set.

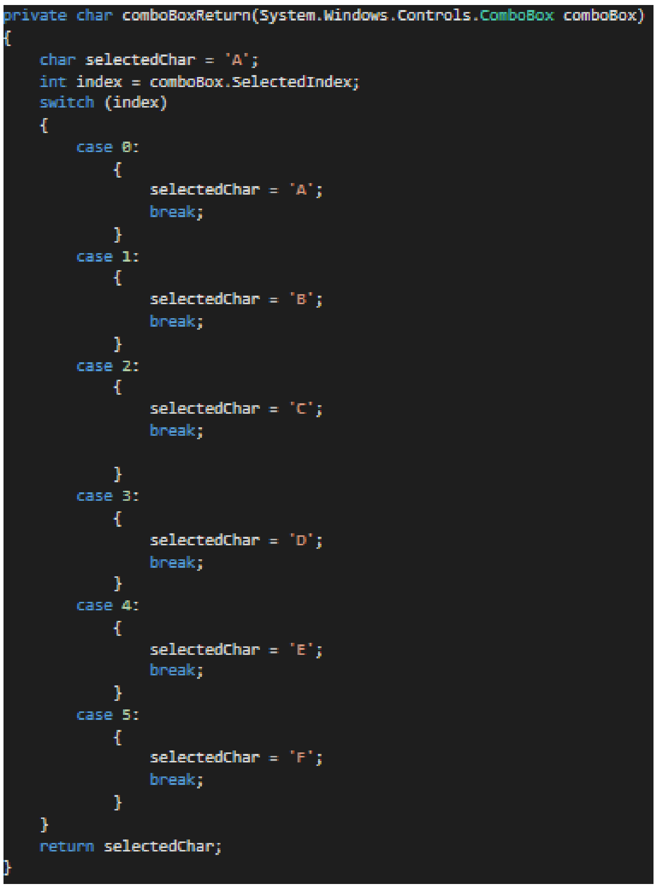


Client Interface Paradigm

It is this software paradigm which is what makes the client code very reliable. It is quick, short and simple meaning there isn't much that can go wrong. In regards to the SQL sending there were several Run time errors that popped up due to the use of SQLite, the first of which being the lack of a 'truncate' command. Whenever code was run that tried to execute said commands, the program would crash and spill out a run time error. This was all fixed when the SQLite documentation[2] was referenced and the replacement keywords were found.

Alongside this there was a very abstract error where the program crashed whenever an apostrophe was used. This happened because the program wasn't sanitizing the string before it was sent to the server. Allowing this to happen is a **very** bad practice as it meant that the program would have been able to have SQL Injection, where the attacked fakes user input for commands to a server. Eventually, this problem was solved by taking parameterized queries meaning that strings would be interpreted as strings and nothing else.

A note to make about the combo boxes in the time table form, in order to retrieve each of their values all of them needed to be integrated through; this was accomplished by creating a custom method that had a 'char' return value, from here the input was of the form 'Systems.Windows. Controls.ComboBox'. This meant that all that was needed to be achieved to get the combo boxes index was feeding the name of the variable into the function. This can be noted below.



```
private char comboBoxReturn(System.Windows.Controls.ComboBox comboBox)
{
    char selectedChar = 'A';
    int index = comboBox.SelectedIndex;
    switch (index)
    {
        case 0:
            {
                selectedChar = 'A';
                break;
            }
        case 1:
            {
                selectedChar = 'B';
                break;
            }
        case 2:
            {
                selectedChar = 'C';
                break;

            }
        case 3:
            {
                selectedChar = 'D';
                break;
            }
        case 4:
            {
                selectedChar = 'E';
                break;
            }
        case 5:
            {
                selectedChar = 'F';
                break;
            }
    }
    return selectedChar;
}
```

Combo Box iteration

[2] http://www.sqlite.org/src/doc/trunk/src/test_demovfs.c

Testing

While the project was in the midst of being created a comprehensive job was done to test and test locally (pre-alpha phase) where many errors were found, with the some of the most notable being Syntax errors with the HTML being shown in the abstract of the Server signage interface, this of course being solved by implementing Regular Expressions into the code allowed for more accurate sorting and replacing of the string.

The next error would be the run time error given by the SQLite database whereby it struggled to create the database due to Visual Studio locking the file; this resolved by restarting Visual Studio. Following this, a difficult logic error was given by the mix-up of two delegates names inside of a verbatim string literal this caused a lot of confusion as the year 11 and 12 labels were showing the exact same thing. Eventually this error was found and amended.

Although, no matter how much pre alpha testing occurs nothing suffices from having fellow students test and give feedback on the functioning prototypes of the software. Fellow classmates, Alex VanLint and Duron Prinsloo were asked to give feedback on the software and various observations made about the program. Refer to Appendix 10 – 13.

Originally, in the Alpha tests it was noted that the RSS Feeds loaded slowly; while they attributed the problem to simply being the internet, there was a problem with the way the feeds were being scraped from the server. A simple rewrite of some of the more basic XML code meant that the RSS was able to be loaded a lot more quickly. Overall the responses seemed very good, although another comment was made by Duron Prinsloo where he stated that the forms needed to populate automatically to make it more enjoyable for the user. While what he said was correct it is beyond the scope of this project and could be addressed at a later date.

To conclude the testing and evaluation of the product, the client was shown the final product to gauge the feedback. Which as can be noted on Appendix 14 was overwhelmingly positive. This feedback summarises the optimal result of this project as the interface was not only designed from the ground up based on Shneiderman's eight rules of interface design and the C.A.R.P. Principles but it even on par with the client's expectations! This all leads to a successful project.

Evaluation

When first taking up this project, it quickly became prevalent how big it was going to be. There were a lot of complicated parts working together meaning there wasn't time to perfect just one part, there needed to exist a working prototype, which was then added on. This was accomplished in part, by creating each part of the program slowly and on a part per part basis; resulting in very modular, easy to implement code. Overall, this code functions very efficiently with not many resources being wasted on unnecessary computations and as can be noted on Appendix 15 continuously there are notes saying this is beyond the scope and there isn't enough time. Creating this project, pointed out how much potential there was for improvements; for example, sometimes the RSS feeds only include the Title and Abstract of the page so to get around this I devised some simple proof of concept code that pulled images from Bings search engine and filtered them and displayed them on the screen (Refer to Appendix 16 for UML Diagram of code). This would have been a good improvement for this project purely because it would allow no RSS feed be without a picture to drag and audience in.

Alongside this another useful inclusion that wasn't able to be achieved within the given time period was the addition of an auto populate command. Where the form would automatically populate with the information that is already in the database and the user would just need to edit it. Having this feature would save the user a lot of time when manipulating the program with it overall relating to the Human Computer Interface of the software. The human computer interface for this project was very highly accredited purely because it follows the strict rulings of C.A.R.P. and Schneiderman. C.A.R.P. was followed all over the program, firstly with the use of Contrast in the Server interface. Effectively using a gradient to highlight the title on the Server interface allowed the user to gain a very obvious perception to what the text wrote. This along with the Z rule on that page meant the very last thing the person would see would be the meeting reminders.

Following from this, Alignment was effectively demonstrated through the implementation of the grid elements, separating each element and having their own clear spots helped the user feel a sense of consistency throughout the program which draws their appeal to the program stronger. Following from this, Repetition was shown throughout all of the client interfaces. This was achieved through the consistent submit button down the bottom of the page and the overall repetition in the way the elements were laid out (in a grid).

Finally, Proximity was incorporated by allowing each of the relevant pieces of content to stay together within 'proximity' of each other. This again allows the user of the software feel a sense of consistency and safety when using the program as it allows them to step back and feel in control of what the computer is doing.

Moving on from this, Shneiderman was a very famous computer scientist who developed 8 rules to follow for a successful human computer interaction. These rules begin with striving for consistency, this is true for this project as consistency across all forms (client and server) were followed. The next rule was 'Enable frequent users to use shortcuts', this is untrue for this project but could be taken as an improvement in the future. Having shortcuts allows more fluent users a faster way around the program resulting in a better user experience. Next is 'Offer informative feedback'. Having quick simple feedback is key to having a successful program. This is achieved within this project by incorporating the use of message boxes to subtly alert the user of the current status of the program.

Following, the next rule is 'Design dialog to yield closure'; having a form like within this project which opens up and tells the user they have successfully updated the form to then have the form close, follows this rule. Having this sense of closure welcomes the user to make changes as they know when they make changes it is saved and it is done. From this, 'Offer simple error handling' this is done in the fact that whenever there is an error on the server it is gracefully echoed to the console. Having this means the program won't crash in the middle of the display but it also means if there is something going awry with the program an administrator can simply open it up within a console and retrieve the input from there.

'Permit easy reversal of actions', incorporating this into the program allows the user to feel safe and comfortable with using it. Unfortunately, a lot of the mechanisms involved with setting that up are beyond the scope of this project; but could be implemented in future versions. 'Support internal locus of control'; in other words, make a responsive and quick interface. This is achieved within this project as the client interface and the server interface both allow for active real time updates of the program, with no lag being experienced between any part of the communication between them.

Finally, 'Reduce short-term memory load.' This is achieved in this project by incorporating the use of the digital sign to display all of the information. Doing so, allows users to see the information in front of them without thinking about it. Overall, around about 6 of those 8 rules were successfully followed making this an interface that is statistically effective in accordance to the Schneiderman and C.A.R.P. Design rules.

Extending from this, overall a project was created which fit all of the client's expectations, a final product was produced which made all of the testers happy (to varying degrees) and it was built from the grounds up in accordance to the various design standards and techniques. All of these leave in summary, that a successful product has been produced.

Glossary

Thread-Affinity - The programming paradigm in which objects within a thread cannot be accessed outside of said thread, this is due to the processor distributing the load over various CPU's.

Delegate - Due to Thread Affinity, it incorporates a chain of objects to display content to the main window.

Manifest - File that contains meta data or the specific instructs required to build a project/program.

WinPE - Shorthand for Windows Portable Executable, commonly refers to byte arrays and read only memory of program

TCP – Transmission Control Protocol, defines how network handshakes work to share data between two devices.

UDP – User Datagram Protocol, defines network wide requests (useful when letting users know you are online).

Verbatim String Literal – A string literal that can span over multiple lines.

Synchronous - Occurring at the same time.

Asynchronous - Not occurring at the same time.

Delimiter - A character separator, used to convert strings to arrays and vise-versa.

Syndication – A news feed, one that gets pushed out.

Atom - A standard of news syndication (designed to rival/overtake RSS 2.0) different way of executing RSS

RSS 2.0 – The new standard to the age old RSS. Comes with added features but is nothing compared to the more modern Atom standard

Sockets – Virtual communication lines between programs.

Encode/Decode – The way data is sent over a network, conversion between readable information and binary

ASCII – American Standard Code of Information Interchange, tells computer how to convert between characters and binary.

Bytes – Eight bits, stores up to $2^8 = 256$ unique permutations of data.

Bits – A one or a zero, the foundation to binary and computational informatics.

Gradient – A smooth blending from two colours from one to another.

Regular Expressions (Regex) – A standard of expressions (shorthand) for manipulating strings.

Bibliography

Anon, 2016. *CARP Design Principles | Digest Web Design | Free Website Design Lessons, HTML, CSS*. [online] Digestwebdesign.com. Available at: <http://digestwebdesign.com/carp_design_principles.html> [Accessed 14 Mar. 2016].

Anon, 2016. *How to: Search Strings Using Regular Expressions (C# Programming Guide)*. [online] Msdn.microsoft.com. Available at: <https://msdn.microsoft.com/en-AU/library/ms228595.aspx> [Accessed 14 Mar. 2016].

Anon, 2016. *Implementing XmlReader Classes for Non-XML Data Structures and Formats*. [online] Msdn.microsoft.com. Available at: <https://msdn.microsoft.com/en-us/library/ms973822.aspx> [Accessed 14 Mar. 2016].

Anon, 2016. *Shneiderman's "Eight Golden Rules of Interface Design" | Design Principles FTW*. [online] Designprinciplesftw.com. Available at: <http://www.designprinciplesftw.com/collections/shneidermans-eight-golden-rules-of-interface-design> [Accessed 12 Mar. 2016].

Anon, 2016. *Socket Constructor (System.Net.Sockets)*. [online] Msdn.microsoft.com. Available at: <https://msdn.microsoft.com/en-us/library/system.net.sockets.socket.socket(v=vs.110).aspx> [Accessed 14 Mar. 2016].

Anon, 2016. *SQLite: Documentation*. [online] Sqlite.org. Available at: <http://www.sqlite.org/src/doc/trunk/src/test_demovfs.c> [Accessed 14 Mar. 2016].

Anon, 2016. *System.Data.SQLite (x86/x64)*. [online] Nuget.org. Available at: <https://www.nuget.org/packages/System.Data.SQLite> [Accessed 16 Mar. 2016].

Appendix

Appendix 1 (First Server interface):

See Attached Document

Appendix 2 (Signage Client main interface):

See Attached Document

Appendix 3 (Signage Client notices and RSS interface):

See Attached Document

Appendix 4 (Signage Client timetable interface):

See Attached Document

Appendix 5 (Signage Server tree diagram):

See Attached Document

Appendix 6 (Signage Client tree diagram):

See Attached Document

Appendix 7 (Signage Server Nassi-Schneiderman):

See Attached Document

Appendix 8 (Signage Client Nassi-Schneiderman):

See Attached Document

Appendix 9 (Finalised design of Signage Server interface):

See Attached Document

Appendix 10 (Alpha testing Alex VanLint):

See Attached Document

Appendix 11 (Alpha testing Duron Prinsloo):

See Attached Document

Appendix 12 (Beta testing Kit O'Brien):

See Attached Document

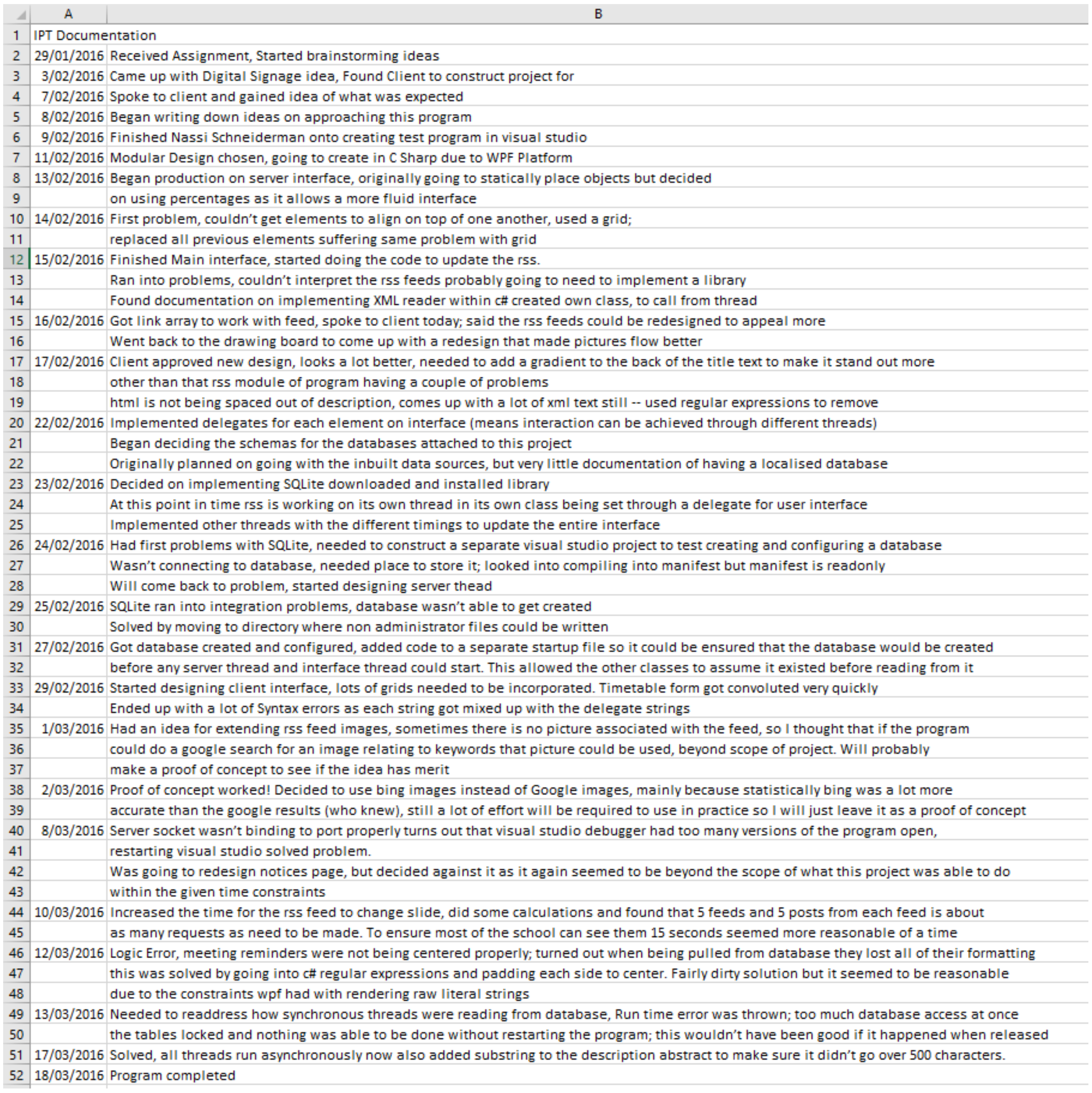
Appendix 13 (Beta testing Bailey Anderson):

See Attached Document

Appendix 14 (Client's Letter regarding feedback):

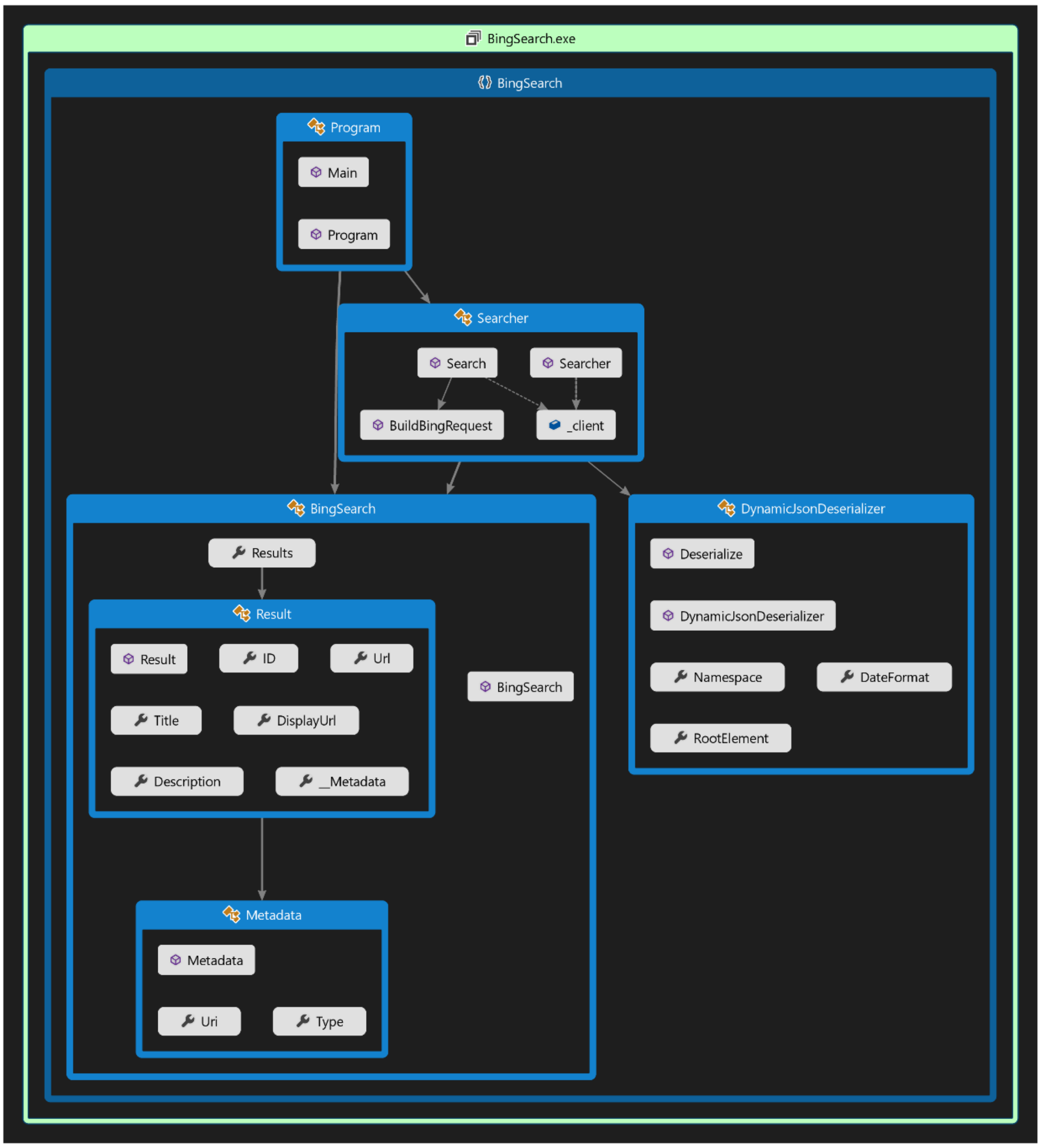
See Attached Document

Appendix 15 (Documentation):

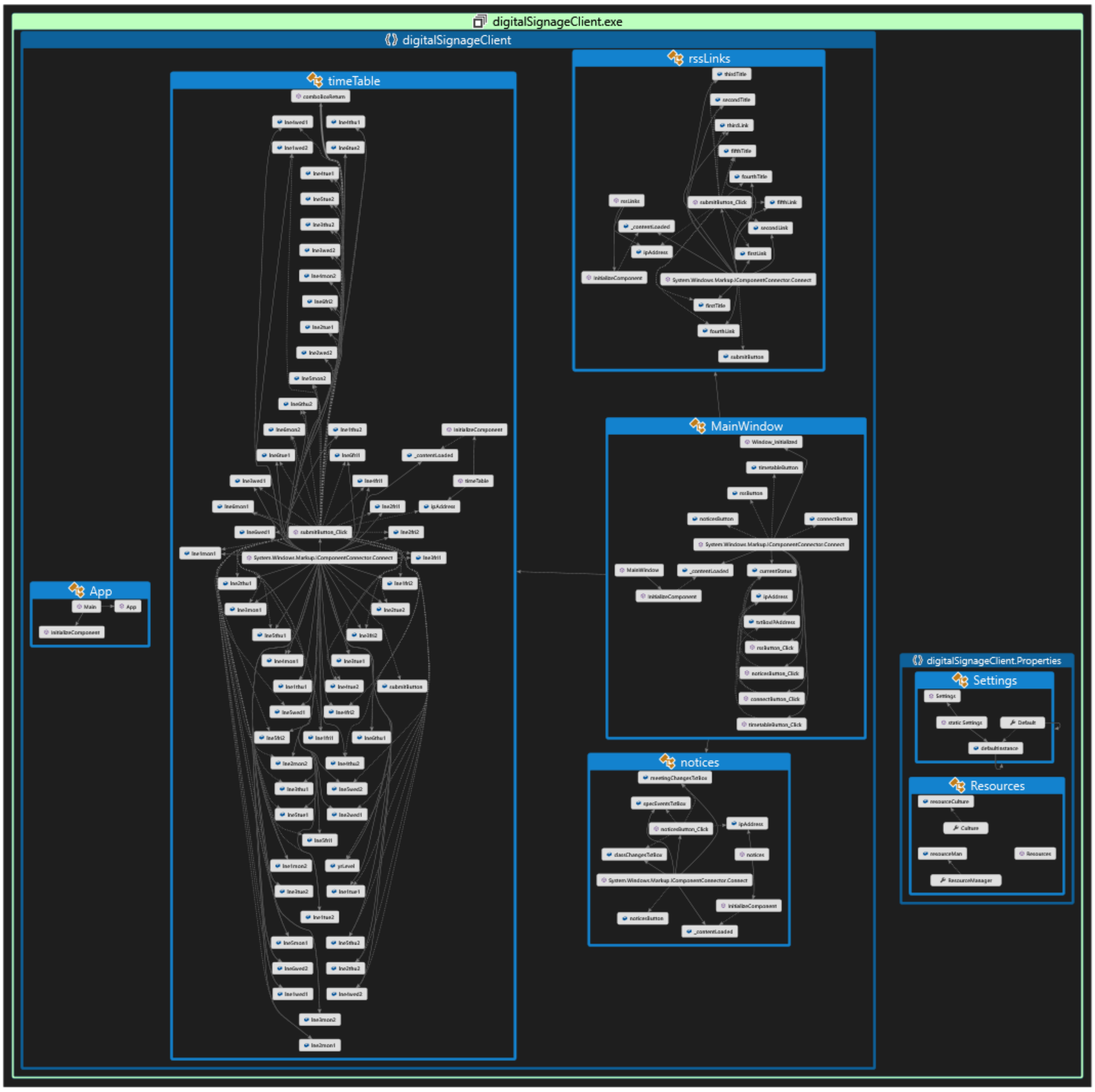


	A	B
1	IPT Documentation	
2	29/01/2016	Received Assignment, Started brainstorming ideas
3	3/02/2016	Came up with Digital Signage idea, Found Client to construct project for
4	7/02/2016	Spoke to client and gained idea of what was expected
5	8/02/2016	Began writing down ideas on approaching this program
6	9/02/2016	Finished Nassi Schneiderman onto creating test program in visual studio
7	11/02/2016	Modular Design chosen, going to create in C Sharp due to WPF Platform
8	13/02/2016	Began production on server interface, originally going to statically place objects but decided
9		on using percentages as it allows a more fluid interface
10	14/02/2016	First problem, couldn't get elements to align on top of one another, used a grid;
11		replaced all previous elements suffering same problem with grid
12	15/02/2016	Finished Main interface, started doing the code to update the rss.
13		Ran into problems, couldn't interpret the rss feeds probably going to need to implement a library
14		Found documentation on implementing XML reader within c# created own class, to call from thread
15	16/02/2016	Got link array to work with feed, spoke to client today; said the rss feeds could be redesigned to appeal more
16		Went back to the drawing board to come up with a redesign that made pictures flow better
17	17/02/2016	Client approved new design, looks a lot better, needed to add a gradient to the back of the title text to make it stand out more
18		other than that rss module of program having a couple of problems
19		html is not being spaced out of description, comes up with a lot of xml text still -- used regular expressions to remove
20	22/02/2016	Implemented delegates for each element on interface (means interaction can be achieved through different threads)
21		Began deciding the schemas for the databases attached to this project
22		Originally planned on going with the inbuilt data sources, but very little documentation of having a localised database
23	23/02/2016	Decided on implementing SQLite downloaded and installed library
24		At this point in time rss is working on its own thread in its own class being set through a delegate for user interface
25		Implemented other threads with the different timings to update the entire interface
26	24/02/2016	Had first problems with SQLite, needed to construct a separate visual studio project to test creating and configuring a database
27		Wasn't connecting to database, needed place to store it; looked into compiling into manifest but manifest is readonly
28		Will come back to problem, started designing server thead
29	25/02/2016	SQLite ran into integration problems, database wasn't able to get created
30		Solved by moving to directory where non administrator files could be written
31	27/02/2016	Got database created and configured, added code to a separate startup file so it could be ensured that the database would be created
32		before any server thread and interface thread could start. This allowed the other classes to assume it existed before reading from it
33	29/02/2016	Started designing client interface, lots of grids needed to be incorporated. Timetable form got convoluted very quickly
34		Ended up with a lot of Syntax errors as each string got mixed up with the delegate strings
35	1/03/2016	Had an idea for extending rss feed images, sometimes there is no picture associated with the feed, so I thought that if the program
36		could do a google search for an image relating to keywords that picture could be used, beyond scope of project. Will probably
37		make a proof of concept to see if the idea has merit
38	2/03/2016	Proof of concept worked! Decided to use bing images instead of Google images, mainly because statistically bing was a lot more
39		accurate than the google results (who knew), still a lot of effort will be required to use in practice so I will just leave it as a proof of concept
40	8/03/2016	Server socket wasn't binding to port properly turns out that visual studio debugger had too many versions of the program open,
41		restarting visual studio solved problem.
42		Was going to redesign notices page, but decided against it as it again seemed to be beyond the scope of what this project was able to do
43		within the given time constraints
44	10/03/2016	Increased the time for the rss feed to change slide, did some calculations and found that 5 feeds and 5 posts from each feed is about
45		as many requests as need to be made. To ensure most of the school can see them 15 seconds seemed more reasonable of a time
46	12/03/2016	Logic Error, meeting reminders were not being centered properly; turned out when being pulled from database they lost all of their formatting
47		this was solved by going into c# regular expressions and padding each side to center. Fairly dirty solution but it seemed to be reasonable
48		due to the constraints wpf had with rendering raw literal strings
49	13/03/2016	Needed to readdress how synchronous threads were reading from database, Run time error was thrown; too much database access at once
50		the tables locked and nothing was able to be done without restarting the program; this wouldn't have been good if it happened when released
51	17/03/2016	Solved, all threads run asynchronously now also added substring to the description abstract to make sure it didn't go over 500 characters.
52	18/03/2016	Program completed

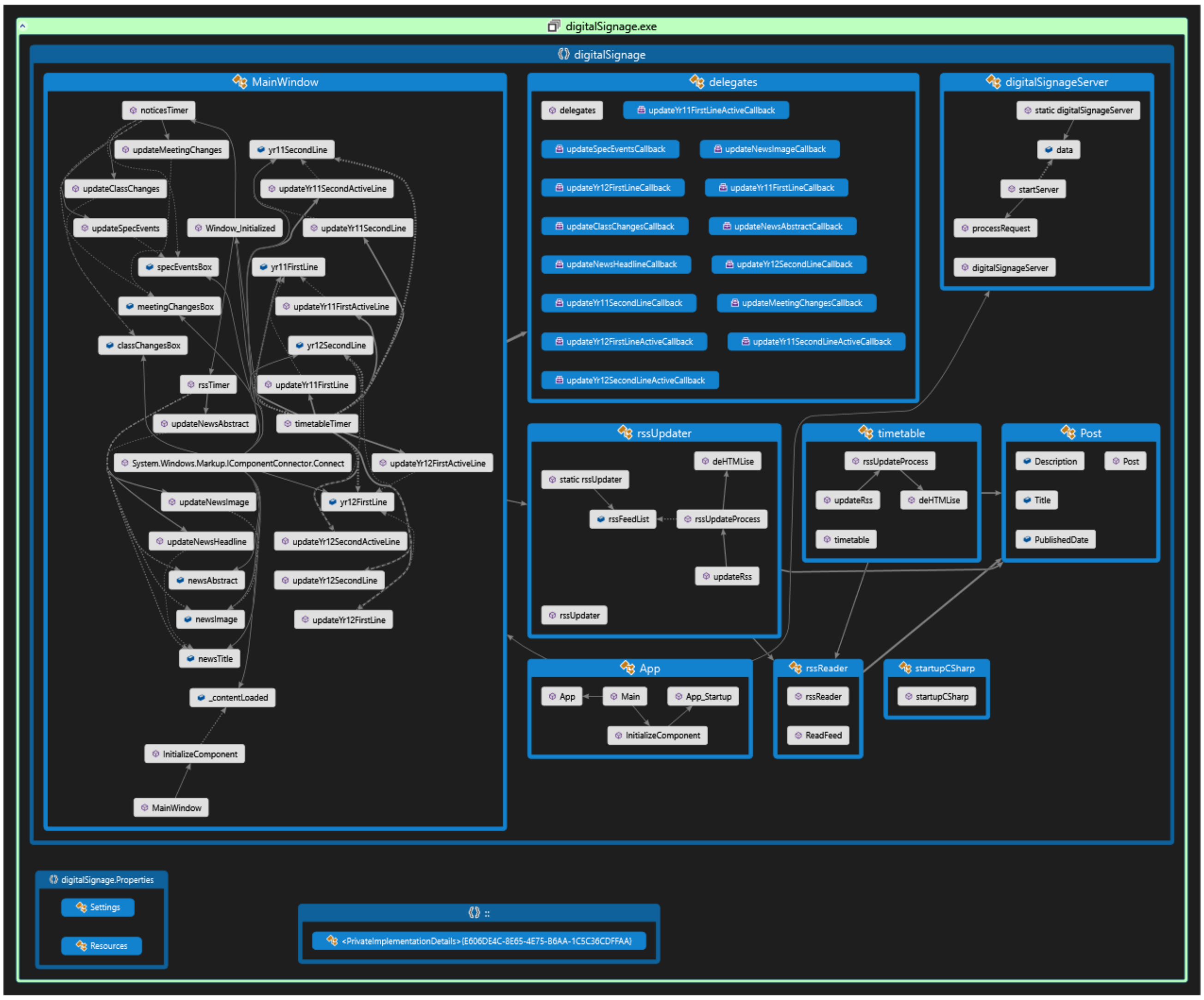
Appendix 16 (Bing image scraper):



Appendix 17 (Signage Client UML Diagram):



Appendix 18 (Signage Server UML Diagram):


digitalSignage.exe
digitalSignage
MainWindow
noticesTimer
updateMeetingChanges
yr11SecondLine
updateClassChanges
updateYr11SecondActiveLine
updateSpecEvents
Window_Initialized
updateYr11SecondLine
specEventsBox
yr11FirstLine
meetingChangesBox
updateYr11FirstActiveLine
classChangesBox
yr12SecondLine
rssTimer
updateYr11FirstLine
updateNewsAbstract
timetableTimer
System.Windows.Markup.IComponentConnector.Connect
updateYr12FirstActiveLine
updateNewsImage
yr12FirstLine
updateNewsHeadline
updateYr12SecondActiveLine
newsAbstract
updateYr12SecondLine
newsImage
updateYr12FirstLine
newsTitle
_contentLoaded
InitializeComponent
MainWindow
delegates
delegates
updateYr11FirstLineActiveCallback
updateSpecEventsCallback
updateNewsImageCallback
updateYr12FirstLineCallback
updateYr11FirstLineCallback
updateClassChangesCallback
updateNewsAbstractCallback
updateNewsHeadlineCallback
updateYr12SecondLineCallback
updateYr11SecondLineCallback
updateMeetingChangesCallback
updateYr12FirstLineActiveCallback
updateYr11SecondLineActiveCallback
updateYr12SecondLineActiveCallback
digitalSignageServer
static digitalSignageServer
data
startServer
processRequest
digitalSignageServer
rssUpdater
deHTMLise
static rssUpdater
rssFeedList
rssUpdateProcess
updateRss
rssUpdater
timetable
rssUpdateProcess
updateRss
deHTMLise
timetable
Post
Description
Post
Title
PublishedDate
App
App
Main
App_Startup
InitializeComponent
rssReader
rssReader
ReadFeed
startupCSharp
startupCSharp
digitalSignage.Properties
Settings
Resources
::
<PrivateImplementationDetails>{E606DE4C-8E65-4E75-B6AA-1C5C36CDFFAA}