WEB BROWSER

F20SC: Industrial Programming

Matthew Frankland
H00241997
mf48

Table of Contents

Introduction

The purpose of this report is to discuss my implementation of a simple web browser in C#. This report will include details on the project's requirements, design decisions and core functionality.

The remit of this project is to provide a simple Graphical User Interface (GUI) through which the user could: send Hyper Text Transfer Protocol (HTTP) requests; receive response messages; set and visit a home page; access and edit a list of favourite websites; and view a history list.

This report will also act as both a user guide and a developer guide. The user guide will demonstrate the basic operations of the web browser. The developer guide will demonstrate the code base's main functions in order to help with future maintenance and development of the browser. Test cases, including edge cases, and their output will also be provided.

An assumption was made that the persistent storage used to store the history list, favourites list, and current home page could be left unencrypted. A second assumption was made that the user would wait for an HTTP Request to respond before making any further requests.

Notes

Windows may request an app be downloaded from the app store to run the web browser's executable. Selecting no to this request will run the exe as normal.

The visual studio project attached was developed in **Visual Studio 2019** however the source code files are also included in a separate folder if using an older version of Visual Studio.

Requirements' Checklist

Requirement	Delivered	Comment
Sending HTTP request messages for URLs typed by the user	Yes	User request is sent upon pressing enter
Receiving HTTP responses and displaying content	Yes	
Handling HTTP response error codes and displaying the correct messages at the top of the browser	Yes	
Reloading a page	Yes	
Creating a Home Page URL	Yes	
Editing a Home Page URL	Yes	
Visiting a Home Page URL	Yes	
Loading a Home Page URL from persistent storage	Yes	Stored unencrypted
Adding a URL to Favourites with associated name	Yes	
Support for Favourite URLs modification and deletion	Yes	
Requesting Favourite URL by clicking on title	Yes	Double click required
Loading Favourite list from persistent storage	Yes	Stored unencrypted
Maintaining a History List	Yes	
Navigating to a URL in the History List by clicking on link	Yes	Double click required
Loading the History List from persistent storage	Yes	Stored unencrypted
Previous page navigation	Yes	Works across all tabs and not independently
Next page navigation	Yes	Works across all tabs and not independently
Simple GUI	Yes	
Use of menus (with shortcut keys) and buttons for increased accessibility	Yes	

Design Considerations

The design of the web browser's classes focused on separating the different logical components of the project. This helped differentiate the 'model' component from the 'view' component and lead to better maintainability and reusability. It also meant that the 'view' components could be more publicly accessible to other classes whereas the 'model' components could have stricter access controls.

User's history and favourites are stored in two generic lists of type `List<string []>`. This type allows for no forced constraints on the list's size, dependent upon the amount of memory available. I have however manually constrained the history list to 2000 elements for two reasons. Firstly, because users cannot manually delete history rows, in comparison with favourites where they can. This would have otherwise caused the history list to grow in size until the memory capacity was reached. Secondly, because entries in the history list could be duplicated if a user visited the same website twice, unlike in the favourites list.

The history list, favourites list, and home page address are stored in the class that initially runs the web browser, `Program ()`. Each serialized variable is read from, or written to, its own XML file which acts as persistent storage.

A second GUI acts as a popup form which displays user's history or favourites. This second form reduces the number of elements in the main form and helps keep the main form simple and easy to understand. Data in the popup form is displayed using text boxes to allow users to edit the data e.g. a favourite's title and link. An `EventHandler` is attached to some text boxes so that links can be visited if a user double clicks on the text box.

The `Connection ()` class handles errors and unexpected cases in HTTP requests by using exceptions. Two exception types are handled in the `Connection ()` class, `URIException` and `WebException`. A `URIException` is raised if a valid URI could not be constructed from the URL string passed to `Connection ()`. A `WebException` is raised if an error is sent back to the `HTTPResponse`, for example, a 404 Not Found error or a 401 Unauthorized error.

User Guide

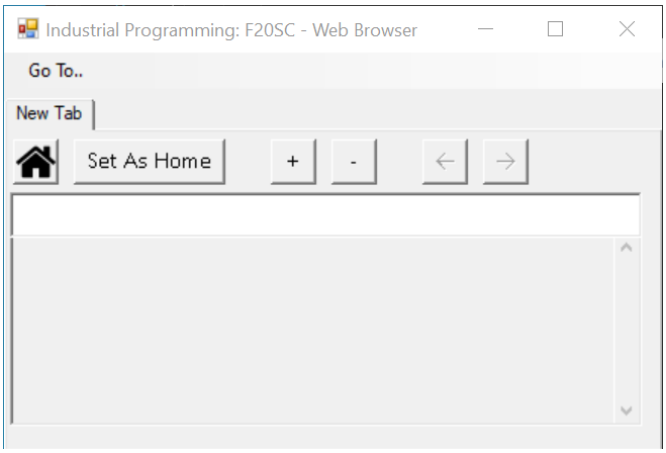

Figure 1 shows the web browser when it initially starts up. The web browser starts with no text in the response box or the address bar. On this screen the user can enter an address in the URI bar, press enter, and wait for a response; add or remove tabs; go back or forward in history; and set or visit the home page.

Figure 1: Start-Up

Figure 2 shows a pop-up box that indicates that the home page has been successfully updated to the current address. The user must visit a valid URI before being able to set it as the home page.

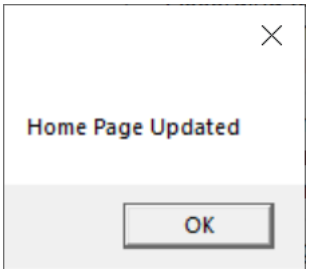

Figure 2: Home Page Updated

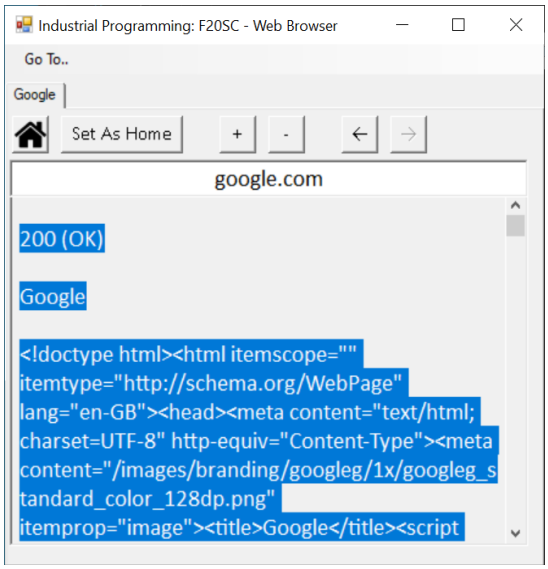

Figure 3 shows the response format for a HTTP Request. This includes the response status, the websites title if the tag exists, and the response source code.

Figure 3: HTTP Response

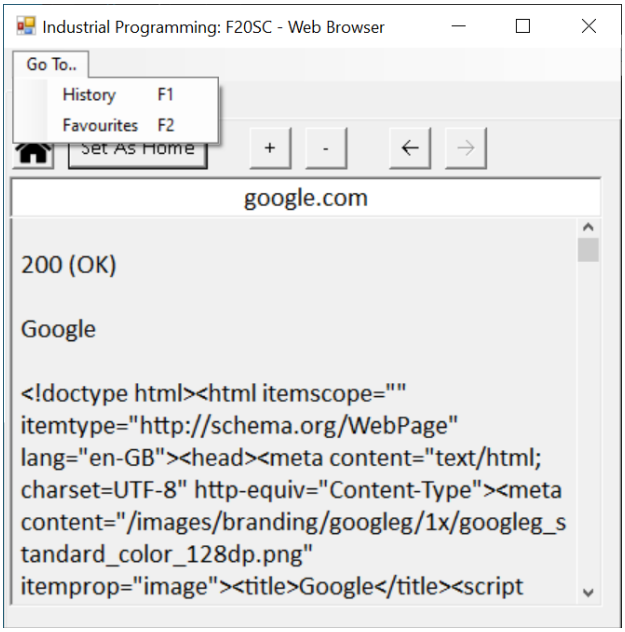

Figure 4: Drop Down Menus

Figure 4 shows the drop-down menus that are used to access the favourites and history pop up forms. These menus can also be accessed via shortcut keys F1 and F2.

Figure 5 shows the history pop up form which lists all previously visited websites. Users can visit a website by double clicking on a link.

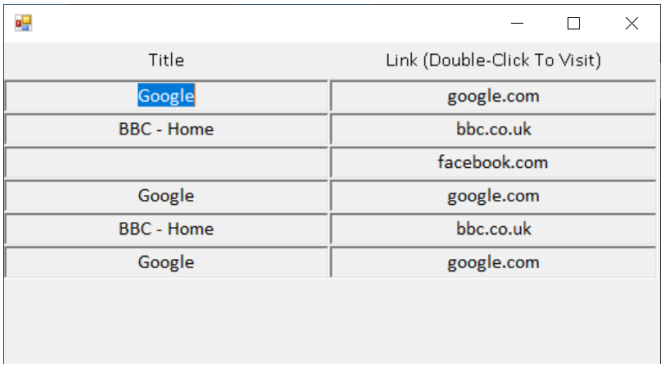

Figure 5: History

Figure 6: Favourites

Figure 6 shows the favourites pop-up form which lists all currently stored favourites. Users can add a new favourite by clicking the button 'Add New Favourite' and filling in the appropriate fields. Users can also edit existing data or visit a favourite by double clicking on the title field.

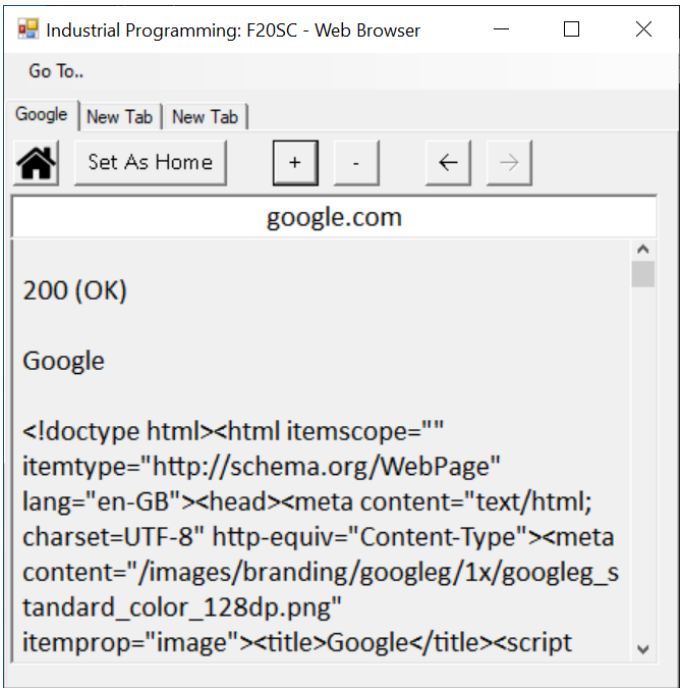

Figure 7: New Tabs

Figure 7 shows multiple tabs open at one time. Users can add new tabs to their current session by hitting the “+” button and remove tabs by pressing the “-“ button. Users can navigate through their history using the back “<-“ and forward “->” buttons if available in the given context. If these buttons are not available in the current context, they are disabled.

```
public Program()
{
    MinimumSize = new Size( 450, 200 );
    Text = "Industrial Programming: F20SC - Web Browser";
    KeyPreview = true;

    /*
     * Setup Menu Strip
     *
     */
    menu = new MenuStrip();
    menu.Location = new Point( 0, 0 );

    var goTo = new ToolStripMenuItem();
    goTo.Text = "Go To..";

    var history = new ToolStripMenuItem();
    goTo.DropDownItems.Add( history );
    history.Text = "History";
    history.Click += new EventHandler( show_history );
    history.ShortcutKeys = Keys.F1;

    var favourites = new ToolStripMenuItem();
    goTo.DropDownItems.Add( favourites );
    favourites.Text = "Favourites";
    favourites.Click += new EventHandler( show_favourites );
    favourites.ShortcutKeys = Keys.F2;

    menu.Items.Add( goTo );

    /*
     * Setup Tab Control
     */
    control = new TabControl();
    control.Width = Width - 15;
    control.Height = Height - 65;
    control.Location = new Point( 0, 30 );
    control.Anchor = ( AnchorStyles.Top | AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Bottom );
    control.SelectedIndexChanged += new EventHandler( change_tab );

    buttonHandlers = new EventHandler[] { go_home, home_update, new_tab, close_tab, go_back, go_forward };
    currentTab = new Tab( ref buttonHandlers, ref control );
    ( ( WebControl ) currentTab.Controls.Find( "webControl", true ) [ 0 ] ).setCurrentTab( currentTab );
    Controls.AddRange( new Control [] { menu, control } );

    /*
     * Get Stored Data
     */
    deserialize();
}
```

Figure 8

`Program ()` is the class that initially runs the web browser application. The constructor in `Program ()` initialises a menu strip and adds it to the applications controls. This menu strip provides access to the history and favourites pop up form. As the `ToolStripMenuItems` do not change between tabs they are initialised here and given their event handlers.

The tab control is set up in `Program ()` and provides overall control of open tabs. `Program ()` also has a public variable which stores the tab that is currently in focus.

A separate class `Tab ()` is called (figure 9) to initialise each tab page. The constructor for `Tab ()` requires a tab control reference to be passed in each call to its constructor. This allows `Tab ()` to add the initialised tab page to the tab control without having to pass back the tab page object.

`Tab ()` is responsible for initialising the GUI controls that each tab page contains. This includes buttons to go to and set the browser's home page; add or remove tab pages; and move back or forward in history. Each tab page contains a class `WebControl ()` which is responsible for the address bar and response text box. `WebControl ()` is also responsible for processing requests and displaying the returned information in an appropriate format (figure 10). An integer index 'historyIndex' is kept in `WebControl ()` that indicates which item in history is currently in focus. The program determines whether the forward and back buttons should be enabled based upon this index.

```
public Tab( ref EventHandler[] buttonHandlers, ref TabControl controller )
{
    Text = "New Tab";
    Width = controller.Width;
    Font = new Font( "calibri", 14 );

    setup_button( ref tabButtons[ 0 ], 30, new EventHandler ( buttonHandlers[ 0 ] ), new Point( 2, 5 ), "" );
    setup_button( ref tabButtons[ 1 ], 100, new EventHandler ( buttonHandlers[ 1 ] ), new Point(tabButtons[ 0 ].Location.X + tabButtons[ 0 ].Width + 10, 5 ), "Set As Home" );
    setup_button( ref tabButtons[ 2 ], 30, new EventHandler ( buttonHandlers[ 2 ] ), new Point(tabButtons[ 1 ].Location.X + tabButtons[ 1 ].Width + 30, 5 ), "+" );
    setup_button( ref tabButtons[ 3 ], 30, new EventHandler ( buttonHandlers[ 3 ] ), new Point(tabButtons[ 2 ].Location.X + tabButtons[ 2 ].Width + 10, 5 ), "-" );
    setup_button( ref tabButtons[ 4 ], 30, new EventHandler ( buttonHandlers[ 4 ] ), new Point(tabButtons[ 3 ].Location.X + tabButtons[ 3 ].Width + 30, 5 ), '\u2190'.ToString() );
    setup_button( ref tabButtons[ 5 ], 30, new EventHandler ( buttonHandlers[ 5 ] ), new Point(tabButtons[ 4 ].Location.X + tabButtons[ 4 ].Width + 10, 5 ), '\u2192'.ToString() );

    tabButtons[ 0 ].BackgroundImage = Properties.Resources.home;
    tabButtons[ 0 ].BackgroundImageLayout = ImageLayout.Stretch;
    tabButtons[ 4 ].Enabled = false;
    tabButtons[ 5 ].Enabled = false;

    Controls.AddRange( tabButtons );

    var webControl = new WebControl();
    webControl.Name = "webControl";
    webControl.Width = controller.Width - 10;
    webControl.Anchor = ( AnchorStyles.Top | AnchorStyles.Right | AnchorStyles.Left | AnchorStyles.Bottom );
    webControl.Location = new Point( 0, tabButtons[ 2 ].Location.Y + tabButtons[ 2 ].Height + 5 );
    Controls.Add( webControl );

    controller.TabPages.Add( this );
}
```

Figure 9

Separate public variables in `Program ()` store the web browser's history, favourites, and home address. This allows for these variables to be retrieved from anywhere in the

program. As multiple copies of each variable do not need to be maintained, these variables help decouple the web browser's classes.

```
/// <summary>
/// Called To Attempt All URL Requests
/// </summary>
/// <param name="address"></param>
/// <param name="backForward"></param>
public void goToAddress( String address, Boolean backForward )
{
    if ( address == "" )
    {
        response.Text = addressBar.Text = currentTab.Text = "";
    }
    else
    {
        Program parent = ( Program ) ( ( TabControl ) currentTab.Parent ).Parent;
        string[] responseText = conn.GET( address, "text/html" );

        currentAddress = addressBar.Text = address;
        response.Text = responseText[ 1 ];
        currentTab.Text = responseText[ 0 ];

        if ( responseText[0] != "Error" )
        {
            if ( !backForward && ( parent.getIndex() != parent.historyList.Count - 1 ) ) // If Address Not From Back or Forward Buttons And Not On Final Index
            {
                parent.setIndex( parent.historyList.Count - 1 ); // Jump To End of History
                currentTab.set_forward( false ); // Disable Forward
            }
            if ( !backForward )
            {
                if ( parent.historyList.Count > 2000 ) // If History List Getting Too Long
                {
                    parent.historyList.RemoveRange( 0, 500 );
                    parent.setIndex( parent.getIndex() - 500 );
                }
                parent.historyList.Add( new string[] { responseText[ 0 ], currentAddress } ); // Add To History
                parent.setIndex( parent.getIndex() + 1 );
            }
            if ( parent.getIndex() > 0 )
            {
                currentTab.set_back( true );
            }
        }
        response.Focus();
    }
}
```

Figure 10

Web requests are completed by first checking if the user has sent an empty request. If the user has sent an empty request the response and the address bar are set to empty. Otherwise a custom HTTP connection is used to make a GET Request (figure 11) to the address specified by the user. This address is placed in the address bar and the response is placed in the response box. The current tab's title is changed to the title of the website that is now showing. If the request contains no errors, the history functions of the web browser are updated. This includes updating the current history index.

```
/// <summary>
/// GET HTTP Request
/// </summary>
/// <param name="url"></param>
/// <param name="contentType"></param>
/// <returns>String Array Denoting Status Code, Website Title (Optional) and Code (Optional) </returns>
internal string[] GET( string url, string contentType )
{
    try
    {
        Uri uri = new UriBuilder( url ).Uri;
        HttpWebRequest request = ( HttpWebRequest ) WebRequest.Create( uri );
        request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
        request.ContentType = contentType;

        // <code>using()</code> closes readers and response automatically after use
        using ( HttpWebResponse response = ( HttpWebResponse ) request.GetResponse() )
        using ( Stream stream = response.GetResponseStream() )
        using ( StreamReader reader = new StreamReader( stream ) )
        {
            var code = reader.ReadToEnd();
            var match = new Regex( @"<title>(.+)</title>" ).Match( code );
            var title = match.Success ? match.Groups[ 0 ].Value.Substring( 7, match.Groups[ 0 ].Value.Length - 15 ) : "";

            return new string[] { title, Regex.Replace( String.Format( "\n200 (OK)\n\n{0}\n\n{1}", title, code ), "(?<!\r)\n", "\r\n" ) };
        }
    }
    catch ( UriFormatException e )
    {
        return new string[] { "Error", "The URI Could Not Be Parsed" };
    }
    catch ( WebException e )
    {
        var response = e.Response as HttpWebResponse;

        if ( response == null || response.GetResponseStream() == Stream.Null )
        {
            return new string[] { "Error", "Connection To Web Page Could Not Be Established" };
        }
        return new string[] { "Error", ( (int) response.StatusCode ).ToString() + " " + response.StatusDescription };
    }
}
```

Figure 11

Within the GET request method, a small and efficient regular expression is used to match and retrieve the website's title from the `<title></title>` tag (figure 10). If no tag exists, the title string is set to an empty string. The advantage of using a regular expression was that it was a short and simple method call compared with a string operation. There was however a slight degradation in readability.

The GET request method surrounds the `HttpWebResponse`, `Stream`, and `StreamReader` in a `using ()` statement. This statement removes the need to close the reader, or the response, as they are both automatically destroyed once the statement's boundaries are reached.

```
/// <summary>
/// On Close Save Variables Into Local XML Files
/// </summary>
protected override void OnFormClosing( FormClosingEventArgs e )
{
    base.OnFormClosing( e );
    var xs = new System.Xml.Serialization.XmlSerializer( typeof( List<string[]> ) );
    var xy = new System.Xml.Serialization.XmlSerializer( typeof( string ) );

    using (FileStream fs = new FileStream( AppDomain.CurrentDomain.BaseDirectory + @"\history.xml", FileMode.OpenOrCreate ) )
    {
        fs.SetLength(0);
        xs.Serialize( fs, historyList );
    }

    using (FileStream fs = new FileStream( AppDomain.CurrentDomain.BaseDirectory + @"\favourites.xml", FileMode.OpenOrCreate ) )
    {
        fs.SetLength(0);
        xs.Serialize( fs, favouritesList );
    }

    using (FileStream fs = new FileStream( AppDomain.CurrentDomain.BaseDirectory + @"\homeaddress.xml", FileMode.OpenOrCreate ) )
    {
        fs.SetLength(0);
        xy.Serialize( fs, homeAddress );
    }
}
```

Figure 12

The application's history, favourites, and home address are stored in local XML encoded files when the browser is closed. This is done by

serializing the local variables. Before the data is saved, the files are first wiped of old data (figure 12).

```
/// <summary>
/// Event Handler For Saving Favourites Upon Popup Window Closing
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void save_favourites( object sender, FormClosedEventArgs e )
{
    string name = "";
    Boolean textBoxExisted = false, nameFound = false;

    foreach ( Control x in Controls)
    {
        if ( x is TextBox )
        {
            if ( x.Name == "name" )
            {
                foreach( string[] s in toDisplay )
                {
                    if ( s[ 0 ] == x.Text ) // If Name Found Already In Favourites List
                    {
                        if ( ( ( int ) x.Tag ) == 0 ) // If Text Box Was AutoFilled On Load
                        {
                            textBoxExisted = true;
                        }
                        nameFound = true;
                        break;
                    }
                }
                name = x.Text;
            }
            else if ( x.Name == "address" )
            {
                if ( textBoxExisted && nameFound ) // If Data Looking At Was Autofilled and Name Found Twice
                {
                    foreach ( string[] s in toDisplay )
                    {
                        if ( name == s[ 0 ] )
                        {
                            s[ 1 ] = x.Text;
                        }
                    }
                    textBoxExisted = nameFound = false;
                }
                else
                {
                    if ( name != "" || ( nameFound && !textBoxExisted ) ) // Name Not Empety & Name Not Already In Favourites List But WasFound Twice
                    {
                        toDisplay.Add( new string[] { name, x.Text } );
                    }
                }
            }
        }
    }
```

Figure 13

The final part of the web browser is the pop-up form that is used to display the history (figure 10) and the favourites list (figures 13, 14 and 15). This form is a simple GUI consisting of labels denoting the columns of data; a button to add rows to the favourites list; and a button for deleting favourites rows.

The event handler to save favourites items is complicated in its design. This is primarily due to no two favourites being able to have the same title. Therefore, textboxes are tagged with values representing whether the data has been presented from storage or has been added by the user. If the same title is found twice, and the data has been entered by the user, the value is not added to the favourites list upon the pop-up window closing. This check is carried out by looping over all available controls and checking titles before moving to save the links (figure 13).

Rows in favourites are removed by first identifying their position. This is calculated via the delete button's Y position as this value is the same for all textboxes that are to be removed. Each text box is removed from the applications controls when it is found in a loop over all controls. The favourites list in `Program()` is also updated to reflect any changes. Finally, the button that activated the event handler is deleted (figure 14).

```
/// <summary>
/// Event Handler For Button To Delete Row In Popup Form
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void delete_fav( object sender, EventArgs e )
{
    string[] a = new string[] { "", "" };
    restart:  foreach ( Control x in Controls )
    {
        if ( x is TextBox && x.Location.Y == ( ( Button ) sender ).Location.Y )
        {
            Controls.Remove( x ); // Jumps One Controller So Need To Restart
            if ( a[ 0 ] == "" )
            {
                a[ 0 ] = x.Text;
            }
            else
            {
                a[ 1 ] = x.Text;
                for ( int i = 0; i < parent.favouritesList.Count; i++ )
                {
                    if ( parent.favouritesList[ i ][ 0 ] == a[ 0 ] && parent.favouritesList[ i ][ 1 ] == a[ 1 ] )
                    {
                        parent.favouritesList.RemoveAt( i );
                        break;
                    }
                }
                a = new string[] { "", "" };
            }
            goto restart;
        }
    }
    Controls.Remove( ( ( Button ) sender ) );
}
```

Figure 14

The pop-up form allows users to visit links by double clicking on the appropriate textbox. This event closes the pop-up form and retrieves the appropriate data to make this request. This retrieval is done in the same manner as delete i.e. based on the location of the text box. The web control object is retrieved from the tab that is currently in focus on the main form. A request to go to the web page selected by the user is made before the loop is returned out of.

```
/// <summary>
/// Event Handler For If Address Is Double Clicked In Popup Form
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void address_click( object sender, EventArgs e )
{
    Close();
    int n = ( ( ( TextBox ) sender ).Location.Y - 3 ) / 20; // Get TextBox Based On Location
    if ( !history ) { n = n - 2; }
    int i = 0;
    foreach( string[] p in toDisplay )
    {
        i++;
        if ( i == n )
        {
            ( ( WebControl ) parent.currentTab.Controls.Find( "webControl", false ) [ 0 ] ).goToAddress( p[ 1 ], false );
            if ( !history )
            {
                parent.currentTab.Text = p[ 0 ];
            }
            return;
        }
    }
}
```

Figure 15

Testing

Test	Expected	Success	Comment
Loading www.bbc.co.uk	200 (OK)	Yes	
Loading bbc.co.uk	200 (OK)	Yes	
Loading httpstat.us/404	404 Not Found	Yes	
Loading httpstat.us/403 while previous request is in progress	403 Forbidden	No	GUI frozen until request completed
Loading httpstat.us/401	401 Unauthorized	Yes	
Loading httpstat.us/400	400 Bad Request	Yes	
*Loading #!"£$%^&U*IO*	URI Exception	Yes	
Visit the Home Page	Response and address should be blank as Home set to “”	Yes	
Add facebook.com to Favourites	Row created containing title and address	Yes	
Visit facebook.com Favourite by double clicking on Title	Favourites pop up form closes; visits facebook.com; and displays user entered title in tab	Yes	
Add bbc.co.uk as a Second Favourite	Row created containing title and address	Yes	
Edit facebook.com Favourite title	Title changed upon removing focus from text box	Yes	
Delete bbc.co.uk from Favourites	Whole row marking bbc.co.uk is removed from Favourites	Yes	
Close the Favourites window and open again	Favourites pop up form has just facebook.com as a favourite	Yes	
Go to google.co.uk and set to the Home Page	Pop up message saying home page has been updated	Yes	

Go to bbc.co.uk and then go to the Home Page	Response changes from bbc.co.uk to google.co.uk	Yes	
Press the Back Button	Visits bbc.co.uk (200 (OK))	Yes	
Go to facebook.com	Visits facebook.com (200 (OK))	Yes	
Press the Back Button	Visits google.co.uk (200 (OK))	No	As a website is entered into the address bar in the previous step it is placed at the end of the history list. The back pointer is then reset to this position. This is why google.co.uk is returned as it is the last item to be entered into the history list before facebook.com
Press the Back Button	Visits bbc.co.uk (200 (OK))	Yes	
Press the Forward Button	Visits google.co.uk (200 (OK))	Yes	
Press the Forward Button	Visits facebook.com (200 (OK))	Yes	
Add a New Tab	New tab page opened in control	Yes	
Visit google.co.uk	Web page visited does not affect previous tab	Yes	
Close the Web Browser and open again	History and Favourites lists have persisted.	Yes	

Conclusions

In conclusion I think the web browser is accessible, easy to use, and the GUI works well. If I had additional time, I would have stopped the user interface from freezing while it waited for a HTTP response. I would have also added the ability to switch between a response's source code and graphical rendering. Additionally, I would have added the functionality for back and forward buttons to work independently between tabs rather than universally between all tabs.