Web Browser

Coursework 1

F20SC: Industrial Programming

Sam Fay-Hunt — `sf52@hw.ac.uk`

October 29, 2020

Contents

1 Introduction

This report is to provide supporting information for those wishing to use or critically analyse the software described within. Information such as design choices, instructions for using the software and details of the implemenation will all be avaliable in the following sections. The task is to build a simple web browser using C#, and to make effective use of the functional and object oriented features of a novel language to me.

It is assumed any users will already be familiar with commonly used web browsers such as Google Chrome or Mozilla Firefox, particularly GUI design descisions are based on this assumption.

2 Requirements

This Section will provide a comprehensive requirements checklist. The requirements have been split into 2 sections: Section. 2.1 (Model Requirements) contains all the requirements related to the "business logic", Section. 2.2 (User Interface Requirements) will describe all the requirements related to the user interface view and control components.

Requirements marked with the Priority **Essential** are requirements explicitly requested in the task brief.

The priority for each requirement is encoded as follows:

- **Essential** - This priority indicates that this requirement must be implemented to satisfy basic functionality of the browser.
- High - A high priority indicates that this requirement is important for providing a good quality user experience.
- Medium - A medium priority requirement is nice to have but if it is missing it is acceptable.
- Low - Low priority indicates that this is unlikely to be fulfilled within the time frame of the project, a stretch goal at best.

2.1 Model Requirements

Requirement	**Description**	**Priority**	**Status**
Send and recieve HTTP requests/responses	Send HTTP request messages for URLs typed by the user then recieve and store HTML code & response status codes	**Essential**	Complete
Display HTTP response	Display HTTP statuscode, the title of the web page & the HTML source code	**Essential**	Complete
Home page	Create and edit home page URL	**Essential**	Complete
Favourites	Add name and URL to a list of favourite web pages, these need to be editable and can be used for navigation	**Essential**	Complete
History	A list of visited URLs should be maintained, that can be used to navigate	**Essential**	Complete
Persistant home page	Store the home page url locally and load it to the browser on startup	**Essential**	Complete
Persistant favourites and History list	Serialize the lists and load them to the browser on startup	**Essential**	Complete
Modify history	The user should be able to clear all or specific history items	Medium	Complete
History names	The list of URLs should have reference to the associated page title	Medium	Complete
Sort history/favourites	Sort the list using an appropriate metric	Medium	Complete
Prepend incomplete URL with protocol	Prepend the protocol ”http://” & ”www.” to the URL when missing	Low	Incomplete
File watcher	A file watcher will observe changes in the history and favourites files and trigger reads and writes consquently	Low	Incomplete

2.2 User Interface Requirements

Requirement	**Description**	**Priority**	**Status**
URL input box	Enter a URL to send HTTP requests	**Essential**	Complete
Display HTML code, status code, title	The Gui should make the status code, HTML and title visible	**Essential**	Complete
Navigate to and set home page	Able to set a home page and navigate at the click of a button	**Essential**	Complete
Set custom favourite	A button to set a favourite with a user defined title	**Essential**	Complete
View favourites list	A selectable menu to display all favourites currently represented in the model favourites list	**Essential**	Complete
Edit favourites window	A window that enables the user to delete or update favourites	**Essential**	Complete
View history list	A selectable menu to display all history items currently represented in the models history list	**Essential**	Complete
Edit history window	A window that provides controls enabling the user to delete or edit history items	**Essential**	Complete
URL input box	Enter a URL to send HTTP requests	**Essential**	Complete
Shortcut keybinds	Make use of shortcut keys to improve accessibility	**Essential**	Complete
Display sorted history menu items	Render the history list in sorted order	Medium	Incomplete
Display sorted favourites menu items	Render the favourites list in sorted order	Medium	Incomplete

3 Design Considerations

3.1 Class design

In general the high level design of this browser has loosely followed a MVC pattern.

3.1.1 PageContent

The most important class to the underlaying model of this web browser is the **PageContent** class, this class contains references to the **PageHistory**, **History**, **Favourites**, and **BrowserResponse** classes. It abstracts away alot of the async behaviour by using an Event called **ContextChanged** which is triggered any time a HTTP get request returns a new **BrowserResponse** instance, this means the GUI elements can subscribe to this event and repaint the relevant elements when it gets triggered. This class also exposes all the navigation functionality to the GUI and the information such as the HTML code, web page title and status code returned via get requests.

3.1.2 BrowserResponse & HttpRequests

HTTP GET requests are handled by the *static Get(string url)* method of the **HttpRequests** class, this method returns an instance of the **BrowserResponse** class which is instantiated asynchronously and contains properties including the page title, HTML source code, URL, and status code. The **BrowserResponse** class uses the AngleSharp library to read and access DOM elements [1].

3.1.3 History & Favourites

There is a significant degree of overlap in the behaviour of the **History** and **Favourites** classes so the common functionality has been implemented in a class called **EntryRecord**. The main distinction between them is the behaviour when there is a duplicated title in their respective lists, this is realised by overriding the *KeyExists()* method.

The **EntryRecord** class handles all the list operations (adding, removing, updating elements), and raising events when the list has changed to notify the GUI. This class also has access to the **Persistance** class that has serilization and deserilization methods.

3.1.4 PageHistory

PageHistory is a session based history navigation class, it only exposes methods for moving back and forwards through the history for a given session, and adding new history items to its own list. It also has properties that provide information about the current node being pointed to in the list. This class is independent of the **History** class in terms of data, but the nodes representing a single page in the history do inherit from the same abstract class called **Entry**. It was a deliberate design descision to seperate the **PageHistory** and **History** classes, this behaviour is reflected in several prominent web browsers, for example Google Chrome and Mozilla Firefox.

3.2 Data structures

The **EntryRecord** class uses the generic form of the list (List<T>where T is the class **EntryElement**) because it has the convienience of built in search, sort and list manipulation methods as a result of implementing IList<T>, IEnumerable<T>& ICollection<T>. [2]

PageHistory uses a hand built doubly linked list for maintaing the list of pages that could be traversed within the session. This was a convienience descision because modelling it in this manner makes the behaviour very readable but it may not be the most performant solution.

3.3 Gui Design

The primary component of the graphical user interface is the browser window itself, this window features a large textbox to display the HTML source code, along the top of the window there are a series of buttons arranged around a url input text box: back, forwards, refresh, home, go, and menu. The bottom left of the window displays a status code and message. This design was heavily inspired by existing browser designs so a user should be able to use this browser fairly intuitively.

3.4 Advanced language constucts

Delegates have been used extensively throughout this body of work, particularly when paired with event handlers where delegates are used to invoke methods, see Fig. 10. Anonymous delegates have been very useful during unit testing, specifically to check if an event had been triggered see Fig. 10. [3]

Favourites and **History** both use a singleton pattern for instantiation (there should only be a single instance of each), to do this in a thread safe manner the Lazy<T>class was used with the Enum LazyThreadSafetyMode.ExecutionAndPublication parameter see Fig. 1. [4], [5]

Built in generic types are heavily featured in this codebase such as List<T>, and Task<T> [6]. The most notable custom generic class is **Persistance<T>** which makes use of generics to serialize and deserialize List<T>objects. It should however be noted that with the code in its current state it is unnecessary to use a generic because only a single class is used to instantiate an instance of **Persistance<T>**, however for future changes to the program it would be useful to keep it generic to potentially serialize a collection such as **List<PageEntry>**.

```
// Sealed to aid the JIT compiler
public sealed class SingletonClass
{
    private static readonly Lazy<SingletonClass> singleton =
        new Lazy<SingletonClass>(() => new SingletonClass(true),
        LazyThreadSafetyMode.ExecutionAndPublication);

    public static SingletonClass Instance { get { return singleton.Value; }}

    private SingletonClass() { }
}
```

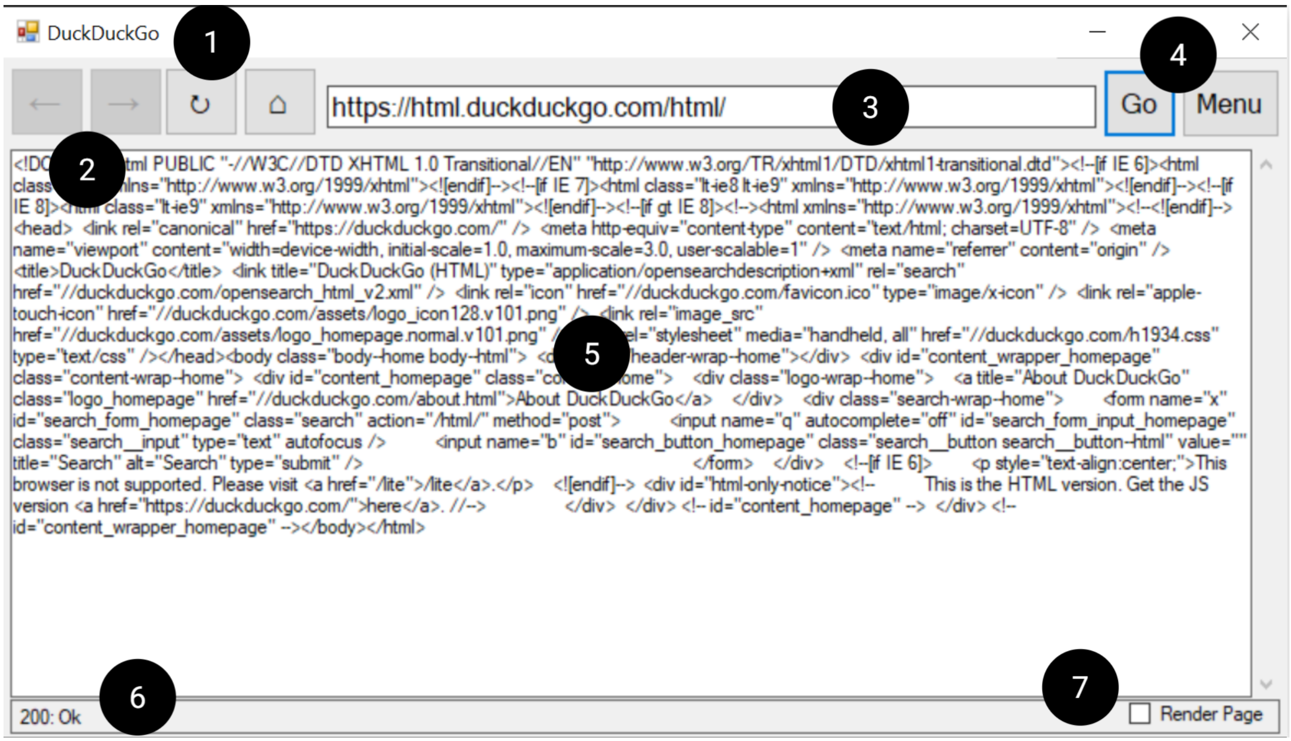
Figure 1: Using Lazy<T>to safely instantiate a singleton class

Small optimisations have been make like sealing classes without children inheriting from them to help the JIT compiler do its job. [7]

4 User Guide

4.1 Browser Page

This is the basic view of the browser, here you can navigate around the web, check resonse statuses and view the pages source code.



```
<!DO      tml PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><!--[if IE 6]><html
class      mlns="http://www.w3.org/1999/xhtml"><![endif]--><!--[if IE 7]><html class="lt-ie8 lt-ie9" xmlns="http://www.w3.org/1999/xhtml"><![endif]--><!--[if
IE 8]><     class="lt-ie9" xmlns="http://www.w3.org/1999/xhtml"><![endif]--><!--[if gt IE 8]><!--><html xmlns="http://www.w3.org/1999/xhtml"><!--<![endif]-->
<head>  <link rel="canonical" href="https://duckduckgo.com/" />  <meta http-equiv="content-type" content="text/html; charset=UTF-8" />  <meta
name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=3.0, user-scalable=1" />  <meta name="referrer" content="origin" />
<title>DuckDuckGo</title>  <link title="DuckDuckGo (HTML)" type="application/opensearchdescription+xml" rel="search"
href="//duckduckgo.com/opensearch_html_v2.xml" />  <link rel="icon" href="//duckduckgo.com/favicon.ico" type="image/x-icon" />  <link rel="apple-
touch-icon" href="//duckduckgo.com/assets/logo_icon128.v101.png"  <link rel="image_src"
href="//duckduckgo.com/assets/logo_homepage.normal.v101.png"      el="stylesheet" media="handheld, all" href="//duckduckgo.com/h1934.css"
type="text/css" /></head><body class="body--home body--html">  <d      header-wrap--home"></div>  <div id="content_wrapper_homepage"
class="content-wrap--home">  <div id="content_homepage" class="co      ome">  <div class="logo-wrap--home">  <a title="About DuckDuckGo"
class="logo_homepage" href="//duckduckgo.com/about.html">About DuckDuckGo</a>  </div>  <div class="search-wrap--home">   <form name="x"
id="search_form_homepage" class="search" action="/html/" method="post">     <input name="q" autocomplete="off" id="search_form_input_homepage"
class="search__input" type="text" autofocus />      <input name="b" id="search_button_homepage" class="search__button search__button--html" value=""
title="Search" alt="Search" type="submit" />                      </form>   </div>   <!--[if IE 6]>     <p style="text-align:center;">This
browser is not supported. Please visit <a href="/lite">/lite</a>.</p>   <![endif]--> <div id="html-only-notice"><!--      This is the HTML version. Get the JS
version <a href="https://duckduckgo.com/">here</a>. //-->          </div>  </div> <!-- id="content_homepage" -->  </div> <!--
id="content_wrapper_homepage" --></body></html>
```

Figure 2: Basic browser view

1. Web Page title

2. back, forwards, refresh & home buttons - Standard, familiar browser navigation buttons

3. URL text input box - Input desired URL here

4. Go & Menu buttons - click Go to navigate to the URL contained in the text box

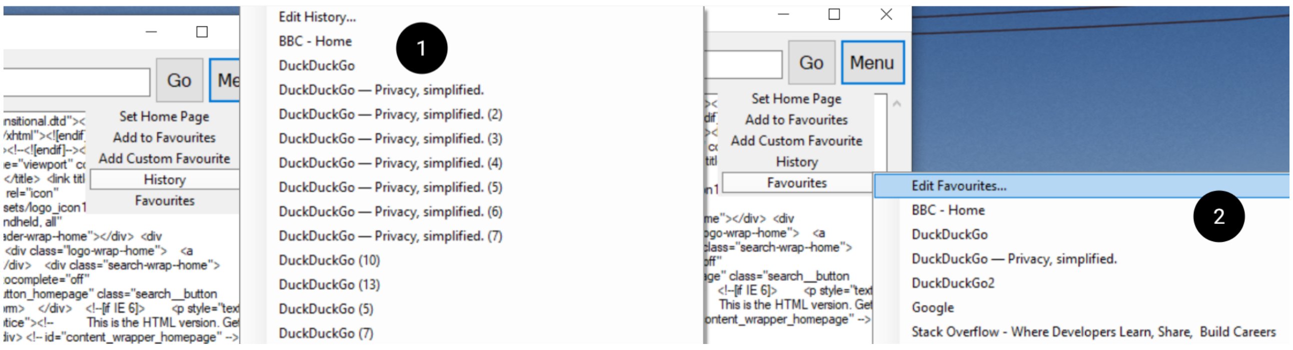
5. HTML source code view

6. HTTP status code / message

7. Render page toggle check box - Check this to toggle to a rendered view of the page

4.2 Menu expanded

Fig. 3 shows the menu open, options for setting the home page, adding to favourites, adding custom favourites, viewing the history/favourites are within this menu.



Go
Me
Set Home Page
Add to Favourites
Add Custom Favourite
History
Favourites
Edit History...
BBC - Home
DuckDuckGo
DuckDuckGo — Privacy, simplified.
DuckDuckGo — Privacy, simplified. (2)
DuckDuckGo — Privacy, simplified. (3)
DuckDuckGo — Privacy, simplified. (4)
DuckDuckGo — Privacy, simplified. (5)
DuckDuckGo — Privacy, simplified. (6)
DuckDuckGo — Privacy, simplified. (7)
DuckDuckGo (10)
DuckDuckGo (13)
DuckDuckGo (5)
DuckDuckGo (7)
1
Go
Menu
Set Home Page
Add to Favourites
Add Custom Favourite
History
Favourites
Edit Favourites...
BBC - Home
DuckDuckGo
DuckDuckGo — Privacy, simplified.
DuckDuckGo2
Google
Stack Overflow - Where Developers Learn, Share, Build Careers
2

Figure 3: Menus opened

1. This is the opened menu with History selected, you can select any one of these elements and navigate to the page.

2. This is the Favourites menu opened, highlighted is ‘Edit Favourites...’ clicking this will open the favourites editor window.

4.3 Edit Favourites

Fig. 4 shows the window that opens upon selecting the edit favourites button in the menu.

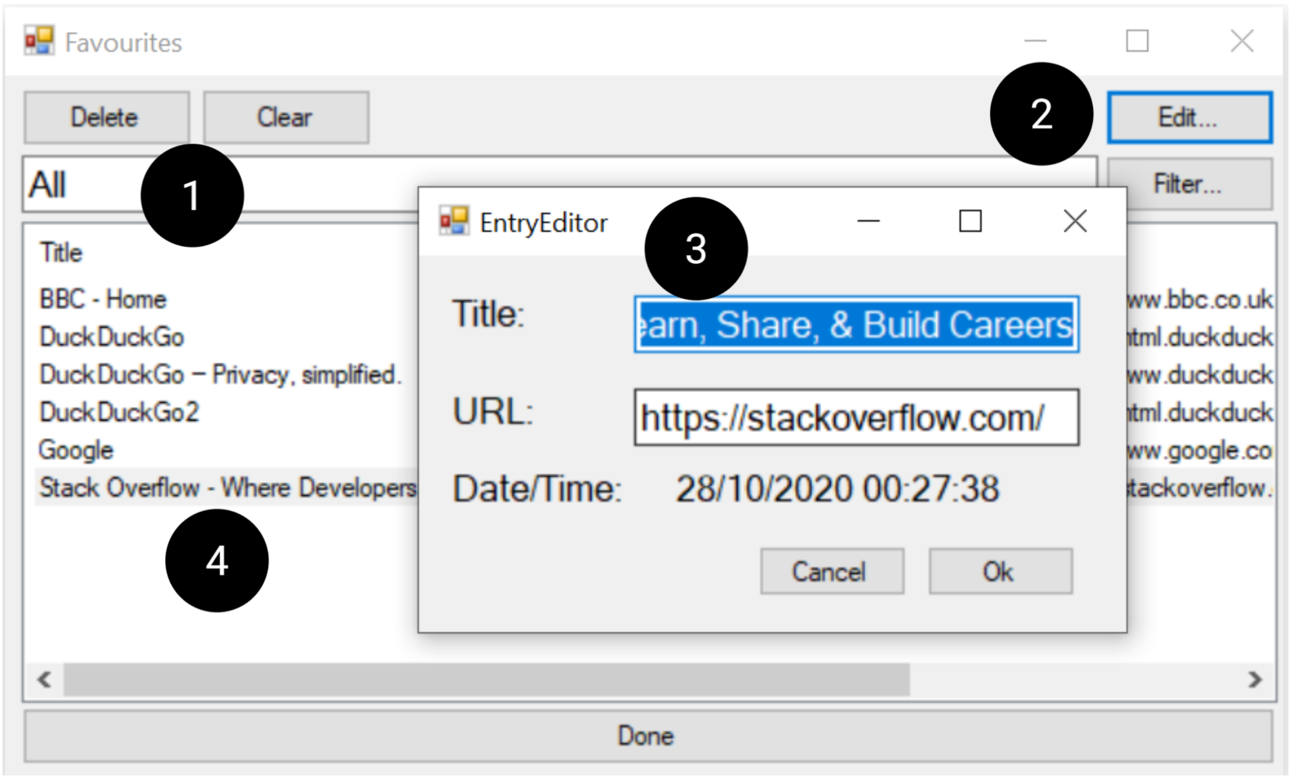


Figure 4: Editing a favourite

1. Here you can see the delete and clear buttons, along side a filter text box(non-functional)
2. This is the edit button, this is only avaliable when a single item in the list of elements is selected
3. The Entry Editor window lets you modify the Title or URL of the entry but changing the contents of the text boxes
4. Here we can see a list of all the current favourites, their titles, urls, and access times.

5 Developer Guide

A high level description of the underlaying model class design can can be found in Section. 3.1. This section will focus on the code required to get the model running and linked to the gui.

5.1 Libraries and Frameworks

- **Windows Forms** - the graphical library, this library is very accessible and can be picked up at a very rapid pace so its almost ideal for a first time project using a GUI in C#.
- **AngleSharp** - A DOM parsing library, this allows us to access the DOM using a logical class structure greatly simplifying accessing specific DOM elements when necessary.

- **Microsoft Visual Studio Unit Testing Framework** - This framework has been used to write the unit tests, it has built in integration to the IDE used for developing this software: Visual Studio 2019

5.2 Interacting with the model

Assuming you have representations of the GUI elements described in Section. 3.3, first insantiate the **History** & **Favourites** classes so you can assign their event handlers. See Fig. 5

```
public partial class ExampleBrowser
{
    private Favourites fav;
    private Histroy hist;
    public ExampleBrowser()
    {
        fav = Favourites.Instance;
        hist = History.Instance;
        fav.EntryChanged += Favourites_Changed;
        hist.EntryChanged += History_Changed;
        // Deserialize any Favourites or History elements stored locally
        fav.DeserializeCollection();
        hist.DeserializeCollection();
    }
    private void Favourites_Changed(object sender, EntryRecordChanged e)
    { /* Update Favourites gui elements */}
    private void History_Changed(object sender, EntryRecordChanged e)
    { /* Update History gui elements */}
}
```

Figure 5: Get the history and favourites instances and assign event handlers

Now that we have our History and Favourites instances we can instantiate a **PageContent** object. See Fig. 6

```
public partial class ExampleBrowser
{
    private PageContent content;
    // This method is called by a gui element event that is triggered
    // upon first loading the gui
    private async void ExampleBrowser_Load(object sender, EventArgs e)
    {
        content = await PageContent.AsyncCreate(fav.HomeUrl, hist, fav);
        content.ContextChanged += content_OnContextChanged;
        // at this point we can use the properties of content to assign
        // values to gui elements.
    }
    private void content_OnContextChanged(
        object sender,
        ContextChangedEventArgs e)
    { /* Update all gui elements realated to the current web page here */ }
}
```

Figure 6: Async instantiate PageContent with the Home URL

At this point you just need to use the public methods in the **PageContent**, **History**, and **Favourites** classes to update the view, and to send control commands to the model. See Fig.8 for a description of the **PageContent** class important public methods and their signatures. Likewise see Fig. 9 for a non-exhaustive summary of useful public **History**/**Favourites** methods.

5.3 GUI elements

This solution uses 4 Windows Forms classes, **BrowserWindow**, **EntryCollectionEditor**, **EntryEditor**, and **FavouritesDialogue**.

BrowserWindow is the class representing the actual browser itself, all the GUI components for this class but the dropdown menu are visible in Fig. 2. Once the basic shape of the Windows forms controls has been defined its just a matter of linking up all the events with appropriate methods called within the **PageContent**, **Favourites** and **History** classes.

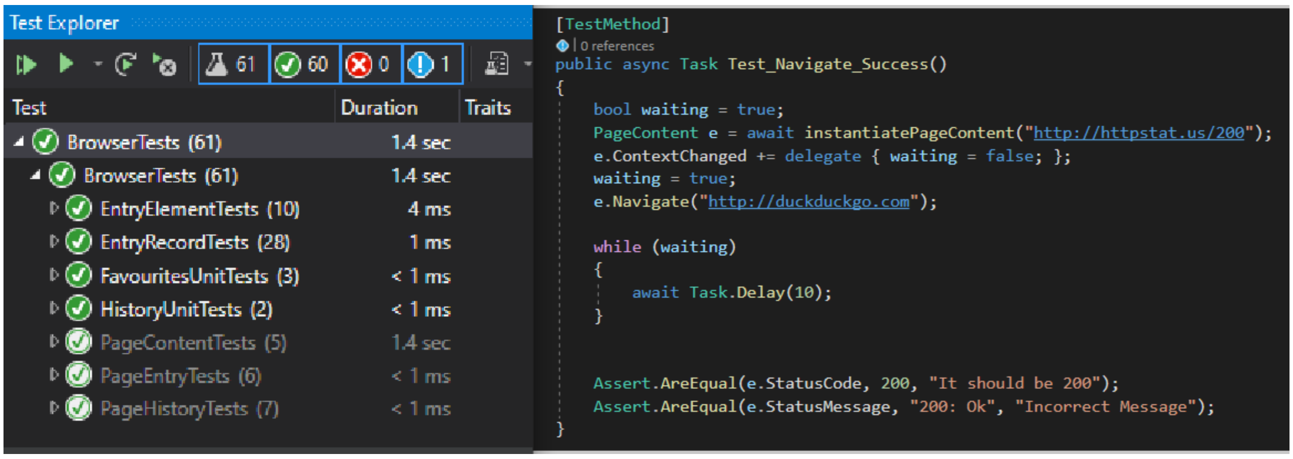
EntryCollectionEditor holds a reference to the **EntryRecord** that contains the List being painted onto the ListView that way any change made to the underlaying source List means that the source List can be used to repaint the ListViewContents. The rest is mostly event handlers that trigger behaviour based on button presses, selection changes ect.

EntryEditor is a very simple component, it is comprised of 2 text input boxes corresponging to the title and url of the entry. If the user makes a change and presses ok, the underlaying element gets mutated an event gets triggered and the window closes.

FavouritesDialogue has very similar appearence to **EntryEditor** but it doesnt use any event handlers. it simply allows the user to instantiate a new EntryElement for the Favourites object.

6 Testing

Because parts of this project were as much a learning exercise as an engineering one it took some experimenting with Windows Forms to build the proper mental model for how it all fits together, as a result writing unit tests was delayed until much later in development. This resulted in a design with numerous tight coupling issues between the model and the gui elements. Expanding the coverage of unit testing revealed many such issues, most (but not all) were resolved. There was a consistent issue with coupling the serializer class **Persistance** with the **Favourites** & **History** classes, this was resolved in an unsatisfactory manner by creating a seperate static method to instantiate a testing version of these classes that are unlinked to the **Persistance** class. The unit testing also presented issues when testing methods with async void signatures, resolving this issue would take a significant amount of refactoring and would mainly serve the purpose of making those methods unit testable. An unsuccessful attempt at testing such a case can be observed in Fig. 7 (right).



```
Test                          Duration   Traits
◢ BrowserTests (61)           1.4 sec
  ◢ BrowserTests (61)         1.4 sec
    ▷ EntryElementTests (10)  4 ms
    ▷ EntryRecordTests (28)   1 ms
    ▷ FavouritesUnitTests (3) < 1 ms
    ▷ HistoryUnitTests (2)    < 1 ms
    ▷ PageContentTests (5)    1.4 sec
    ▷ PageEntryTests (6)      < 1 ms
    ▷ PageHistoryTests (7)    < 1 ms
```

```
[TestMethod]
0 references
public async Task Test_Navigate_Success()
{
    bool waiting = true;
    PageContent e = await instantiatePageContent("http://httpstat.us/200");
    e.ContextChanged += delegate { waiting = false; };
    waiting = true;
    e.Navigate("http://duckduckgo.com");

    while (waiting)
    {
        await Task.Delay(10);
    }


    Assert.AreEqual(e.StatusCode, 200, "It should be 200");
    Assert.AreEqual(e.StatusMessage, "200: Ok", "Incorrect Message");
}
```

Figure 7: (Left) Summary of the number of tests for each class. (Right) Unsuccessful attempt at testing async void method

The classes related to the history/favourite do have excellent test coverage (aside from interactions with **Persistance**), these classes were mostly engineered in a loosely coupled manner and as a result it was quite easy to write tests for them. On the other hand the **PageContent** class has many async void methods, so the test coverage is poor.

Throughout development manual integration tests were performed, these were adequte at the beginning but as the code base grew the need for a more formal testing suite was becomming increasingly apparent. Ultimately the project had 61 unit tests to complement the regular integration tests.

7 Conclusions

The basic functionality of this software was very easy to engineer, windows forms has a fairly low learning curve. This encouraged me to try an implement things beyond the spec such as sorting algorithms for the history and favourites. I even began to implement a filewatcher, with the intent to handle multiple browsers running in parallel reading and writing the same files, while all maintaining history and favourites in sync with each other. I strongly feel that there are so many areas to improve the software, for example the GUI lists do not reflect the sorted order of the entries in History and Favourites. If starting again I would implement a function to convert the sorted List into a ToolStripItemCollection type instead of just adding the most recently modified one to the end. Another area I feel could be really improved is error handling, C# feels quite relaxed about handling errors explicitly compared to other languages that use the functional paradigm such as Rust where whenever an error might be raised you have to handle it or explicitly tell the compiler it wont be an error (.unwrap()) [8]. Unfortunately this means its very likely there are a number of unhandled exceptions in my code because I often found myself handling only one of several exceptions (the easiest one to throw), and without going through it with a fine tooth comb I may not encounter the unhandled exceptions at all. I also feel I didnt make much good use of the existing interfaces and likely re-engineered alot of algorithms due to my lack of familiarity with the subtlelys of C#. Overall it felt like a productive project that required revisiting a wide range of programming skills such as async, inheritence, considering the effect of immutability and working with collections.

8 References

References

[1] (). "AngleSharp - Home," [Online]. Available: `https://anglesharp.github.io/` (visited on 10/29/2020).

[2] dotnet-bot. (). “List<T> Class (System.Collections.Generic),” [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1` (visited on 10/29/2020).

[3] BillWagner. (). “Delegates - C# Programming Guide,” [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/` (visited on 10/29/2020).

[4] dotnet-bot. (). “Lazy<T> Class (System),” [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/api/system.lazy-1` (visited on 10/29/2020).

[5] ——, (). “LazyThreadSafetyMode Enum (System.Threading),” [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/api/system.threading.lazythreadsafetymode` (visited on 10/29/2020).

[6] ——, (). “Task Class (System.Threading.Tasks),” [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task` (visited on 10/29/2020).

[7] BillWagner. (). “Sealed modifier - C# Reference,” [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed` (visited on 10/29/2020).

[8] (). “Std::result::Result - Rust,” [Online]. Available: `https://doc.rust-lang.org/std/result/enum.Result.html#method.unwrap` (visited on 10/29/2020).

A Appendix

```
class PageContent
{
    // Asyncronously instantiate a PageContent object, replaces constructor
    public static async Task<PageContent> AsyncCreate(
        string url,
        History singletonHistory,
        Favourites singletonFavourites)

     // Navigate to a new page, add the new page to history
    public async void Navigate(string url)

    // Navigate to a page without adding to history
    public async void NavigateNoHistory(string url)

    // Navigate back through local history
    public void Back()

    // Navigate forwards through local history
    public void Forwards()
}
```

Figure 8: Useful PageContent public method signatures with descriptions

```
class EntryRecord
{
    // Param: filename - The name of the file used to
    //        serialize the contained collection.
    // Param: CompareBy - An enum that determines the order that
    //        the collection will be sorted by.
    // Param: withPersistence - Serialize or not
    public EntryRecord(
        string filename,
        CompareBy sortOrder,
        bool withPersistence)

    // Add a new entry to the collection
    public void AddEntry(string url, string title, bool write)

    // Remove entry from the list, control thrown event
    public void RemoveEntry(string title, bool write, bool bEvent)

    // clear all elements from the list
    public void ClearList(bool write)

    // get the list object
    public List<EntryElement> GetList()

    // Method to serialize the list
    public void SerializeCollection()

    // Method to deserialize the list
    public void DeserializeCollection()
}
```

Figure 9: Some useful EntryRecord public method signatures with descriptions

```
        {
            public class Example
            {
                public event EventHandler ExampleHandler = delegate { };

                public void ExampleEvent()
                {
                    EventHandler handler = ExampleHandler;
                    // No need to check if ExampleHandler is null because
                    // it has the empty delegate assigned
                    handler(null, EventArgs.Empty);
                }

                public void TriggerEvent()
                {
                    ExampleEvent();
                }
            }

            [TestClass]
            class Test
            {
                [TestMethod]
                public void TestEventDelegate()
                {
                    bool eventTriggered = false;
                    Example e = new Example();
                    // Named delegates are more commonly used here
                    e.ExampleHandler += delegate { eventTriggered = true; };
                    Assert.AreEqual(eventTriggered, false);
                    e.TriggerEvent();
                    Assert.AreEqual(eventTriggered, true);
                }
            }
        }
```

Figure 10: Using delegates to raise, trigger, and test events