ASSESSED COURSEWORK 1 —DEVELOPING A SIMPLE WEB BROWSER

F21SC: Industrial Programming

Abstract

This report documents the design, implementation and testing of a basic web browser. It contains a user guide and development guide to gain insight into the program.

Stuart Paterson

H00123967

1. Introduction

This report documents the design process of a web browser created in C#. It covers the requirements set and how the program met these. It also documents some design considerations that led to the provided features. Next, a user guide and developer guide are provided. There are details of the testing carried out on this web browser and analysis of how the results compared to what was expected. Finally there is some reflection on how well the language and implementations were suited to this program.

The web browser is operated using a graphical user interface (GUI) developed using Windows Forms. An example image of the browser in operation is shown in Figure 1.

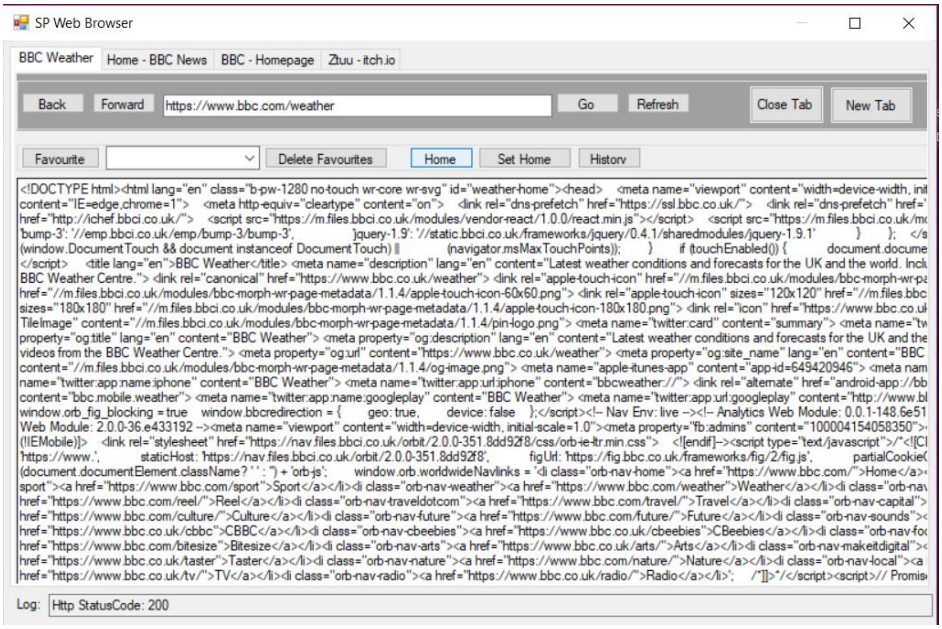


Figure 1: The GUI of the web browser showing the bbc weather website html

2. Requirements' Checklist

Table 1 shows a list of the project requirements whether they were met. Later in this report how each requirement was implemented will be expanded on in more detail.

Table 1: A table of the project requirements and whether they were met

Requirements	Status
Send an http request to a url typed by the user.	Fully met.
Receive an http response message from a request and display html to user.	Fully met.
Handle http status code 400, 403 and 404 errors as well as 200.	Fully met.
Display web page title at the top of the page and http status response code.	Fully met. (Status code is displayed in log box at bottom of page)
Reload current page by resending http request to current url.	Fully met.
Create and edit a home page. This page should be loaded on startup.	Fully met.
Add a web page to a personal list of favourites and associate a name with the url of that favourite.	Fully met.
Favourites can be modified and deleted.	Fully met.
Favourites are loaded on browser startup.	Fully met.
User history is stored and loaded on startup.	Fully met.
Local history on each tab allows back and forward for web pages.	Fully met.
User can create multiple tabs and browse separate pages in each. Each tab contains information about its local state.	Fully met.
Browser-server communication separated from gui support using different threads. Different threads for each tab.	Not met. Browser does not support multi-threading. Http request is sent asynchronously but is not prevented from locking gui thread.
Menus and shortcuts to increase accessibility	Partially met: Return in text boxes to submit instead of button, ctrl-n opens new tab, ctrl-w closes current tab, ctrl-q closes the application. Attempted: ctrl-f to open favourites, ctrl-h to open history. Bugs detailed later.

3. Design Considerations

Rather than build my GUI inside a single windows form I used user controls to encapsulate the different functionality. The main browsing functionality would have to be instantiated multiple times to create multiple tabs. However some controls, for example the new tab button, should not be contained in each tab but just once at a higher level in the program. This saves memory by not duplicating unnecessary controls for each tab.

The user data to be stored was history, home page and favourites. Home page has been stored in the same file as favourites as it is only one address and does not warrant an additional file. An identifier is attached to each line to let the parser know whether it is the home address or a favourite it is reading. History was stored in a separate file as this may be quite large relative to the favourites list. The history file stores which urls the user has visited and the date and time they

visited them at. The history will only need to be loaded sometimes and including it in the same file as the favourites list would slow the loading of the favourites unnecessarily. Comma separated values(CSV) was the format chosen to store the history, home page and favourites. JSON, XML and a database were also considered but CSV was deemed the most suitable. JSON although similar to CSV did not offer any distinct advantage over it for the added complexity. JSON provides better support for hierarchical data (Krus, 2018) and large data sets (Datafiniti, 2018) but neither of these are appropriate. The users favourites will not likely contain a large number of entries and the history is not all loaded simultaneously. JSON is good for data exchange but this is not useful as the data is local to each user. This data will likely stay local as users will not want their personal favourites and history being available to anyone else. XML does not offer any advantage over CSV for the data in question. The added complexity in the parsing and writing would make the code less readable, the files take more space and would potentially slow loading times.

Storing the data in a database is not a viable solution. The data must be local to each user so that it can be accessed by whoever is using the program and creating and maintaining a database would be a lot of work. If the user data were to be stored in a remote database this would introduce issues with data protection aswell. The data was stored as ASCII characters instead of binary because it is only read and written to by this program and there are no privacy issues with users reading the source files as they are stored locally.

I chose to split history into local history and global history. Local history is used for the back and forward functionality on each tab whereas global history is list of pages accessed in all tabs and when they were accessed. The global history is written to a file and loaded on the browser's startup.

I created both a go button and go on enter key for the url text box. It is common for return or enter to be used to submit a completed text entry so it seemed natural to add this support. Enter support will streamline the usage for familiar users whereas the go button will make it clear to unfamiliar users how to operate the browser.

I considered creating a web page class for storing information about pages the user visited. This would store the url and any other desired data. Classes could then inherit from this base class to create a favourite class and a history class. The favourite class would additionally store a name and the history class would store the date and time. These classes could be made serializable so that they could be written and read from files for storing the user's data. Ultimately I decided this was overkill for current functionality required by the browser but may be a valid design configuration if more information about the pages was required.

I made several variables that belonged to the class that held each tab's information static. This was because these fields would be identical for all tabs and should be updated across them all simultaneously. For example the user favourites and overall history were both static collections. This allows whichever tab is active to add or remove from these easily while having the changes reflected and updated across all tabs.

An additional control was intended to be created for the favourites bar, as seen in Figure 2. This would be independent from the tabs as the favourites and history are identical for all tabs. It would improve performance and save resources by not reloading these controls for every tab. However this was not implemented as there were issues within the form when trying to display it and buttons were being created out of the visible window. Due to time constraints the code was not refactored and this feature never made it in to the program. It was deemed more important to implement as many features from the requirements as possible rather than refactor existing code.

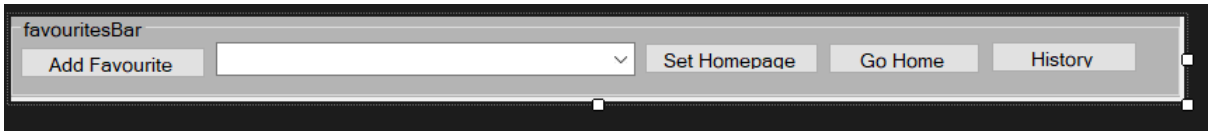


Figure 2: The intended favourites bar user control to be separated from the tabs.

At first it was intended to use a class library to store the user controls. However publishing the project it would not run. This was determined to be due to cyclic dependencies as the history browser user control required a reference to the top level program to open new tabs. Cyclic dependencies are poor practice as they eliminate the point of class libraries establishing a hierarchy within the program (Barow, 2018). At this point it was clear the classes belonged as part of the same program so they were merged together. This cyclic dependency is clearly visible in the class diagram in Appendix A.

4. User Guide

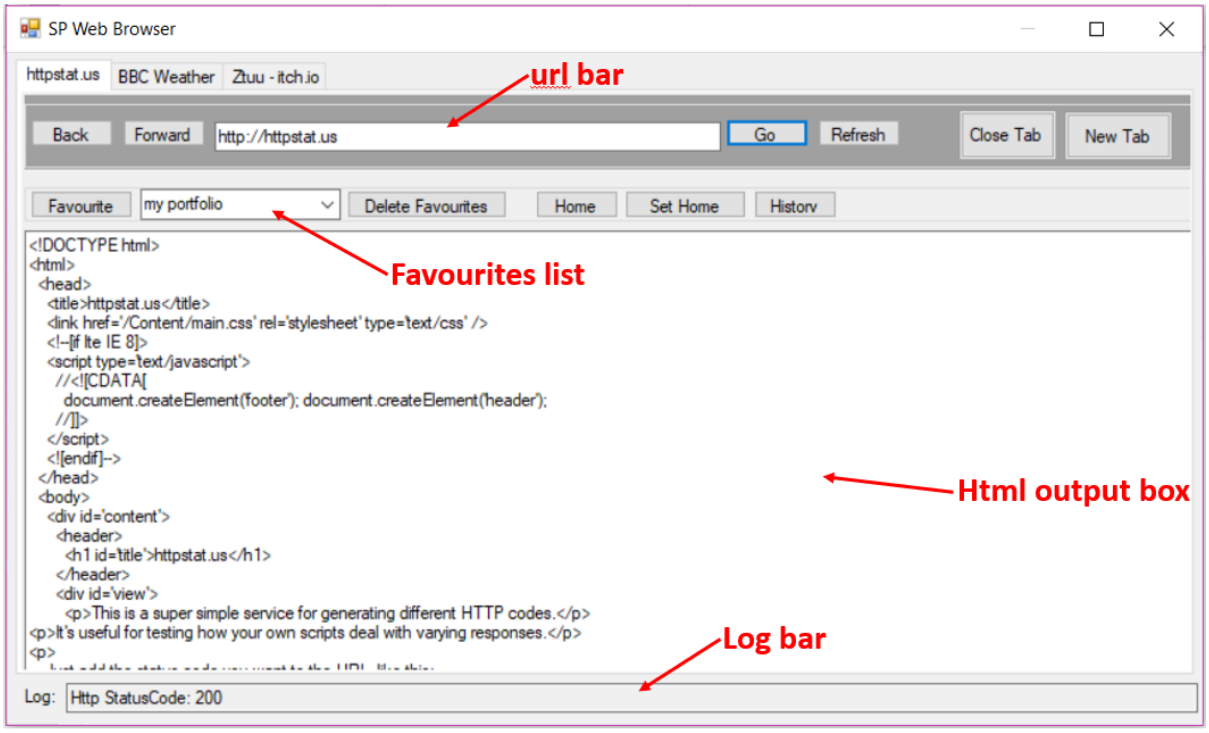


Figure 3: A partially annotated image of the browser. Names buttons have not been labelled

The url bar allows the user to enter a web address to search for. The user can input an address with or without “http://” or “https://”. If not it will be automatically added when a request is sent. The user can either click “go” or press enter to send a request for the current address.

The html output box shows the html response received from the last http request sent for this tab. It will also display some error information in conjunction with the log box if the request fails.

The refresh button will re-send an http request for the current page to reload it. The html output is cleared before re-sending the request to make it clear to the user the page is refreshing.

The tab name is updated to the title of the most recent web page accessed on that tab. If the last request sent was invalid the tab name is just “Error”.

The log bar is a single line textbox at the bottom of the screen used to display program messages to the user. Some errors will be displayed here but most commonly it is used to shown the http response code that returns when an http request is sent.

The tabs the user currently has open are displayed in a bar along the top of the screen. When the browser is first opened only one tab will open and direct to the home page. The add tab button can be used to create new tabs which will start on the home page by default. If a large number of tabs are opened arrows will appear allowing the user to horizontally scroll through their tabs. There is also a close tab button which will close the current tab. The close tab button will do nothing if there are no tabs open. Ctrl-n will open a new tab and ctrl-w will close current tab.

The back and forward buttons allow the user to browse through their recent history in that tab. Back will take the user to the previous web page they visited and forward will take them to the next more recent page if they have gone back. If the user hasn't gone back, forward will do nothing and if they are at the oldest page for that tab, back will do nothing. Back and forward support is local to each tab.

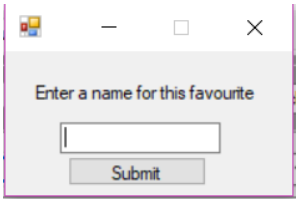


Figure 4: The name favourite dialog

Pressing the favourite button allows the user to add the current web page in the currently active tab to be stored as a favourite. When the button is pressed an additional dialog window will pop-up asking the user to input a name for this favourite to be stored under. If the user enters a non-empty string that contains no commas and presses submit, the favourite will be saved. If the dialog is closed the favourite will not be saved. If the user enters a name already used for another favourite the new entry will not be added and the log bar will display an error message informing the user they have entered a duplicate name and what that name is. The users favourites are written to a file every time they are changed and are loaded every time the browser is opened.

The users favourites are displayed in a drop down list. This will only show the name the user has assigned to this favourite, not the url itself. Clicking an entry in this list takes the user to the page associated with that favourite.

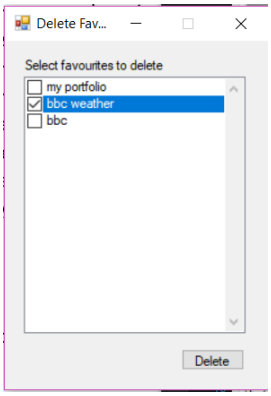


Figure 5: The favourite deletion dialog

The delete favourite button opens a new window where the user can remove saved favourites. A checked list shows all the favourites the user has stored, identified by the names given to them. The user can check as many entries as they like then press delete to remove those pages from their favourites. This will automatically close the dialog box and remove the favourites from the user's settings. This is reflected across all open tabs and saved data.

The home button will take the user to their saved home page. If the user is opening the browser for the first time or has not set up a home page the homepage will be set to the browser default. The set home button can be used to set the current page as the user's homepage. This will be saved to their settings and be loaded every time the browser is started. Only one page can be stored as the home page.

The history button will open a new window where the user can view their browsing history. This shows the urls the user has visited and the date and time when they were visited. This list is comprehensive of all uses of the browser and all tabs open. The history displays a set number of items per page and can be browsed backwards and forwards. Clicking an entry in the history will

open a new tab and direct it to that url. Websites will not be added to the history when using the back and forward buttons or the refresh button to avoid duplication.

The browser can be resized to be larger than default if the user drags the window or presses the maximise button. Making the window smaller is not advised as there will not be space to fit all controls on the screen.

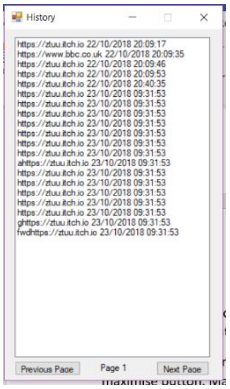


Figure 6: History viewing window

5. Developer Guide

The overall structure of the program is that the top level form contains a collection of tabs. It also contains the functionality to create new tabs and close tabs. A new tab is created by first making a new page in the TabPage object and then instantiating a new TabUserControl onto it. The TabUserControl class contains the functionality for each tab. The top level is also where the keyboard shortcuts are defined (Dey, 2018). A UML class diagram to demonstrate this structure has been created for the program and can be found in Appendix A. Button click functions generated by windows forms have been omitted to aid in clarity of the diagram.

Each tab contains information about its local state. There are also static fields that contain information common to all tabs. The additional dialog windows that open are the name favourite window, delete favourites window and the history window. These controls are instantiated on new forms when the corresponding buttons are clicked.

The history dialog is created with a reference to the top level of the browser and the list of history items. This is because it needs the ability to create a new tab when the user clicks a page to go to.

```
public HistoryBrowser(List<KeyValuePair<string,string>> historyIn, SPWebBrowser browser, Form newWindow)
```

The favourite name dialog is created with simply the url to be favourited. This is because the add favourites method is static because it affects all tabs, so a reference to the tab it was instantiated from is not required.

```
public FavouriteNameBox(string url, Form favNameWindow)
```

The favourite deletion dialog similarly does not need any references to a specific tab as remove favourites is also a static method.

```
public FavouriteDeletion(Form newWindow)
```

All of the controls that open in new windows are passed a reference to the window they are in so that they can close it when their task is complete.

The primary method in the TabUserControl class that is used most often is the GetRequest method. This asynchronous method is used for sending http requests and showing the resulting html. This method is used in conjunction with fields holding the current page, history and favourites to provide the user access to whichever page they need. It takes an optional Boolean to determine whether this url should be added to history.

```
/// <summary> Sends an http request to a given url
private async void GetRequest(string url, bool addToHistory = true)...
```

This is so that duplicates are not added to the history when a page is refreshed or a user travels back and forwards. Most of the functionality only occurs if the http request is successful, for example the current url being updated and the url being added to history. If the request fails then an error message is printed to the log box dependant on the reason for it failing. This will usually be due to http syntax which throws an HttpRequestException.

The favourites, home page and history are all stored in CSV files. These files are read and written to using StreamWriter and StreamReader. The saved data is loaded when the first tab is initialised. The files are written to when data is changed so that if the program is closed suddenly all saved data has been captured. When adding a new favourite or adding a new item to history the corresponding files will simply have the new addition appended to them as the existing data will not change. When deleting favourites or overwriting the homepage the files are rewritten to reflect these changes. Table 2 shows the format the user data is stored in within the CSV. The user has been prevented from entering a favourite name that contains a comma to ensure no errors result from this.

```
if (Favourites == null)
{
    Favourites = new Dictionary<string, string>();
    LoadFavourites();
}
else
{
    RefreshFavouritesBox(); //If the favourites are already in menu just refresh the display box to show them
}
```

```
//If this is the first tab assign the log box
if(logBox == null)
{
    logBox = browserLogBox;
}

//If this is the first tab to be created it will need to load history, others wont
if(GlobalHistory == null)
{
    LoadHistory();
}
```

Figure 7: Examples of some of the saved data being initialised only if this is the first tab

Table 2: Storage format of user data

Data being stored	Storage format
Favourite with name	“f,name,url”
Home page	“h,url”
History item	“url,date” (Note: date is stored as a string)

The back and forward methods that allow the user to move through a tabs history are created using delegates. (show code) This means the code is flexible and easy to maintain. In the future additional methods could be written to pass to this delegate, adding new functionality with ease. For example if a function incremented a value by 5 it could be used to skip forwards 5 pages. The user could enter a number and skip that many pages, or even skip to beginning or end.

```
private void MoveInHistory(ChangePage moveFunc)
{
    //Make sure the position in history is always valid
    Contract.Ensures(localHistoryPosition > -1);
    Contract.Ensures(localHistoryPosition < localHistory.Count());

    int newHistoryPosition = moveFunc(localHistoryPosition);
    if(newHistoryPosition > -1 && newHistoryPosition < localHistory.Count())
    {
        localHistoryPosition = newHistoryPosition;
        GetRequest(localHistory[localHistoryPosition], false);
    }
}
```

Figure 8: The method for moving backwards and forwards pages using delegates

Full docu-tag comments are present in the code for all classes. All classes are also commented to convey functionality and help with readability.

6. Testing

Known bugs:

- No multithreading.
- Favourite names and urls can't contain commas.
- New tabs load to slightly different size even though they use same method as first tab, as shown in Figure 9.
- Open history and favourites shortcuts don't work.
- Http request doesn't stop when tab is closed. It will not display anywhere but is still being processed.

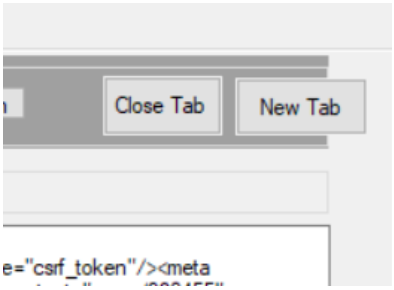


Figure 9: New tab size incorrect so doesn't line up with edge of window

Test: The url "https://httpstat.us/" was used to test http response code 200 and http error codes 400, 403 and 404 (Powell & Oddie, 2018).

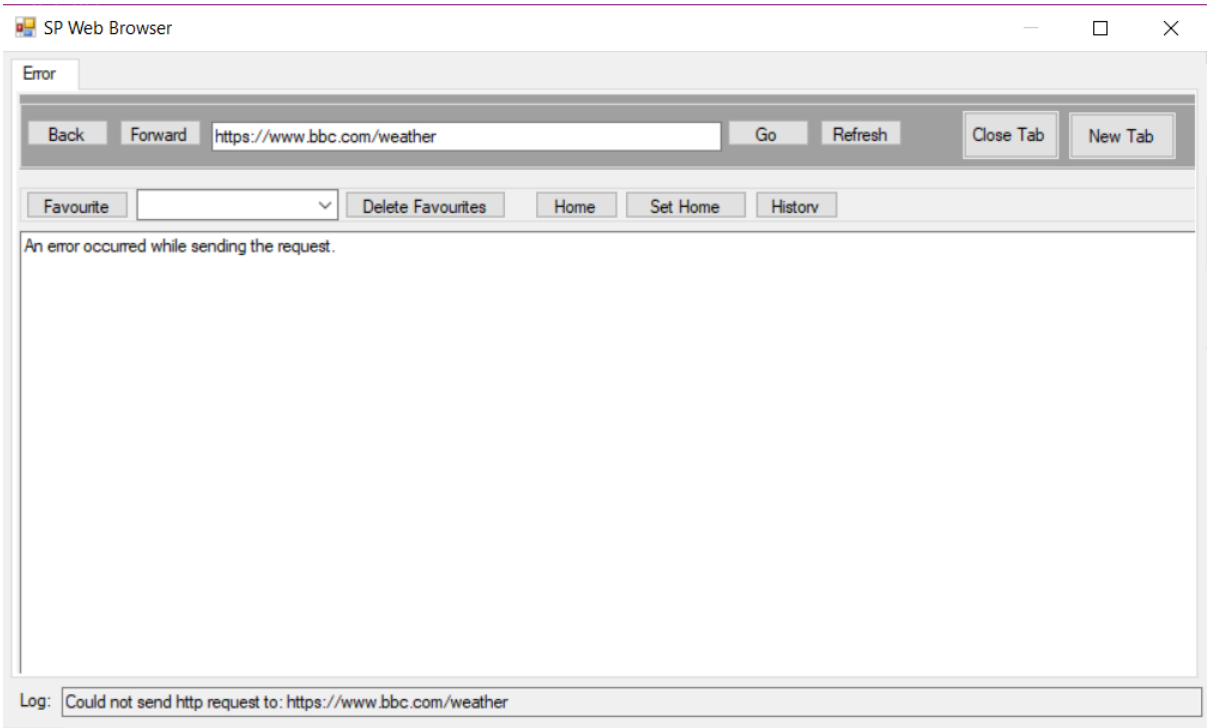
Expected Result: Correct status code displayed to user in log box for all cases.

Actual Result: As Expected.

Test: No internet connection.

Expected Result: Request fails and displays no connection error to user.

Actual Result: Request fails and error shows but not specific about connection.



Test: Edit saved pages directly to include errors and invalid entries.

Expected Result: Invalid entries ignored.

Actual Result: Dependant on case. Empty newline: ignored, empty name : all favourites redirected to this url, no url : request fails and generic error displayed to user, no tag : ignored and no http: http added and request successful.

```
h,https://www.bbc.com/weather
f,my portfolio,https://ztuu.itch.io
f,bbc weather,https://www.bbc.com/weather
f,bbc,https://www.bbc.com

f,,https://www.google.com
f,no url,
,no tag,https://www.yahoo.com
f,no http, www.bbc.co.uk
```

Figure 10: Some of the invalid saved data used for testing. Not all necessarily tested simultaneously.

Test: Open new tab which loads home page, change home page on first tab. Open another new tab to home page.

Expected Result: Second tab opens to new home page.

Actual Result: As expected.

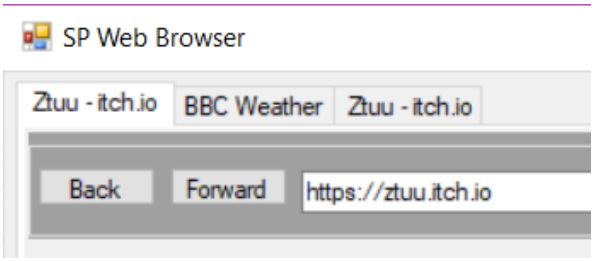


Figure 11: Home page testing

Test: Saved pages file contains only favourites and no home page.

Expected Result: Default home page loaded.

Actual Result: As expected.

Test: No saved page or history files.

Expected Result: Files created when first call to write to them happens. Favourites and history initialised empty, homepage uses default address.

Actual Result: As expected.

Test: Go back when at oldest page or forward when at most recent page.

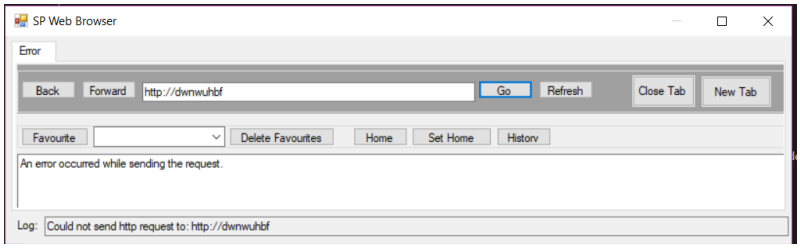
Expected Result: Stay on current page and do nothing.

Actual Result: As expected.

Test: Enter invalid url syntax.

Expected Result: Request fails and displays invalid url error to user.

Actual Result: Request fails and error shows but not specific about url syntax.



Test: Try and close tab when no tabs open.

Expected Result: Nothing happens.

Actual Result: As expected and log box displays “No tabs to close!”.

Test: Search with empty url.

Expected Result: Invalid url error.

Actual Result: As expected.

7. Reflections on programming language and implementations

This project was well suited to an object oriented approach. Each tab could be created as a separate instance of one object. This object contained some local information (like current page) and some static information common to all instances(like favourites). A hierarchy was established where the top level browser object contained a collection of the tab objects. This allowed easy control of making new tabs and closing tabs. This kept the code readable as functionality was contained where it was relevant. The top level program was short as it only had to manage the tabs, and the user control that contained all the tab functionality was longer. New windows that were only used infrequently (like deleting favourites) could be captured in their own object and only instantiated when needed. This helped prevent unnecessary resource usage by not loading these when not required.

The windows forms library made it easy to quickly prototype a gui and saved a lot of time writing the code to create the gui. Developing the gui in a visual interface also made the process easier than hard coding numbers for positions and sizes and then testing and refining them. User controls were a useful structure for creating this program. By creating a user control a set of form controls could be grouped together as an object. As discussed this object oriented approach was well suited to this program.

The dictionary data structure was used to store the favourites. This stores the data in key value pairs. A string was used for both the key and the value, where the key corresponded to the name the user assigned to the favourite and the value was the url. Dictionaries do not allow duplicate keys but do allow duplicate values. This matches the desired functionality that users cannot create two favourites with the same name but could favourite the same page twice. Duplicate names would make it impossible to distinguish in the interface how the favourites differed as url is not displayed. Dictionaries are not ordered (Microsoft, 2018) which is no problem as there is no ordering required for the favourites.

Global history required the storing of key value pairs similarly to favourites. However here a list of key value pairs was used instead of a dictionary. Although both are collections of key value pairs it was important to retain the order for the history. This is because history should be displayed to the user in order. As date is stored with the history items they could theoretically be stored without order then sorted by date. However due to the likely large number of history items sorting like this would be very inefficient and may require the user to wait a long time for the history to load. The performance would of course depend on the sorting algorithm but will inherently be slower than loading ordered data as it has to be loaded and sorted. Key value pairs with two strings were used to store the url and time the page was accessed. The date and time do not need to be manipulated, they are just displayed to the user. This means casting them to a string is not a problem. Another advantage of a list for the history, over dictionaries is that duplicate keys can be stored. We want the

history to store information about all the times a user has accessed a page, not just the most recent. As the url is the key here a dictionary would only allow storing one visit to a page.

Regular expressions were used several times in the program, for example to parse the page title from the html returned for a request. This was done by making use of the System.Text.RegularExpressions library (Microsoft, 2018). This allowed this searching to be done in just a few lines of code making it easy to read. The regular expressions were compiled at initial run time as they are used many times while the program runs. Using compiled regular expressions also yields better performance (Allen, 2018). It would be inefficient to compile them every time they were used.

The http library was used for the sending and receiving of http request functionality (Microsoft, 2018). This library nicely encapsulates these operations making the method to send the user's requests short and readable. The methods for sending an http request and retrieving the html can be asynchronous which is necessary as these operations may take some time.

The streamwriter and streamreader classes from the System.IO library weres used for writing the saved data to files. The streamreader's ReadLine method was used to parse the saved pages line by line. This was important as each line contained one stored page which needed to be interpreted one at a time to determine if it was the home page or a favourite. The streamwriter was used in two cases. In the case of a new user or data wipe it was creating a new file and writing the data to it. In the case of adding favourites the streamwriter was just used to append the new favourite to the existing file. This functionality was advantageous as it meant the entire file did not need to be rewritten every time a favourite was added. The streamreader and streamwriter classes worked well in conjunction with the CSV storage format to create simple, readable code. This means it is easy for future developers to understand and maintain or change.

The variety of access modifiers in C# were used to control how fields and methods were accessed. Private was used were the field was not required by anything outside of the class, for example local history of a tab. Static was used for several fields that belonged to the tab object. This means the field belonged to the class rather than any one instance. These values, for example the users favourites, are common to all tabs and should never differ between instances. This was used in conjunction with private/public access modifiers. For example add favourite is a public and static method. This allows an entity outside of the class to add favourites to the users saved data without a reference to an individual instance of that class.

The public modifier was not used on its own for a variable as it is bad practice to make fields public. This is because access to said fields is not controlled, anywhere in the program could access and edit the field. Some fields were declared as public in conjunction with the auto property functionality in C#. For example a static list of all tabs currently open was stored within the tab class. Every time a tab is opened it adds itself to this list so that there is a single reference that can be used to loop through and apply an operation to every tab.

```
//static reference to all current tabs. Can be used to loop through and apply operation to all tabs
public static List<TabUserControl> AllTabs { get; private set; }
```

This is declared as public in conjunction with an auto property that defines a public getter and a private setter. This was done so that only new tabs can add themselves to the list, no external classes can affect it. However external classes can get access to the list of all open tabs through the public getter. This conjunction of static, public and auto properties made it easy to achieve the desired functionality of this field.

The readonly modifier was used for values that should not be changed during runtime. For example the filenames were stored as readonly. The file names should not change while the program is running or this could introduce issues with continuity of data storage. Storing the filenames as readonly strings allows the developer to change the storage files in the code if necessary but ensures this will not change when the program is in use. The readonly modifier allows the developer to place this constraint on certain fields which protects the program from unintentional behaviour and is therefore very useful.

An optional parameter was used in the method that sent the http requests to capture whether that url should be saved to the history or not. Optional parameters mean the user does not have to pass an argument to that parameter in order for the method to execute (Microsoft, 2018). Optional parameters require a default value that will be used if no argument is passed. This is good for capturing if the design intent is for there to be a default value. In this program the default case is that a web page should always be added to the history. In the rare circumstances where a web page is not to be added to the history, for example when refreshing, the optional parameter can be used to easily specify this. Saving the developer entering the default value almost every time they use the method makes the code more readable and allows a developer reading the code to immediately see the cases that go against the default.

The use of delegates for history has been described in the developer guide.

Inheritance was used for the user controls. These inherited from the UserControl superclass which provides them with the gui interaction support and allows them to be developed like a windows form. This made developing the controls fast and efficient as this superclass handled a lot of the menial work and allowed the developer to focus on implementing the browser functionality. This also makes each user control's class easy to read as it is not full of code to support the gui, it just holds that class' intended functionality.

Generics have been used when creating the lists and dictionaries present in the program. Generics highly increase code reusability and can streamline the development process. For example in this situation the desired collection of objects could be very quickly created as both lists and generics support generic parameters. As long as generics are properly constrained so that invalid objects cannot be used they are a powerful language feature.

Code contracts were used to help enforce important constraints and aid testing. For example when creating a new tab they check the controls are being initialised correctly. When using static checking the code contracts can be relied on to check for validations of these conditions even before runtime. They also make the intention of the code clear to anyone reading, in this case it's immediately obvious neither of the parameters can be null.

```
private void NewTab(TabUserControl newTabControl, TabPage newTab)
{
    Contract.Requires(newTab != null);
    Contract.Requires(newTabControl != null);
```

8. Conclusions

I was most proud of the encapsulation and abstraction of the program I created. In particular the low level of functionality in the top level class. The top level simply managed the tabs and each tab contained all of its own functionality. I think this is good object-oriented design.

Along with fixing the known bugs I previously mentioned there were some design features I would do differently. Although the CSV file format worked well for this program, I think it limits future

expansion. This format will perform worse with increased amounts of data and also has some flaws. The main one is that commas cannot be stored in favourite names. Although uncommon and considered bad practice, urls may also contain commas which would cause problems when stored in the current program.

The history currently displays in chronological order. I would rather it show in reverse order so that the most recent pages are first as the user is more likely to be concerned with recent history. I of course would've like to get multi-threading working. I tried sending the http requests on a new thread but was having issues communicating the returned data with the text boxes on the gui thread.

Overall I was happy with the program. It met almost all the requirements, was neat and well laid out and encompassed some good design practices behind the scenes. I tried to consider best practice and good quality code while I designed the browser but there are of course also areas with room for improvement.

9. References

Allen, S., 2018. *Regex performance.* [Online]
Available at: https://www.dotnetperls.com/regex-performance

Barow, S., 2018. *Why Cyclic Dependencies are Bad.* [Online]
Available at: https://lattix.com/blog/2017/07/26/why-cyclic-dependencies-are-bad

Datafiniti, 2018. *4 Reasons You Should Use JSON Instead of CSV.* [Online]
Available at: https://blog.datafiniti.co/4-reasons-you-should-use-json-instead-of-csv-2cac362f1943

Dey, S., 2018. *C#: Implementing Keyboard Shortcuts In A Windows Form Application.* [Online]
Available at: https://social.technet.microsoft.com/wiki/contents/articles/50920.c-implementing-keyboard-shortcuts-in-a-windows-form-application.aspx

draw.io, 2018. *draw.io.* [Online]
Available at: draw.io

Krus, A., 2018. *CSV, JSON or OLAP CUBE.* [Online]
Available at: https://www.flexmonster.com/blog/blog-perform-faster/

Microsoft, 2018. *Dictionary<TKey,TValue> Class.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?redirectedfrom=MSDN&view=netframework-4.7.2

Microsoft, 2018. *Named and Optional Arguments (C# Programming Guide).* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/named-and-optional-arguments

Microsoft, 2018. *System.Net.Http Namespace.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.net.http?view=netframework-4.7.2

Microsoft, 2018. *System.Text.RegularExpressions Namespace.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions?view=netframework-4.7.2

Perry, S., 2018. *Object-oriented programming concepts and principles.* [Online]
Available at: https://www.ibm.com/developerworks/library/j-perry-object-oriented-programming-concepts-and-principles/index.html

Powell, A. & Oddie, T., 2018. *httpstat.us.* [Online]
Available at: https://httpstat.us/

10. Appendices

Appendix A

UML Class diagram created using draw.io (draw.io, 2018)