Assessed Coursework 1 — Developing a Simple Web Browser

F20SC Industrial Programming

Name: Lachlan Woods (H00223249)

Due Date: Thursday 25th of October 2018

Online Documentation: https://lachlanwoods.github.io/annotated.html

Introduction

The purpose of this coursework was to develop a multithreaded web browser in C#. A simple GUI was to be constructed using the Windows Forms library.

Advanced programming concepts were used in my implementation of the web browser. Great care was taken to ensure object orientated best practices were maintained throughout the entire development of the web browser. Classes have been designed to ensure that they have high cohesion. Each class has a single well-defined purpose. Classes have also been designed to have low coupling, so that changes can be made to a single class, with minimal impact to other classes. This has been achieved through the use of private variables, with globally accessible properties and methods.

My implementation has been designed with code readability, maintenance and extendibility in mind. All code has been commented, and every method and class has XML documentation comments. These comments have been used to generate a documentation website, which gives information of all publicly accessible methods.

I am very pleased with the final web browser that I developed. I am confident that I have met all requirements, and have added additional features, which are discussed under the additional functionality section of this report. I have used advanced concepts such as code contracts, anonymous and lambda expressions, and multiple different multithreading techniques. Exceptions have been used to mitigate the impact of any errors thrown by the application. Advanced concepts used have been outlined in further detail under the design considerations section of this report.

This coursework was of particular interest to me, as it was the first assessed coursework using C# that I have received. It is also the first time I have been tasked with creating a GUI for a moderate sized program by the University. GUI programming is a concept that I believe is very important, particularly for developing applications for industrial purposes. This coursework gave me experience in developing an application with a GUI under a strict timeframe, which mimics the working conditions I expect to find when working as a software engineer outside of university.

It was interesting to see the differences the use of C# made to the development process compared to Java, which is the language most commonly used in previous years of study. I have noticed that the .net framework is targeted towards the Windows operating system, which means that the GUI developed using the framework has a Windows like aesthetic by default, which is very convenient, and reduces the development time needed for styling the interface, compared to Java's swing library. A downside is that .net applications are not as portable as applications developed in Java, which run in the JVM, and therefore support cross-platform execution. It is my understanding that a C# application would need to be developed using the mono framework, rather than .net, to be executable on Linux and Mac machines, which is something that would need to be taken into consideration when choosing a language.

Assumptions

I made the following assumptions about this project before beginning development of the web browser:

1. The primary purpose of this coursework was the use of advanced C# constructs and code quality, rather than producing a GUI with good user experience principles. Therefore, priority was put towards the design decisions of my code, rather than the design of my user interface. Despite this, effort was still made to produce an easy to use, and visually appealing interface.
2. Some websites may require special conditions to receive the correct HTTP response, such as support for cookies or JavaScript. Websites may also incorporate redirects, which may impact the response received by my application. Therefore, I have assumed HTTP responses will often contain error responses. As such, effort was made to catch as many exceptions as possible to ensure that the web application handles HTTP errors appropriately, rather than crashing. I have also assumed that we will be marked on how our application handles HTTP errors, rather than whether or not our application receives the correct HTTP response from all websites.
3. When the user launches the web browser for the very first time, there will be no homepage set. In this case, I have assumed that a default webpage will be used. I have set www.google.com as the default homepage.

Requirements Checklist

Requirement	**Achieved**	**Comment**
Sending HTTP request messages	Yes	HTTP requests are sent using the System.Net library. Requests are of type HttpWebRequest, and are made using the WebRequest.Create(url) method. My program performs requests in background workers, which are starter by the SearchInNewThread() method. Each tab handles its own HTTP request.
Receiving HTTP response messages	Yes	HTTP responses are of type HttpWebResponse. The response code of a request is accessed using the HttpStatusCode class. Errors in a HTTP request are represented by the WebException class, and are handled using a try-catch statement.
Display of HTTP Requests	Yes	The HTTP response code and title of a webpage (when available) is displayed as the main window title when a response is received. The HTML response, or error message is displayed by the web browser.
Refresh webpage	Yes	A webpage can be refreshed by clicking the refresh button, or pressing control+r. This action will resend a HTTP request for the current page.
Home Page	Yes	A home page can be set by right clicking on a webpage and pressing the “Set as home page” option from the contextual menu, or by pressing control+h. A prompt will be displayed, allowing the user to edit the url of their homepage. By default the url displayed in the prompt will be that of the current page. The homepage will be loaded when a new tab is opened. The homepage is saved between sessions using a Properties.Settings value.
Favourites	Yes	Favourites can be saved by right clicking on a webpage and clicking the “Add to favourites” option from the contextual menu, or by pressing control+f. Favourites can be edited, by changing the name and url associated with the favourite entry. This can be done by opening the favourites list and clicking the edit button. Favourites can also be removed by clicking on the “X” button on a favourite entry in the favourites list.

		Favourites are saved to an XML file and are loaded at start-up. Clicking on a favourite entry in the favourites list will navigate the current tab to the url specified by the favourite entry.
History	Yes	My implementation of history has been achieved in two different ways: A list of global history is saved and loaded from an XML file. All webpages visited by any tab are saved to global history. Global history can be viewed in a list by clicking on the history button. Clicking on a global history entry will redirect the current tab to the url of the selected history entry. A global history entry can be removed from history by clicking on the "X" button in the history list. Webpages visited by one tab can be accessed in a different tab through the global history list. Local history has been implemented for each individual tab as a linked list. This history is used to navigate back and forwards to previously visited webpages. Only the webpages visited by a specific tab will be recorded in the history local to that tab. For every webpage that is visited, the webpage is added to global history, which is persistent between sessions, and the local history of the tab that was used to navigate to the webpage.
Multithreading	Yes	Multithreading has been used for three different actions that would have potential to block GUI actions if performed on a single thread: 1) Each tab has its own Background Worker. All browser-server communication is performed by the background worker, which performs HTTP requests in a new thread. The use of a background worker ensures that multiple HTTP requests can be performed at the same time by different tabs, without blocking GUI interactions. My reasons for choosing a background worker over other multithreading techniques is outlined in the Design Considerations section of this report. 2) History and favourites are saved in an XML file. On start-up these two files are read simultaneously in separate threads using the System.Threading library. A Thread.Join statement is used to synchronise the completion of these two threads, so that the user can only interact with the web browser GUI once both files have been loaded. 3) History and favourites are written to their respective files often throughout the running of the web browser. To prevent this action blocking the main thread, writes are performed in a new thread, again using the System.Threading library. Locks are used when reading and writing to files to prevent race conditions.

Additional Functionality

In addition to the required functionality, I have added the following features to my web browser:

1) **Rendered HTML display:** An option to display rendered HTML instead of HTML code has been added. The user may toggle between the rendered HTML and code view by clicking a button. Rendering of HTML has been achieved by using an external library[1]. I chose to use this library as it does not make use of the Web Browser class, which was forbidden for this coursework. The library also provides a HtmlPanel Windows Form control, which made it very easy to integrate the library into my existing application, without any changes to pre-existing code. I was able to create a new HtmlPanel control, and access it from code as if it was a default Windows Form control. The HtmlPanel supports rendering of HTML 4.01 and CSS level 2.
2) **Automatic tab resizing:** The width of tabs are updated whenever a new tab is opened, or the web browser window is resized to ensure that all tabs fit within the web browser view, without needing horizontal scroll controls. This feature was added to improve user experience. Tab widths are calculated based on the number of opened tabs, and the width of the web browser. Callbacks were assigned to listen for window resize events.
3) **Sliding side panels:** I extended the Panel class to implement sliding side panels. My extended panels class adds methods to slide the panel in and out from the side of the web browser window with an animation. These sliding panels are used to display the history and favourite webpage list. Each sliding panel object creates a Timer control, which is used to increment or decrement the width of the panel over time, which is used to create a sliding effect when the panel is opened or closed.
4) **Url fixer:** A CorrectUrl(string input) method has been created in the Tab class, which appends "http://" to any search query that does not begin with "http://" or "https://". All HTTP requests must begin with this prefix to be valid. This addition also allows for HTTP requests to be performed by entering an IP address into the search bar. Therefore, simply entering “216.58.210.46” into the search bar will perform a request to Google.
5) **Remove from history:** Webpages may be removed from history by clicking on an “X” button, which can be found beside each history entry while viewing the history list. Removing a webpage from history will be reflected in all tabs, and between sessions.
6) **Closing the final tab:** Closing all tabs will close the entire web browser. This was done to improve user experience. I observed this functionality in Google Chrome, and replicated it in my web browser.
7) **Add to history flag:** The NavigateToPage(Webpage page, bool addToHistory) method, which is used to make all HTTP requests takes a Boolean value indicating whether to add a search to history. This is useful, as not all requests should be added to history. For example, I decided that refreshing a page or navigating back and forwards in history should not re-add the page to the global history. This decision was made to prevent unnecessary duplicates in the history list.

[1] https://github.com/ArthurHub/HTML-Renderer

Design Considerations

Class design:

The use of a singleton pattern for the BrowserManager class is a defining design decision of my program. The BrowserManager class contains many methods that are useful for multiple different classes, such as reading and writing to files, accessing history and favourites, and accessing the reference to the current tab and main web browser objects.

A BrowserManager object is created on start-up and stored as a private static field, that is accessible by a global static property. Checks are performed to ensure that only one BrowserManager object can be created.

The use of a singleton pattern reduces code duplication, as common methods can be written once in the BrowserManager class and be called by any other object. The singleton pattern also reduces the need to pass references to controls around between objects. Important controls, such as the current tab and main window control can be accessed from the BrowserManager.

In all classes, only private variables have been defined. When required, these variables can be accessed through getter/setter methods. These methods are usefully implemented as properties. The decision to use private variables, with globally accessible properties reduces coupling, and aids in maintenance and future development.

Data Structures:

A linked list has been used to store the local history of each tab. The linked list data structure is part of the System.Collections.Generic library. Generics were used to create a linked list of type Webpage.

I chose to use a linked list to represent a tab's local history, as a pointer to the currently viewed webpage can be represented by a LinkedListNode object. All nodes before the current node represent the pages that the user can go back to by pressing the back button. All pages after the current node represent the pages that the user can go forward to by pressing the forward button. We can easily check if there are pages before or after the current page, by using the LinkedListNode.Next and LinkedListNode.Previous methods. As the user navigates around through history, the current node object can be updated to point to the current position in the linked list, without having to shuffle nodes around in the list. This is an efficient way to represent the state of history.

Global history and favourites have been implemented as Lists (System.Collections.Generic). Generics have been used so that the history list stores HistoryEntry objects, and the favourites list stores FavouriteEntry objects. The HistoryEntry and FavouriteEntry classes have been implemented by extending the UserControl class. These two classes store variables such as the webpage url, access date, and website name corresponding to the entry. The decision to extend the UserControl class was made so that each entry could be displayed as a visual item in the history or favourites list. Each class can then make use of callbacks, so that variables can be updated based on user interaction with the GUI. This design decision removes the need to maintain a link between a history or favourite control displayed in the GUI with an object stored in a list.

GUI Design:

I have aimed for a minimalistic user interface, to keep in-line with modern design principles. A "flat" design has been used for all GUI elements. Buttons are displayed with a simple image, instead of text. All buttons have been assigned tooltip text, which is displayed when the mouse is hovered over a button. The tooltip gives a description of the button's purpose.

In addition to buttons, hotkeys have been assigned to actions. These hotkeys are displayed in tooltip text. Hotkeys remove the need to click on buttons, and can be used by advanced users to speed up actions.

Custom sliding side panels have been created to replace dropdown menu lists. These sliding menus have animations when opening and closing and allow items to be displayed in a scrollable list. These sliding menus provide a better user experience than dropdown menus, as they provide a larger area to display elements, and they do not loose focus if the user miss clicks.

Custom tabs have been created by extending the TabPage class. My custom tabs automatically resize to fit in the browser window without needing horizontal scroll bars. This feature was developed with the aim of improving user experience, and is a technique used by larger web browsers such as Chrome and Firefox. Close buttons and the url of the current webpage displayed by the tab are presented on the tab entry, making it easy to navigate between tabs and close unused tabs.

Advanced Language Constructs:

Code contracts were used to perform runtime checking during static analysis. Preconditions were frequently used to ensure that objects required by a method were not null, for example:

```
Contract.Requires(favouritesPanel != null && historyPanel != null);
```

Code contracts were useful for testing, as errors were given if a method was to be called before a requirement was met, without needing to compile the program.

Inheritance was used by many of my classes. Typically, my classes inherited from Windows Forms control classes, so that they could be displayed as a GUI element. My implementations of Tab, SlidingPanel, FavouriteEntry and HistoryEntry all inherit from a Windows Forms class.

Using inheritance allowed me to call methods from base classes on my own objects, as if they were a Windows Form control. This allowed me to call the Controls.Add method to add my own custom controls to Windows Form panels.

The use of inheritance was very important for my implementation of tabs. By inheriting the TabPage class, and defining my own variables and methods, I was able to create tabs that look like the default Windows Form tabs, while also adding methods to perform tab specific multithreaded HTTP requests and storage of local history.

Overriding the ToString() method was conducted in the WebPage and FavouriteEntry classes so that the classes returned an XML object, rather than a string. This was done so that web pages and favourites could be written to an XML file. By default, the ToString() method would return an object name, which is not useful for this project. My overridden method presents the value of variables as XML attribute values, which can then be loaded and used to re-create objects in different sessions.

The **using keyword** was used whenever an IDisposable object was created. The using keyword creates a block indicating the scope of an IDisposable object. When the block of code is finished, the IDisposable object will be disposed. This ensures that the object is always correctly closed, which helps prevent memory leaks. The use of the using keywork is prevalent when fetching and reading HTTP requests.

Exceptions were handled whenever an error could be thrown. This was accomplished using try-catch statements. These statements help to mitigate the impact of errors, by executing code defined in the catch statement, rather than the program crashing.

Lambda expressions were frequently used to define code that should be run on a button click, or a new thread. Lambda expressions allowed for anonymous methods to be defined with a local scope. This was very useful for reading files in a new thread, as the new thread could run a lambda expression, which would read a file and update a variable in the parent method on completion.

Other Key Decisions:

Multithreading of client-server communication was performed using **background workers** (System.ComponentModel). Each tab manages its own background worker, which is used to perform HTTP requests.

I chose to use background workers rather than standard threads (System.Threading), as the result of a background worker is saved in a variable that is accessible in the background worker's RunWorkerCompletedEventArgs argument in the callback method used when work is completed. This made it easy to access a HTTP response. In comparison, the method ran by a new thread cannot return a value, unless an anonymous function is used within another method, and the thread writes to a variable within the scope of the parent method. I did not want to have client-server communication performed as an anonymous function, as it is a major part of the application. Therefore, I chose to implement the client-server communication as a method that was treated as the background worker's "DoWork" method.

Background workers can also be queried using the IsBusy() method, and cancelled by calling CancelAsync(). These methods make background workers appropriate for web browsing, as it is important to know when a HTTP request is in progress. The user may also wish to cancel a request if it is taking too long.

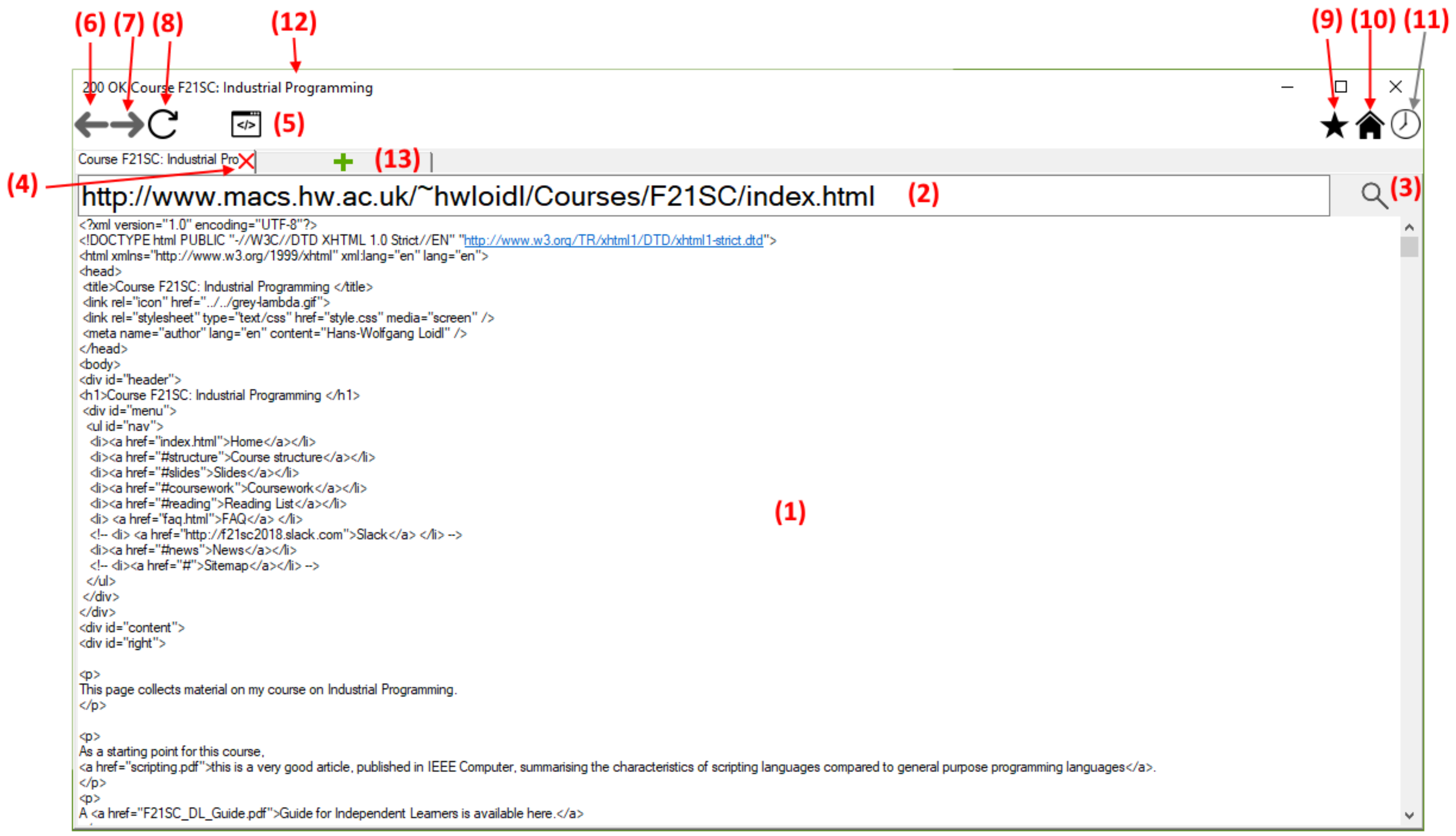
History and favourites are saved between sessions using **XML files**. I chose to save persistent data in XML format, as it is a widely used format for data representation. This means that I could easily load data, and select individual attributes using the System.Xml.Linq library. If a custom format had been used to save data, I would have needed to write my own function to search and read the important information from files. Since XML is a widely used format, third party tools are available to process and display history and favourite items created by my web browser.

JSON format could have been used to save data, however I decided to use XML instead due to the availability of the inbuilt System.Xml.Linq library, which provided all functionality I needed.

Files were written to and read in separate **threads** to avoid blocking GUI interactions during reads and writes. Both the history and favourites files are loaded simultaneously in separate threads on start-up. A join statement is used to synchronize the completion of these threads, which ensures that the required files are fully loaded before the user can interact with the browser. **Locks** are used to ensure that a single file is not written to or read by more than one thread at the same time, as this could create a race condition.

XML documentation comments have been written for all methods and classes. These comments have been used to automatically generate documentation files using Doxygen. These comments help improve code readability and maintainability.

User Guide


(6)
(7)
(8)
(12)
(9)
(10)
(11)
200 OK Course F21SC: Industrial Programming
(5)
Course F21SC: Industrial Pro
(13)
(4)
(2)
(3)
(1)

http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/index.html

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <title>Course F21SC: Industrial Programming </title>
 <link rel="icon" href="../../grey-lambda.gif">
 <link rel="stylesheet" type="text/css" href="style.css" media="screen" />
 <meta name="author" lang="en" content="Hans-Wolfgang Loidl" />
</head>
<body>
<div id="header">
<h1>Course F21SC: Industrial Programming </h1>
 <div id="menu">
  <ul id="nav">
   <li><a href="index.html">Home</a></li>
   <li><a href="#structure">Course structure</a></li>
   <li><a href="#slides">Slides</a></li>
   <li><a href="#coursework">Coursework</a></li>
   <li><a href="#reading">Reading List</a></li>
   <li> <a href="faq.html">FAQ</a> </li>
   <!-- <li> <a href="http://f21sc2018.slack.com">Slack</a> </li> -->
   <li><a href="#news">News</a></li>
   <!-- <li><a href="#">Sitemap</a></li> -->
  </ul>
 </div>
</div>
<div id="content">
<div id="right">

<p>
This page collects material on my course on Industrial Programming.
</p>

<p>
As a starting point for this course,
<a href="scripting.pdf">this is a very good article, published in IEEE Computer, summarising the characteristics of scripting languages compared to general purpose programming languages</a>.
</p>
<p>
A <a href="F21SC_DL_Guide.pdf">Guide for Independent Learners is available here.</a>
```

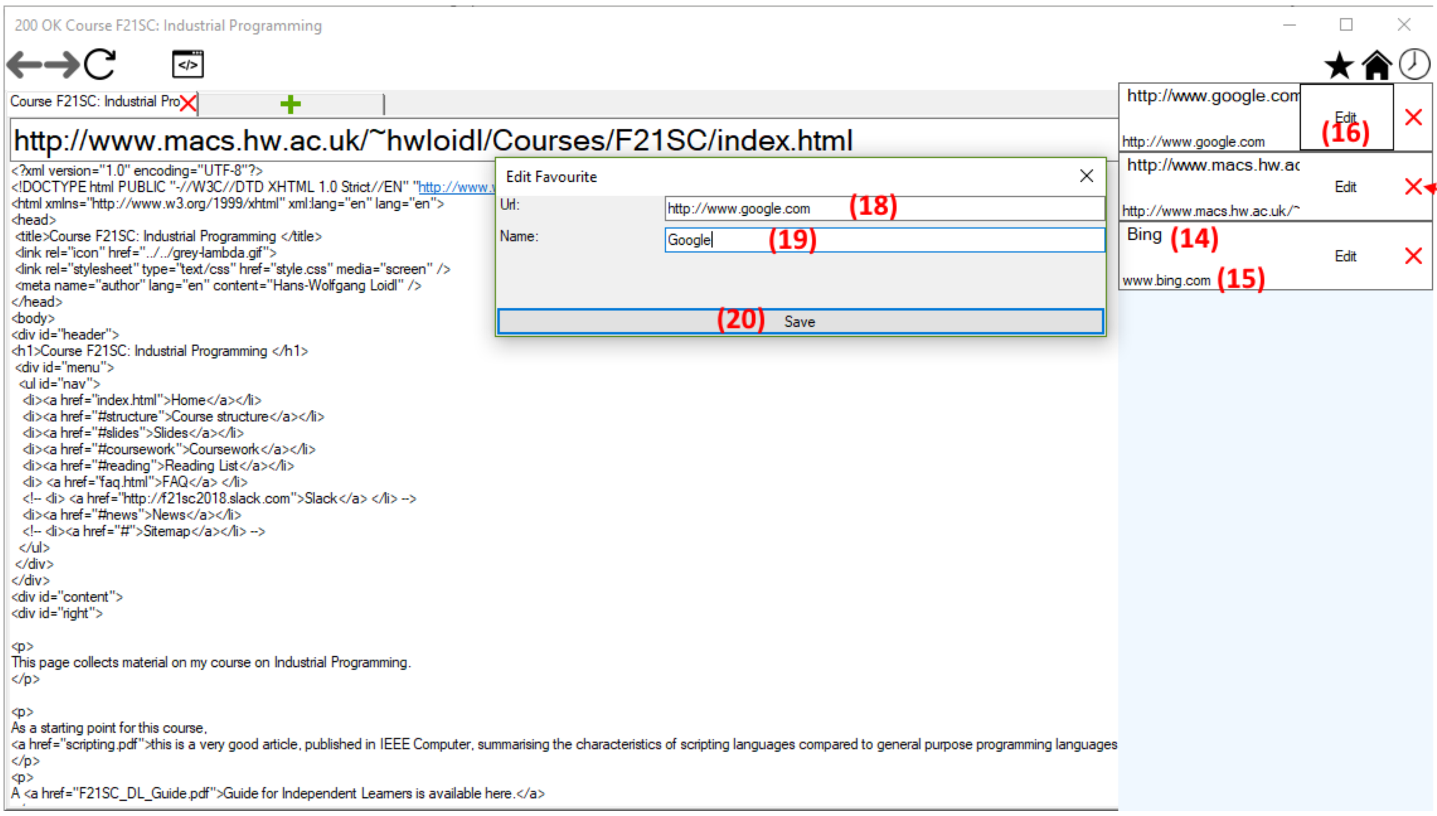
(1) Web display: Displays the response of an HTTP request. This may be HTML code or a Rendered HTML page based on the state of button (5).
(2) Search bar: Enter a URL here and press the enter key or button (3) to make a HTTP request.
(3) Search button: Click this button to make a HTTP request using the url specified in the search bar (2).
(4) Close tab: Click this button to close a tab. If you close all tabs, then the web browser will exit.
(5) Toggle view: Click this button to toggle the web display (1) between rendered HTML and code view.
(6) Back button: When there are previous pages that you have visited in the current tab, this button will be enabled. Clicking this button will navigate the current tab to the previous page.
(7) Forward button: When there are previous pages that you have visited in the current tab, and navigated back from, this button will be enabled. Clicking this button will navigate the current tab forward to the pages previously visited.
(8) Refresh: Clicking this button will re-send a HTTP request to the current webpage, which will refresh the page.
(9) View Favourites: Clicking this button will open a side panel displaying all webpages that you have added to favourites. Click the button again to close the side panel. You may add a webpage to your favourites by right clicking on the web display (1) and clicking “Add to favourites”, or pressing control+f.

(10) Home: Click this button to navigate to your homepage. You may set your homepage by right clicking on the web display (1) and clicking “Set as homepage” in the contextual menu (or pressing control+h. You will be prompted to enter the url of your homepage. By default, this is the url of the page you are currently viewing.

(11) History: Clicking this button will open a side panel displaying all webpages that you have previously visited. Click the button again to close the side panel.

(12) Title bar: The status returned from a HTTP request will be displayed here. When available, the title of a webpage will also be displayed here.

(13) New tab: Click on the final tab in the tab bar to create a new tab. The new tab will automatically go to your homepage.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
 <title>Course F21SC: Industrial Programming </title>
 <link rel="icon" href="../../grey-lambda.gif">
 <link rel="stylesheet" type="text/css" href="style.css" media="screen" />
 <meta name="author" lang="en" content="Hans-Wolfgang Loidl" />
</head>
<body>
<div id="header">
<h1>Course F21SC: Industrial Programming </h1>
 <div id="menu">
  <ul id="nav">
   <li><a href="index.html">Home</a></li>
   <li><a href="#structure">Course structure</a></li>
   <li><a href="#slides">Slides</a></li>
   <li><a href="#coursework">Coursework</a></li>
   <li><a href="#reading">Reading List</a></li>
   <li> <a href="faq.html">FAQ</a> </li>
   <!-- <li> <a href="http://f21sc2018.slack.com">Slack</a> </li> -->
   <li><a href="#news">News</a></li>
   <!-- <li><a href="#">Sitemap</a></li> -->
  </ul>
 </div>
</div>
<div id="content">
<div id="right">

<p>
This page collects material on my course on Industrial Programming.
</p>

<p>
As a starting point for this course,
<a href="scripting.pdf">this is a very good article, published in IEEE Computer, summarising the characteristics of scripting languages compared to general purpose programming languages
</p>
<p>
A <a href="F21SC_DL_Guide.pdf">Guide for Independent Learners is available here.</a>
```

(14) Favourite name: The name assigned to a favourite webpage.

(15) Favourite url: The url of a favourite webpage.

(16) Edit button: Press this button to edit the name or url of a favourite entry.

(17) Remove favourite: Click this button to remove a webpage from your list of favourites.

(18) Url edit field: Edit the url of a favourite item and press Save (20) to update the url of a favourite webpage.

(19) Name edit field: Edit the name of a favourite item and press Save (20) to update the name of a favourite webpage.

(20) Save button: Click this button to save the changes made to a favourite webpage.

Developer Guide

Online documentation can be found online at: https://lachlanwoods.github.io/annotated.html

The documentation linked above provides a description of all publicly accessible classes and methods.

The documentation below describes how to use the main classes and methods of the web browser.

BrowserWindow.cs

The BrowserWindow class is an extension of the Form class, and is responsible for the display of the main web browser window. The class' main purpose is to handle GUI callbacks. The majority of GUI controls exist as a child of the BrowserWindow display. BrowserWindow.cs is also responsible for creating and closing tabs, which are performed on button clicks.

Creating and closing tabs:

Tabs can be created by calling BrowserWindow.CreateTab(int index). Index specifies the position the tab should be displayed at in the tab bar, usually specified by :

```
int lastIndex = tabBar.TabCount - 1;


public void CreateNewTab(int index) {
	Tab newTab = new Tab();
	tabBar.TabPages.Insert(index, newTab);
	ResizeTabs(); //update the tabs width to fit within the tabbar
	tabBar.SelectedIndex = index;
}
```

Likewise, tabs can be closed by calling BrowserWindow.CloseTab(int index). This will close the tab at the given index.

```
public void CloseTab(int index) {
	tabBar.TabPages.RemoveAt(index); //remove the tab
	tabBar.SelectedIndex = index; //select the tab beside the one we just closed
	ResizeTabs(); //update the tabs width to fit within the tabbar
	if (this.tabBar.TabPages.Count <= 1) {//if we closed all tabs close the browser
		Application.Exit();
	}
}
```

Tab.cs

The Tab class extends the TabPage class, and is used to represent a single tab. Each tab can perform a HTTP request in a background worker. Each tab stores its own local history in a TabHistory object, which can be used to navigate back and forward in history.

Managing HTTP Requests:

Each tab is responsible for managing its own HTTP requests. Requests are performed by a background worker. Therefore, multiple HTTP requests can be executed simultaneously by different tabs. To perform a HTTP request, call the NavigateToPage() method, that will start the background worker and display the result of the request upon completion.

```
public void NavigateToPage(Webpage page, bool addToHistory) {

       //contrtact to ensure that the requested page is not null
       Contract.Requires(page != null);

        if (page == null) {
               return;
        }

       searchBar.Text = page.GetUrl(); //get the text from the search bar

       //if there is already a running search request, cancel it
       if (backgroundWorker.IsBusy) {
              backgroundWorker.CancelAsync();
       }
       else {
              //run the background worker, to fetch the requested page in a new thread
              backgroundWorker.RunWorkerAsync(page);
       //add the page to local and global history if the addToHistory flag is set
              if (addToHistory) {
                     localHistory.AddToHistory(page);
                     BrowserManager.Instance.History.AddToHistory(page);
              }
       //update the back and forward buttons (i.e) enable them if we can now go back or forward
        localHistory.UpdateNavButtons();
       }
}
```

Refresh page:

A page can be refresh by resending a HTTP request to the currently viewed webpage. This page should not be added to global or local history again.

```
public void RefreshPage() {
       //refresh and do not add the page to history again
       NavigateToPage(localHistory.GetCurrentPage(), false);
}
```

TabHistory.cs

TabHistory.cs stores the local history of a single tab. This class allows a tab to navigate back and forwards between previously visited webpages.

History is stored as a LinkedList of Webpage objects. The current webpage is pointed to by a LinkedListNode object.

Get current page:

You can retrieve the current webage that a tab is displaying by calling the GetCurrentWebpage() method. This returns a Webpage object.

```
public Webpage GetCurrentPage() {

Contract.Requires(currentPage != null); // ensure that the current webpage exists

       if (currentPage == null) {
              return null;
       }
       //return the currently viewed page, held by the currentPage node.
       return currentPage.Value;
}
```

Go back and forwards in history:

When possible, the user can navigate back and forwards to previously visited page. The user can go back when there are nodes before the current webpage in the linked list. Similarly, the user can go forward when there are nodes after the current webpage in the linked list. The back and forward buttons will change between being interactable and non-interactable to indicate these two states. The following two methods are used to get previous and next pages in history, which should be passed to the NavigateToPage() method of Tab.cs.

```
public Webpage GetBackPage() {
	//if we visited a page before the current page
	if (currentPage.Previous != null) {
		//get the node before the current page in the linked list
		Webpage backPage = currentPage.Previous.Value;
		//move current page back one in the linked list
		currentPage = currentPage.Previous;
		return backPage;
	}
	return null;
}


public Webpage GetForwardPage() {
	if (currentPage.Next != null) { //if we visited a page after the current page
		//get the node after the current page in the linked list
		Webpage nextPage = currentPage.Next.Value;
		//move current page forward one in the linked list
		currentPage = currentPage.Next;
		return nextPage;
	}
	return null;
}
```

Add a page to local history:

It is important to add webpages to the local tab history to ensure the tab can go back and forwards. This can be done by calling the AddToHistory() method of TabHistory.cs

```
public void AddToHistory(Webpage page) {
	//code contract to ensure that the webpage exists
	Contract.Requires(page != null);

	LinkedListNode<Webpage> newEntry;

	if (currentPage == null) { //if this is the first page visited by the tab
		newEntry = history.AddFirst(page);
	}
	else {
		//clear forward history if we search for a new page
		if (currentPage.Next != null) {
			history.Remove(currentPage.Next);
		}
		//add the new page after the current page in history
		newEntry = history.AddAfter(currentPage, page);
	}
	currentPage = newEntry; //set the current page as the new page
	UpdateNavButtons(); //update the back and forward buttons to make them
interactible, if they were previously not interactible
}
```

BrowserManager.cs

The BrowserManager class uses a singleton pattern, so it can be accessed from any object. A BrowserManager object is created on Start-up, and can be accessed by calling BrowserManager.Instance. You can use the BrowserManager instance to access common functions, such as reading or writing to files, and to access the main BrowserWindow control, without needing to pass references as parameters. The BrowserWindow object can then be used to access other GUI controls from other objects. You can also use the BrowserManager instance to access global history, the favourites list and the current tab.

Accessing global history, favourites, the current tab and BrowserWindow objects:

Global history, favourites the current tab and the BrowserWindow control can be access through properties defined in BrowserManager.cs. You can access the two lists by calling BrowserManager.Instance.History and BrowserManager.Instance.Favourites. Likewise, you can access the current tab with BrowserManager.Instance.CurrentTab and the BrowserWindow with BrowserManager.Instance.Window.

Reading the history and favourites files:

History and favourites are stored in history.xml and favourites.xml in the root directory of the program. These two files are loaded on start-up simultaneously in two separate threads. It is important to wait for these two files to load before the user can interact with the web browser to ensure that all required data is available on launch. This is achieved by using the Thread.Join() method. Files are read by calling the ReadFilesMultithreaded() method.

```
public void ReadFilesMultithreaded() {

       Thread favThread = new Thread(new ThreadStart(favourites.LoadFavourites));
       Thread historyThread = new Thread(new ThreadStart(globalHistory.LoadHistory));

       //run the two threads
       favThread.Start();
       historyThread.Start();

       //Wait for both threads to complete before continuing the main thread. This
ensures both files will be loaded before the user can interact with the web browser.
       favThread.Join();
       historyThread.Join();
}
```

Writing to files:

Files can be written in a new thread to avoid blocking the main thread. This can be achieved by calling the WriteToFileInThread() method in BrowserManager.

```
public void WriteToFileInThread(string content, string filePath) {
       //uses a lambda expression to write to a file in a new thread
       Thread newThread = new Thread(() => {
                using (StreamWriter writer = new StreamWriter(filePath)) {
                    writer.WriteLine(content);
                }
            });

       newThread.Start(); //run the new thread to write to the file.
}
```

Favourites.cs and GlobalHistory.cs

Favourites.cs and GlobalHistory.cs are responsible for managing the favourites and history lists. Favourites.cs performs operations on FavouriteEntry objects, whereas GlobalHistory.cs performs operations on HistoryEntry objects.

Add a webpage to global history:

Call BrowserManager.Instance.History.AddToHistory(Webpage page) to add a webpage to global history.

Add a webpage to favourites:

Call BrowserManager.Instance.Favourites.AddToFavourites(FavouriteEntry favourite) to add a webpage to favourites. A FavouriteEntry object can be created by calling:

new FavouriteEntry(string favouriteName, Webpage page)

Remove a webpage from global history:

Call BrowserManager.Instance.History.RemoveFromHistory(HistoryEntry page) to remove a webpage from global history.

This action is typically performed by clicking the "X" button on the GUI display of a HistoryEntry object. In this scenario, you can use the current object (using the "this" keyword) as the HistoryEntry parameter.

Remove a webpage from favourites:

Call BrowserManager.Instance.Favourites.RemoveFromFavourites(FavouriteEntry favourite) to remove a webpage from favourites.

This action is typically performed by clicking the "X" button on the GUI display of a FavouriteEntry object. In this scenario, you can use the current object (using the "this" keyword) as the FavouriteEntry parameter.

Testing

Code contracts were used to perform runtime checking during static analysis.

The following tests were conducted on the web browser. The expected and actual results have been recorded.

Action	Expected Result	Actual Result	Pass?
Perform a HTTP request and receive a 200 (ok) response	The html of the requested webpage should be displayed on the web browser. The title of the webpage and "200 (ok)" should be displayed on the web browser title bar.	http://www.google.com was requested. The html of the webpage was displayed, and "200 ok Google" was displayed on the web browser title bar.	Yes
Perform a HTTP request and receive a 400 (Bad Request) response	The title of the web browser should display "400 Bad Request". The program should not crash.	https://httpstat.us/400 was requested to receive a 400 status code. "400 BadRequest" was displayed in the title bar. The program did not crash.	Yes
Perform a HTTP request and receive a 403 (Forbidden) response	The title of the web browser should display "403 Forbidden". The program should not crash.	https://httpstat.us/403 was requested to receive a 403 status code. "403 Forbidden" was	Yes

		displayed in the title bar. The program did not crash.	
Perform a HTTP request and receive a 404 (Not Found) response	The title of the web browser should display "404 Not Found". The program should not crash.	https://httpstat.us/404 was requested to receive a 404 status code. "404 NotFound" was displayed in the title bar. The program did not crash.	Yes
Add a webpage to history, and ensure it is persistent between sessions.	Web pages should be visible in the history list after they have been visited. The list should be loaded when the program is closed and opened again.	The history list was persistent between sessions. Web pages that had been visited in previous sessions were visible in the history list when the program was re-launched.	Yes
Add a webpage to favourites, and ensure it is persistent between sessions.	A user should be able to add webpages to their favourites list. This list should be loaded when the program is closed and opened again.	I was able to add a webpage to my favourites list. The favourites list was persistent between sessions.	Yes
Set a homepage, and check that it is loaded by default.	A user should be able to set a webpage as their homepage. This homepage should be loaded when a new tab is opened.	A homepage http://www.google.com was set. This homepage was loaded whenever a new tab was opened, and when the web browser was first opened. The homepage set was persistent between sessions.	Yes
Refresh a page	The webpage being displayed in a tab should be refreshed when the refresh button is pressed.	When the refresh button is pressed, a HTTP request is resent to the current webpage, which refreshes the page.	Yes
Go back to a previous page	When a previous page has been visited, the back button should be interactable. When this button is clicked, the current tab should navigate back to the last visited page.	The back button was not interactable until a previous page was available. When possible, clicking the back button navigated the current tab to the previously visited webpage.	Yes
Go forward to a previous page	When a page has been visited, and then gone back from, the forward button should be interactable. When this button is clicked, the current tab should navigate forward to the previously visited page.	The forward button was not interactable until a previous page was available to go forward to. When possible, clicking the forward button navigated forward in history.	Yes
Edit the name and url of a favourite page	A user should be able to edit the name and url associated with a favourite page. These changes should be immediately visible and saved between sessions.	The name and url of a favourite entry could be changed by clicking the "edit" button beside a favourite entry. A prompt was shown to change the url and name of the entry. Changing these values and clicking "save" immediately updated the values shown on the favourites list, and were saved between sessions.	Yes

Add a new tab	A new tab should be added when the new button is clicked.	Clicking the final tab on the tab bar with the "+" icon added a new tab. This tab was automatically selected and navigated to the homepage.	Yes
Close a tab	A tab should be closed when the "X" button is clicked.	Clicking the "X" button on a tab closed that tab and removed it from the tab bar. Closing all tabs closed the entire application.	Yes
Perform multiple HTTP requests at the same time	Multiple HTTP requests running at the same time should be supported through multithreading. This should not block interactions with the GUI.	Multiple HTTP requests could be performed at the same time by using multiple tabs. Interactions with the GUI were not blocked when HTTP requests were running.	Yes
Override an ongoing HTTP request, by making a search while a previous request is still running.	The user should be able to search for a new request, while a HTTP request is currently running. The currently running HTTP request should be cancelled, and the new request should be run instead.	A request to A slow loading page was made. While this request was processing, I tried to search for another page. The original request was not cancelled. Even though I make a call in code to cancel a background worker when a new request is made, the cancellation is never processed by the background worker. This is due to the call to request.GetResponse() blocking the background worker until a HTTP response is received. Therefore the cancellation of a background worker will only take affect if the cancel call is received before request.GetResponse() is reached. It is unlikely that this will happen. This is a downside of using background workers that have blocking method calls. backgroundWorker.CancelAsync(); is better suited for background workers that perform work in a loop, as the worker can check the cancel flag in each iteration of the loop. When a blocking call such as GetResponse() is used, the background worker cannot check the cancelation flag until the blocking call is completed.	No
Display Rendered HTML	Clicking the HTML view button should toggle between displaying HTML code and rendered HTML	Clicking the HTML view button toggles between displaying HTML code and rendered HTML. Some	Yes

		webpages do not display correctly as the web browser only supports CSS 2 and does not support JavaScript.	
Remove history and favourite webpages	Favourite webpages and history should be removeable. Removals should be reflected between sessions.	Clicking the "X" button beside a history or favourite entry immediately removes the webpage from the history or favourites list. This is reflected between sessions.	Yes

Conclusion

I am confident that I have met all requirements of the coursework. I am proud to have included substantial additional features to the web browser, such as rendered HTML and sliding side panels. Effort was put into removing all bugs detected during development. As such, I do not believe there are any actions that result in unusual or unexpected results.

I am very happy with the quality of my code, and believe that software engineering best practices have been used for all my code. Code has been written to be easily readable and maintainable. This has been achieved by creating classes that have high cohesion and low coupling. All classes and methods have been commented with XML documentation. I was interested to find that these comments could be used to automatically generate documentation files using tools like Doxygen. The documentation of my program can be found here.

If I was to repeat this coursework, I would like to try a different approach for multithreading client-server communication. My current implementation creates and manages tabs in the main thread. Each tab has access to its own background worker, which is used for HTTP requests. This means that HTTP requests are multithreaded, but the actual tab objects are all controlled by the main thread. I would be interested in seeing the impact of creating new threads for each tab, and creating the entire tab object in the new thread. I originally though this approach would cause issues, as GUI controls cannot be updated by threads they were not created in. Since starting the project, I have discovered that this may not be the case due to the Invoke() method. Regardless, the approach that I chose for multithreading works well for the purpose of this coursework.

References:

HTML Renderer: https://github.com/ArthurHub/HTML-Renderer

Documentation generator: http://www.doxygen.org/index.html

Displaying a close icon on tabs: https://stackoverflow.com/questions/36895089/tabcontrol-with-close-and-add-button/36900582#36900582