0.1 Class design

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

0.1.1 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.

0.1.2 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].

0.1.3 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.

0.1.4 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.

0.2 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.

0.3 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.

0.4 Advanced language constucts

Delegates have been used extensively throughout this body of work, particularly when using delegates to invoke a method from an event handler see Fig. **??**. Anonymous delegates have been very useful during unit testing, specifically to check if an event had been triggered see Fig. **??**. [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>**.

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

```
// 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