Milestone 3 : Design Document

Matthew Pike, 523355@swansea.ac.uk
Supervisor: Dr Max Wilson

Swansea University
Prifysgol Abertawe

Swansea University
Computer Science Department

Contents

1 Introduction

In this design document we will provide the necessary documentation for the design of the User Study System (USS). We do so by detailing the design of the current implementation of the system.

This document will be expressed in a semi-formal notation, using a variety of UML diagrams. Additionally, we detail the key design decisions made during the construction of the system, and discuss our rationale for making those decisions.

Term Definition

As with any technical project, there exists numerous technical abbreviations that are used in place of long descriptors. We present a table below of the various abbreviations used in this document, which should serve as an aid to the reader.

Term	Definition
Visualiser	The part of the system responsible for displaying the data visualisation.
Document	The web site or web based document being investigated in the user study.
Experiment	The study that is being conducted. Typically will contain a User and a Conductor.
Data Source	A device/software/location that is recorded by the Web Browser.
Recorded Instance	Since a single experiment on a single user can contain numerous conditions and tasks, we must distinguish what one unique combination of these properties are called. We have chosen 'Recorded Instance' to represent this. A Recorded Instance is an identifier for: Experiment -> User -> Condition -> Task.
Stack	A Stack is a part of the Visualiser UI that represents a single Recorded Instance. A Stack is therefore a UI component.
WPF	Windows Presentation Foundation - The visual framework used to develop the user interface(s) in this project.
ORM	Object Relational Mapping - a way of linking entries in a database to a usable format in application code.

SQLCE	SQL Compact Edition, the embedded database product we use to store application data.

Personas

In addition to technical abbreviations, the project features unique individuals that partake or feature somehow in the system. Below is a table documenting these individuals and their role in the system. Again, this table is meant as an aid to the reader.

Individual	Role
Conductor	The person responsible for performing the user study. This is typically a researcher who is aiming to prove a particular hypothesis.
Researcher	The person who is responsible for gaining insight from the user study. The Researcher and Conductor can typically be the same person, however this is not always the case, especially in a large research group. The Researcher is therefore a member of the research team who is wishing to use the data collected from the study.
User	The person who is sitting the user study. Their responsibility is to perform tasks provided to them by the conductor.
Client	The person who will be receiving the finished software project. In this case it is Pingar.

Electronic Document

Due to the limited real estate of physical paper, some of the technical diagrams may be unreadable. For a high resolution, electronic copy of this document, please navigate to the following URL to download an electronic copy:

http://dl.dropbox.com/u/21780/Docs/Design%20Document.pdf

2 System Overview

In this section we will provide a brief overview of the User Study System (USS) and the subsections that form it. The intention for this section is to provide an overview of the components that form the USS and their interactions with one another in order to gain a basic understanding of the USS's structure.

The USS was designed as a combination of two major subsystems; Browser and the Visualiser. However, the subsystem Browser, can be further broken into two separate components; Setup and Web Browser. This relationship is shown in Figure 2.1.

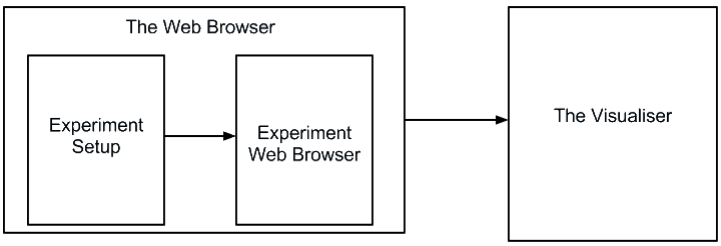

Figure 2.1: The Major subsystems and components that form the USS.

We see from Figure 2.1 that each component in the overall system is linearly dependant on the component before it. This gives the system a seemingly flat hierarchy. However, in terms of actual implementation, additional components are required in order for the system to operate correctly. As such, Figure 2.2 is an accurate representation of the major subsystems that exist in the USS.

The purpose of this document is to explain the design of these subsystems and how they interact to form the USS. Figure 2.2 shows the dependencies of each component in the system, and the remainder of this document will aim to detail each component.

The document has been organised according to the components identified in Figure 2.2. The diagram has been generated according to the namespaces in the actual implementation of the project. The remainder of the documented systematically goes through each major component and documents its purpose and any design decisions.

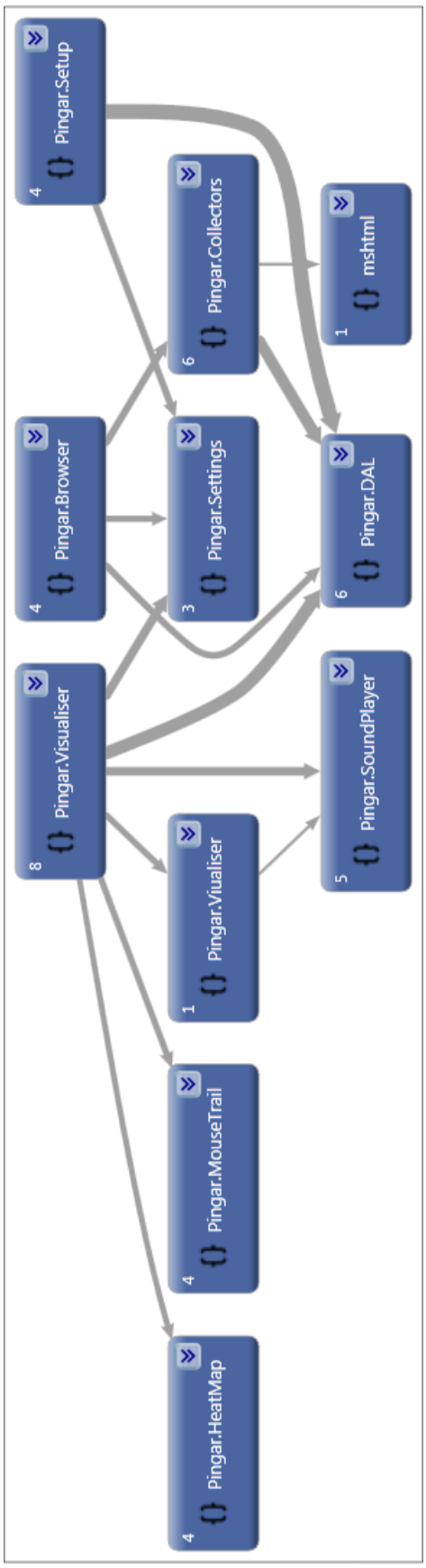

Figure 2.2: The components of the USS and their interaction/dependency with one another.

3 Data Abstraction Layer (DAL)

Namespace

- Pingar.DAL

Key Files

- USS/DAL/Context.cs
- USS/DAL/DatabaseSetup.cs
- USS/DAL/Experiment/Condition.cs
- USS/DAL/Experiment/Experiment.cs
- USS/DAL/Experiment/ExperimentInstance.cs
- USS/DAL/Experiment/Participant.cs
- USS/DAL/Experiment/Task.cs
- USS/DAL/Settings/AudioSettings.cs
- USS/DAL/Settings/BrainSettings.cs
- USS/DAL/Settings/ScreenSettings.cs
- USS/DAL/Settings/WebEventSettings.cs

3.1 Purpose

The Data Abstraction Layer (DAL) should be thought of as the gatekeeper to the application data. The publicly exposed functionality of the DAL provides the interface to the USS database, which is responsible for storing the data relating to the conducted user studies.

The DAL specifies the data entities stored in the system database, these are identified by their representative data type e.g. the ***Participant*** entity represents a participant in a user study. These entities and their relations to one another are then used to form the system database automatically, via the Entity Framework.

Additionally the DAL is responsible for the administrative side of the system's data. The component includes functionality to create, populate and validate the USS database.

The primary and only public responsibility of the DAL however is to provide a ***Context***, which other components can use to access the system database.

The **Entity Framework** technology has been used exclusively in the DAL. The framework is simply an object-relational mapper which automatically translates classes into representative database tables. It can be thought of as structured serialization.

3.2 Design

In Fig. 3.1 we provide the UML diagram representation for the DAL. The majority of the component consists of entity types. That is, the majority of the DAL consists of the actual data being represented in the database. We see that as well as defining individual entities (e.g. **BrainSettings, Experiment, ...**), we have also created a relationship between entities. The primary example, can be seen in ***ExperimentInstance*** which has a many-to-one association with all other entities in the DAL. ***ExperimentInstance,*** represents a single User Study recording and the various settings for that individual recording.

Additionally, there are 3 stand alone classes in the DAL. The first two ***DbInitializer*** and ***DatabaseSetup*** simply define how a database file should be generated and what should be inserted as default values. ***Context*** is however of more interest. We see from Fig. 3.1, that ***Context*** also exposes each of the entities stored in the DAL, however ***Context*** acts as the intermediary for the actual data in the database. Therefore it is responsible for committing and retrieving the specified entities from the system database.

3.3 Design Decisions

The primary design decision taken for the DAL was the representation of the data using a Code First representation. Under the Entity Framework, we could have followed a number of alternative routes:

Database First In this mode, we would develop the system's database using an external tool (e.g. SQL) and then interface to the DB using the entity framework. We decided against this approach since it tied us to a specific database technology and was fairly inflexible.

Model First Using this mode, we could develop abstract data models using a graphical IDE (Visual Studio), and then instruct the Entity Framework to generate the representative classes. We did find this approach productive, however we found that the generated code was unnecessarily complex.

Code First Our chosen approach, Code First allowed us to specify data entities and their relations using standard C# code. From this simple (hierarchical) structure, the Entity Framework generated the representative database and allowed for massive code reuse. We found this approach to be the most intuitive also.

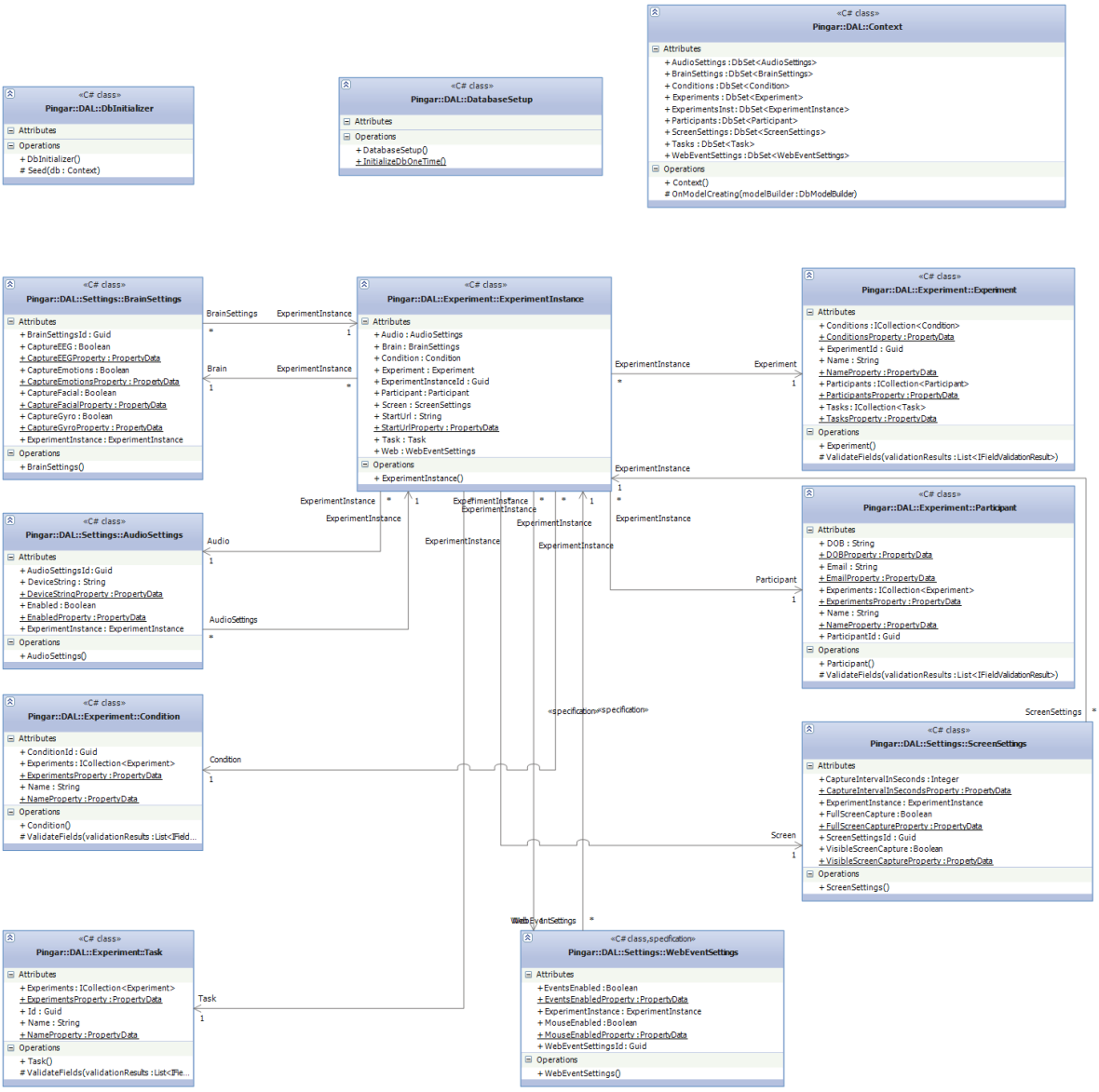

Figure 3.1: The UML diagram for the DAL.

4 Setup

Namespace

- Pingar.Setup

Key Files

- USS/Setup/ViewModels/MainWindowViewModel.cs
- USS/Setup/ViewModels/Pages/BeginExperimentPageModel.cs
- USS/Setup/ViewModels/Pages/CollectorsPageModel.cs
- USS/Setup/ViewModels/Pages/ExperimentPageModel.cs
- USS/Setup/ViewModels/Pages/HomePageModel.cs
- USS/Setup/Views/MainWindow.cs
- USS/Setup/Views/Pages/BeginExperimentPage.cs
- USS/Setup/Views/Pages/CollectorsPage.cs
- USS/Setup/Views/Pages/ExperimentPage.cs
- USS/Setup/Views/Pages/HomePage.cs

4.1 Purpose

The Setup component is responsible for providing the User Interface for Conductors to configure user studies. The interface is simply a front end for the data held in the USS database, and does not perform any advanced/ atypical operations.

4.2 Design

The Setup component is intended to interface with the USS database. To accomplish this we have implemented the MVVM architectural pattern, which separates the various application layers into defined sections:

Model This is the data represented in the USS database. We get this layer for "free" from the DAL, which is an indicator that we are following good engineering principles, since we are achieving high code reuse.

View This is the presentation layer that the conductor interacts with. This should simply define how a particular screen in the interface should appear visually. The viewer should have no knowledge or preconceptions of the data that it is displaying.

View Model The view model is the "glue" that ties the data (Model) to the user interface (View). The View Model mediates between the View and the Model based on the data bindings in the View.

We can see from the component's dependency graph (Figure 2.2) that the Setup component's only external dependency (excluding application Settings) is the DAL. This confirms that the component's only intention is to behave as a front end to the application data.

In terms of the structure of the Setup component, the UML diagram shown in Figure 4.2 provides the necessary detail of the component's structure. We see that the design matches our chosen architectural pattern MVVM.

In total there are 4 *Pages* in the Setup application. Each *Page* has an associated View Model and View, denoted by the class name:

1. Welcome
    a) View - HomePage
    b) View Model - HomePageModel
2. Experiment
    a) View - ExperimentPage
    b) View Model - ExperimentPageModel
3. Collectors
    a) View - CollectorsPage
    b) View Model - CollectorsPageModel
4. Begin Experiment
    a) View - BeginExperimentPage
    b) View Model - BeginExperimentModel

This hierarchy can also be seen in Figure 4.1. Additionally there is a ***MainWindow*** which is simply the container that holds each of the pages and provides the page switching logic.

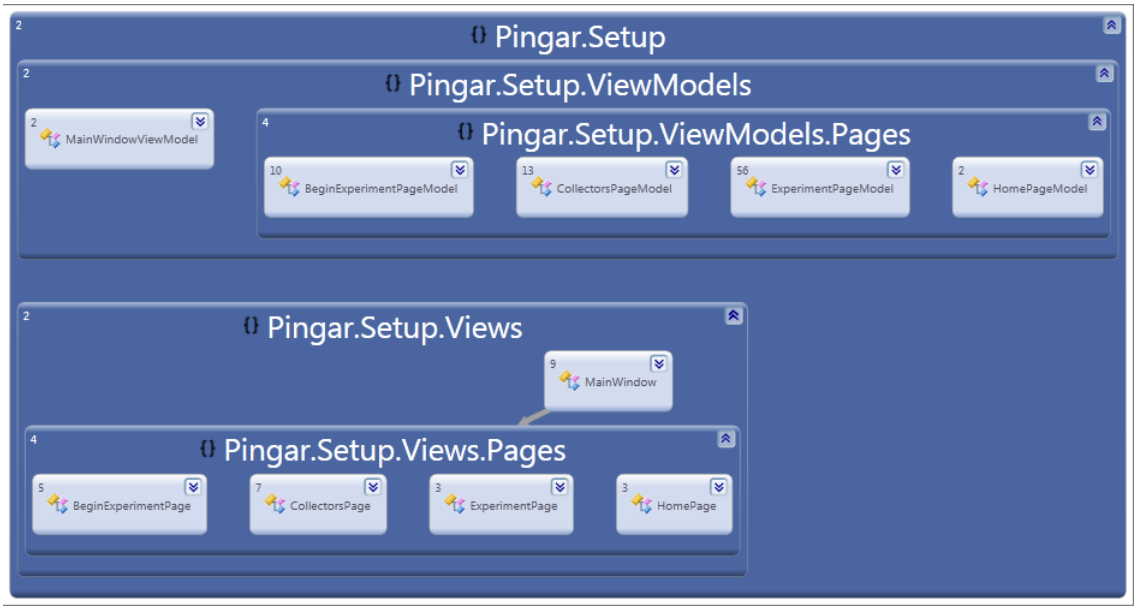

Figure 4.1: The hierarchy of the Setup component.

4.3 Design Decisions

The primary design decision made for the Setup component was the architectural pattern. There are several patterns designed for interfacing user interfaces and data collections. The two primary patterns are : Model View Controller (MVC) and Model View View Model (MVVM). Both are very similar in structure, with each separating the View and Data from one another, and interfacing them together through some other layer.

However our decision to use MVVM over MVC is based on the following criteria:

Library Support Despite there being many third party libraries supporting MVC and WPF (the User Interface Library), there are more mature and extensive libraries supporting MVVM. Our chosen library for the Setup component was Catel http://catel.catenalogic.com/. Catel was chosen for its simplicity and its use of annotations to tag classes as having certain functionality.

Event Driven Programming MVVM is specifically designed for event driven programming such as a user interacting with a user interface. Whereas MVC is designed to be very flexible, allowing controllers to be used interchangeably, the view model (used in MVVM) tends to be a lot more specific to the view that it serves. This may seem like a negative since MVVM is therefore less flexible than MVC, but in actuality, MVVM exploits event based programming principles such as data binding which would break the MVC pattern. As such it requires less set-up code using the MVVM when compared to MVC.

The decision to use MVVM also structured the Setup component in a very organised

manner. We see from Figure 4.1 that the component is split into 2 distinct namespaces - Views and ViewModels. Being well structured should reduce the complexity of future maintenance.

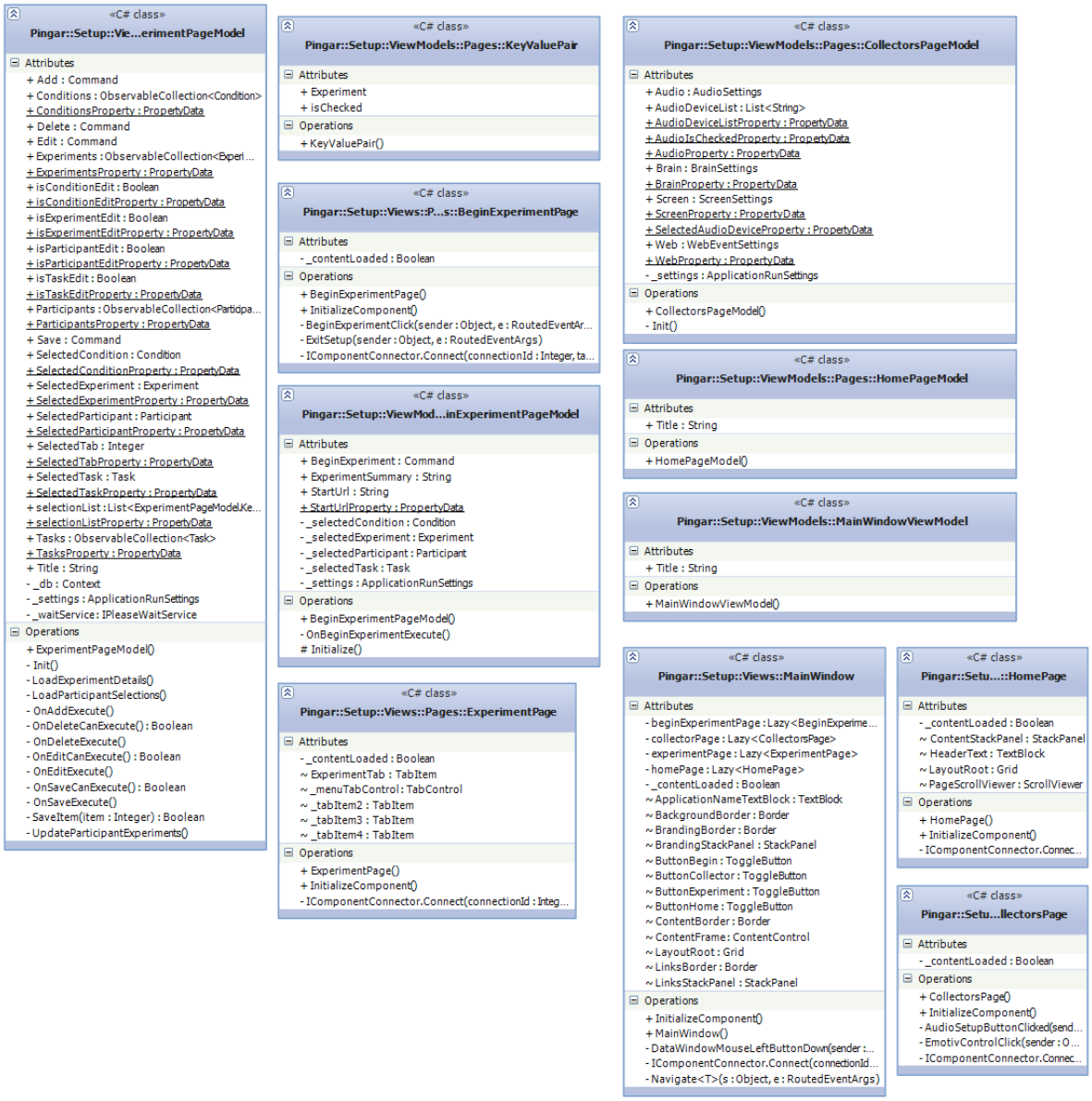

Figure 4.2: The UML diagram for the Setup component.

5 Browser

Namespace

- Pingar.Browser

Key Files

- USS/Browser/Program.cs
- USS/Browser/WebBrowser.cs
- USS/Browser/CommObject.cs

5.1 Purpose

The Browser component is the front end web browser that the participant uses during the User Study. Its direct functionality is simply to behave as a standard web browser and provide the standard navigation features that are familiar to most users. Additionally the Browser is responsible for initiating the Collectors (See chapter 6) which are responsible for collecting the various data points during the user study.

5.2 Design

Due to the relative simple requirements of this component, the design of the Browser component is fairly straightforward. There are only 2 classes that are of interest in this component.

The first class ***WebBrowser***, provides the front end for the web browsing functionality. Since the browser is fairly simple and does not interact with any external data, we simply implemented all navigation logic in the code-behind file for the web browser.

The second class of interest is ***CommObject***. This class is tagged with the annotation - ***ComVisible.*** By tagging the class with this annotation, we are instructing the compiler to make the class publicly available to anything implementing the COM interface. By doing this we are able to address ***CommObject*** directly from

within our JavaScript code, which is responsible for capturing JavaScript derived events when the participant browses the web. Therefore this class can be thought of as the bridge between our JavaScript library and the ***Collectors*** component (See chapter 6) which logs the web events to file.

5.3 Design Decisions

The primary design decision here was the use of the COM as the communication bridge between JavaScript and the client code. Our decision to use COM over other potential methods, such as event handlers in the ***WebBrowser*** component, was down to completeness. We discovered that COM was the only approach that allowed us to capture all JavaScript events during a study. Other approaches (such as custom event handlers) provided access to some, but not all events and as such were not suitable.

Alternatively we could have chosen to use an alternative Web browsing component, in favour of the standard WPF ***WebBrowser*** variant. Alternative technologies exist (such as http://awesomium.com/), but were not used in this current implementation because of technical difficulties and time constraints.

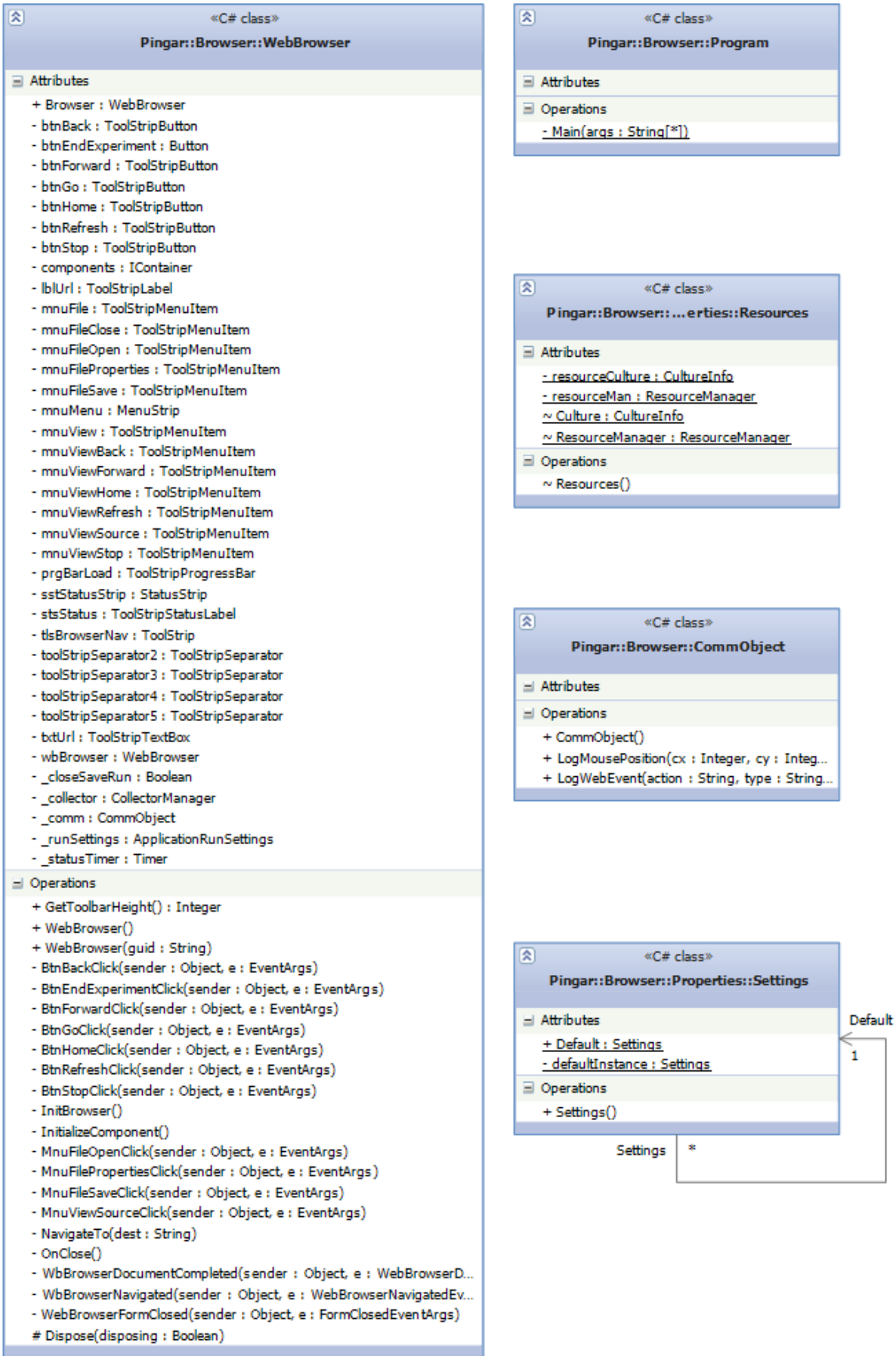

Figure 5.1: The UML diagram for the Browser Component.

6 Collectors

Namespace

- Pingar.Collectors

Key Files

- USS/Collectors/Brain/Data/Facial_Data.cs
- USS/Collectors/Brain/Data/Emotion_Data.cs
- USS/Collectors/CollectorManager.cs
- USS/Collectors/WebEvents/MouseLogger.cs
- USS/Collectors/WebEvents/WebEventCollector.cs
- USS/Collectors/WebEvents/WebEventLogger.cs
- USS/Collectors/Brain/BrainCollectorRunner.cs
- USS/Collectors/Brain/Data/Pair.cs
- USS/Collectors/ScreenCapture/ScreenCaptureCollector.cs
- USS/Collectors/Audio/AudioCaptureCollector.cs
- USS/Collectors/BaseCollector.cs
- USS/Collectors/Brain/BrainCollector.cs
- USS/Collectors/Audio/AudioUtilities.cs

6.1 Purpose

The ***Collectors*** namespace forms a vital part of the USS. It is responsible for capturing the specified data during a user study. The component currently supports a variety of data acquisition sources:

1. Emotiv EPOC brain scanner
2. Audio from microphone
3. Screen capture of the web browser
4. Web Events from the web browser

The component is designed to provide the collection functionality from each of these sources and store them permanently on disk. The component is also required to be extensible, allowing for additional data sources to be added in the future.

6.2 Design

The design of the ***Collector*** component required a great deal of consideration as there were many factors to consider.

First and foremost, the component needed to be extensible, allowing for additional data sources to be added in the future. To facilitate this we created a base class ***BaseCollector*** which contains two delegate methods - ***Start*** and ***End.*** The ***BaseCollector*** class includes the necessary scaffolding code for registering a collection source in the application. All that is required to add a new data collector therefore is the implementation of the ***Start*** and ***End*** delegate methods, which are called when the user study itself is begun and is about to end respectively. This provides very simple extensibility.

Secondly, the ***Collector*** must have some way of simultaneously starting and ending all of the necessary collectors. To accomplish this we created the class ***CollectorManager*** which is responsible for managing each data source.

The remainder of the classes in ***Collector*** component (shown in Figure 6.1) are implementation of data sources. Each data source follows a similar structure:

Data Structures Each data source defines a data structure representing a single unit of data.

Collector The class which implements ***BaseCollector.*** This class is responsible for collecting the data, inserting it into the associated data structure and saving it to disk.

Utility Additionally, many data sources have helper utilities which provide additional functionalities not associated with data collection, but aid in the general process e.g. routines for processing the collected data.

We have used this technique very successfully on a number of diverse data sources and can recommend this approach for future data source development.

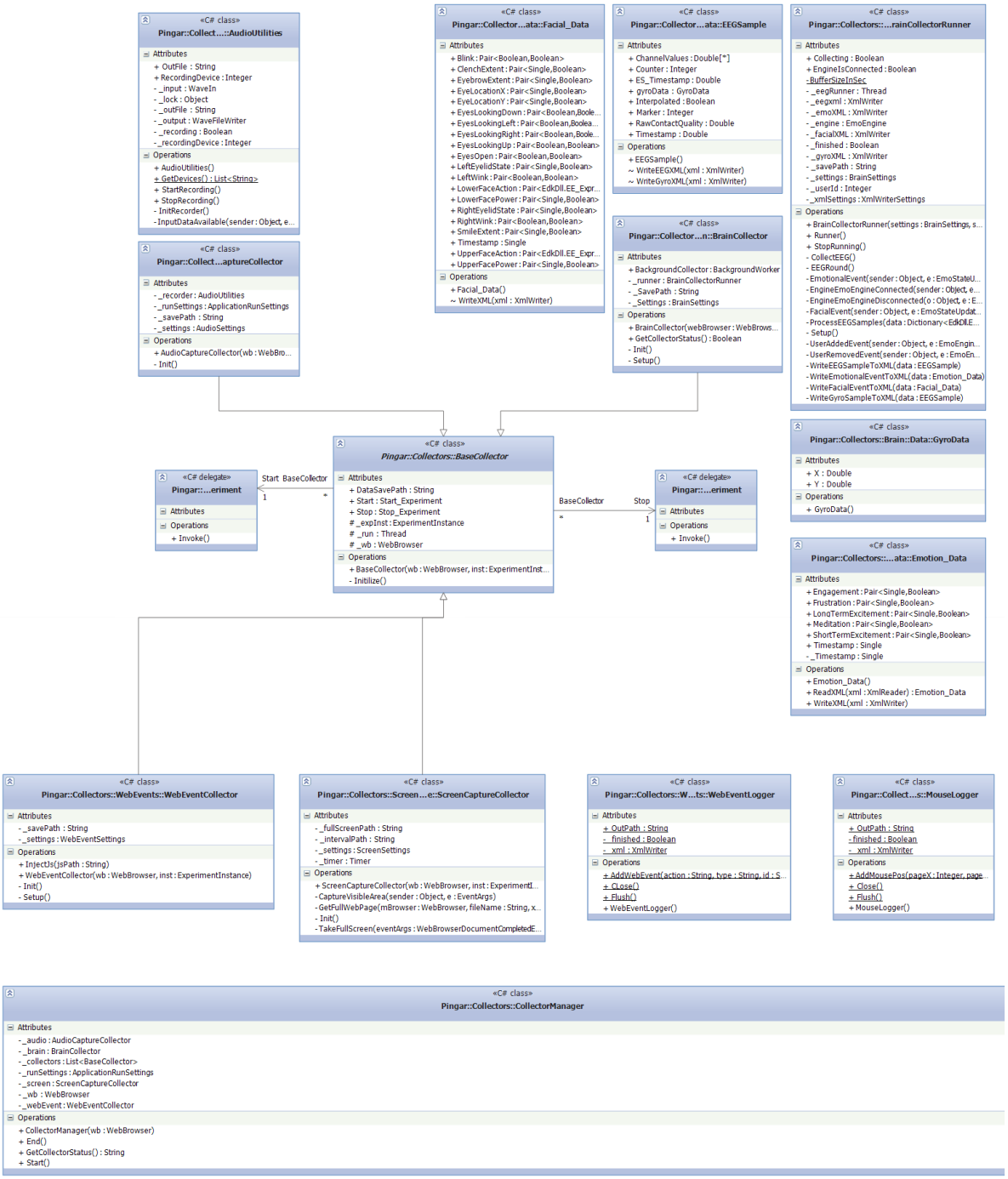

Figure 6.1: The UML diagram for the DAL.

6.3 Design Decisions

The requirement for the ***Collector*** component to be extensible immediately made us consider the use of an existing plug-in architecture, such as Managed Extensibility Framework (MEF - http://mef.codeplex.com/). We researched the framework considerably, and even constructed a prototype utilising the library. MEF is perfectly suited to solving this problem, and the functionality and documentation provided with the framework is of a high quality. However, due to the considerable time constraints imposed upon this project, we were not able to utilise the framework.

We instead opted for a simple, tried and tested approach using the State design pattern. Using this pattern we expose certain functionality in the client application (***BaseCollector***) which allows plugins to "hook" into. The approach provides a high level of integration between client and plugins, without the need for extensive integration code in the plugin.

7 Visualiser

Namespace

- Pingar.Visualiser

Key Files

- USS/Visualiser/MainWindow.cs
- USS/Visualiser/Pane.cs
- USS/Visualiser/ExperimentLoader.cs
- USS/Visualiser/BassEngine.cs
- USS/Visualiser/ExperimentTime.cs
- USS/Visualiser/ImageTime.cs
- USS/Visualiser/ImageViewer.cs

7.1 Purpose

The Visualiser Component is the front end responsible for visualising and comparing User Study recordings. The component displays numerous visualisations of a single recording and is responsible for ensuring all visualisations occur in the correct time-frame, according to the researcher's input.

7.2 Design

The Visualiser component is responsible for a variety of tasks and fulfils many responsibilities. In order to document this correctly we detail the design according to each responsibility. The UML diagram for the Visualiser component is shown in Figure 7.1.

Container - MainWindow The ***MainWindow*** class is the User Interface that contains the visualised recordings within it. Since the Visualiser component was required to display many recordings at once, it was designed to incorporate a tabbed interface allowing for massive customisation in the display of recordings. To accomplish this the ***MainWindow*** uses AvalonDock (http://avalondock.codeplex.com/), a WPF control library which is used to create a docking layout system like that present in Visual Studio.

A Single Recording - Pane A single recording is visualised within its own "Pane" (referred to as ***DockableContent*** by AvalonDock). Each pane has references to:

Heatmap Image Viewer The component responsible for displaying a heatmap visualisation. See chapter 10 for details.

MouseTrail Image Viewer The component responsible for displaying a mouse trail visualisation. See chapter 9 for details.

AudioPlayer The component responsible for displaying and playing the associated audio recording. See chapter 10 for details.

ExperimentLoader This class is responsible for loading the recorded experiment data from disk into memory. This is done dynamically according to the information needs. For example, if image X is required at time Y, then ***ExperimentLoader*** only loads the image into memory at time Y. This conserves memory usage and ensures that the application is responsive.

ExperimentTime This class is a simple data structure that conveys the current experiment time. This is not as trivial as it first seems however, since an experiment has both a Start and End time, as the Researcher is able to select periods of time from within the experiment. This class also implements the ***PropertyChangedEventHandler*** interface, allowing for the automatic implementation of the Observer design pattern on the ***ExperimentTime*** event.

BassEngine This is the class responsible for decoding and gathering statistics from the audio recording associated with the user study. The class simply acts as a helper to the underlying BASS engine (http://www.un4seen.com/) which performs the actual operations.

7.3 Design Decisions

For the Visualiser component, a lot of the key design decisions were in fact handled transparently through our chosen docking library - AvalonDock. The key design decisions, such as how to represent a single recording and manage many recordings, were handled by the library. The design of the Visualiser was therefore based heavily on the recomended patterns provided by AvalonDock - http://avalondock.codeplex.com/documentation.

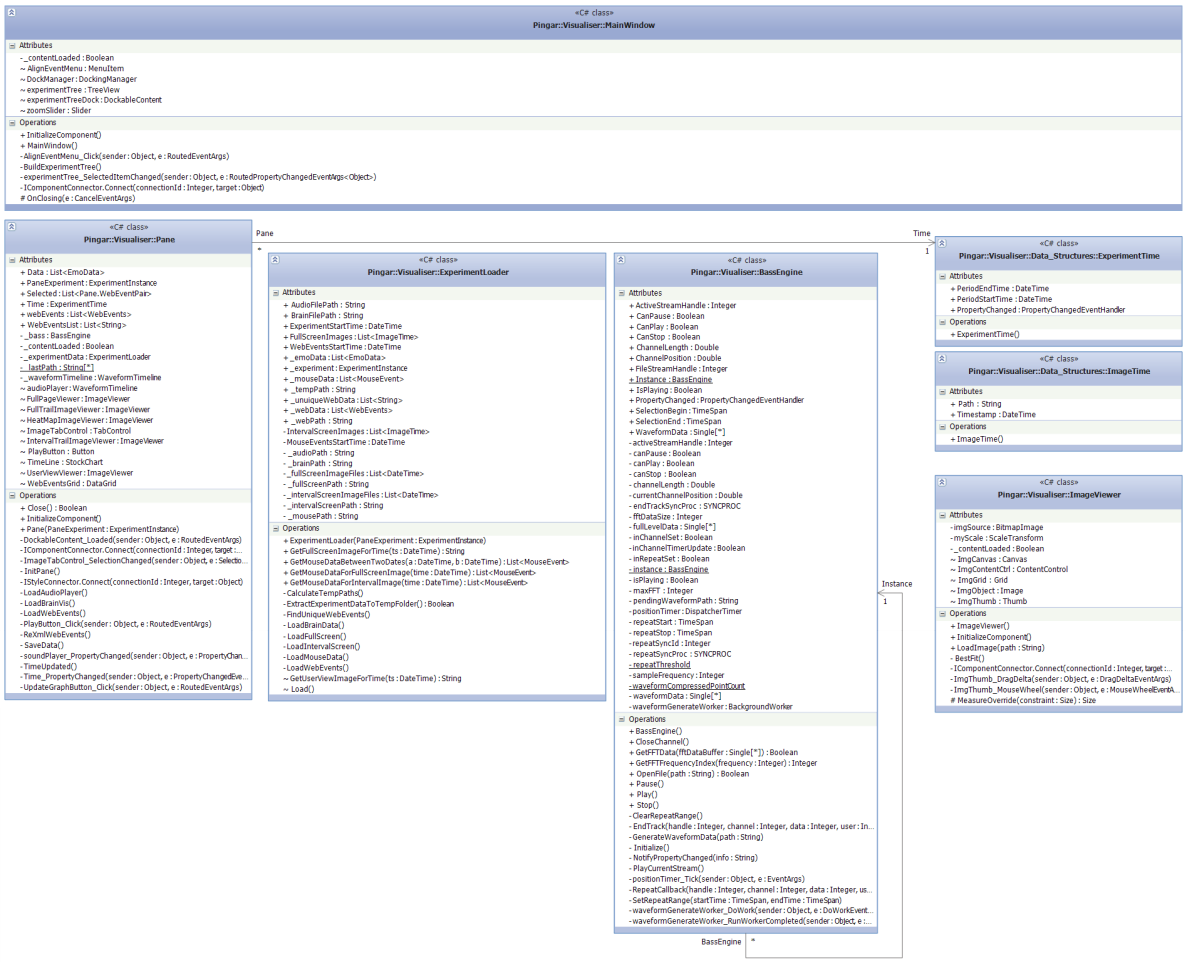

Figure 7.1: The UML diagram for the Visualiser Component.

8 Heatmap

Namespace

- Pingar.HeatMap

Key Files

- USS/HeatMap/heatdot.png
- USS/HeatMap/HeatMapManager.cs

8.1 Purpose

The HeatMap component is designed to overlay a colour heatmap visualisation on-top of a provided image using a collection of coordinates.

8.2 Design

The UML diagram for the HeatMap component (Shown in Figure 8.1), shows that the component consists of a single class. The class can be thought of as a utility class. The class is informally divided into 3 stages:

1. **Load Base Image (LoadImage) -** During this initial stage, the base image (i.e. the image which is to be overlayed with the heatmap) is loaded into the heatmap manager, and consequently into memory.
2. **Add Hits (AddHit) -** At this point the class expects for the points that the HeatMap will visualise, to be loaded into the manager. A check is made to ensure that the Hit falls within the image boundary.
3. **Generate the Heatmap (GenerateHeatMap) -** During this final stage, the HeatMap itself is created. The method proceeds to successively add the heatdot.png image file onto each Hit stored in the manager. This collection of "heat dots" is then colourised according to a custom defined colour scale and overlayed onto the original image.

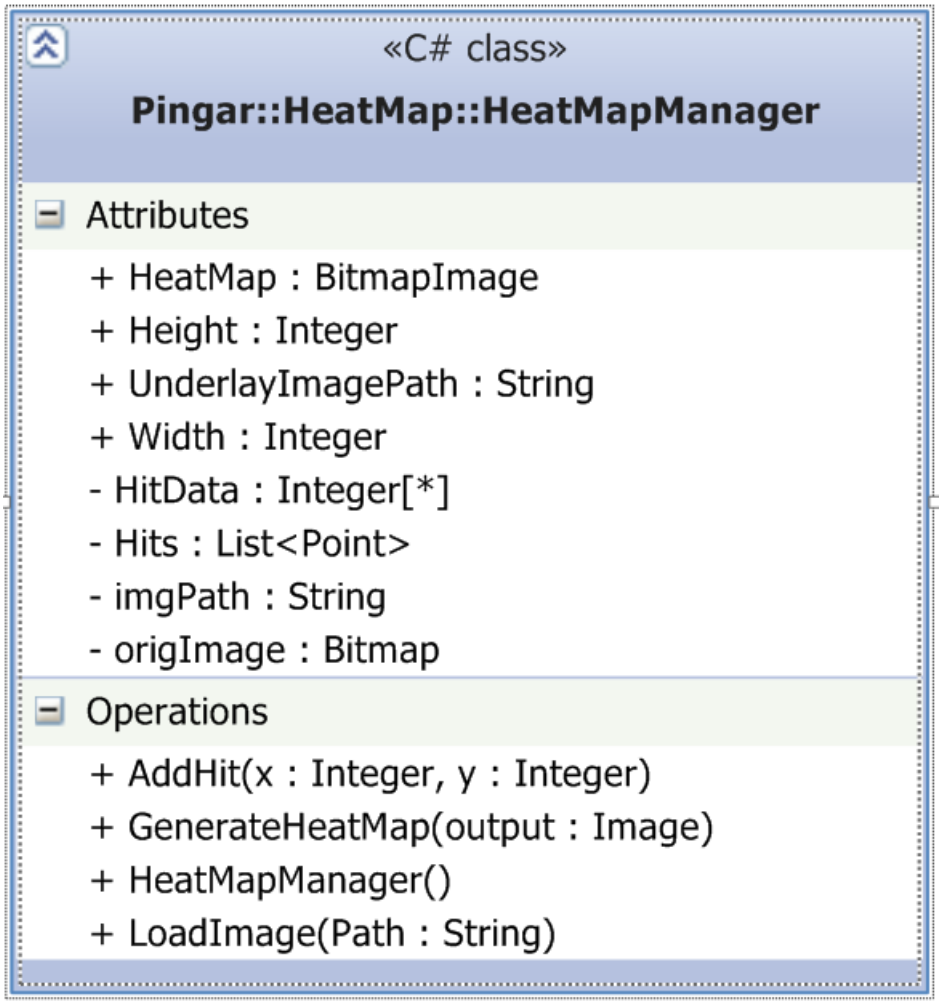

Figure 8.1: The UML diagram for the HeatMap Component.

8.3 Design Decisions

The primary design decision made here was to do with memory management. The process of generating the heatmap is fairly memory intensive and the process allows for many resource locks to be left open unnecessarily. We therefore ensured in our design that every effort was made to reduce and reuse as much allocated memory as possible. A primary example is the loading of the heatdot.png image file into memory. This occurs only once throughout the lifetime of the manager, but the image file is referenced thousands of times during a single HeatMap generation. This approach therefore save many cycles, since the class is not constantly loading the image file.

9 MouseTrail

Namespace

- Pingar.MouseTrail

Key Files

- USS/MouseTrail/MouseTrailManager.cs

9.1 Purpose

The MouseTrail component is designed to overlay a Mouse Trail visualisation on-top of a provided image using a collection of coordinates.

9.2 Design

The UML diagram for the MouseTrail component (Shown in Figure 9.1), shows that the component consists of a single class. The class can be thought of as a utility class. The class is informally divided into 3 stages:

1. **Load Base Image (LoadImage) -** During this initial stage, the base image (i.e. the image which is to be overlayed with the mouse trail visualisation) is loaded into the MouseTrail manager, and consequently into memory.
2. **Add Hits (AddHit) -** At this point the class expects for the points that the MouseTrail will visualise, to be loaded into the manager. A check is made to ensure that the Hit falls within the image boundary.
3. **Generate the Mouse Trail (GenerateTrail) -** During this final stage, the Mouse Trail itself is created. The process consists of drawing a successive number of lines between each point.

9.3 Design Decisions

There were no noteworthy design decisions made for the MouseTrail component.

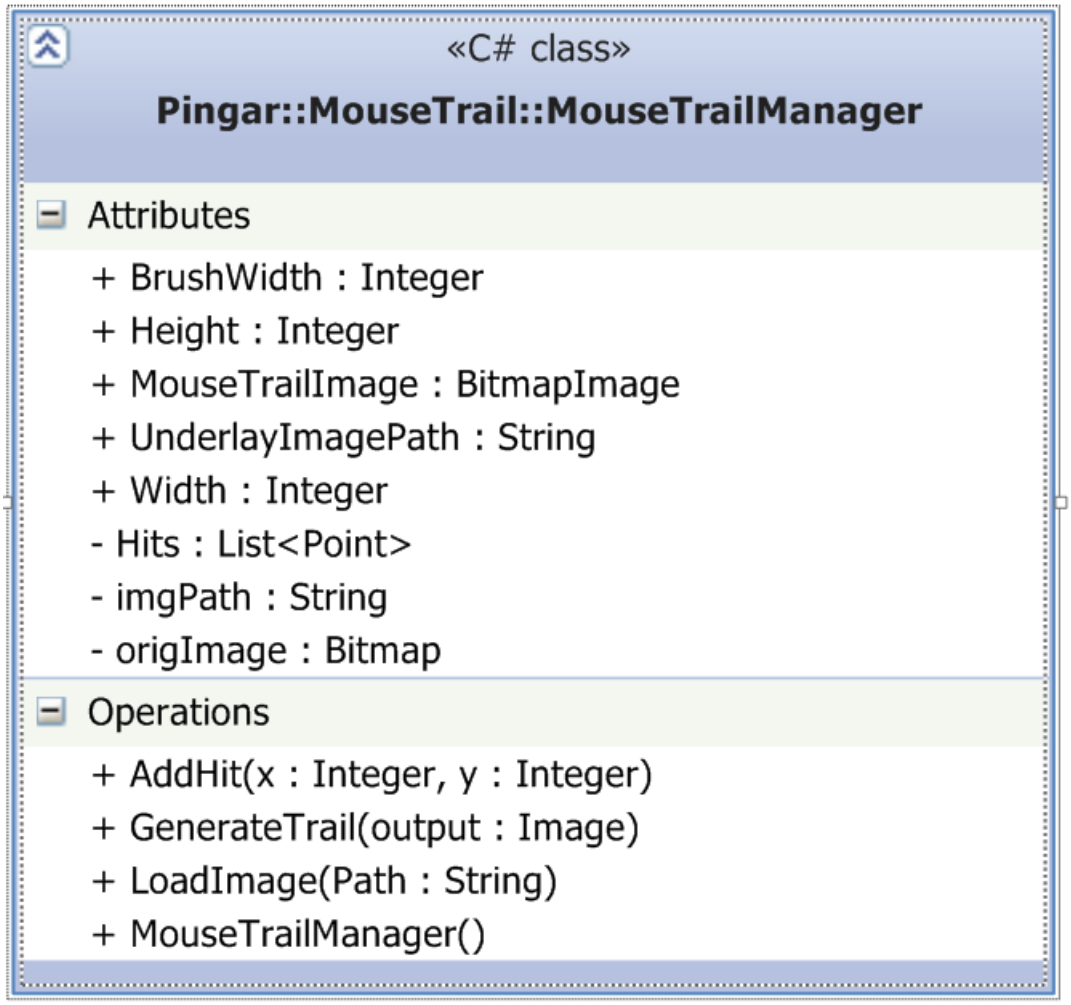

Figure 9.1: The UML diagram for the MouseTrail Component.

10 SoundPlayer

Namespace

- Pingar.SoundPlayer

Key Files

- USS/SoundPlayer/FFTDataSize.cs
- USS/SoundPlayer/ISoundPlayer.cs
- USS/SoundPlayer/IWaveformPlayer.cs
- USS/SoundPlayer/WaveFormTimeline.cs

10.1 Purpose

The SoundPlayer component is responsible for providing a visualisation of the audio recording track associated with the user study. The component must also provide a user interface which allows the Researcher to pause and play the audio track.

10.2 Design

The SoundPlayer component has a fairly simple design and is intended to be included within a WPF application. From the UML diagram representing the component, shown in Figure 10.1, we see that the majority of the component's logic occurs within a single class ***WaveformTimeline.*** The component was designed to integrate with WPF applications using the MVVM design pattern. As such there are many dependency properties which an application can hook into and utilise. For the most part however, using the class is fairly straightforward and requires only one single method call to setup the component. That method is - ***RegisterSoundPlayer,*** which takes a parameter of type ***ISoundPlayer,*** which is an interface type representing an audio playing engine.

The ***WaveformTimeline*** is self managing and calculates the waveform visualisation when a track is loaded into the object. The remaining data structure - ***FFTDataSize*** is an enumeration value that is used when calculating the amplitude of the waveform.

This component is based on the open source project WPF Sound Visualization Library (http://wpfsvl.codeplex.com/), but has been significantly re-written to handle multiple instances of the visualisation being open at once, along with other minor alterations.

10.3 Design Decisions

The primary design decision made here was the approach taken for data binding in the component. We followed standard MVVM practices here to ensure the component integrates easily into existing WPF applications. Lesser used, but simpler patterns, such as MVC and MVP, could have been used to simplify the development of the component, but could introduce additional integration complexity.

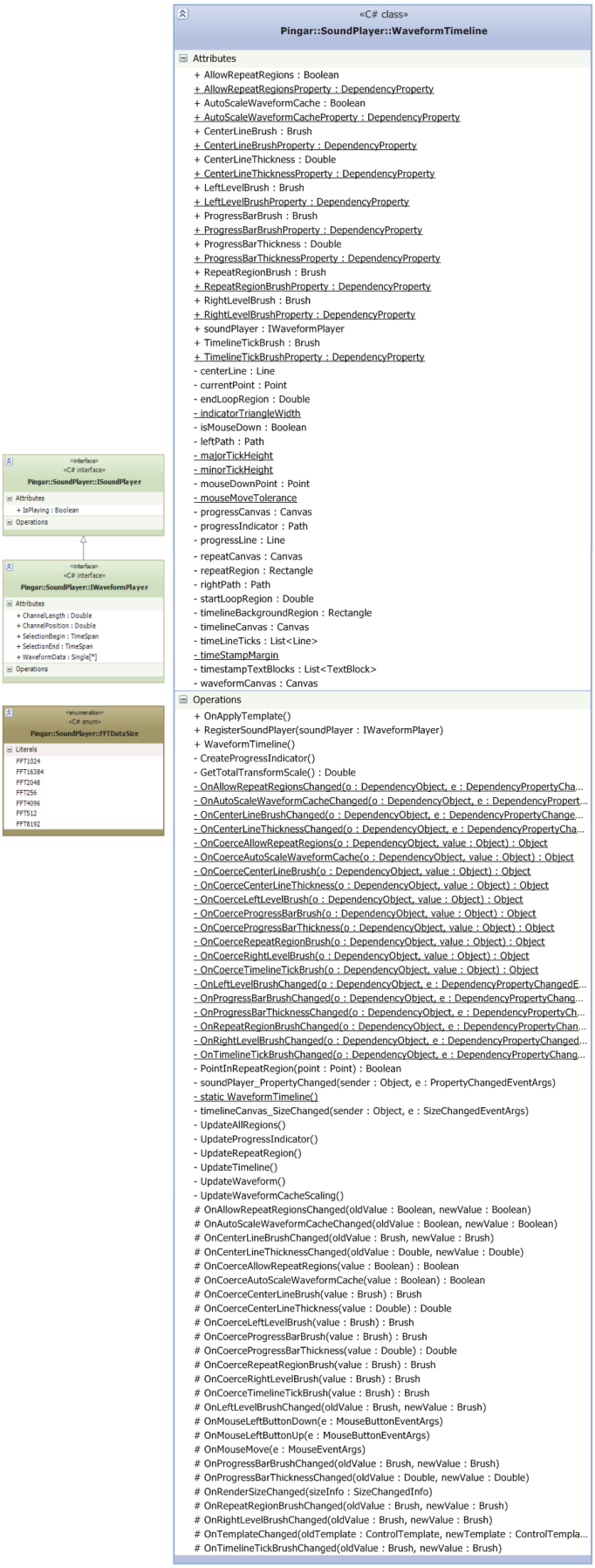

Figure 10.1: The UML diagram for the SoundPlayer Component.