Creating an interactive interface for writing and illustrating articles and essays

---

Craig Harbottle

MSc Computer Science

2014/2015

Word count: 11000

Supervisor: Dr Anna Jordanous

Contents

Introduction……page 4

Problem Area……page 4

Requirements……page 4

Aims and Objectives……page 5

Chapter 1: Research……page 6

1.1 Technology review……page 6

1.2 Literature review……page 11

1.3 Look and Feel……page 12

Chapter 2: Development & Design……page 13

2.1 Development Methodology……page 13

2.2 Design……page 17

Chapter 3: Implementation……page 18

3.1 Languages……page 18

3.2 Building the parts……page 19

3.3 Linking the parts……page 23

3.4 Changes……page 25

Chapter 4: Analysis & Results……page 26

4.1 Testing……page 26

4.2 Results……page 27

4.3 Analysis……page 31

Chapter 5: Evaluation……page 35

5.1 Successes and Criticisms……page 35

5.2 Further Research……page 37

5.3 Further Development……page 38

Conclusion……page 40

Bibliography……page 41

Appendices……page 41

Abstract

This project aimed to produce an interactive interface for the University of Kent Film Studies Department which would allow film students and academics to embed images, moving image, and audio clips within essays and articles. Furthermore, the interface would enable both author and reader to annotate the contents of a project — whether that be the central article or one of the supporting pieces of embedded multimedia. Such interactivity would therefore enable the writer to highlight and group the thought processes behind an idea and would also enable the reader to comment on and enhance these ideas.

Introduction

Problem area

Central to this project is a situation which animates and pervades all software development: the client with the problem. In this case the client is Dr. Aylish Wood of the University of Kent's Film Studies Department, and in this case the client's problem is the absence of a writing interface which would enable students and academics to illustrate their essays and articles with embedded video, sound-clips, and images. In written, text-based disciplines such as Literature, the author is able simply to quote and embed words and passages verbatim from the text being studied. This form of direct quoting is, save for the use of still images, a practice largely denied students of film.

Another key facet of the problem area is the need for the interface to enable authors and readers of an article to annotate the central text and all of its illustrative materials. Those annotations should themselves be available for annotation and be manipulable so that the author can highlight the development of and relationship between key ideas and groups of ideas.

Requirements

To help delineate the problem area, a requirements analysis occurred very early on in the project. Those requirements were outlined, clarified, and prioritised over the course of a series of meetings and email exchanges with the client. They were then formalised as two documents: a user requirements document (Appendix A), and a Use Case diagram (Appendix B). Both provided a blueprint - for the project supervisor - Dr Anna Jordanous, for the client, and for the developer - of the interface's principal functionality.

The requirements gathering process did also recommend potential development methodologies and these shall be discussed in future chapters.

Aims & Objectives

- *writing, editing of essays and articles*
- *functionality to enable placing of content within GUI*
- *functionality to embed multimedia within essays and articles*
- *functionality to annotate article and multimedia*
- *functionality to group and display key concepts within an article & show progression of thinking*
- *functionality to tag points of interest in a video clip as it moves*

The central objective of the project was necessarily the production of a piece of software which, ideally, provided a solution to all of the client's requirements. However, from this general objective, priority was placed on firstly producing software which enabled users to write articles and essays, and to annotate them with illustrative multimedia. Once this objective had been met, there would be a focus on implementing the required annotation functionality. Such prioritisation was necessary, due to the large scope of the requirements, to the inherently difficult nature of some of those requirements, and to the time available. The fourth and sixth bullet points above were, for example, perceived as being challenging to implement. This was an observation based on research into existing, similar projects, about which there shall be more in Chapter 1.

A name

The client eventually decided to name the software '*Bubble*'. This was inspired by the idea of the 'thought' bubble and would reflect something of the nature of the overall concept of the software.

Chapter 1: Research

1.1 Technology Review

Research into existing technologies centred on analysis of word processors, annotation tools, and media players. Both commercial and open source products were considered, with analysis focusing on functionality but also on GUI design as look and feel would be a key consideration. During the review, there was also a special focus on identifying open-source technologies whose functionality might be adapted for use in *Bubble*. The review encompassed a plethora of such technologies but only those seen as pertinent are discussed here.

word processors

Scrivener

In the preliminary meetings with the client, the writing software *Scrivener* (Appendix C) was mentioned as an application which the software might be modelled on, in terms of its GUI design and some of its functionality. Indeed, there was much about *Scrivener's* layout which was transferred to the design of *Bubble's* interface. Most obvious is the inclusion of a Project Files window similar to *Scrivener's* 'binder' section. This seemed an ideal way for user's to organise and manoeuvre through their projects and the different types of files each project will contain. *Scrivener* also inspired thinking about screen organisation. Like *Scrivener*, *Bubble* would be divided into sections, with each responsible for its own type of media or functionality. *Scrivener's* 'comments & footnotes' section, for example, was the inspiration for *Bubble's* Annotations window. Although much of the screen organisation seems in keeping with received wisdom on designing GUIs (Preece, 1993, p.82), it is useful to observe it working successfully in a similar interface.

There are, though, many aspects of *Scrivener* which would fail to meet the client's requirements. The entire GUI was, for example, housed within a single window and not

manipulable as single, stand-alone windows. Furthermore, while their was support for multimedia in the software, their was no functionality to read or write an article and experience embedded media concurrently. Whenever a hyperlink is clicked in a *Scrivener* article, the central window, which houses the text document, is replaced by a viewport for the media. Finally, and most importantly, their is no support for multimedia annotation within *Scrivener*. Clearly a deficiency in terms of the client's requirements.

annotation tools

Mae Annotation

Mae Annotation (Appendix D) is a tool designed for linguistic analysis of texts. Written in Java, it was designed to enable users to annotate parts of a text and to make cross references between those annotations. This sort of functionality, to cross-reference ideas, was close to that described by the client and which were stated in the user requirements (Appendix A, no.5). *Mae Annotation* is open-source and it was thought that this tool and some its ideas could be adapted to develop the text annotation function of *Bubble*. In its original form, though, the tool did not offer the ability to annotate texts with extensive comments. Furthermore, the tool was hindered by a cumbersome file loading process. This involved first loading an XML document which would contain the annotation metadata. Once loaded, the user would then need to open the '.txt' file which contained the text and would host the annotations contained within the XML document. This loading process was perceived as a problem and to achieve the sort of ease of use required by the user, it would need refining. The process by which annotations were made also proved problematic. Part of the code which dealt with this functionality contained a bug resulting in temperamental performance. This would also need refinement.

Transana

Transana (Appendix E) is a transcription tool enabling users to load and transcribe the contents of audio, video, and still images. The entire interface is written in Python and supported by an SQL database. The project is open source with source code available through GitHub.

The layout of *Transana* fit very closely with the user's final, required layout for *Bubble*. The interface is made up of several moveable, resizable windows, including a project files window where all files are organised in a tree view. It was useful to have observed this in design terms, and it was also beneficial to have had the source code available to analyse how the layout was written and furthermore how the inter-window communication within the interface was achieved. This last point took on increased significance as it became apparent that multi-window GUIs, in contrast to single window GUIs, are much more challenging to write and coordinate the data of.

That *Transana* contained two of the key features - a text editor and a video player - which the user required made it a candidate for the basis of *Bubble* . However, neither of these components fit exactly what was needed. The text editor had limited functionality for the needs of *Bubble*. The media player, too, would not be fit for purpose as it supported only a limited, out-of-date set of codecs (H.263) and file types (MPEG1). Use of both the text editor and video player would have required considerable improvements to make them suitable for the requirements of the project.

Viper

One of the most challenging user requirements was to implement a functionality to enable users of *Bubble* to highlight objects in a frame of a video clip and then to be able to track that object for the duration of the clip. Research highlighted *Viper* (Appendix F), a video annotation tool which enables objects within a video clip to be tracked by using 'bounding boxes'. The user would highlight the object they wanted to track and annotate at the start

of a video. They would then highlight that same object's end point in the video. The movement of the object would be interpolated from key frames between the user-defined object start and end point.

As the entire project was open source, *Viper* was potentially an ideal tool to implement within *Bubble*. However, a key challenge in achieving this successfully would be to streamline the process of loading video files into the software, ready for annotation. Within *Viper*, the user was required first to load an XML file which would store the annotation metadata. The user would then need to load the video separately. This was, arguably, far too cumbersome. *Viper* also suffered from having been written in the Java Media Framework, an API which supports only a limited number of codecs and file types. Another challenge would, then, be to update the bounding box functionality to make it compatible with *Bubble's* media player and the associated codecs and file formats.

Annotator

Annotator (Appendix G) is an open source annotation tool, written in Javascript and built for use in web pages. The tool is a simple component, requiring the user to click on a piece of text to be annotated. Once highlighted, the text would produce a pop-up window containing a text area for the user to add an annotation. Once the annotation had been completed the text would remain highlighted to indicate an embedded annotation. The clean, minimal design and simplicity of use recommended it as a possible tool to use in Bubble. Furthermore, its seeming flexibility made it appear highly adaptable to any text.

However, considering *Bubble* would be a standalone, offline application, written in Java, the use of Javascript did present a problem. Although there exist techniques to embed Javascript within Java, there was a worry about performance issues. With Java 8, however, Oracle released *Nashorn*, an official Javascript engine designed to integrate Javascript with Java. This could, then, make it possible to embed the *Annotator* tool within *Bubble*.

Another concern with using this tool was how annotations would be saved. In its current use, the tool makes use of a server to 'rewrite' annotations back to the html of the web page in which the *Annotator* plugin is embedded. User annotations are, then, stored within a new <p> tag, nested within a new element. For this project that implementation would be unfeasible as the software being built would be standalone and offline. As to how this functionality could be reproduced would be a key challenge.

Finally, a problem with *Annotator*, for our project at least, was that in its current use it is, by design, static. The plugin must be embedded as part of a webpage's html which must be set up with the text which will be set for users to annotate. How, then, could this be adapted to make it dynamic? Could it be adapted to enable users to load files, as is required in *Bubble*?

Annotorious

Annotorious (Appendix H) is an open source image annotation tool, based on the *Annotator* tool discussed above and so almost identical in design and functionality. The obvious difference with this plugin being that annotations are made on images.

Consideration of how to use Javascript within Java and how to save annotations within Java were again important considerations, as were the ability for users to load images they want to annotate.

Open Video Annotation

The *Open Video Annotation* (Appendix I) project again builds on the work of the *Annotator* plugin. This particular implementation enables users to annotate video clips. As with *Annotator*, the design of the tool is simple and lends itself to be easily embedded within the media player of *Bubble,* but the compatibility of Java and Javascript would be a challenge, as would be adding the ability for users to load video files rather than them being statically embedded in html.

media players

There are a variety of existing 'out-of-the-box' media players which could have been implemented within *Bubble*. However, it was felt that the media player should be built from scratch to ensure a full understanding of the underlying code and functionality. Such an understanding would make it easier to add the requisite annotation functionality and to integrate with the other components of *Bubble*. With this precondition, a search was made in to how such a media player might be built. During this search JavaFX was identified as a library which could be used. A part of Java 8, JavaFX is a library designed for building GUIs and tools such as multimedia components such as media players.

1.2 Literature Review

The literature review focused on texts which might inform thinking about the cognitive process of writing. This could then, it was hoped, influence design decisions. *The Fluid Text: A Theory of Revision and Editing for Book and Screen* was an informative text in this regard. The author presents the idea of the 'fluid text', arguing that works of literature, specifically, are not static, and that their meaning is in fact subject to change depending on a variety of factors. Although the book has no direct technical value, it does provide an excellent conceptual model for how to think about *Bubble* and the sort of texts it should enable writers to produce. It also provided an insight into the way that digital annotation is being used to enhance the reading and writing process.

To inform thinking about GUI design two key texts were reviewed. *Human Computer Interaction* (2011) and *A Guide to Usability* (1993). While the first offers a simple overview of best practises, the second offers some detailed insight into the cognitive and memory processes of interface users. Several chapters describe different types of computer users and how systems can be tailored to suit them. This will undoubtedly be useful for *Bubble* as one of the key aims is to produce a piece of software which maps the cognitive processes of writers.

Research into current digital annotation standards was also undertaken. The *Open Annotation Data Model*[1] specifies a set of guidelines for the annotation of digital resources - "an interoperable framework for creating associations between related resources, [and] annotations […][2] ". This framework covers issues such as the provenance of an annotation and the identification of annotations through digital signatures. Although not in the client's requests, it will be important to attempt to implement this data model and its recommendations to ensure that all annotations created within the software are W3C compliant.

1.3 Look and Feel

Although the user requirements make no explicit mention of the look and feel of the software, they were, based on early discussions with the client, clearly a consideration. It was important then to produce a piece of software which looked the way the client wanted, and which functioned in they way they wanted, but to do so in a way which would not divert from the main task of completing the central requirements. To achieve this end, prototypes were used from very early on to give the client a 'real' experience of what the end software would be like. The GUI illustrated in Appendix J represents one such prototype, and one which actually formed the basis for the final design of *Bubble*. These prototypes were useful in keeping the client updated and in providing a way of gleaning design preferences. Indeed, the use of the aforementioned prototype during an early meeting enabled the client to further clarify the design requirements so that the prototype could be adapted to more closely fit what was required. While the minimal colour scheme was in keeping with what was required, the one window layout was not.

Another method used to hone the required look and feel was to suggest existing software as possible models. *Final Cut Pro* (Appendix K), a video editing tool for Mac, was

---

1 http://www.openannotation.org/spec/core/

2 http://www.openannotation.org/spec/core/#Aims

one such example used to define the look and feel of *Bubble*. This particular example was, in fact, key in solidifying the client's concept of a multi-window GUI (ability to place, move blocks of content).

These early meetings, the dialogue arising out of them, and the use of prototypes to frame the discussions were crucial in helping to refine exactly what the client wanted in terms of look and feel. Indeed, the first prototype which was produced (Appendix J), and the end version (Appendix L) indicates some of the disparity between the developer's conception of what the look and feel should be and what in fact the client's required look and feel was.

As Bubble would primarily be a writing tool, it was important to design the GUI so that the user's experience of writing would not be unduly affected. It was believed that there could be a cognitive burden created by the 'clutter' of the software's ancillary components. Ideally the user should only have to focus on those tools essential to their writing. The use of a multi-window tool was ideal in this regard as it would enable superfluous windows to be minimised. Furthermore, the TextEditor window could, it was thought, be enhanced by adding a 'full-screen' editor which would create a more 'immersive' writing experience where the ancillary windows and desktop are concealed. This was an idea inspired by a feature used in *Scrivener* (see above). While simple tweaks, they would, it was hoped, ensure the user's writing experience would be approximate to that when using any other word processor.

Chapter 2: Development & Design

2.1 Development Methodology

With *Bubble* being built solely by one developer, there might have been an instinct to just begin coding, and not to consider which, if any, development methodology should be

chosen to guide the development process. On the contrary, from a very early stage it was paramount to establish a way of working which would, firstly, enable development to take place in a fluid, exploratory, and creative manner; which would, secondly, enable an open, discursive dialogue between the developer and the client; and which would, thirdly, allow for the work of the developer to be monitored by the project supervisor. However, selecting and following just one development methodology to facilitate these goals proved difficult and, arguably, erroneous. The project fluctuated due to a variety of factors and did, as a result, make use of techniques from several development methodologies.

The process of considering a single development methodology with which to build a piece of software did in fact highlight a variety of intriguing questions about the very nature of development methodologies, about their usefulness and validity, and about whether it is the project, with its inevitably inconstant, ever-changing idiosyncrasies which defines and creates the illusion of a development paradigm rather than there being any coherent, 'off-the-shelf' methodologies.

What follows is a list of methodologies used during this project, with a note on how aspects of their ideas influenced the development process of *Bubble*.

Component-based development

Due to the large scope of the project, the demanding aspects of some of the user requirements, and the time-scale, a component-based approach was both desirable and probably inevitable. Much of the research - as outlined in Chapter 1 - focused on sourcing components which already contained the functionality required by the user. One such component was that which formed the basis for the Project Files window. Although requiring a small amount of reverse-engineering, it was overall reasonably easy to adapt to the project's needs. A second component used within the project was the *Annotorious* image annotation tool described in Chapter 1. This particular component required a

significant amount of work to implement and is, as of yet, not fully implemented in the way outlined by the client. This failure does, perhaps, point a key problem with the component-based approach. Readily available solutions are rarely tailor-made solutions.

Agile development

In initial meetings with both the project supervisor and client, there was an emphasis on using an Agile methodology to develop the software. The supervisor would act as the 'scrum master' and would orchestrate and monitor the incremental delivery of key functionalities over the course of several 'sprints'. The first of these would see the delivery of the image annotation feature. The advantages of this approach to our software were many. Firstly, the process of producing complete, detailed documentation was minimised and only those documents thought crucial to the development process and to its end user would be created. Secondly, the client would be heavily involved in the process of building the software and would be available to evaluate and propose changes to subsequent versions.

However, quite early on it became apparent that it would be important to build the software from a more secure structure and code foundation than a fully Agile development might allow. The software would be, for example, composed of several, intimately related, crucially inter-dependant, yet different parts. It was thought that producing each of these 'parts' separately, in isolation from their companion parts, without careful thought as to how they would link together, could compromise the structure of the software, in many places damage its functionality, and also hinder its future development. Therefore, a more careful, planned methodology was needed.

The confidence and inexperience of the developer also influenced the decision not to follow the Agile approach completely. To have begun work on producing key requirements such as the image annotation or video annotation, as part of a 'sprint', was perceived as

too much of a challenge. Failure to have fulfilled these requirements, at the early stages of the project, could have been demoralising. Truly Agile development was, it was thought, the preserve of experienced programmers, not relative novices. A careful, planned approach was also required to enable the developer to build confidence.

Nevertheless, many aspects of the Agile methodology were used throughout the project. Indeed, supervision followed a scrum-pack dynamic, enabling work to be monitored and achievable milestones to be set in an interactive, discursive way. Also, the relationship between developer and client was dialogic, enabling the software to be refined as the project developed. There was an emphasis on producing 'versions' of the software for appraisal. However, those versions did not deliver key functionality quickly - something which typifies software developed in a truly Agile environment. Testing, too, followed an Agile approach as elements of the software were tested as and when they were created to ensure their rigour in isolation and as part of the whole application.

Another key aspect of the Agile methodology which was useful to the project was its emphasis on creativity as part of the process of software development. Indeed, this freedom afforded the opportunity for consideration and, in some cases, inclusion of enhancements not included in the original specification. The autosave and proposed export to multi-media PDF functionality, for example, were born out of necessity, and the developmental freedom which an Agile approach allows.

The Waterfall model

Although in no way can the development of *Bubble* be said to have followed the Waterfall model, it did, without doubt, follow some of its principles, if not by the letter, then in tone. As has been noted, the project required an approach which would allow the software, and the interdependence of each individual functionality, to be considered and designed holistically, as part of the whole. Agile methods, with its emphasis on delivering

key functionality quickly, could affect that. The Waterfall model was useful, then, in its emphasis on “establishing an overall system architecture” (Sommerville, p. 31). However, planning of this overall architecture did not involve the continued production of formalised design documentation which would need to be approved, as such time-consuming practises were impractical. Rather, the overall structure was developed mostly from an informal specification, and formalised through continuous implementation and testing. That is not to say that some necessary yet minimal documentation took place. Clearly this approach is only possible were projects are being developed by individuals and small groups.

It could be argued also that the Waterfall model’s idea of moving a project through ‘phases’ which are ‘signed off’ (Somerville, 2011, p.31) was evident in the project’s emphasis on having the software appraised by the client at every possible step. Although, far less formal than the client-developer relationship outlined in the Waterfall model, ours did provide the project with a sense of purpose and focus.

2.2 Design

The software structure was to a large extent dictated by the client’s request for a multi-window GUI. This requirement enabled development to treat each key component (ProjectFiles window, TextEditor window, Annotations window, ImageViewer window, MediaPlayer window) of the software as a single Class and forced a certain Class coherence. The ImageViewer window would, for example, deal only with activities concerning the displaying of images. Appendix M shows a table which was used to plan and design the content and behaviours of each window and how each window would interact.

Design of the software also aimed to follow the Model-View-Controller (MVC) design paradigm. It was thought that this model would provide a helpful way to visualise the structure of the software and to conceptualise exactly the role of each part and how it

would interact with other parts of the software. Future development would also, it was hoped, be eased by the use of MVC. As well as providing a model, it did also form the thinking behind very real implementations. Indeed, the presence of 'Controller' classes in the final software was inspired by the MVC model and, as shall be discussed presently, crucial to the implementation of the multi-window GUI.

Appendix N shows a Class diagram which provides an overview of the final software structure. The final structure did not differ from the original conception (Appendix O), apart from the addition of several ancillary Classes such as the 'JavaScriptBridge' Class. As can be seen in the latest diagram, adherence to the MVC paradigm was not fully refined, with elements of the 'model' and the 'view' still combined as 'model_view'.

Chapter 3: Implementation

The implementation of the software followed a mostly linear path, dictated in the large part by the client's request for a multi-window GUI. The process was, then, a case of building each window individually, testing it, and then building the next window. Once all the windows were completed and offered the requisite functionality, they were then linked together and work on the annotation features was begun. Exactly how that process was completed, and the tools used, shall be discussed presently.

3.1 Languages

Java was chosen as the base language with which to build the software. The obvious benefit of programming in Java is its portability. Although developed primarily for Mac OS, the software could, it was hoped, be easily adapted to run on other platforms. Furthermore, Java offers support for multithreading which would be relatively easy to implement. This would be essential, as the multi-window GUI, with its multimedia

capabilities, would make significant demands of the CPU. Multithreading would also prove invaluable to enable features such as the auto-saving of text documents.

Another reason for choosing Java was the opportunity to use its more recent software platform JavaFX. As has been mentioned, JavaFX can be used to build and manipulate multimedia components with relative ease, and this would be essential to the project. JavaFX also enables the styling of components with CSS. For the required look of the software, this too would be essential and therefore made JavaFX an useful language for this project.

As well as Java and JavaFX, the project utilised HTML and Javascript. Exactly how shall be covered in the following sections.

3.2 Building the parts

ImageViewer window

The first window to be completed was the ImageViewer (Appendix L, bottom left), albeit without the annotation functionality. That would come later, once the other windows were completed. It was important that the ImageViewer enabled the use of several, current image formats and their variant file extensions to be viewed and so support for ‘png’, ‘jpeg’, and , ’jpg’ was implemented. The loading and display of these files was made relatively simple by the use of the JavaFX ImageView Class. Images were opened by simply passing an image object to the constructor or by using the ‘setImage()’ method.

There were some issues with preserving the aspect ratio of imported images while keeping them centred within the viewer window. This sort of aberration would, it was thought, have seemed odd to the user and so small effort was invested in correcting the problem. Eventually it was solved by a combination of replacing the Border Layout Class

with the more dynamic Grid Pane layout, and by use of a simple method call to 'setPreserveRatio()'.

MediaPlayer window

The MediaPlayer (Appendix L, bottom right) followed a similar course of implementation as the Image Viewer. Again, no annotation function would be attempted at this stage as it was perceived as being too demanding to implement. However, instead of an ImageView, the MediaPlayer makes use of the MediaView Class, another default JavaFX Class. To display media, a simple call to 'setMedia()' could be used, or the media object could be passed to the constructor.

Only 'mp4', 'm4v', and 'mp3' files are supported within the MediaPlayer. However, it was thought that these would be adequate and were representative of currently used file types. One useful feature of the MediaView Class is its inbuilt ability to manage codecs and to automate the reading and loading of media files. If the developer wanted, however, they could customise the codecs to optimise playback. Although for this project, and its timescale, it was not possible.

TextEditor window

Without doubt the most demanding and therefore most time-consuming window to complete was the TextEditor window (Appendix L, top-centre). As this component was integral to the overall functionality of the software - more so than any other component - a significant amount of work was expended in getting it right. Its final form, however, was much different to the initial conception, and the decision to deviate from that initial plan was not taken easily. It was, ultimately, a pragmatic decision.

The original plan for the TextEditor was for it to use a common, portable file type, one which enabled styling of texts and creation of hyperlinks. The styling of text was important

as the TextEditor must offer the features a user would expect of a basic word processor and which are essential in writing essays and articles - functions such as text alignment and ability to change font size and type. The use of hyperlinks would also be integral, as it would enable users to embed illustrative media in their articles and essays. Research into suitable file formats recommended the 'rtf' file format. Not only did it offer styling capability, it also supported hyperlinks. Research also highlighted the Java 'RTFEditorKit', a complete set of tools to enable the reading and writing of 'rtf' files. With these benefits, the 'rtf' format was chosen as the base file type for the TextEditor. There were future plans to implement a function to enable users to export articles to different file types but for use, locally, within the software, the 'rtf' was thought to be ideal.

After building the 'rtf' TextEditor (Appendix J, top-centre) and adding the styling capabilities, work began on implementing the hyperlink functionality. After several attempts it became apparent that such a feature would be more demanding than research had suggested. Indeed, while it is possible to implement hyperlinks within the 'rtf' file format, to do so using Java's 'RTFEditorKit' would require the kit to be almost completely rewritten and customised. It would also require a good knowledge of the actual code underpinning the 'rtf' file format. To have completed this customisation would, it was observed, have been a huge undertaking and one thought unfeasible within the project's time-constraints.

A workaround was found by use of a combination of the HTMLEditor and WebView JavaFX components[3]. Casting the HTMLEditor as a WebView enabled the WebView's 'getEngine()' method to be called on the HTMLEditor. This created a Javascript engine. With this engine Javascript functions could now be called on JavaFX objects and ultimately used to get and manipulate user selected text from the HTMLEditor. That text could then be returned to the HTMLEditor, formatted as a hyperlink. As the HTMLEditor

[3] Research into using the HTMLEditor for the TextEditor lead to these posts: http://stackoverflow.com/questions/17555937/hyperlinklistener-in-javafx-webengine and https://rajeshkumarsahanee.wordpress.com/2014/11/01/hyperlink-button-in-htmleditor/ Both of these were instrumental in adapting the HTMLEditor to add hyperlink functionality.

was cast as a WebView, it therefore has the capacity to display web content in the form of html, meaning it could display that newly Javascript-formatted hyperlink. However, the HTMLEditor still retained its native ability to enable the editing of text.

Furthermore, the HTMLEditor supported the basic styling of text which would be more than adequate for writing essays. Although a departure from the initial plan, the combination of HTMLEditor and WebView did provide the functionality required. If only for the present it was a sufficient, if inelegant, solution.

ProjectFiles window

To make the implementation of the ProjectFiles window (Appendix L, top left) easier, an open source component was used. The component was firstly reverse engineered and then adapted so it could form the basis for the ProjectFiles component. In its original form, the component enabled users to navigate their computer's filing system and to create or delete files. The current version, however, demonstrates three significant and necessary changes. Firstly, the component was adapted so that users could create, import, and open project folders within the file viewer. The user could then import files into these projects and also create new files. A second change was to add a functionality to enable users to load files into their associated window by clicking on the file name. The original component did not allow this ability to 'open' files.This functionality was further enhanced to automatically load files to the parent window once a project is created or opened, and to display that file name at the top of the parent window. This was essential in enabling the user to keep track of the file they are currently working on. In an earlier version this was not the case, and this would have, for example, potentially seen users writing at length before realising a document was not loaded. Finally, the component was adapted to house the autosave function which would ensure all text documents are autosaved.

Annotations window

Full implementation of the Annotations window (Appendix L, top right) did not occur until after the ProjectFiles, TextEditor, ImageViewer, and MediaPlayer windows were completed and linked together. Much of the functionality of the Annotations window relies on those windows mentioned and, particularly, on the TextEditor window which is where, for the most part, annotations are created or 'opened'. However, the template for the Annotations window was set out using separate tabs to house the different forms of annotation (article annotations, image annotations, film annotations, annotations about annotations). A tab to host user's notes was also added which, although not specified in the requirements, was mentioned by the client in several meetings.

3.3 Linking the parts

Having built the basics of each individual window, the task now was how to link them. It was envisaged that 'Controller' Classes could be utilised to facilitate this communication and enable data to be passed between the windows. How to implement that concept, however, presented the real challenge. Eventually, it was realised that the Controller Classes should be ***final*** and that their fields and some of their methods should be ***static*** (Figure 1). That way, no instances of the Controller Classes could be created, ensuring that all data stored within and passed from them remained constant. There would, then, be Classes whose sole purpose would be to store and distribute data required for the successful communications between each of the windows. Each Class which needed to extract data from a given Controller Class would need only call methods from the ***static*** Controller type. A representative call from the ProjectFiles window to open a new article in the TextEditor window would read as follows:

'TextEditorWindowController.getTextEditorWindow().setFile(file)'.

Initialisation of the Controller Classes occurred within the AnnotationTool Class - the 'main', entry Class which deals with the initialisation of each window. A window object was

passed to its Controller's 'init()' method ensuring that it would have a Controller attached to it which would be responsible for its communication with other windows (Figure 2).

The implementation of these Controller Classes was a huge breakthrough and crucial to fulfilling the client's requirements for a multi-window GUI. Failure to have completed it in this way would have meant having to compromise on the design.

```
public final class AnnotationsWindowController {

    private static AnnotationsWindow annotationsWindow;
    private static Stage stage;

    public AnnotationsWindowController(AnnotationsWindow annotationsWindow ) {
        AnnotationsWindowController.setAnnotationsWindow(annotationsWindow);
    }

    public static AnnotationsWindow getAnnotationsWindow() {
        return annotationsWindow;
    }

    public static void setAnnotationsWindow(AnnotationsWindow annotationsWindow) {
        AnnotationsWindowController.annotationsWindow = annotationsWindow;
    }

    public void init(Stage annotationStage) {
        this.setStage(annotationStage);
    }

    public  void setStage(Stage stage) {
        AnnotationsWindowController.stage = stage;
    }

    public static Stage getStage(){
        return stage;
    }

}
```

Figure 1 sample of the Controller Class

```
final AnnotationsWindowController annotationsWindowController = new AnnotationsWindowController(annotationsWindow);
annotationsWindowController.init(annotationStage);
```

Figure 2 initialising the Controller Class

3.4 Changes

As is perhaps the case with any software project, how the software was intended to be built is not the way it is built. So was the case with this software and the project benefited from several serendipitous discoveries, most notably with the ImageViewer. From having to adapt the TextEditor window to use html as its base format, and to use the WebView Class to enable that, it was observed that the image ImageViewer Class could too make use of the WebView Class to 'host' images by having them embedded in a html file as <img> elements. If this were possible, it was thought that the images, as they would be embedded in html code, within <img> tags, could also be easily accessed and manipulated by using Javascript. This would eliminate the need to use the Nashorn Javascript engine to enable Javascript to be called on Java objects. The use of the *Annotorious* (see above) Javascript plug-in would then be much simplified. Having adapted the ImageViewer to use a WebView Class instead of the ImageView Class, the implementation was achieved. The *Annotorious* plugin was placed in the application package, along with its associated CSS file. Images imported into the ImageViewer were embedded in a html file. To enable the *Annotorious* plugin to access the image, the <img> 'class' attribute was set as: **class="annotatable".** Whenever an image was loaded into the ImageViewer, the user could activate the *Annotorious* plugin by clicking and dragging on an image. The plugin's JavaScript would then be activated, launching a dialog box in which users could make annotations about the loaded image.

4.1 Testing

Both verification and validation testing were conducted as an integral part of the project. There was an emphasis on ongoing validation testing to ensure that the "right product" (Somerville, 2011, p. 207) was being built for the client, and this was sought mainly through the use of prototypes for the client to use and appraise. This is evident from the very first meeting were a prototype was used to demonstrate the design of the GUI. Feedback on this version enabled changes to be instantly made. Validation followed this form throughout the project with each subsequent 'version' inspiring instant feedback and so enabling development to stay focused and produce the requisite software.

This approach was also complimented by using validation techniques whenever a component was completed. It would firstly be tested in isolation to ensure robustness and expected functionality. Once the component's behaviour was consistent with what was expected, it was then tested within the context of the whole software, again to ensure robustness and ultimately to ensure that the "right product" was begin built.

A final way validation of the software was achieved was through the use of user testing. The software was given to a group of six testers typical of the people who would use it - academics and students with a connection to film. The testers were required to spend several days using the software and the user guide (Appendix P) and then complete a questionnaire (Appendix Q) which would glean feedback on their experience. The questionnaire was composed of multiple choice questions, and a mix of closed and open questions. It was hoped this mix would produce a variety of feedback - both qualitative and quantitative - as to the successes of the software and possible improvements. Also, all

questions were based on set of criteria outlined by the *Software Sustainability Institute*[4]. This would ensure that test criteria was in keeping with industry standards.

Verification was also a continual process rather than an activity completed at the end of the project. Dynamic verification techniques were a necessity and necessarily utilised at every stage of the project to encompass as large a scope of the code as possible. These took the form of simple unit testing of methods and Classes to ensure expected behaviour and required interoperability with their associated Classes. They also involved the wider testing of larger, more substantial 'modules' such as completed windows. From this modular testing, integration testing was performed to ensure that, for example, the ProjectFiles window performed as expected when working with its associated windows. Finally, once separate modules were proven to have performed as expected with each other, the overall system was tested to ensure its holistic behaviour was consistent with what was expected of the software.

Static verification techniques were also instrumental from the outset, and there was an emphasis on ensuring code was well-commented, clear, and followed Object-Oriented principles. There was also a recognition that the software should adhere to a wider design model to make the code structure more robust and so facilitate future development. To this end the model-view-controller paradigm was chosen and partially implemented.

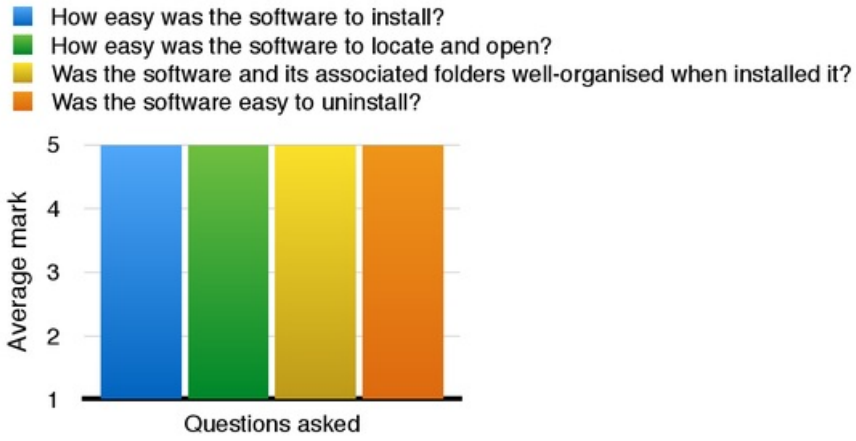
4.2 Results

'Results' here concern the feedback gathered from the questionnaires used in user tests. Those results are organised in the same categories as they were grouped in the questionnaire and are presented as a set of graphs illustrating the average mark for each question. The mark relates to a number which reflects the strength of tester's feelings concerning the question asked, with 5 meaning 'very' or 'highly' and 1 meaning 'not at all'.

[4] http://www.software.ac.uk/

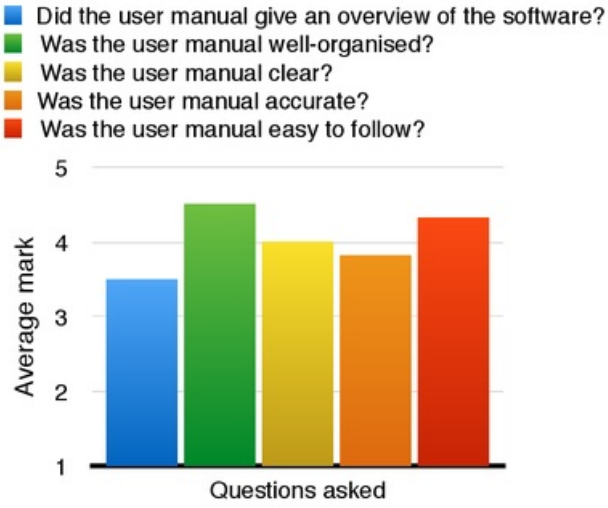
There is also a brief summary of the key comments from answers to those questions designed to extract qualitative feedback.

Installability



No additional comments are made by testers as to how the experience of installing, locating, opening, and uninstalling the software could be improved.

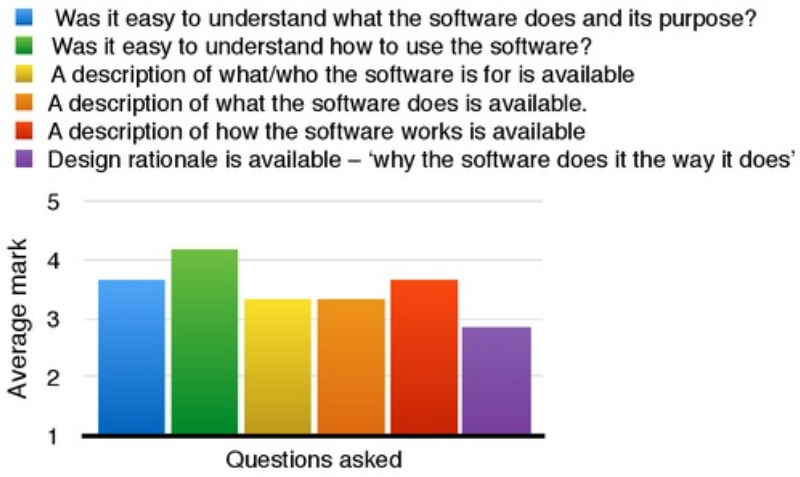
Documentation



Overall, testers remarked how the manual was mostly clear and logically ordered. However, there were comments recommending the addition of a troubleshooting section.

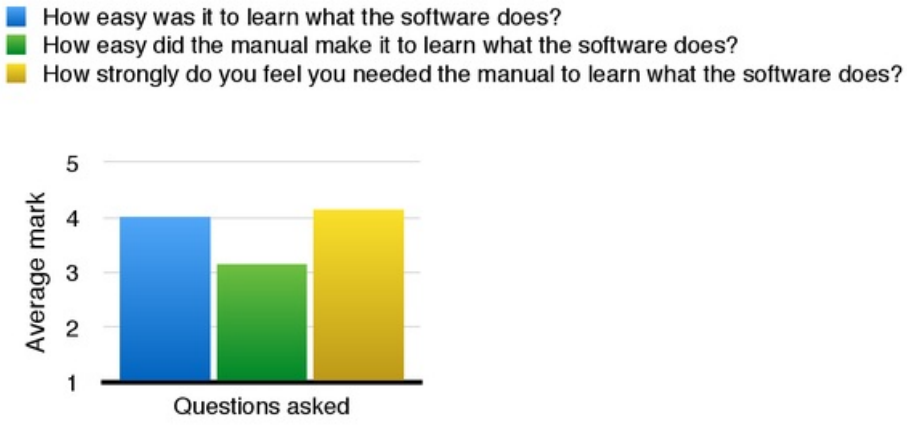
One user has an issue with deleting annotations and a troubleshooting section would have been useful for this. Two testers made comments about the need for the manual to provide more detail to inform future users about precisely what the software does. Was it primarily a reading tool? Was it primarily a writing tool? One tester also commented on the need for more clarity on how documents would be read once completed. Were they meant to be read in the software or where there other options?

Understandability



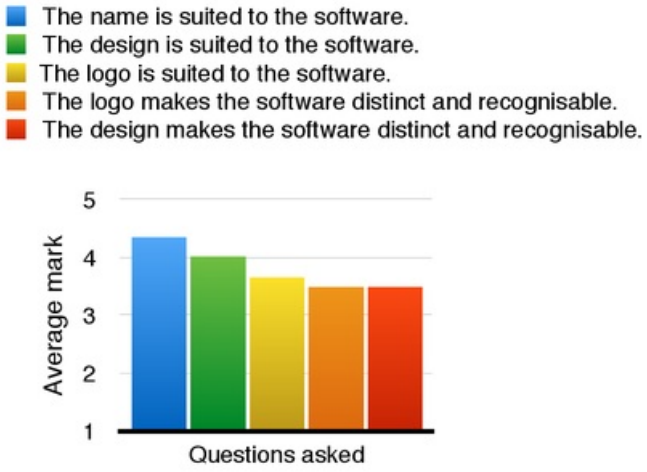
In general, testers wanted more clarity on what the software exists for and for whom it exists. Also, more information about exactly what users will gain from using it was requested. The process of importing files, and of finding them to create links was a process which testers thought needed refinement to make it easier.

Learnability



Testers made general comments about adding more detail to the manual to aid learning how to use the software. Although there was also general agreement that the software was easy to use once initial difficulties in learning how to use it had been eliminated. There was a specific request to add a function which enables users to re-link an existing annotation to a new piece of text in an article.

Sustainability and maintainability



The comments about the logo and design presented wildly differing views. One group of comments questioned the suitability of the logo and design to the software. While another group were satisfied that the design and logo 'fit' with the software.

There were specific comments about the layout of the software. Firstly, the choice of multi-window GUI should be replaced with a single window GUI, similar to the one used in the new Final Cut Pro. Also, there was a suggestion to reduce the amount of windows to three and focus on using only the ProjectFiles, TextEditor, MediaPlayer window. The MediaPlayer window, could, it was suggested, be used to house the media and annotations whenever a user clicks on a link.

General comments

Towards the end of the questionnaire there were questions designed to obtain general feedback. These are summarised here.

Some testers raised concerns about images being warped on import into the ImageViewer.

Additionally, there were comments made about an inability to load some files or open some links in the to the ImagerViewer. This was also an problem highlighted with the MediaPlayer.

One tester made a comment about exploring the possibility of creating and opening links which 'lived online'.

Overall, testers were enthused by the software and keen to use it once it was completed. Specifically, they were excited by the possibility of embedding multimedia in their documents.

4.3 Analysis

Analysis here involves analysis of both the results obtained from user testing and the testing procedures performed throughout the project. Firstly, the user testing.

In terms of the software's 'installability', all criteria for success were met. The criteria for 'documentation' too appears to have been mostly met, although there is area for improvement. Testers made a requirement for further detail to be added to the manual and specifically wanted a 'troubleshooting' section. These comments were surprising as it was thought that the manual was too simplistic and that users would feel insulted. This would suggest that users do crave simplicity of instruction in a manual and that 'dumbing down' instructions may not be detrimental.

The results for the criteria 'understandability' were initially disappointing and suggested flaws with the design of the software. However, closer inspection indicated that there was nothing inherently wrong with the design - save for some refinements (see chapter 5) - rather they were related to an issue with the user guide and a demand for more clarifying detail within it. Indeed, the question on whether the software was easy to understand how to use tested well. But, more than was thought, or expected, users do, it seems, rely on a manual to understand how a piece of software works. There was an expectation that testers would just want to 'dive in' and use the software and that, as has been mentioned, using the manual might be demeaning. Interestingly, there was only one male tester. It would be interesting to have had more male testers to analyse whether their is a gender bias in the use and expectations of a user guide.

The results for the criteria for 'learnability' were similar to those for 'understandability' with many of the issues with the software related to deficiencies in the use manual. Again, this was fascinating and not in keeping with the perception users would be able to just use the software. This indicates, obviously, that the manual should be further refined, and also that the software needs work if it is to be able to be used 'out-of-the-box'.

The results for the criteria 'sustainability and maintainability' are hard to analyse as concerns with matters such as design and logo appear so entrenched in personal taste and preference. What the results do suggest, though, is that a better way to have designed

the software - and to have pleased the most amount of testers - would have been to have sought user opinion about design before designing the software. In this project that was not done as the design and logo were not a key consideration. Also, with regards to the design it was not possible as the client specified a multi-window GUI.

A criticism of the project's testing was that more formal, independent verification - both static and dynamic - should have taken place. Both code and structure were informally appraised by fellow students throughout the project; however, this can by no means be considered to have been sufficient or rigorous enough. Code inspections should have been conducted from the start, and should have been conducted by an objective party. Furthermore, they should have been woven into the development process. The focus on producing working software, in the tight time-scale was, arguably, a reason for the lack of such regular, formal verification. Nevertheless, it should not be an excuse. This a criticism which perhaps points to an inherent problem with projects developed by individuals.

An area in which the user testing might be improved would be to have used more testers to enable more feedback to have been obtained. More male testers would have also been beneficial to have provided possible insights as to how gender influences experience of the software and manual. Finally, it would have been useful to have observed testers using the software to have gained more substantive insights into a user's experience. To further support this, interviews could have been used as part of the process. However, the development time available limited such possibilities.
Finally, the tests allowed for some interesting observations concerning the user experience. It became apparent that user expectations of the ease of use of software is incredibly high. Indeed, there was a significant disparity between the developer believing the software was easy to use and whether the user thought it was easy to use. This was an observation which provided a good lesson for future development.

Bugs

There were several technical issues which emerged as a result of testing - both the user testing and ongoing testing. One such bug concerned the annotation feature. Some testers reported no problems. However, some testers commented that sometimes annotations did not appear in the Annotations window.

Testers also reported that some image, video, and audio file names would not load in their associated windows. Analysis found that these files contained the special characters ( ) , [ ], and " ". Such characters were not properly parsed by the current file-cleansing mechanism. This needs special attention as users should have total confidence that their files will be compatible with the software no matter how unorthodox their file names are.

In both user tests and general testing the MediaPlayer was highlighted as having a critical fault. When users scrolled through large video files the entire application would freeze and crash. Also, there were early problems with the MediaPlayer which emitted a momentary buzz when videos were played. This appears to have vanished as of writing, but having not been properly diagnosed or solved, it should remain as a bug.

Finally, a bug was found which relates to the 'DS_File' in found in directors created on Macs These sometimes appear in projects in the ProjectFiles window. Their presence can create indexing issues with the ProjectFiles tree view, resulting in files being placed in the wrong folder. This only happens when folders have been accessed and manipulated outside of the software.

Chapter 5: Evaluation

5.1 Successes and Criticisms

A measurement of success for this project might be made by crosschecking those functionalities which were implemented against those specified in the client's requirements. However, to measure success in this way completely would, arguably, be crude and would, importantly, fail to consider the challenges of the project. Namely, the tight time-scale and the difficulty of some features. In under three months development, all of the features which it was assumed could feasibly be completed were completed. Amongst those features were a word processor, the development of which made huge demands on development time. There is, though, still work to be done to refine and complete some of these features. The image annotation, as will be discussed in section 5.3, is one such feature requiring work.

In drawing up the requirements document, the client was under no illusion that the video annotation functions would be particularly hard to complete within the given time frame. That only two requirements of those specified remain unimplemented, and that one of those requirements is the video annotation feature, perhaps suggests some degree of success for the project.

Another measure of success should also be derived from the feedback obtained from user testing. As has been noted, the feedback for the software was generally positive for all the criteria used in the questionnaire - but there are indeed areas for improvement. However, that the software tested this positively, against industry standards, again indicates some level of success.

One final measure of success, for any software development project, should be an ability to to reflect on the development process and its perceived flaws. Without doubt there are question marks over aspects of the project, and many of those concern the

approach taken to the work. One such question is whether following a more purely Agile methodology might have produced more of the client's requirements. Should the project have just focused on achieving each user requirement? Or was it better to have "establish[ed] an overall system architecture" (Sommerville, 2011, p.31) ? Would it have been possible to have achieved this within an Agile approach? Having reflected on this decision, as much as it was for the benefit of the software structure, it was also one made out of necessity, bearing in mind the relative inexperience and confidence of the developer. To have begun development immediately on something as demanding as, say, the image or video annotation feature would have been too large a task and would have been demoralising. Particularly so within the confines of a "sprint". In building the software, step-by-step, the confidence of the developer too increased and with that came the requisite skill set to implement the more difficult of the client's requirements.

Another criticism concerns the efforts expended to build the TextEditor window and the initial 'rtf' word processor. The realisation of the difficulties involved in implementing the hyperlink functionality into the RTFEditorKit produced great frustration. Having come to this realisation, thought turned to how this situation could have been avoided. Research should have been more thorough in investigating the difficulties of implementing the hyperlink functionality within the RTFEditorKit. Whilst the editor was being built, a demanding task in itself, there was a failure to stop and ask the necessary questions. At best this failure demonstrates a naivety and lack of experience, and at worst it suggests an irritating lack of foresight.

The problems with the 'rtf' word processor meant that the project would need to make unplanned implementation decisions. One such decision was the replacement of the 'rtf' word processor with the HTML TextEditor component. This obviously meant that the base file type would need to be 'html'. Although both enable the project to fulfil the client's

requirement for a text editor, they do so in a way which is perceived as inelegant, and which does not fit with the original development conception - and this is hugely irritating.

Having to change to the 'html' file does also highlight another question which emerged towards the end of the project and which still remains now the project is complete. Would it have been better to have built the software as a web-based application? This thought and some of the potential benefits emerged a month before the project's end. However, with too little time left to reconsult the client, or to redevelop the work already completed, it could, sadly, only be a thought.

5.2 Future Research

An area for future research would be to analyse and understand the cognitive processes involved in digital reading and writing in more detail than this project allowed. Such research could then be used to optimise the software, to account for best practises. It could also inform thinking about the client's requirement for a function which enables writer to highlight their thought processes. Some effort was, as has been noted, devoted to this within the project and was instrumental in aspects of the final design.

Research should also be made into redeveloping the software as a web-based application, as mentioned previously. Such a decision would, though, need to be made in collaboration with the client. However, such a redevelopment would open up exciting possibilities. Moving to a web-based application could make the implementation of a video annotation feature easier to implement through use of the *Open Video Annotation* plugin. However, moving the software to the web would also provide challenges. The application would, inevitably, require the use of a server and a database both of which would be demanding, in terms of implementation and maintenance work. There would also need to be, potentially, a significant amount of administration work to enable users to use the software in a way which secure and robust.

5.3 Further Development

There are those features, as outlined in the client's requirements, which are as yet unfinished and so obviously recommend themselves as future work. Namely: including functionality to annotate moving images, and to track and annotate objects within a video clip; and including functionality for writers to group and display key concepts within an article and to show the progression of their thinking.

As well as these unfinished client requirements, there is also scope for further refinement to existing features. One such feature would be the image annotation function. As yet, image annotations are not saved when the software is closed. In its original usage, the *Annotorious* image annotation plugin works with a web server to 'save' annotations by returning those annotations, embedded in the html of the web page hosting the image. A solution would, perhaps, see the implementation of a 'server' Class which would replicate the required functionality. Also, the image annotation feature would benefit from work to ensure imported images retain their native aspect-ratios imported. Currently, the hosting page has a static aspect ratio, resulting in image warping where images use a different ratio.

In general, use of threading throughout the software would too need refining. With time running out during the project, they were left 'as is', and, as a result, are inefficient and inelegant. The autosave, for example, should be perfected to 'listen' for changes and save when a change is made. At present documents are automatically saved every 4 seconds which is potentially unnecessarily CPU-intensive. Threading in the MediaPlayer window needs also to improve to enable users to scroll through videos more smoothly. At present, some HD videos can cause scrolling to stutter, resulting in video lag at best and software crash at worst.

As well as these refinements, there were also several ideas for possible future enhancements which emerged while building the software. One such idea was the

implementation of a function to export a document as a multimedia PDF. The PDF would contain the document, along with linked embedded media as part of a singe standalone, multimedia document. This was an exciting possibility, one which would make user articles and essays truly portable, allowing readers, not in possession of the *Bubble* software, to access documents written with it. Work was well under way into achieving this functionality. Future work would see the implementation of a PDF writer and would also make use of compression techniques to ensure files sizes remained small.

The software must have the ability to enable users to export articles to a variety of file types, not just PDFs. Work has begun on enabling work to be exported to 'rtf', 'docx', and and future work would see file writers for each of these filetypes fully implemented. Further filetypes, such as the 'Open Document Format' could also be added to allow the user greater flexibility. Implementing these features would negate the frustration of having to replace the proposed native file type of 'rtf' with 'html'. As long as users could export to the a final format of their choosing, the native filetype would be, perhaps, incidental.

There is also the scope for more extensive development work on the current software. One such change would involve properly implementing the 'rtf' word processor, with the required hyperlink functionality enabled in Java's RTFEditorKit. This would, however, require skill and time.

In Chapter 1, the Open Annotation Core Data Model was mentioned as a framework which could be used to ensure that all annotations made within the software are W3C compliant. Future work would ensure that the software produces annotations which are in line with the standards recommended in the Open Annotation Model. The use of 'digital signatures' to indicate the identity and province of an annotation would, for instance, be a worthwhile development.

Finally, the code should be refactored to follow more closely the model-view-controller paradigm. Again, time constraints did not allow this to be fully executed. It is

believed that a full implementation of this model will result in a cleaner code structure and hopefully make future development easier, whether in its current standalone format or as a web-based application. The move to a web-based application would, arguably, necessitate the use of MVC, with the 'model' dealing with all database records, the 'view' taking care of the display of these records, and the 'controller' handling user input.

Testing also highlighted areas for further development and those mentioned in section 5.3 should be addressed to further improve the software. Following on from comments concerning usability, a refinement, specifically, to the linking of media should be improved. This could be done by ensuring that when the user wants to create a link they are navigated straight to their current project folder. This would eliminate having to navigate through unnecessary folders to find the media they would like to link.

Conclusion

This list of future research and development outlined above is by no means exhaustive. Indeed, there are many refinements and improvements which could be made and which time did not allow for. However, the list represents key areas for such refinements and hopefully points the way for genuinely important future work. It is hoped that the foundations which this project has laid can be built upon over the coming months. Indeed, there is enthusiasm from the client, the developer to continue to develop the software. There does also, based on user testing, appear to be a genuine enthusiasm and need for Bubble. So, although this project is finished, the development of *Bubble* is not.

Bibliography

Bryant, J., 2002. *The Fluid Text: A Theory of Revision and Editing for Book and Screen.* University of Michigan Press.

Preece, J., 1993. A Guide to Usability. 2nd ed. Open University

Preece, J., Rogers, Y., Sharp, H., Carey, T., 2011. *Human Computer Interaction.* 3rd ed. Wiley

Sommerville, I., 2011. *Software Engineering.* 9th ed. Pearson

Appendices

Appendix A

User requirements document. These were derived and prioritised through several meetings and email exchanges.

User requirements

1. Writing, editing of essays and articles

2. Embedding sound, image and moving images - be able to watch/view/listen multimedia

3. Annotation of images and moving-images, with a facility to tag points of interest

4. If possible, to have the tag associated with point of interest as it moves (in video)

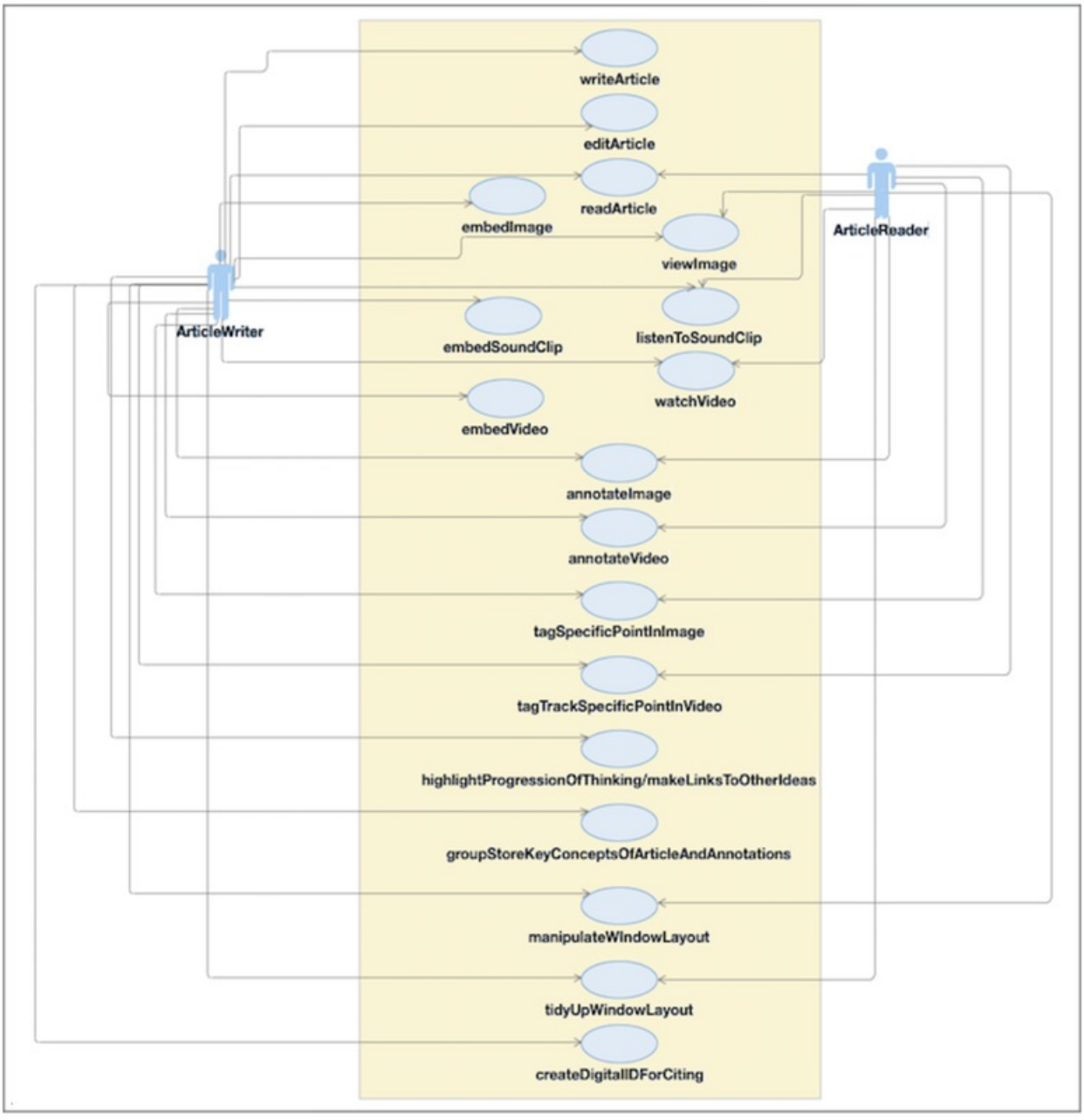
5. Provide a way of showing progression of thinking - nested links and ability to cross to another set of ideas to cross-reference ideas

6. Be able to have a list of key concepts or questions for someone else to access and find out what's on the page - both as a drop down overview and as activated when rolling a cursor over an area

7. Some ability to control placing of blocks of text and other content

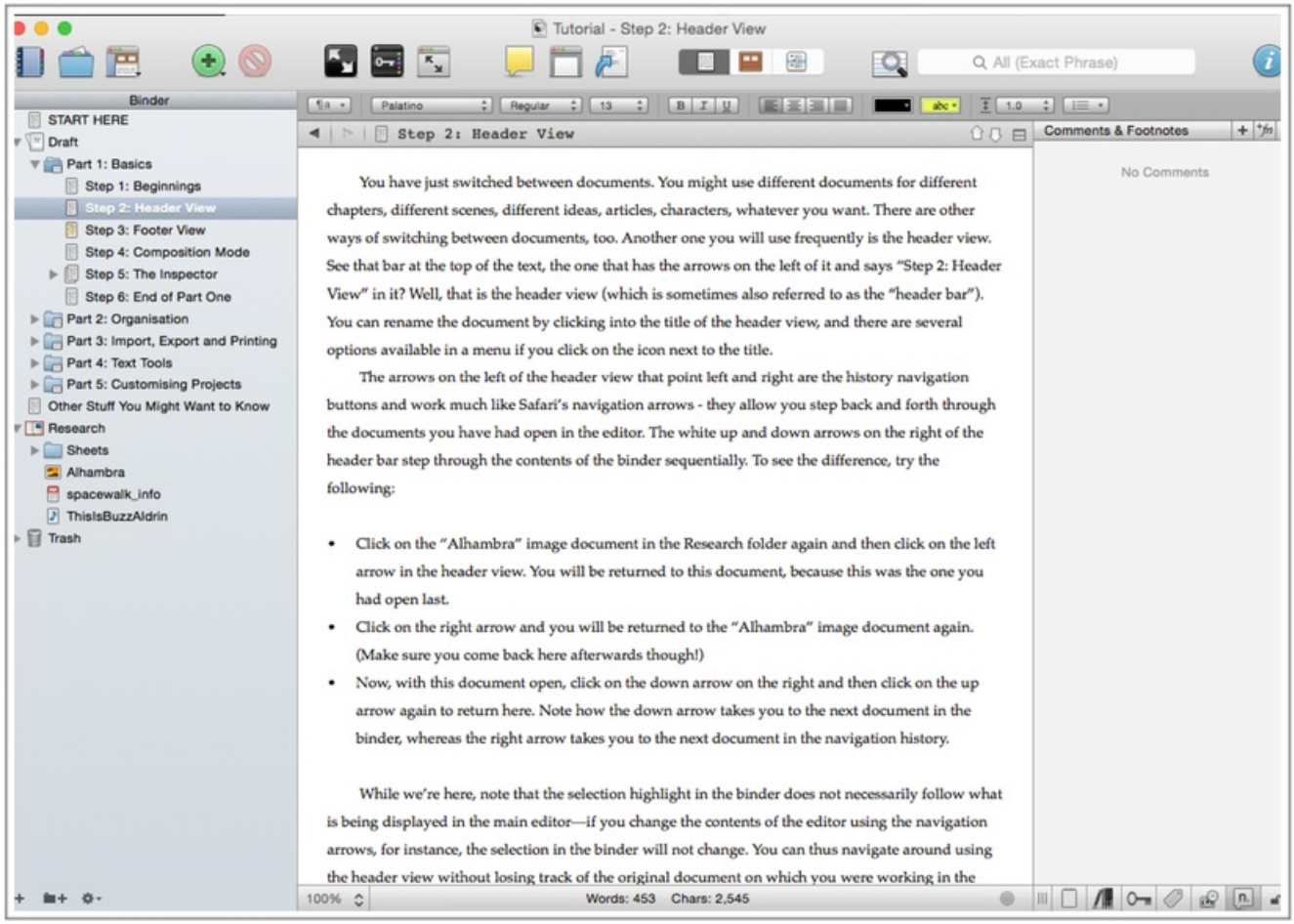
Appendix B

UseCase diagram documenting key functionality of the interface.



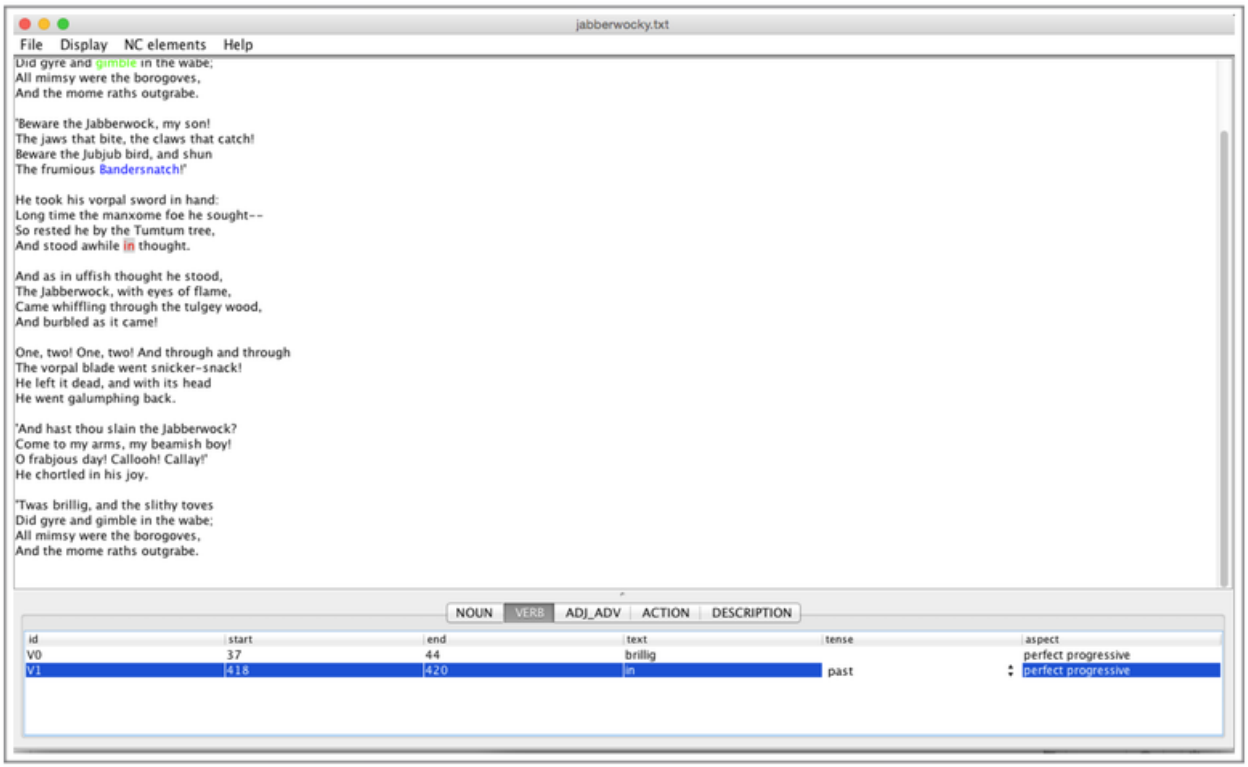
Appendix C

Screenshot of Scrivener



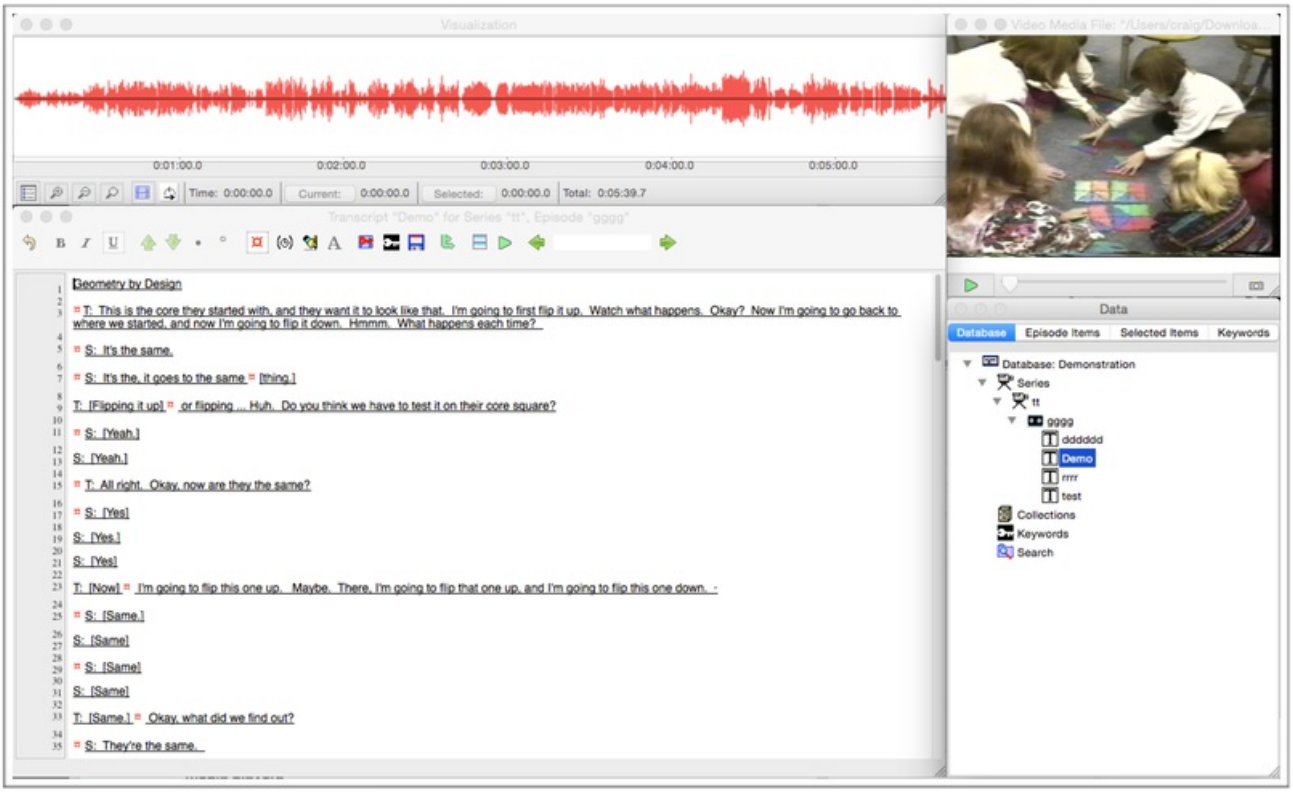
Appendix D

Screenshot of Mae Annotation. Code can be found here: https://code.google.com/p/mae-annotation/



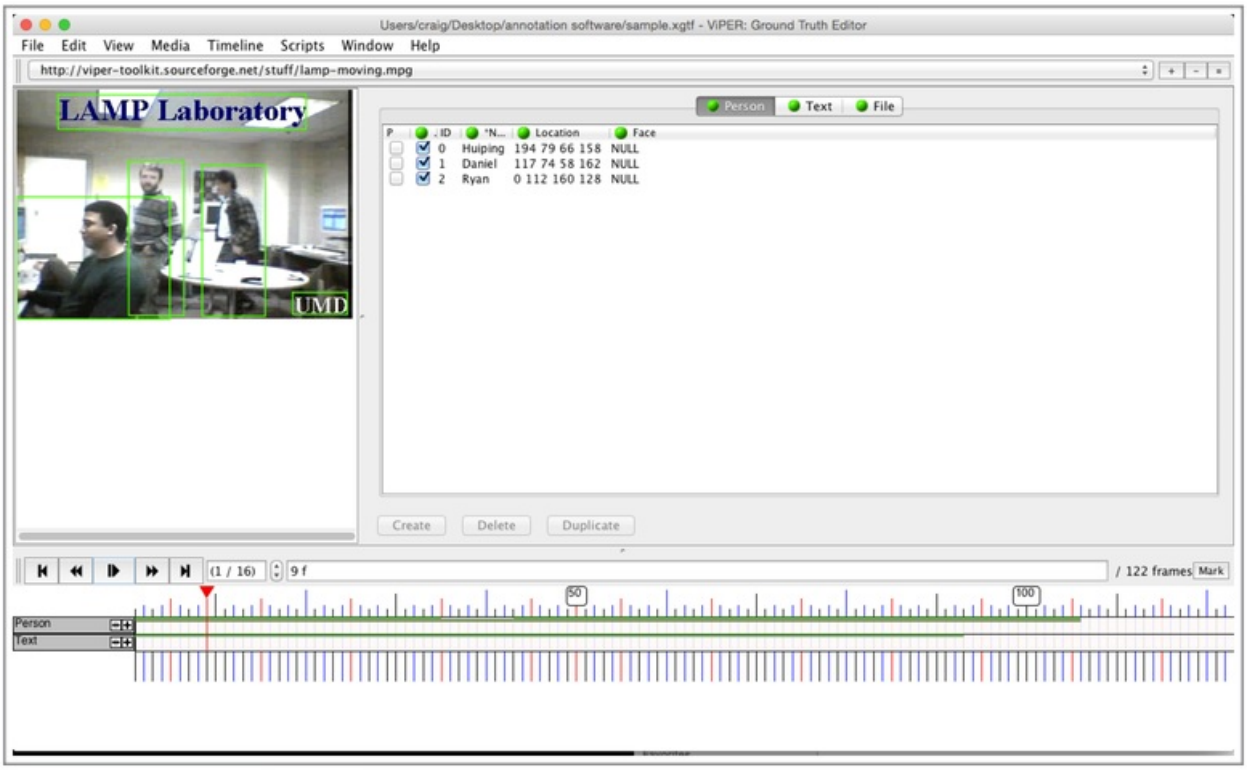
Appendix E

Screenshot of Transana (http://www.transana.org/)



Appendix F

Screenshot of Viper (http://viper-toolkit.sourceforge.net/)

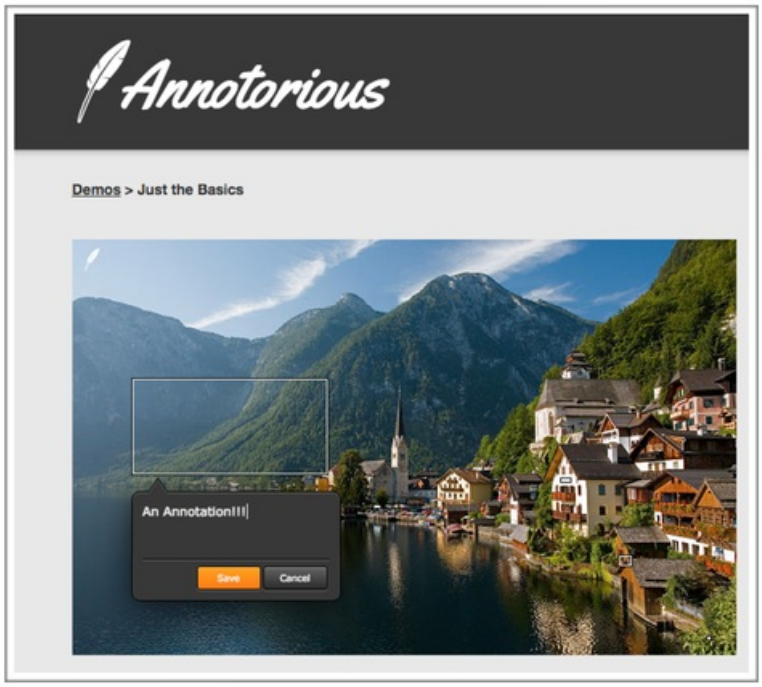


Appendix G

Screen shot of the Javascript Annotator tool (http://annotatorjs.org/)

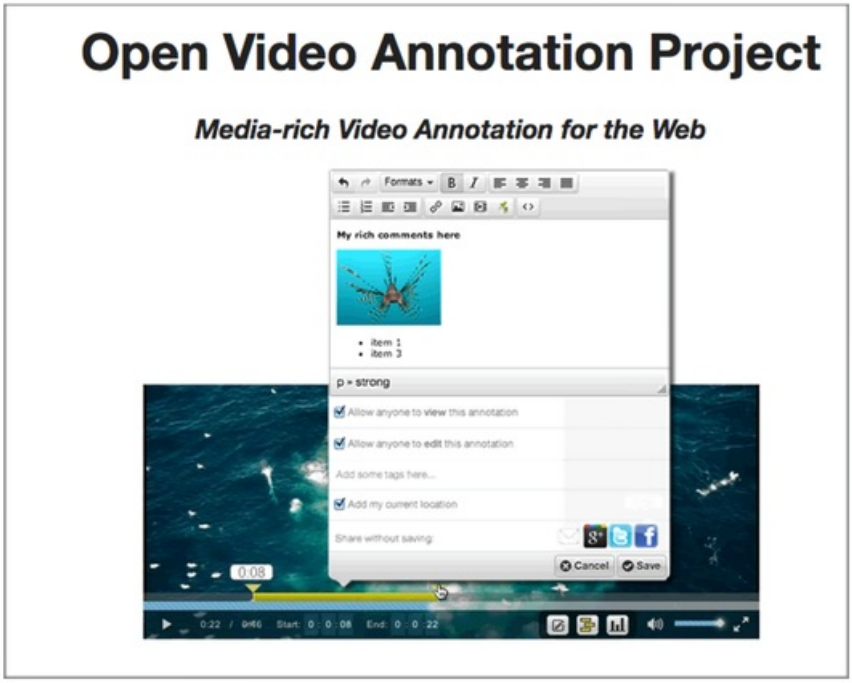
Appendix H

Screen shot of Annotorious, an image annotation tool written in Javascript (http://annotorious.github.io/)



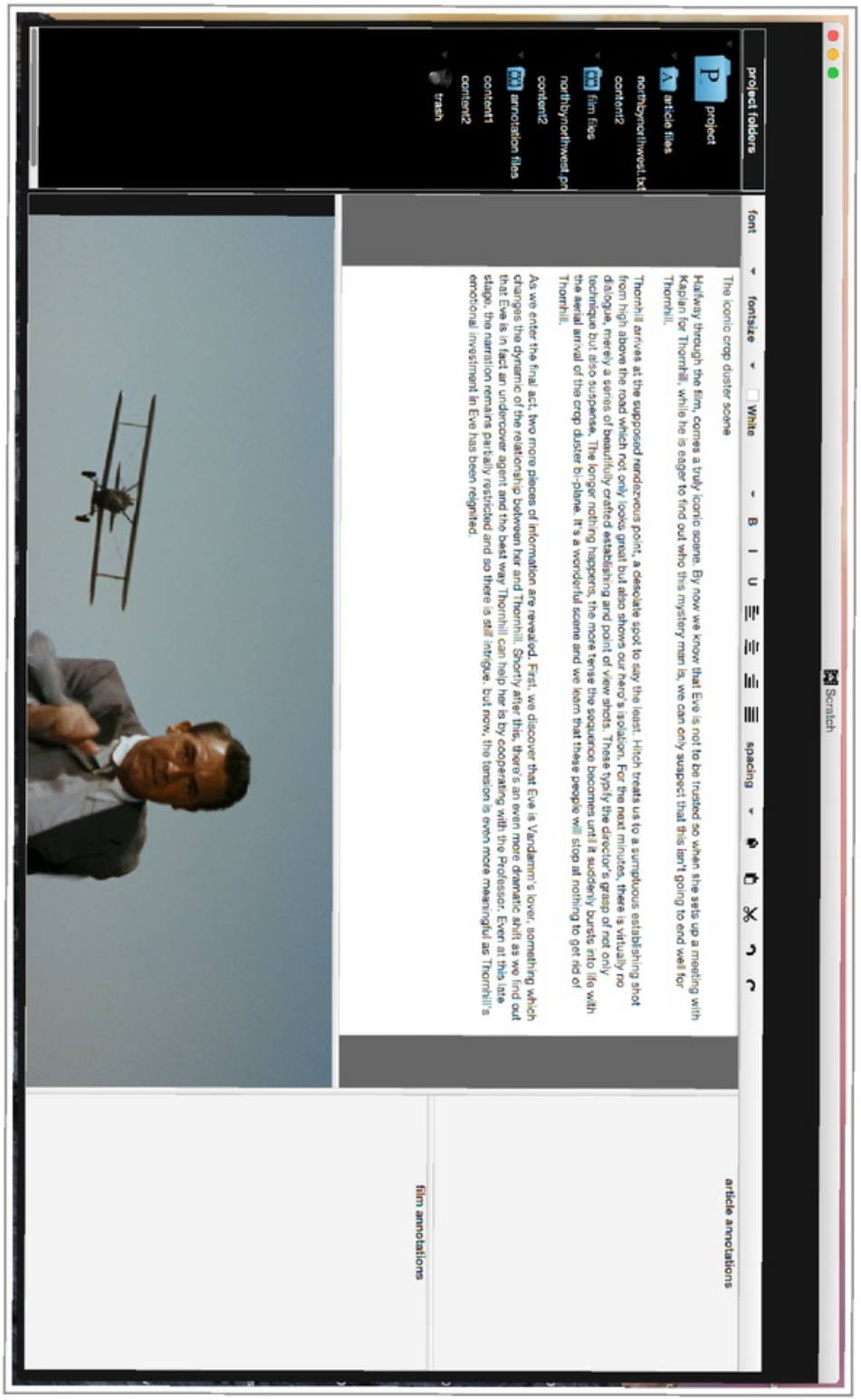
Appendix I

Screenshot of *Open Video Annotation,* a Javascript annotation tool for video *(*http://openvideoannotation.org/*)*



Appendix J

Screenshot of early prototype - used to establish design of the GUI

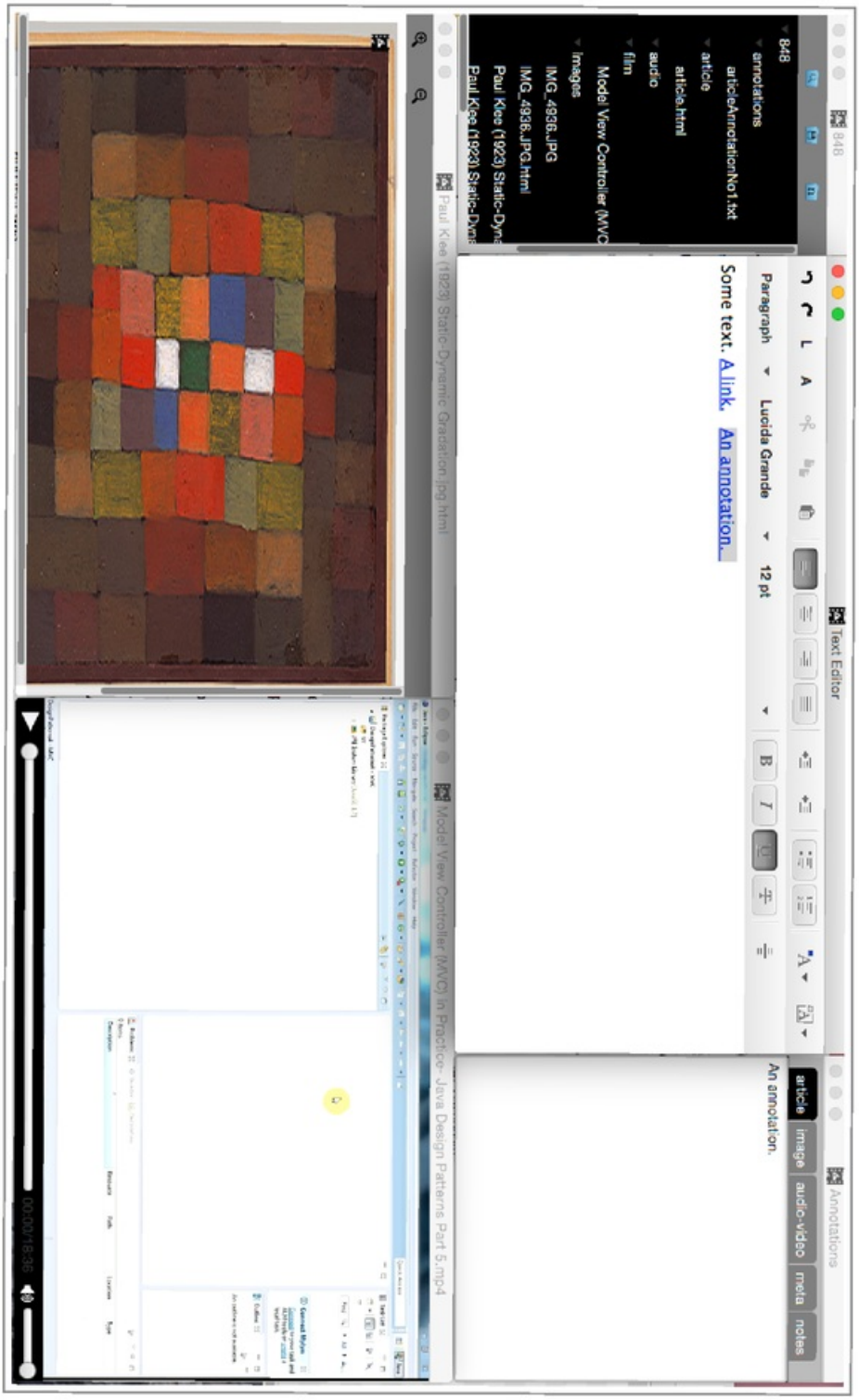


Appendix K

Screenshot of Final Cut Pro - this was used in early meetings to clarify design decisions - specifically the requirement for a multi-window GUI

Appendix L

Screenshot of final version of the software



Appendix M

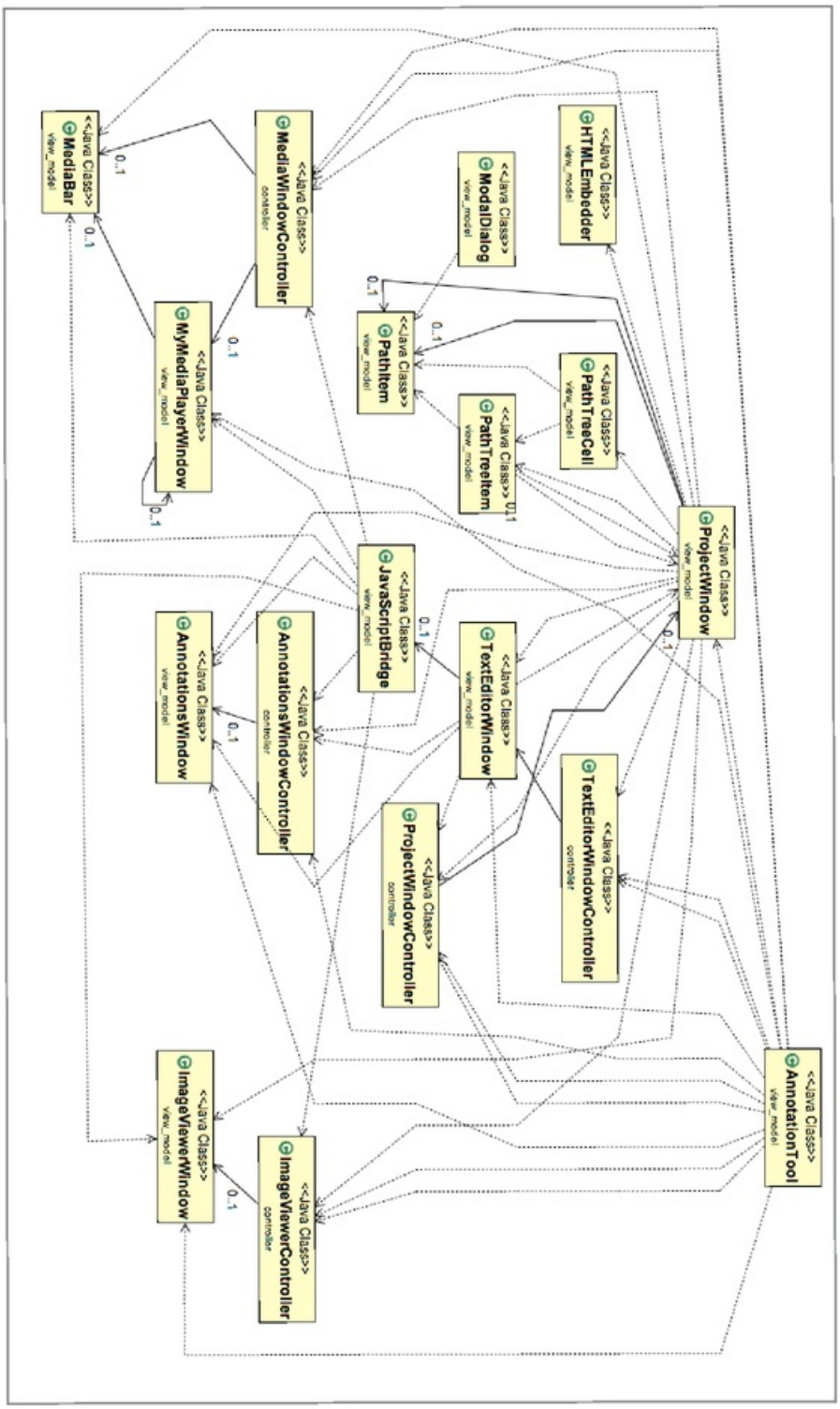
Planning document used to design each window, and the key functionalities they should have

Bubble - Each window and key functionalities

Project Files window	TextEditor window	Annotations window	ImageViewer window	MediaPlayer window
1. displays info on currently loaded project folder/files 2. allows user to open/create/ import projects 3. allows user to import media into project folder (images, videos, sound files) 4. allows user to create new article documents 5. allows user to delete files within a project - BUT NEVER the project folder 6. enables user to load a file in the associated window (images in ImageViewer) 7. allows user to export an article/essay to a required file types 8. allows users to print their articles 9. contains function to save article and annotation documents - maybe autosave? 10. allow user to save a project by a different name 11. deals with setting annotation in annotation window when created 12. when window exited, entire application is exited	1. display opened article/ essay documents 2. allow users to write & edit documents 3. allow styling of text 4. allow links to media be created / embedded in article 5. button to allow annotations/links to annotations to be created & embedded in article 6. button to allow links/ annotations to be deleted from a document 7. when fullscreen, text editor should be centralised - to offer more 'immersive' writing experience 8. clicks on links in this window should open them in their associated window 9. when window exited, entire application is exited	1. hosts annotations for images, films, essays/ articles 2. annotation should be editable 3. when window exited, entire application is exited 4. contains 'notes' window 5. toolbar with buttons/ functions to annotate and add to existing annotations - to create 'cross-links' to other annotations	1. displays images 2. zoom in/zoom out functions 3. user should be able to create annotations in this window 4. when window exited, entire application is exited	1. allows video and audio files to be opened 2. allows video and audio files to be played, paused, scrolled 3. contains timer counter 4. contains volume control 5. user should be able to create annotations in the window 6. when window exited, entire application is exited

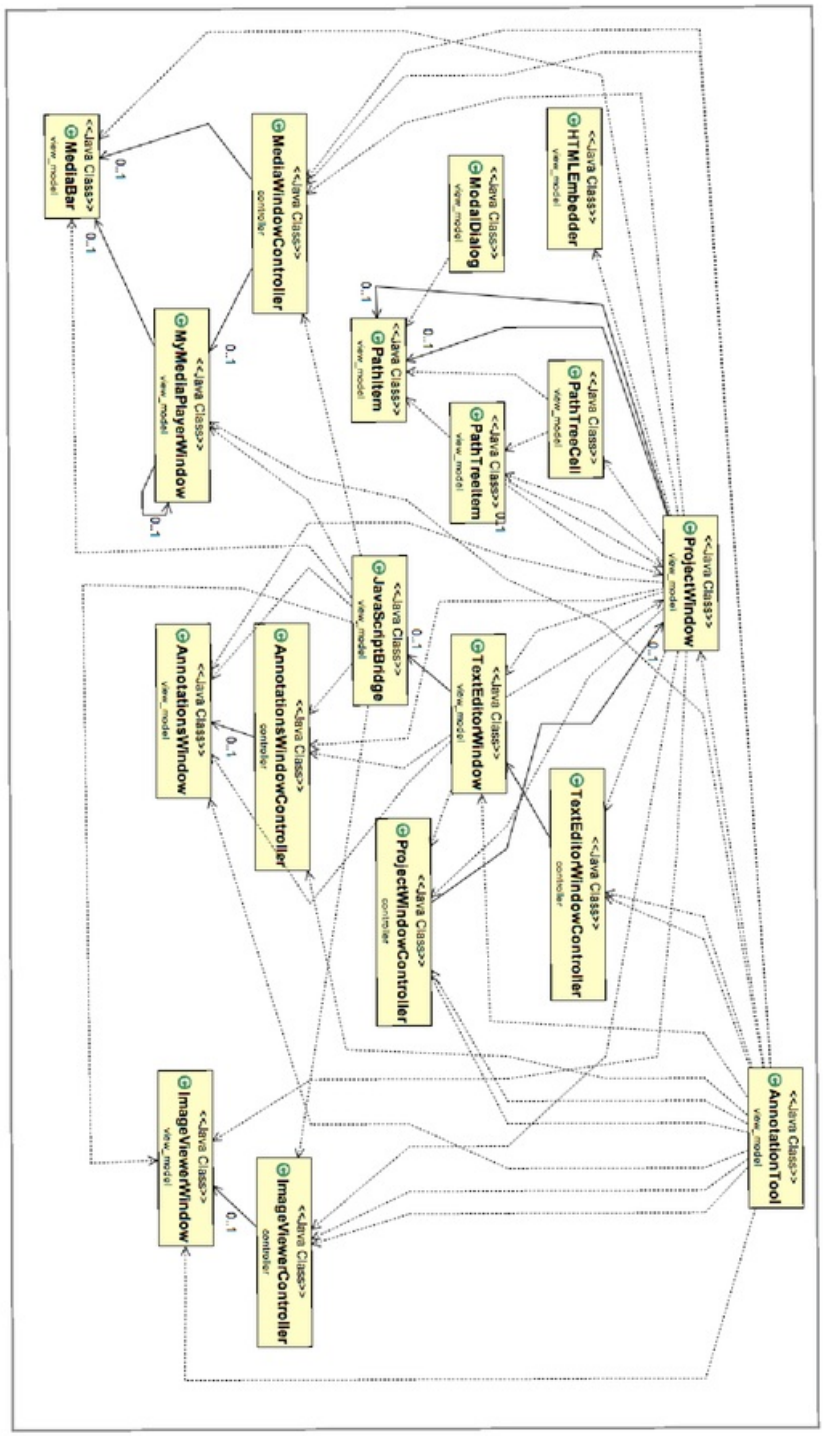
Appendix N

Class diagram of final software structure (note: fields, methods, and some dependancies & relationships have been omitted for clarity)



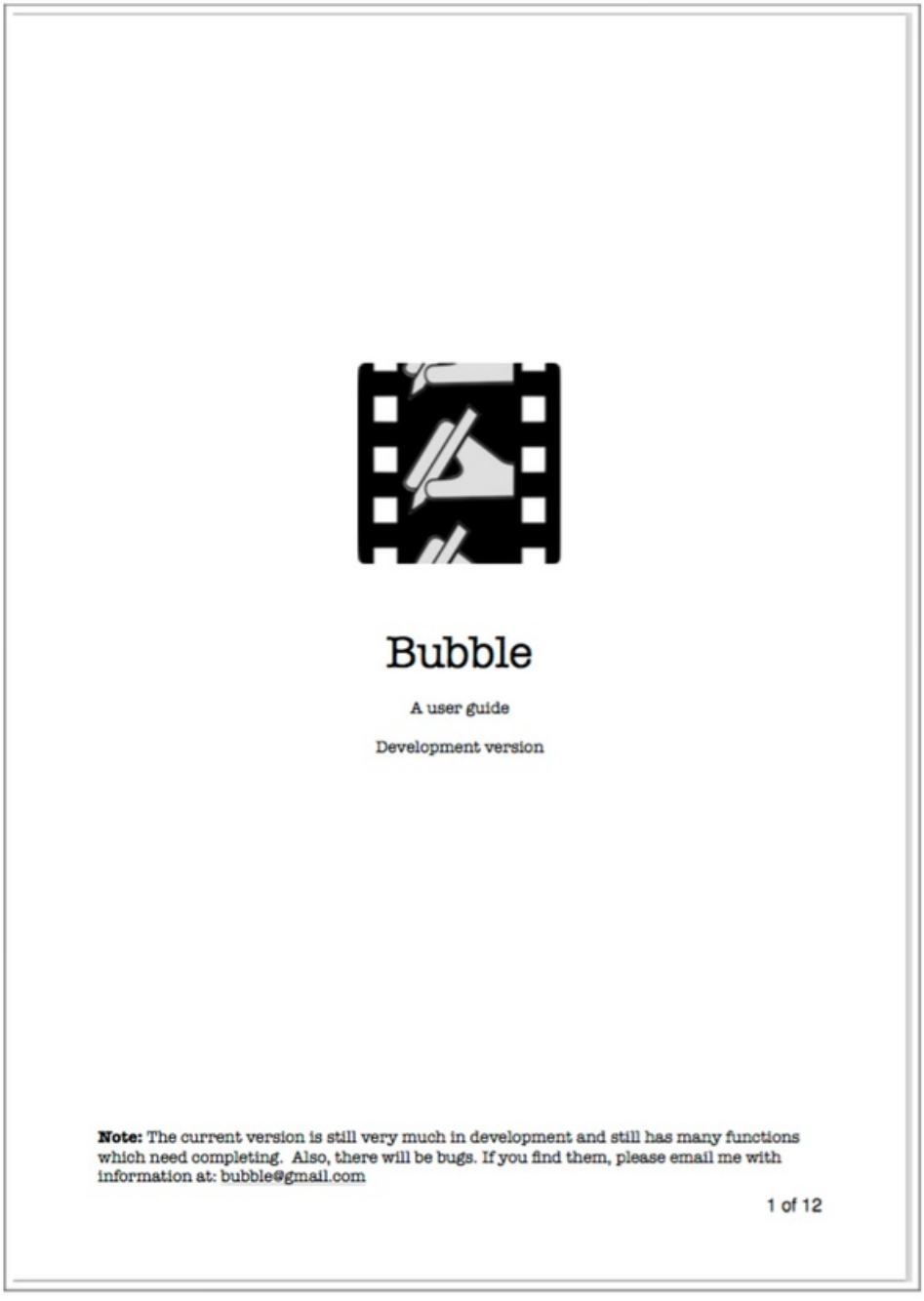
Appendix O

Initial Class diagram of software structure (note: fields, methods, and some dependancies & relationships have been omitted for clarity)



Appendix P

User guide designed for use with the prototype (written based on guidelines from the Software Sustainability Institute)



Bubble

A user guide

Development version

Note: The current version is still very much in development and still has many functions which need completing. Also, there will be bugs. If you find them, please email me with information at: bubble@gmail.com

1 of 12

1. Before you start

- what is this software?

Bubble was developed from a brief by Dr Aylish Wood of the Kent University Film Studies department. The requirements were to produce a piece of stand-alone software which would enable film students to write articles and essays in which multimedia could be embedded through the use of links. The software should also facilitate the annotation of the embedded multimedia, as well as the annotation of the actual article.

Bubble was developed by an MSc student over the course of three months as part of his final project. It was the first piece of proper software he had ever written. It may show in places.

Bubble is, as of writing, still in development.

- what do I need?

You will need a Mac. Ideally, you'll need OS 10.10.5. It may work with other versions but at present this was the version it was developed with and tested on. You will also need at least 4 GB RAM.

Note: At present Bubble has only been tested on Mac OS X so it is not advisable to use the software on any other platform.

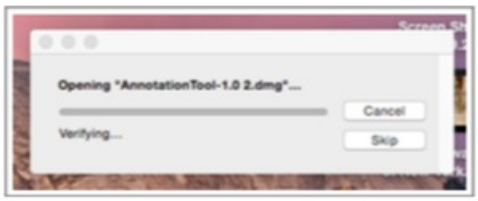
2. Getting started

- how do I install it?

You will have a zip file which looks something like **this** on your desktop. Double click on it.

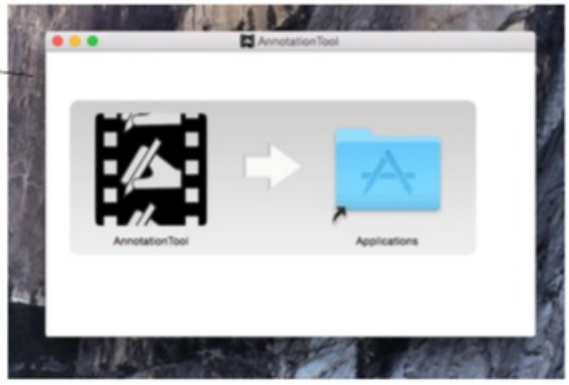
You should then get a .dmg file which looks like **this**. Double-click on that.

Once you have double-clicked the .dmg, **this** will happen.



And then **this**.

Drag the Bubble app into your Applications folder.



- where is it?

Go to your Applications folder and if all is well, the app should have been copied to it.

- how do I open it?

Double click on the Bubble app and then it should open.

3. The Windows

Bubble uses a multi-window interface. Each window can be moved around, minimised, maximised, etc.....

To close Annotation Tool, you can click this  button in the top left of any of the windows.

- Project files window

This window is home to your project folders and files. When files are imported, they can be clicked on and the file will appear in its associated window.

- Text Editor window

This is where you'll write your articles. Links and annotations are created here too. More about that later.

When you are focused on writing, try the full screen writing mode by clicking on the  button.

- Annotations window

This window is home to all of your annotations. It also has a 'notes' tab where you can make notes about your current project. Or anything.

- Image Viewer window

This window enables you to view links to images within your project.

- Media Player window

This window allows you to play links to audio and video files.

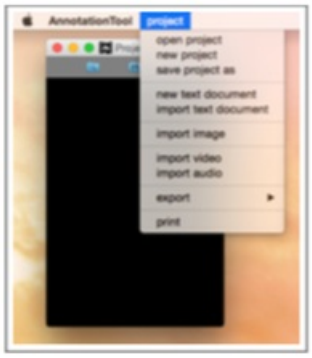
5 of 12

4. Now it's on my computer, what next?

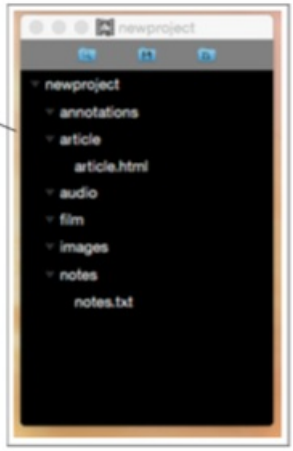
- creating a new project

There are two ways to create a new project. You can create one by pressing the icon which looks like this: 

Or you can use the project dropdown menu:



Once you have given your project a name and clicked ok on the file chooser, you should see **this** in the Project Files window:



A new project has been created, containing 6 folders, with a default article.html file in the 'article' folder and a default 'notes.txt' file in the 'notes' folder.

By default all projects are stored in your Mac's 'Documents' folder.

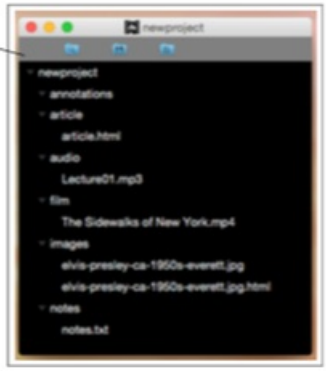
- opening a project

To open an existing project, click on the button which looks like this: 

A file chooser will pop-up. Find and select the project and click on 'Open'. The project should appear in your Project Window.

- importing media

To import media, click on the project drop down menu. You will be given options to import image, video, and audio files. Clicking on any of these options will launch a file chooser. Once a file chooser is open, find the media you want to import and either double-click on it or select it and press OK in the file chooser. Once you have imported your media, your Project Window will start to look like **this.**



- supported file types

The following files and file extensions are supported:

- mp3, mp4, .m4v
- jpeg, jpg, png
- html, txt

Note: when images imported, they are embedded in an 'html' file. For example, 'image.png', when imported, would become 'image.png.html'. These are the files you should use to create links, not the image file you imported.

- creating new article files

To create a new article file use the drop down menu. A new file will appear in your Project Window. Its default name is 'article' with an incremented number. To change the name, double click on the file name in the Project Window.

A new article file can also be created by right-clicking the article project folder. You will then be provided with a menu with the option to create a new file.

5. Now my projects is set up, what next?

- creating links to media

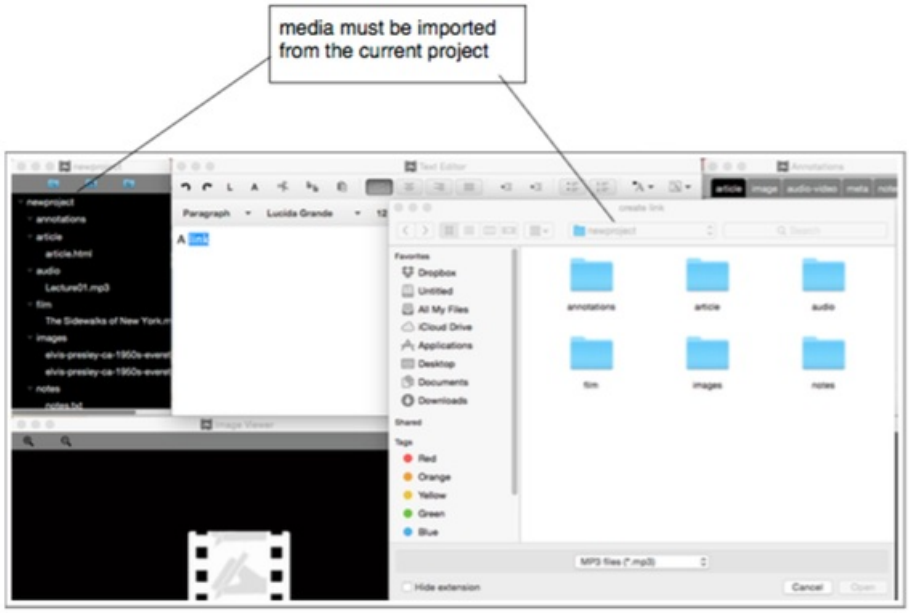
Now you have your media imported in your project, you can start to write an article, create links to media, and add annotations in your article.

To create a link, select the text in the Text Editor which you would like to create a link from. Now click on this button: 

A file chooser will pop up. Now find the media from your current project folder. Select the media by double-clicking or selecting and pressing the OK button. The text should look like thistext, and a link to your media should now be created.

Note: it is crucial that links are created to media which is already in your project. This is so when you re-open your project or when you want to forward your project to someone else, the computer will know where to find the media.

Note: To link to image files, select the files ending with 'jpeg.html', 'jpg.html', or 'png.html'.

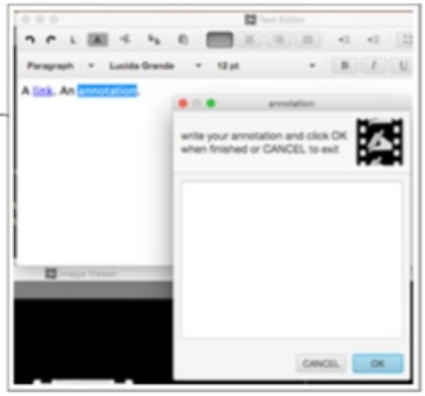


To create an annotation, first select the text in the Text Editor. Now click this button: A

A pop-up should appear which looks like **this**.

Write your annotation and click OK.

Your annotation will appear in the Annotation Window and a file containing the annotation will appear in your Project files window, in your project folder.



Once the annotation has been created it can be edited and added to in the Annotations window's 'article' tab. Like **this**.

Note: As of writing, the image, audio-video, and meta annotation tabs have not been finished.

- deleting links

Just select the text and delete the link.

- deleting annotations

As above, select the text and delete. The annotation file will remain in your project. To delete that, right-click the file in the Project Files window and select delete from the pop-up menu.

Now your links and annotations have been created you can click on them and they will appear in their associated window. Images in the Image Viewer, etc....

6. Saving/Deleting

- saving article documents

Nothing to do! All documents - annotations, notes, or articles, are automatically saved.

- saving projects using a different name

There are two ways to save a project.

You can click on this button: 

Or, you can select the project drop down menu and select 'save project as'.

Both methods will launch a file chooser. Here you can enter the new name for your project and click OK.

- deleting projects

Projects can only be deleted in your Documents folder.

7. Annotating imported media

- annotating images

To annotate an image, load it into the Image Viewer. Now click and drag on the part of the image you would like to annotate. A **dotted outline** should appear. Drag to create the size area you would like. A **text box** should appear. Enter your annotation and press save.

Here's what it should look like:

Note: This function is still under development and annotations cannot yet be saved. However, the version gives a good idea of how the image annotation will work.

- **annotating audio**

This function is still currently under development.

- **annotating audio**

This function is still currently under development.

8. Exporting

These functions are currently under development.

9. Printing

This function is currently under development.

10. Information for developers

Bubble is open source and the code will be soon available on Github. This will be accompanied with information of known bugs and required functionality. Any help is welcome.

Acknowledgements

Bubble makes significant use of the work of these:

http://d.hatena.ne.jp/tomoTaka/20130323/1364040196

http://annotorious.github.io/

A huge thanks for their efforts.

Appendix Q

Questionnaire used in user tests (all questions based on criteria from Software Sustainability Institute - http://www.software.ac.uk/)

Bubble - User testing - questionnaire

Firstly, thank you! Your time is greatly appreciated in completing this questionnaire.

Bubble is a piece of software currently in development. Your feedback will provide insights on how it can be further developed and improved.

The questions are based on a set of criteria form the **Software Sustainability Institute** (http://www.software.ac.uk/sites/default/files/SSI-SoftwareEvaluationCriteria.pdf) who provide guidelines on how software should be evaluated.

Each question will ask for you to give a mark out of 5.

5 = 'very' or 'highly'.

1 = 'not at all'.

You are then asked to offer further comment to provide detail to your answers.

Usability

Installability

1. How easy was the software to install?

   1 2 3 4 5

2. How easy was the software to locate and open?

   1 2 3 4 5

3. Was the software and its associated folders well-organised when installed it?

   1 2 3 4 5

4. Was the software easy to uninstall?

   1 2 3 4 5

How could the installation process be improved?

Are there any other comments you would like to add to help answer the questions above?

1 of 6

5 = 'very' or 'highly'.

1 = 'not at all'.

Usability

Documentation

1. Did the user manual give an overview of the software?

   1 2 3 4 5

2. Was the user manual well-organised?

   1 2 3 4 5

3. Was the user manual clear?

   1 2 3 4 5

4. Was the user manual accurate?

   1 2 3 4 5

5. Was the user manual easy to follow?

   1 2 3 4 5

Please add any comments on how the user manual could be improved:

5 = 'very' or 'highly'.

1 = 'not at all'.

Usability

Understandability

1. Was it easy to understand what the software does and its purpose?

   1 2 3 4 5

2. Was it easy to understand how to use the software?

   1 2 3 4 5

How far do you agree with any of the following:

1. A description of what/who the software is for is available

   1 2 3 4 5

2. A description of what the software does is available.

   1 2 3 4 5

3. A description of how the software works is available

   1 2 3 4 5

4. Design rationale is available – 'why the software does it the way it does'

   1 2 3 4 5

Was there anything you found difficult to understand about the software?

Do you have any comments to help you further answer any of the questions above?

5 = 'very' or 'highly'.

1 = 'not at all'.

Usability

Learnability

1. How easy was it to learn what the software does?

   1 2 3 4 5

2. How easy did the manual make it to learn what the software does?

   1 2 3 4 5

3. How strongly do you feel you needed the manual to learn what the software does?

   1 2 3 4 5

Did you find any function difficult to learn or use?

Are there any other comments you would like to add to help answer the questions above?

5 = 'very' or 'highly'.

1 = 'not at all'.

Sustainability and maintainability

Identity

1. The name is suited to the software.

   1 2 3 4 5

2. The design is suited to the software.

   1 2 3 4 5

3. The logo is suited to the software.

   1 2 3 4 5

4. The logo makes the software distinct and recognisable.

   1 2 3 4 5

5. The design makes the software distinct and recognisable.

   1 2 3 4 5

What do you think could be improved about the design?

Are there any other comments about the above statements you would like to add?

Were there any aspects of the software which you particularly liked?

The software currently only has a development name. Do you have any suggestions for a name for the software?

Please use this section to offer any thoughts or ideas on areas where Bubble could be improved which have not covered in the previous sections.

Thank you for completing this questionnaire