**Code editor with syntax highlighting & autocomplete**

**Interim Report**

Submitted for the BSc in   
Computer science

January 2016

by

**Jacob Plaster**

Word Count: 5143

Table of Contents

1. Introduction 4

1.1 Initial brief 4

1.2 Project context 4

2. Aims and Objectives 5

2.1 Primary objectives 5

2.2 Secondary objectives 6

3. Background 7

3.1 General context 7

3.1.1 Syntax highlighting 8

3.1.2 Auto complete 9

3.1.3 Language support 9

3.1.4 Software functionality 10

3.2 Technologies 11

3.3 Algorithms and data structures 12

3.3.1 Lexical analysis vs. regular expression 12

3.4 Alternative solutions 13

3.4.1 Critical appraisal of Atom 13

3.4.2 Critical appraisal of Sublime 14

4. Designs 15

4.1 User interface 15

4.2 System architecture 17

4.3 Testing 18

4.3.1 Unit testing 18

5. Task List 20

5.1 Personal task list 20

5.2 Project task list 21

5.3 Time plan 22

6. Risk Analysis 23

7. Appendices 25

5. References 26

Atom, 2015. Code editor [online] 26

Biomedcentral, 2015. Efficiency comparison of languages [online] 26

Businessdictionary, 2015. Software engineering definition [online] 26

Microsoft, 2013. Visual studios 2013 [online] 26

Notepad ++, 2016. Code/Rich text editor [online] 26

Yourdictionary, 2015. Software engineering definition [online] 26

Sublime 2, 2015. Code editor [online] 26

# Introduction

Software engineering is:

“The process of making, testing and documenting computer programs (Yourdictionary, 2015). This particular type of engineering has to take into consideration what type of machine the software will be used on, how the software will work with the machine, and what elements need to be put in place to ensure reliability (Businessdictionary, 2015).”

A typical workflow for all software engineers requires heavy use of a code editor: A text editor, which is tailored especially for the production of computer code in various languages. Code editors improve the development speed of programming through the use of various features such as syntax highlighting, code formatting, auto-complete.

Syntax highlighters colour in specific keywords within the code in order to quickly draw the users attention/eye focus towards the more important areas of the code. Research into the speed of program comprehension and syntax highlighting showed that “The presence of syntax highlighting significantly reduces context switches” (Advait Sarkar, 2015).

Autocompleting aids the speeds and accuracy of the user by providing suggestions that cave been calculated whilst the user types. This functions efficiency improves depending on the size of the sequence that the user attempting to type. Code formatting optimises the writing speed of the user since it automatically performs tedious text management tasks such as indentation and inserting syntax symbols.

Both standard coded editors and integrated development environments (IDE’s) will be assessed during the duration of this report. Code editors are multi purpose, rich, text editors that are specialised for code and usually target multiple languages. IDE’s are similar, however they sacrifice the ability to accommodate multiple languages for other useful function such as compilers, smarter grammar understanding and more.

This report will aim to cover:

* The initial brief and project description
* The background on existing code editors and their features
* The aims and objectives of the project
* The design and implementation of technologies in the development of the project
* The task list with detailed timelines and progression
* Ethics and risk analysis

## 1.1 Initial brief

Project description given:

“Although it is possible to program using nothing more than Notepad and a compiler, it is much easier to use an Interactive Development Environment (IDE) as the GUI for programming. Typical features include syntax highlighting, so that the keywords are readily visible, and autocomplete (e.g. like Visual Studio’s Intellisense) to improve efficiency or gain context dependent help. This project would involve creating your own IDE, such as a simple Notepad++ style program” (Walker, 2013)

## 1.2 Project context

The outcome of this project would be a code editor, which gives the user the ability to create and modify code even when under high amounts of stress and dealing with large amounts of data. The Extra features should rapidly increase development time and provide a clean and minimalistic environment for the user to work with. Syntax highlighting will help the user to better understand the code.

Advantages of using a code editor over a text editor:

* Increases speed of workflow.
* Improves readability and understanding of code with syntax highlighting and text formatting.
* Has a knowledge of programming codes/conventions so it can help manipulate data
* Some editors come with a built in compiler and error handling which can help highlight unwanted bugs that would otherwise go unnoticed.
* Source control and team management tools can help with large scale production

The table below (1.1) demonstrates how the current existing code editors share some of the same core functions, which have been described in this report, however fail to provide a solution which meets all of the requirements.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Name | Syntax-Highlighting | Auto-complete | Plug-in  System | Resource  Usage | Machine-  learning | Cost |
| Notepad++  (Notepad++, 2016) | ✓ | 🗶 | 🗶 | LOW | 🗶 | FREE |
| Sublime 2  (Sublime 2, 2013) | ✓ | ✓ | ✓ | MED | 🗶 | £45.61 |
| Atom  (Atom, 2015) | ✓ | ✓ | ✓ | HIGH | 🗶 | FREE |
| Visual Studio  (Microsoft, 2013) | ✓ | ✓ | ✓ | HIGH | 🗶 | £351 (Professional) |

Table 1.1 – Functional comparison of modern IDE’s

This project aims to create a lightweight IDE that combines syntax-highlighting, auto-complete, machine learning and a plug-in system whilst still remaining extremely efficient and reliable. The users experience with the software will be taken into consideration. A minimalistic and clean GUI will be designed to create a friendly and refreshing environment, which would help for long periods of usage.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Name | Syntax-Highlighting | Auto-complete | Plug-in  System | Resource  Usage | Machine-  learning | Cost |
| Able | ✓ | ✓ | ✓ | LOW | ✓ | FREE |

Table 1.2 –implementations for able

# 

# 2. Aims and Objectives

To create an efficient and reliable code editor with auto-complete and syntax highlighting

## 2.1 Primary objectives

The following primary objectives would need to be implemented for this project to be considered successful:

1. Create a clean and minimalistic code editor
2. Add ability to handle files
3. Include additional core features such as auto-correct and syntax highlighting
4. Ensure software efficiency and reliability is at a high standard

1. Code editor

Develop a clean and minimalistic interface which allows the user to write and manipulate code. The GUI should be responsive, simple and comfortable for users to use for long periods of time.

2. File handling

The software should give the user the ability to manipulate file structures and allow them to:

1. Create files
2. Rename files
3. Remove files

3. Additional core features

Develop and integrate smart algorithms that can successfully auto-complete words and highlight code syntax. The algorithms would be required to be fast, accurate and reliable.

4. Efficiency and reliability

Perform numerous rigorous tests to ensure that the software’s core algorithms perform as efficiently as possible. The software should be reliable and able to perform under a high amount of stress and consistent when running on other hardware configurations.

## 2.2 Secondary objectives

The following secondary objectives have low priority and will only be implemented when the primary objectives have achieved success:

1. User generated customization
2. Machine learning autocomplete
3. Multi-language support
4. Compilers
5. User generated customization

Develop a system which allows 3rd party plug-ins to be creates and implemented into the software. These plug-ins should have the capability to change both the functionality and aesthetics of the software.

1. Machine learning

Develop and provide the auto-complete algorithm with the ability to utilize machine learning in order to predict and complete the user’s word.

1. Multi-language support

Implement a system which allows the syntax-highlighting algorithm to work on multiple languages through the use of configuration files which can be customized for each language.

1. Compiler support

Develop a built in compiler that will compile C++ code and give the plug-in system the ability to add extra user generated compilers.

# 3. Background

## 3.1 General context

A code editor is an essential piece of equipment that all software engineers and computer programmer would struggle to work without. Typical usage of a programming development environment would include high amounts of workload/stress and usage that can last for very long periods of time. Therefore, it is important that the software provides the user with a clean working environment and the ability to work on multiple projects with large amounts of data and no signs of stress. The majority of this project focuses on creating an adaptable, efficient piece of software that can be tailored to the users desires and needs through the use of support files and plug-ins.

This section will describe the relevant concepts and terminologies that were researched during the development of this project. Firstly, a code editor needs to understand the grammar of the specified language in order to provide function such as syntax highlighting and autocorrect. There are 3 key areas that need to be considered when analysing a grammar:

## 3.1.1 Grammar analysis

Lexical analysis

Lexical analysis is the first phase of understanding grammar and is used in compilers, syntax highlighters and auto completers. It takes a stream of characters and converts them into a table of ‘lexemes’ and ‘tokens’. A lexeme is the literal piece of source code and the token is its corresponding label. This is massively important because it means the program is split up into understandable tokens such as ‘keywords’, ‘symbols’, ‘numbers’ and more. These tokens can then be used inside of the syntax analysis and/or in other functions such as syntax highlighting or autocomplete.

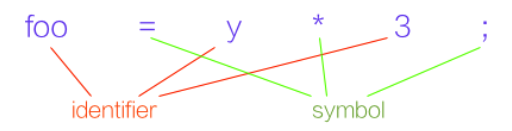


Figure 1.2 – Lexical analysis of assignment statement

Syntax analysis

The syntax analyser is a lot more complex than the lexical analyser since the program must decide on what statements have been used depending on both the syntax and the tokens that have been used within the statement. The program will then generate a data structure called a parse tree using the tokens provided by the lexical analyser, which gives the code syntactic meaning:



Figure 1.2 – Lexical analysis of a ‘C’ styled assignment statement

In order to describe the syntax, a meta-language can be used: a language that is used to describe another languages. Commonly, “Backus-Naur-Form” (BNF) or “Extended Backus-Naur Form” (EBNF) is used. Here is an example a BNF rule



Figure 1.3 – BNF description of assignment and ‘IF’ statement

The combination of lexical and syntax analyser is usually adopted by IDE’s since they contribute to the implementation of a compiler system: A piece of software that converts one language into another. And also provide a higher level of understanding. However, code editors, which aim to accommodate multiple languages often avoid this time consuming approach and use regular to lexical analyse a language and then skip the syntax analysis phase.

## 3.1.2 Syntax highlighting

Syntax highlighting, as mentioned in the Project Context, is a feature that all modern code editors and IDE’s require and it consists of a process which searches through the user written code to find and manipulate keywords, identifiers and operators. The highlighter changes the visual elements of these variables in order to make them easier to distinguish from the rest of the code, therefore improving the readability. A study by Advait Sarkar (Advait Sarkar, 2015) at the university of Cambridge showed that the presence of syntax highlighting vastly reduced the time it took people to comprehend a program. The figure below compares comprehension times of code with and without syntax highlighting:



Figure 3.1.1 – Program comprehension times with and without syntax highlighting (Advait Sarkar, 2015)

Although the study did find that programmers with more experience were less affected by syntax highlighting. Below is an example of some python code; the above block of text has been processed by Notepad++’s syntax highlighter, whereas the bottom block has been left plain:





Figure 3.1.2– with and without syntax highlighting (Notepad++, 2016)

The main aim of the syntax highlighter is to direct user eye focus towards the more important areas of the code. For example, in the figure below you can see that function calls, operators and keywords stand out and are easily recognised, whereas comments are harder to notice and require closer attention, giving the impression of less text:



Figure 3.1.3 – Atom Syntax highlighted piece of C++ code (Atom, 2015)

This helps to improve the understanding and readability of the code, but syntax highlighting also helps to prevent syntax errors by forcing the writer to conform to certain standards. If the user was to accidentally spell a keyword wrong or miss the closing symbol when writing a comment then the highlighter would bring this to the users attention by behaving unexpectedly.

There are many ways to build a syntax highlighter; one of the most popular methods involves a lexical analyser and parser, this method makes it extremely hard to accommodate different programming languages. However, Able purely utilises Qt’s regular expression engine. This gives the software greater flexibility and allows for the usage of language support files, which essentially provides the ability to accommodate any programming language.

## 3.1.3 Auto complete

‘Autocomplete’ is a feature that provides accurate suggestions of words as the user types into the text area. The user can select one of the presented suggestions, which will cause for the word to be instantly completed and therefore increasing typing speed. Like syntax highlighters, auto-completers require the grammar of the language to be defined in order to select tokens as suggestions. Many editors struggle with this problem, especially IDE’s since it is very hard to create syntax and lexical analysers for a large array of languages. So the user is bound to a small set of languages in which he can write within the software. Modern code editors tackle this problem by, instead, introducing ‘plug-ins’ which can be added to the software externally and usually contain a configuration file which ads support for the given language.

Since IDE’s have a stronger knowledge of the defined grammar means that they can provide smarter autocorrect features. . For example, below is a figure taken from Android studio (IDE) (Android Studio, 2016), which utilises the IntelliJ platform in order to produce one of the smartest auto completers available:



Figure 3.1.3 – Android studios IntelliJ autocomplete (Android Studio, 2016)

In figure 3.1.3 we can see that the autocorrect feature is aware of the newly created object and all of the functions that are contained within.

By predicting and replacing the currently typed word, the software reduces the amount of key presses required for the user to reach their goal. Due to the nature of programming, it is often required for the user to re-write the same word countless times and auto-completion removes the need for this tedious task and therefore increasing development speed and making it easier to write code.

## 3.1.4 Software functionality

File systems

Since IDE’s and code editors exist to manipulate data it is important that these pieces of software provide the user with the ability to at least load, save, rename and open files. More powerful editors further improve their functionality by implementing file management systems. Editors like Atom (Atom, 2015) and Sublime (Sublime, 2015) do this through the use of their file tree widgets, which rest beside the text-editing window. This allows the user to manage entire directories with ease and from within the software. IDE’s often package code files as ‘projects’ which helps the user to keep their necessary code files as a bundle, this makes it especially easy to import and export data. The Qt creator IDE (QT Creator, 2015) does this by including a ‘.pro’ file within the directory which describe the projects contents. The software then uses this information in order to style the file system widget in the most efficient way possible, below shows QT neatly separating the header files from the source files:



Figure 3.1.5 – Comparison of project management systems (Atom (Atom, 2015) on the left and QT Creator (QT Creator, 2015) on the right)

Figure 3.1.5 gives an example of how differently typical IDE’s and code editors handle their file systems. IDE’s have a more complex file system since they are built with firm knowledge of the language they support.

Text editing

Whilst researching the common functionalities of text editing with code editors and IDE’s it quickly became apparent that there are a few common features that are very important to the development speed of code. Firstly, auto-indentation simply formats your codes indentation margins as the user types. Whenever the user creates a new line, the cursor is moved to the indentation margin of the above parent statement. Below is an example with all of the parents highlighted in green and the indentation margins marked with a dotted line:

 Figure 3.1.6 – Atoms auto-indentation margins highlighted (Atom, 2015)

Although this may seem very trivial, it reduces the amount of key presses that the user needs to type in order to complete a statement drastically and also improves the readability of the code.

When editing large amounts of code it can be a difficult task if the user wants to quickly locate a word or piece of code. To resolve this, most common editors provide a regular expression search function, which quickly highlights any query matches. More powerful editors may also provide a replace functionality, which removes the search matching and replaces it with some user-defined text. This is especially useful when changing variable names or restructuring code as is shown below:



Figure 3.1.7 – Example of QT’s search and replace function (QT Creator, 2015)

## 

## 3.2 Technologies

Code editors with auto-complete and syntax highlighting work under a heavy amount of stress since there are constant background algorithms that run simultaneously to monitoring the user input. In order to make sure this process runs as smoothly as possible the author has decided to use C++ for its efficiency. However many other languages such as Java and C# were considered due to their ease of use which would have a large effect on the production speed.



Figure 3.2.1 – Efficiency comparison of languages (Neighbour-joining algorithm) (biomedcentral, 2015)

The QT modern framework would be implemented to manage the base GUI components and cross platform capability. QT offers support for SQL, ECM, CSS as well as a fully transformable canvas. The framework can easily be ported onto many different platforms as well as any screen size. Which would be extremely useful when targeting a large customer base.

Implementing a machine learning system is hard since the software is written in C++, which doesn’t have very good support for machine learning technologies. So an API that allows for python commands to be executed would be implemented. The machine learning algorithms would scan the documents and understand which variables/libraries/objects that the users uses regularly and then rank them higher inside of the autocorrect prediction engine.

## 3.3 Algorithms and data structures

The autocomplete algorithm when used for already existing libraries would need a data structure, which is composed of a static dictionary and a dynamic dictionary. The static dictionary would contain a tree of all of the available library modules and would attempt to complete the user-inputted word depending on which branch it resembled.

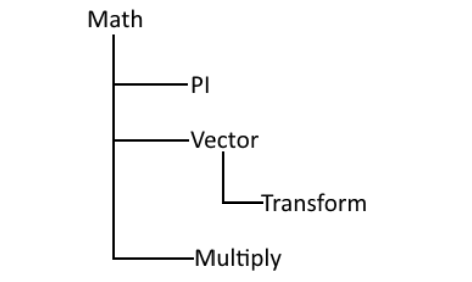


Figure 3.2.2 – Tree data structure for autocomplete

If the user was to enter “Math.” Then ‘Pi’, ‘Vector’ and ‘Multiply’ would be suggested. If the user were to enter “Math.Vector.” then only ‘Transform’ would be suggested.

All scope variables would be contained inside of a dynamic list and would have priority over the static library dictionary. The autocorrect algorithm would have to run every time the user enters a key into the code editor text area, so efficiency is key.

Syntax highlighting would not have to run as frequently as the autocorrect algorithm, it would be sufficient to run this function every time the user presses the space bar to indicate a new word has been created. It would run through the current line of text and match the newly completed word with a cfg file full of colour-coded identifiers. Accommodation for other languages would be handled via the plug-in manager, which would simply change the selected language support file with one that supports the selected language. Adding support for a new language would be as easy as adding another cfg file.

## 3.3.1 Lexical analysis vs. regular expression

There are two popular methods of tackling the syntax highlighting /auto-complete function and each have their own benefits. Most typical IDE’s use parsers and lexical analysers to read through a buffer stream of text and convert certain elements into tokens, these tokens can then be highlighted individually later on or taken into the auto-completer engine this is because they have a greater knowledge of the language and can understand syntax. However, code editors, which are more similar to that of this project, use regular expressions in order to search through the entire block of text at once and process each match at a time.

Lexical analysers are considered a lot smarter than regular expressions since they can take multiple files into scope whereas regular expression can only process what they can match. However, lexical analysers are not flexible whatsoever and a completely new analyser would need to be created in order to support a different language. Regular expressions do not have this problem; they can easily adapted to process different languages. In terms of efficiency, using regular expression highlighting is a lot more efficient; this is because we remove the need to tokenise an entire stream before processing it. IDE’s prefer analysers because they have a greater knowledge of the syntax and can therefore produce better error checking.

Since the project is not attempting to be an IDE there is no need for such overhead and inflexibility. Able aims to support a huge array of languages and regular expression help to make this happen.

## 3.4 Alternative solutions

There are many solutions available on the market that are similar to that of being developed for this project, some of the most popular packages include: Atom, Sublime and Notepad++. Integrated development environments such as Microsoft visual studios and Eclipse are not being considered as alternative solutions since IDE’s work differently. Although they contain syntax highlighting and autocomplete, they usually sacrifice the ability to support multiple languages and plug-ins with the integration of a compiler specifically tailored to that language, they are usually also more complicated to use.

## 3.4.1 Critical appraisal of Atom



Figure 3.4.1 – Atom editing window containing python code (Atom, 2013)

Atom prides itself as “a hack-able text editor for the 21st century” (Atom, 2013) and it calls itself this because it does what it says on the tin, the front end is completely hack-able. Atom works inside of a web environment using NodeJs and node-webkit to render web apps inside of a desktop window natively, which allows for the entire system to be built with JavaScript, Html and CSS (less). This allows for both, the software to be easily hacked since none of its code is compiled and for the software to be cross-platform. This also creates great plug-in opportunities because users can share their hacks on a large GitHub powered marketplace. The UI for Atom is extremely well designed and is similar to both sublime and Able. It attempts to attract users attention to key areas, such as the code editing window and file tree view whilst providing an extremely easy to use interface.

At this point Atom may seem as the perfect editor, but its not, a lot of programmers avoid using the software due to its in-efficiency. Since Atom is written in JavaScript and rendered in a web environment it is much slower than its competitors, and speed is very important. When attempting to control large projects or work on large files, atom will struggle.

## 3.4.2 Critical appraisal of Sublime

Figure 3.4.2 – Sublime editing window containing python code (Sublime, 2015)

At first glance Sublime resembles Atoms UI design greatly, but this is a commonality found in most popular code editors. The users eye focus is directed towards the editing window and file tree view and is again, extremely easy to use. Sublime isn’t hack-able, but it makes developing custom plug-ins very easy which also helped to generate a very large community of developers, so you can get a plug-in for pretty much anything. Apart from the plug in system (which uses python) the entire software package is written in C++ making it fast yet efficient and this is one of the main selling points of the editor. Even though the software is written in C++, sublime still can be downloaded and used on all of the major platforms: OS X, Windows and Linux.

Sublime is an all round great editor and doesn’t have many weaknesses, other than that it can be slow at times when there are a large amount of plug-ins installed (since it uses python) and that a license for the software will cost $70 (£45).

# 4. Designs

## 4.1 User interface

The early stage concept design for the software focused on a dark theme:



Figure 4.1.1 – Initial design concept of Able (10th October 2015)

Both the colour scheme and the layout attempt to replicate a clean and minimalistic design style. It is important that the software directs as much user eye focus towards the code editing text area as possible. Since the software could be used for extremely long durations, only mellow colours/shades were selected. The author was happy with the design and began to build a working version:

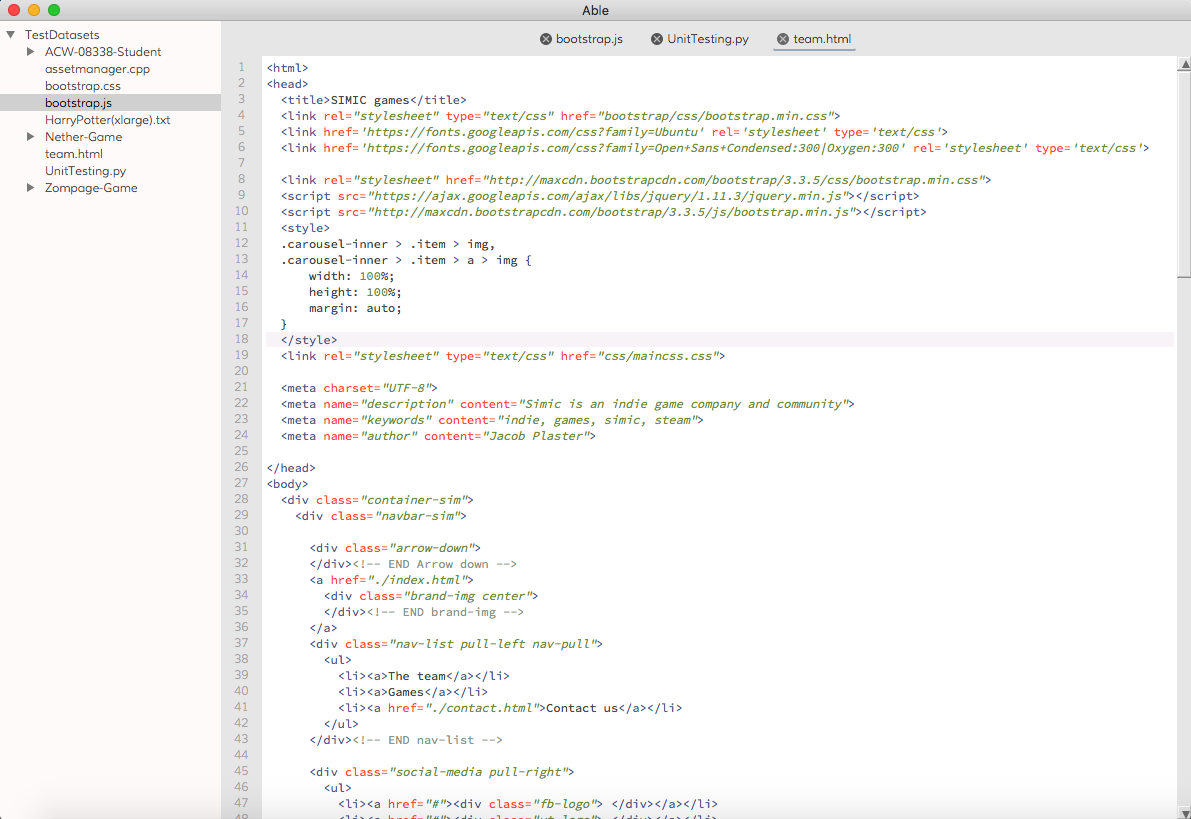


Figure 4.1.2 – working version of Able (26th December 2015)

As displayed in figure 3.1.2 it is easy to see that the base UI elements are exactly the same. The project manager tree, tabs bar and code editor window are all located in the same positions and look similar. However, the author has decided to switch the default theme from dark to light since it seems less stressful on the eyes, but there will be a dark theme included that the user can select and there is always the ability for the user to create their own.

When designing the UI, the author focused on 3 main functions for the user: programming environment, project management and task management (in that order of priority). As shown in figure 3.1.2 it is clear that the UI elements are split into these three categories with the attempt to drive most user eye focus based on its priority. The main programming panel stands out the most since it is the core functionality of the software, secondly, the project manager and lastly, the task bar. In order to minimise any confusion and to achieve a minimalistic design, everything else is removed from the main window and placed into a sub menu elsewhere.

## 4.2 System architecture

Before development began on Able a simplified UML class diagram was designed to outline Able’s main system architecture, which can be seen below. 

Figure 4.2.1 – A simplified UML class diagram of Able’s system architecture

As seen in figure 3.2.1 the simplified class diagram shows how the software is broken down into individual classes. In order to reduce complexity, all of the I/O functions are handled in one place, which is inside of the FileSystemManager class. The user interface has also been split up into separate objects, this allows for the software to more flexible in adding/removing UI objects, Qt has helped with this greatly since it provides templates of UI elements that can be attached to the windows.

The complex algorithms that are contained within the system architecture cannot be seen inside of this diagram due to the simplification. The language support class houses two complex algorithms for both the syntax highlighting and the auto complete functions. Both of these algorithms use regular expressions to search through the code editors ‘PlainText’ variable in order to find auto-complete suggestions and to colour certain elements to produce a syntax highlighting effect.

The asset manager is greatly important to the architecture of the software and this is because the software has been designed to include a plug in system. Upon initial load the main Able class tells the Asset Manager to find and load all plug-in related files and useful files (this can include CSS, language support files and more). The asset manager class is static, this helps to remove the need to constantly re-load resources.

## 4.3 System implementation

|  |  |  |
| --- | --- | --- |
| Language | Rule | Expression |
| Python | Class | [-a-zA-Z\_]+:\s |
| Auto-completer | [a-zA-Z]+[ ]\*= |
| Single line comment | [#][^\n]\* |
| C++ | Class | <.\*> |
| Auto-completer | [a-zA-Z]+[ ]\*[=;] |
| Single line comment | //[^\n]\* |
| HTML | Class | (</?[a-zA-Z0-9]+>?)|> |
| Auto-completer | Only supports keywords |
| Single line comment | <![-]\*.+[-]{2}> |
| CSS | Class | [-a-zA-Z\_]+:\s |
| Auto-completer | .[a-zA-Z]+ |
| Single line comment | //[^\n]\* |

## 3.1.3 Language support

There are over 500 different programming languages that exist today and whilst most code editors and IDE’s can only support a small number of them, able aims to give the user the ability to support all of them. Language support files are like plug-ins that can be implemented into the system directly. They contain a list of regular expressions and keywords which are then used by the syntax highlighter and auto-completer in order provide the same Able experience for that language. By default, Able already has support files for Html, Java, JavaScript, Python, CSS and C++. However, it is designed to be as simple as possible for users to create and implement their own.

Table 3.1.4 – Sample expressions taken from language support files

Table 3.1.3 shows some example rules taken from the language support files: python.cfg, cpp.cfg, html.cfg and css.cfg. The class rule handles functional and keyword related objects, the auto-completer rule handles variables that will be included in the auto-completers prediction engine and the single line comment rule handles code comments. These are just a few examples, a typical language support files would also include rules for multi-lined comments, operators, numeric values, Strings and more.

## 4.4 Testing

Parasoft was used to enforce a good standard of programming quality across the entire project; It also helped to generate a heat map of the most used functions within the program using its “Run time traceability” analysis tool. This helped the author to target key areas of the program when optimising code. Able was designed with Qt with the end goal of being published to an array of different platforms. To ensure that this was possible, the software was testing on all of the major operating systems (Linux, Unix (Mac OS) and Windows) with all of the popular hardware configurations. The below table contains all of the machines used:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Device | OS | Memory | CPU | GPU |
| MacBook Air | El Capitan | 4Gb | 1.3Ghz x2 i5 | Intel HD 5000 |
| Desktop | Windows 7 | 8Gb | 3.8Ghz x8 AMD | Nvidia GTX 660ti |
| MacBook Pro | Yosemite | 4Gb | 2.5Ghz x2 i5 | Intel HD 4000 |
| Desktop | Ubuntu | 8Gb | 3.8Ghz x8 AMD | Nvidia GTX 660ti |

Table 4.4 – List of machines that able was tested on

Standard usage tests were carried out on these machines to simulate a typical user usage scenario. Usage test consisted of: loading/saving and renaming files, editing large resource files, editing simultaneous files at once, editing files of various language and UI unctionality.

## 4.3.1 Unit testing

All unit tests were handled on the MacBook air machine with the El Capitan operating system (see Table 3.4), It is important that unit tests are carried out to ensure that the software does not deviate off of the planned design during the development period. Able aims to cope under high amounts of stress and usage whilst still remaining quick and efficient. In order to make sure this aim was achieved, many unit tests were carried out on the core functions of the editor, such as the load and highlight speed of a code file:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Date |  | Language | Support | File size (bytes) | Time (ms) |
|  |  | C++ | Full | 7208 | 143 |
| 26/12/15  (Full syntax highlighting) |  | Css | Full | 133614 | 854 |
|  | Js | Full | 61884 | 1018 |
|  | Text | None | 56450 | 154 |
|  | Html | Full | 7372 | 96 |
|  | Python | Full | 10732 | 87 |
|  |  | C++ | Med | 7208 | 122 |
| 05/11/15  (Basic syntax highlighting) |  | Css | None | 133614 | 733 |
|  | Js | None | 61884 | 245 |
|  | Text | None | 56450 | 169 |
|  | Html | None | 7372 | 32 |
|  | Python | None | 10732 | 65 |
|  |  | C++ | Med | 7208 | 137 |
| 29/10/15  (Basic syntax highlighting) |  | Text | None | 56450 | 167 |
|  | Html | None | 7372 | 36 |
|  | Python | None | 10732 | 38 |

Table 4.4.1 – Speed tests of Able’s load code function

Full support means the syntax highlighter successfully ran all of its regular expressions, medium means that only a small number of expressions were used and ‘None’ means no expressions were ran. Tests were used to analyse the performance on a regular basis to ensure that the efficiency of the functions was not being affected too much by the changes being made. However, when looking at the above table you can easily see that once the syntax highlighter was fully implemented it had a large impact on efficiently, especially with large files. On the 26/12/15 we can see that the JS language support system is inefficient because even though the file size is only 61884’bytes it still took 1018 milliseconds to run, after viewing these results the author is able pinpoint problem areas and optimise them.

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Name | Number of Items | Time (ms) |
| 26/12/15 | Folder-1 | 329 | 32 |
|  | Folder-2 | 306 | 36 |
|  | Folder-3 | 148 | 17 |
|  | Folder-4 | 8 | <1 |
|  | Folder-5 | 4 | <1 |
| 05/11/15 | Folder-1 | 329 | 30 |
|  | Folder-2 | 306 | 35 |
|  | Folder-3 | 148 | 12 |
|  | Folder-4 | 8 | <1 |
|  | Folder-5 | 4 | <1 |

Table 4.4.2 – Speed tests of Able’s load folder function

Another important process that is required by able is the ability to load entire project directories into the project manager tree view. Here the author measured the time it took to load projects with different amounts of items. This process is a lot less resource intensive than the code loading function because it only needs to quickly crawl a directory and return the absolute file paths of all of its contents, this is why the operation time is considerably less. The table proves that as the file grows in size, as does the load time.

# 5. Task List

## 5.1 Personal task list

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | Task name | Duration  (days) | Start date | Finish date | Notes |
| 1 | 08341 (Development project) initial report | 15 | 01/10/2015 | 15/10/2015 |  |
| 2 | 08341, 08348,08338 lectures | 85 | 28/09/2015 | 21/12/2015 | Lectures continue until exam revision begins |
| 3 | 08341 (Development project) Interim report | 15 | 06/10/2015 | 21/01/2016 |  |
| 4 | 08341, 08348,08338 Revision and exams | 37 | 21/12/2015 | 26/01/2016 |  |
| Christmas break | | 24 | 18/12/2015 | 10/01/2016 |  |
| 5 | 08334, 08346, 08130, 08341 lectures | 121 | 03/02/2016 | 02/06/2016 | Lectures continue until end of year exams |
| Easter break | | 19 | 14/03/2016 | 01/04/2016 |  |
| 6 | 08341 (Development project) Final report | 35 | 01/04/2016 | 05/05/2016 |  |
| 7 | 08334, 08346, 08130, 08341 Revision and exams | 23 | 02/06/2016 | 25/06/2016 | Academic year reaches its end after this event |

## 5.2 Project task list

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | Task name | Duration  (days) | Start date | Finish date | Notes |
| 1 | initial report | 15 | 01/10/2015 | 15/10/2015 |  |
| 2 | Product development | 150 | 31/10/2015 | 28/03/2016 | All development processes |
| 3 | Planning and requirements | 16 | 31/10/2015 | 15/11/2015 |  |
| 4 | Implementation part 1 | 31 | 15/11/2015 | 15/12/2016 |  |
| Christmas break | | 24 | 18/12/2015 | 10/01/2016 |  |
| 5 | Interim report | 15 | 21/12/2016 | 04/01/2016 |  |
|  | Implementation part 2 | 42 | 04/01/2016 | 14/02/2016 |  |
|  | Prototyping & testing | 21 | 25/01/2016 | 14/02/2016 |  |
|  | Software verification | 16 | 14/02/2016 | 29/02/2016 |  |
| Easter break | | 19 | 14/03/2016 | 01/04/2016 |  |
| 6 | 08341 (Development project) Final report | 35 | 01/04/2016 | 05/05/2016 |  |
|  | Final report (1st draft) | 13 | 03/04/2016 | 15/04/2016 |  |
|  | Final report (2nd draft) | 10 | 15/04/2016 | 25/04/2016 |  |
|  | Final report (final draft) | 11 | 25/04/2016 | 05/05/2016 |  |

## 5.3 Time plan

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **University Calendar Weeks** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **#** | **Task Name** | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| 1 | Initial report |  |  |  | D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Development of project product |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | Project planning stages |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 | Interim report |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | D |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Code implementation |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Prototyping & testing |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Software verification |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 | Final report |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | D |
|  | First draft |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Second draft |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

# 6. Risk Analysis

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Risk | Severity  (L/M/H) | Likelihood  (L/M/H) | Significance (Sev. x Like.) | How to Avoid | How to Recover |
| Loss of data | H | M | H | Create contingency backups | Recover from back ups |
| Loss of all data assets | H | L | H | Backup to cloud services | Recover from cloud |
| Failure to produce software that meets requirements | M | M | M | Ensure that the requirements stage of product planning has sufficient attention | Re-plan software requirements |
| External tasks (exams, revision etc.) interfering with time allocation | M | H | M | Plan for contingency time allocation | Adjust method of external task time management |
| Hardware defects | L | L | L | Make use of other university provided equipment | Reinstate from backups |
| Failure to implement 3ed party technologies | M | M | M | Allow for more time allocated to research | Switch to a technology which the author has more experience with |

7. Appendices

* 1. **Appendix A**

1. References

Advait Sarkar. (2015). *The impact of syntax colouring on program comprehension.* Available: https://www.cl.cam.ac.uk/~as2006/files/sarkar\_2015\_syntax\_colouring.pdf. Last accessed 6th March 2016.

Android Studio, 2016. IDE for the development of android applications [online]

Available at: http://developer.android.com/sdk/index.html [Accessed 2016 March 7]

Atom, 2015. Code editor [online]

Available at: http://www.atom.io [Accessed 2015 October 12]

Biomedcentral, 2015. Efficiency comparison of languages [online]

Available at http http://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-9-82 [Accessed 2015 October 14]

Businessdictionary, 2015. Software engineering definition [online]

Available at: http://www.businessdictionary.com/definition/software-engineering.html [Accessed 2015 October 12]

Microsoft, 2013. Visual studios 2013 [online]

Available at: <http://www.visualstudios.com> [Accessed 2016 March 3]

Notepad ++, 2016. Code/Rich text editor [online]

Available at: https://notepad-plus-plus.org/ [Accessed 2016 March 3]

Yourdictionary, 2015. Software engineering definition [online]

Available at: http://www.yourdictionary.com/software-engineering [Accessed 2015 October 12]

Sublime 2, 2015. Code editor [online]

Available at: http://www.sublimetext.com/2 [Accessed 2016 March 3]

QT Creator, 2015. QT IDE [online]

Available at: http: http://www.qt.io/download/ [Accessed 2016 March 7]

Walker, D. M., 2013. *Code editor with syntax highlighting & autocomplete.* Hull: University of Hull.