**Code editor with syntax highlighting & autocomplete**

**Interim Report**

Submitted for the BSc in   
Computer science

January 2016

by

**Jacob Plaster**

Word Count: 10,954 (+2,174) (+850)

Table of Contents

1. Introduction 4

1.1 Initial brief 4

1.2 Project context 5

2. Aims and Objectives 6

2.1 Primary objectives 6

2.2 Secondary objectives 6

3. Background 7

3.1 General context 7

3.1.1 Grammar analysis 7

3.1.2 Syntax highlighting 8

3.1.3 Auto complete 10

3.1.4 Software functionality 11

3.2 Comparison of technologies 12

3.2.1 Programming language 12

3.2.2 Frameworks and GUI API’s 14

3.3 Comparison of algorithms 16

3.3.1 Syntax highlighting techniques 16

3.3.2 Lexical analysis 17

3.3.3 Autocomplete matching 18

3.4 Alternative solutions 19

3.4.1 Critical appraisal of Atom 19

3.4.2 Critical appraisal of Sublime 20

4. Designs 21

4.1 User interface 21

4.2 System architecture 22

4.3 System implementation 23

3.1.3 Language support 23

4.4 Testing 24

4.3.1 Unit testing 24

5. Task List 26

5.1 Personal task list 26

5.2 Project task list 27

5.3 Time plan 28

6. Risk Analysis 29

7. Appendices 31

5. References 32

Android Studio, 2016. IDE for the development of android applications [online] 32

Atom, 2015. Code editor [online] 32

Biomedcentral, 2015. Efficiency comparison of languages [online] 32

Businessdictionary, 2015. Software engineering definition [online] 32

FLTK, 2014. Fast light toolkit C++ framework [online] 32

Microsoft, 2013. Visual studios 2013 [online] 32

Notepad++, 2016. Code/Rich text editor [online] 32

Ultimate++, 2016. C++ GUI framework [online] 32

Yourdictionary, 2015. Software engineering definition [online] 32

Sublime 2, 2015. Code editor [online] 32

QT, 2015. C++ development framework [online] 32

QT Creator, 2015. QT IDE [online] 33

# 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.

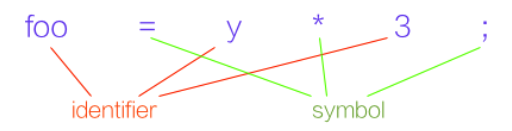


Figure 1.2 – Lexical analysis of assignment statement

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. Some code editors adopt a regular expression finite state machine approach, giving 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 Comparison of technologies

## 3.2.1 Programming language

Code editors are expected to work under extreme amounts of stress and this can be a very hard task when there are constant complex autocomplete, syntax highlighting and input monitoring algorithms running in the background. It is also imperative that the system provides 100% reliability and security since it is potentially handling extremely valuable assets (the users work). Whilst researching into high performance languages, these few were considered:

Java (7)

Java is a high level, general purpose, object-orientated programming language, which is renowned for being reliable and portable. This is due to its compiling stages, which allows for the post-compiled byte-code to be interpreted on a virtual machine. A study by Laxmi Joshi into the popularity of java: “After its birth it became popular because of many reasons like security, robust and multithreadedness but mainly because of its portable and platform independent. The logic and magic behind its platform independence is “byte code”” (Laxmi Joshi, 2014). However, Java’s JIT compiler causes for an average performance rating. When compared to a language like ‘C++’, Java suffers to produce great results when crunching large complex algorithms due to its lack of support for references and pointers.

Python (3.4.2)

Python is also a high-level, general-purpose, object-orientated programming language. But it is renowned for its ease of development. Pythons dynamic typing system and minimal syntax means that you can do more stuff with less code. Therefore, the speed of production is drastically improved. Python’s performance is very bad, even worse than java and this is because of both the JIT compiler and the fact that the language is dynamically typed. Pythons values are not stored in speedy buffers but in scattered objects.

C++ (11)

Like Python and Java, C++ is an object orientated programming language. Although C++ is considered a high level language, many people argue that it isn’t since it allows for doing things outside the abstraction of the language. C++ is renowned for its extreme speed and is used mostly for high performance application such as games, low level processes and general applications. However, C++ is sometimes avoided due to its slow development speeds and difficulty of debugging. Reliability can also be compromised when using C++ since the programmer is allowed to manipulate direct memory addresses.

It was decided that due to the massive performance advantage that C++ has over the other languages it would be chosen for this project. A study by Biomedcentral (Biomedcentral, 2015) provided an accurate display of the comparison of speed between the languages, as shown below:



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

## 3.2.2 Frameworks and GUI API’s

This is the framework that will hold the entire application together. Choosing a suitable solution is very important since it has a large affect on the development speed of the application.

QT (QT, 2015)

Qt is a cross-platform, highly documented and well-funded framework that has a large community of followers. Qt handles both the application life cycle and the GUI of the application; also it is written in C++ for C++. Qt provides a tailored IDE called ‘Qt creator’ (Qt Creator, 2015) which is developed especially for programming with the QT framework. The IDE gives the programmer a drag and drop widget interface which makes designing UI’s extremely quick and easy, which is shown below:



Figure 3.2.2 – QT’s GUI tool (QT Creator, 2015)

Widgets are QT’s GUI objects which are pre installed with every version of QT, programmers can create their own or rebuild upon the pre-existing widgets that are provided. This makes adding complex functionality such as buttons and text edit areas extremely easy.

The CSS parser that comes built into QT makes it much easier to apply style to the GUI of the application and would be extremely useful later on in the development of the project when the ‘plug-in’ system is implemented. This means that a user might be given the ability to create their own style-sheet and completely change the look of the application with out re-compiling the source code of the editor. A sample of QCSS (QT’s adapted version of CSS):



Figure 3.2.3 – Example QCSS code

Qt is renowned for being completely cross-platform, meaning that the code can be written once, but ported onto multiple devices. Qt is published under the ‘GNU’ license which means that developers are free to do what they want with the software as long as its open source.

Ultimate++ (Ultimatepp, 2016)

Like Qt, Ultimate++ is a C++ framework. Ultimate++ aims to reduce code complexity by making it as easy as possible to quickly build application using its personal IDE. Which is shown below:



Figure 3.2.4 – Ultimate++’s IDE (Ultimatepp, 2016)

Since Ultimate++ is not cross platform means that it does not have to regulate the build process as intensely as QT, making it much simpler to both understand and to program. However, this also means that the software produced by this library can only be ran on MS Windows and Linux machines which may restrict the target audience for the end product.

Ultimate++ has an active community with well-documented libraries, a simple problem can be solved pretty quickly simply by browsing through their forums and examples. But it is not as highly funded or as supported as other frameworks such as QT. Ultimate++ is released under the BSD license, which imposes very minimal restrictions.

Fast light toolkit (FLTK, 2014)

Fast light toolkit (FLTK) library is a cross platform graphical control platform. Originally, FLTK was developed in order to house 3D graphics but has been re-directed towards general applications. Using its own widget system, drawing events and openGL interface it allows writing graphical applications easy that look the same on all operating systems.

FLTK does not handle elements that Ultimate++ and Qt handle, such as the application life cycle in order to remain as lightweight as possible. A standard hello world application is usually around 100KiB in size. FLTK is released under the same license as QT, meaning that all source code of the software developed using this library needs to be public.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Name | Cross- platform | Features | Powerful IDE | Documentation/Community | Updates | Licensing |
| QT  (QT, 2015) | ✓ | LOTS | ✓ | GOOD | FREQUENT | GNU |
| Ultimate++  (Ultimatepp, 2016) | 🗶 | LOTS | ✓ | OK | ANNUAL | BSD |
| FLTK  (FLTK, 2014) | ✓ | FEW | 🗶 | VERY BAD | ANNUAL | LGPL |

Table 3.2.5 – Comparison of frameworks

QT’s feature rich, highly documented and powerful framework makes creating cross-platform software in C++ much easier. Due to the results shown in table 3.2.5, QT will be selected as the framework that the project will be developed upon. The extra features that QT provides such as CSS parsing and built in asset management system will help tremendously later on when implementing the secondary aims.

## 3.3 Comparison of algorithms

Comparing and selecting the most efficient algorithms possible is imperative to a project such as this, which aims to build a speedy piece of software that is capable of handling valuable user data. When analysing the projects key program cycle, it quickly became obvious which functions and algorithms would be running most regularly. These key areas would require a major amount of research into optimisation in order to produce the most successful product. These are:

## 3.3.1 Syntax highlighting techniques

In order to provide a fluent code-editing interface, the syntax highlighting algorithms would have to run very regularly. At least, when a new file is opened and whenever the code it manipulated, so it is very important that these tasks take as little time as possible.

There are two possible methods to choose from that are used in modern day editors. Firstly, there is the common standard way, which is plain ‘Code-Highlighting’. This consists of simply searching through entire code files and matching tokens every time the algorithm is run. Alternatively, there is the more efficient ‘Block-Highlighting’ model. This algorithm is more complex and recognizes that the code file can be split up into separate blocks. Then, whenever the user changes the displayed code, the algorithm can update the highlighting of the isolated block instead of re-highlighting the entire file. Before beginning any performance tests, it was theories that the ‘Block-Highlighting’ algorithm would be faster during run time (re-highlighting), but slower when initialising the editing interface.

This test was performed by running both of the algorithms in two different scenarios: Firstly, when the code file is first loaded/initialised (requiring the entire file to be highlighted). Secondly, when a change has been through the editing interface to the displayed code (requiring a re-highlight).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | File-size (Chars) | Language | | Time to initialize (MS) | Time to re-highlight (MS) | |
| Standard CodeHighlight | 7372 | HTML | 44 | | | 42 |
| 133611 | CSS | 269 | | | 280 |
| 10732 | Python | 31 | | | 45 |
| 61883 | JavaScript | 407 | | | 388 |
| Advanced  BlockHighlight | 7372 | HTML | 75 | | | 0.34 |
| 133611 | CSS | 340 | | | 0.36 |
| 10732 | Python | 48 | | | 0.23 |
| 61883 | JavaScript | 622 | | | 0.34 |

Table 3.3.1 – Performance test results

The results show that the standard ‘CodeHighlight’ algorithm has a slightly quicker initialisation speed since it has fewer overheads. However, whenever the algorithm is used for re-highlighting, its speed is considerably slower than the advanced ‘BlockHighlight’ algorithm. For this reason, the advanced ‘BlockHighlight’ algorithm is much more suitable for this project.

## 3.3.2 Lexical analysis

In order to perform functionality such as syntax and auto-completion the software needs to understand the syntax and semantics of the language. This is done through lexical analysers. Typically there are two types of common lexical analysers used in modern code editors.

Ad-Hoc analysing

Ad-hoc scanners work in a very simplistic manner. They iterate through a character stream one-by-one and depending on the value given place the input in either a token list or a buffer. Ad-hoc analysers are not written for general purpose and can only perform on a specific language. For example, a Python ad-hoc scanner could not be re-implemented to work with JavaScript.

The major advantages of using Ad-Hoc analysers is the fact that they can perform unique operations, such as the ability to look ahead: looking at one or more characters ahead in the sequence string in order to make a more informed decision. When used in compiling, this gives the analyser the ability to make small optimizations to the code.

Ad-hoc scanners fail when attempting to accommodate multiple languages, so it is common for this approach do be ditched. In order to support another language, a completely new Ad-hoc scanner must be written.

Regular expression analysis

Instead of Ad-Hoc analyzing, it is also common for modern day editors to use regular expressions in order to describe a grammar. Regular expression analyzers take in a stream of expressions which each represent a single token type, together these form a finite state machine capable of describing the grammar.

One major advantage to regular expression lexical analysis is the fact that they can be re-used to support other languages. Because the finite machine consists of purely expression, simply changing some of the expressions can mean the finite machine can support a different language.

Regular expression analysers fail to implement some of the functionality that Ad-hoc scanners can provide. RegExp state machines cannot use ‘look-ahead’ and can very complicated to write since they do not allow for complex nesting.

In order to gain a better understanding of the performance differences between the two analysers, the author performed a test. Both approaches were used in order to tokenise a simple Python program consisting of different sized input strings.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input size (Chars) | Average time in milliseconds | | | Difference |
| Ad-hoc |  | Regular expression |
| 10732 | 22 | 25 | | 3 |
| 15871 | 35 | 43 | | 8 |
| 22163 | 56 | 70 | | 14 |

Table 3.3.2 – Performance test results

While the results do show that the Ad-hoc scanning approach is slightly more efficient, the difference is not that great. The flexibility that the regular expression finite state machine offers is much more suitable for this project.

## 3.3.3 Autocomplete matching

The core function of the autocomplete is to locate and build a dictionary of frequently used keywords; these are then used to predict the word that the user is typing. For example, In C++ if the user was to type ‘br’ then ‘break’ should be displayed as a suggestion.

In order to build a dictionary, the auto-completer needs to run algorithms which check the recently inputted string for new dictionary entries every time the user interacts with the editing interface. In order to locate the potential dictionary entries a search function is used. In this section we will compare the custom built ‘Contains-token’ search function with QT’s (Qt, 2015) pre-existing simple “string-search” algorithm. QT’s string-search algorithm is based on a linear model, meaning that it searches through the given sequence until a match is found or the resource is exhausted. Although linear searching is the simplest model, there are many more efficient methods such as ‘binary search’ and ‘hash-table searching’. Whereas, the custom built algorithm uses regular expressions to find a match.

This test was performed by measuring the time it took for each of the algorithms to match the tokens of a Python input string

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input size (Chars) | Average time in milliseconds | | | Difference |
| QT’s string-search |  | Custom contains-token |
| 10732 | 0.5337 | 0.033 | | 0.5007 |
| 15871 | 0.9012 | 0.0597 | | 0.8415 |
| 22163 | 1.453 | 0.1108 | | 1.3422 |

Table 3.3.3 – Performance test results

These results clearly show that the custom searching algorithm ‘contains-token’ is much faster than QT’s built-in search algorithm ‘string-search’. Due to this, this algorithm will be used for this project.

## 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. This allows for the entire system to be built with JavaScript, Html and CSS (less). This means the software can be easily hacked since none of its code is compiled and also allows 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 sublime (Sublime, 2015). 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 working on large projects, 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. Technical development

## 4.1 System Design

The following details the planning process and decision made before commencing development. The author named the program “Able”. This relates to the software being ‘able’ to work efficiently, reliably and under high amounts of stress without any complications. In order to understand the users use case scenario, a UML diagram was developed:



Figure 4.1 – Use case diagram

Figure 4.1 displays a UML diagram of the basic Able system. The system displays the user very generically; this is because the software is designed for multi-purpose use and can accommodate all programmers, no matter what language they choose to write in. The UML diagram outlines the 3 key areas of the software: “File management”, “Settings” and “hack”.

The file management system allows the user to manipulate OS directories by giving the user the ability to create, rename and delete files. This will make managing projects a much simpler task. The file system also houses the most important function of Able, which is the ‘Code editor’. This is where the user is able to manipulate code files with the assistance of syntax highlighting, auto-completion and indentations. Furthermore, it will give the user basic text editing functionality such as copy/paste, write/delete and undo/redo. The user will be given the option to change any settings that affect the functionality of the software via the ‘Settings’ system. Here, the user will be able to change line spacing, font and indentation margins.

The ‘Hack’ system, which allows users to manage their custom themes and language support files, exists to give the software a customizable functionality. Users will be able add QSS (.qcss) and CFG (.cgf) files in order to give the entire software a different look or support a different language.

## 4.1.1 Specifications

The below specification displays the key requirements of the software and how each element will collectively come together in order to form a successful piece of software. It also describes the kind of functionality that a typical user might expect when using software such as this.

1. **File Management**
   1. Load
   2. Save
   3. Rename
   4. View
2. **Text editing**
   1. Manipulate code
      1. Paste/Cut/Copy
      2. Undo/Redo
      3. RegExp Find/Replace
   2. Syntax-highlighting
      1. Change syntax support
   3. Auto-completing
      1. Change auto-completion support
   4. Indentation/auto-indentation
3. **Usability**
   1. Cross-platform
   2. Efficient
   3. Reliable
   4. Light-weight
4. **User Interface**
   1. Change theme
   2. Control file management
   3. Control text editor
   4. Clean/Minimalistic
5. **Customization** 
   1. Plug-ins
      1. *Theme*
      2. *Language support*
   2. Load plugin
   3. Change editor preferences
      1. *Font*
      2. *Indentation*
      3. *Line-spacing*

## 4.2 System architecture

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

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

As seen in figure 4.2.1 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.

The complex algorithms that will be implemented within the system architecture cannot be seen inside of this diagram due to the simplification. The language support class will house two complex algorithms for both the syntax highlighting and the auto complete functions. Both of these algorithms will 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 will tell 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 will work statically, this is so all objects will be able to use the contained resources without having to re-load them.

## 4.2 Modular design

Modular design is a software design technique, which theorises that separating the software’s core elements into independent modules will improve the author’s ability to control the system later on. The act of splitting into modules is known as a “Separation of concern” (Phillip Laplante, 2007). Since the system is composed of separate modules it makes it a lot easier to interchange, update or remove modules in the future. “Computer programming is an intensive, hands-on design process. Modular Program Strategy (MPS) makes the computer programming process less challenging” (Wangping Sun, 2012). The Figure below illustrates how Able’s core functionality will be separated in to a series of widgets and modules:



Figure 4.2.2 – Simple diagram of Able’s widgets and modules

Figure 4.2.2 shows how Able will be split into a collection of widgets and modules. It also shows how each widget is connected to a parent node via a hierarchy structure. This means that widgets can be further modularised and large widgets may be comprised of multiple children widgets. The ‘Module’ tree depicted in figure 4.2.2 shows how the containing modules do not directly communicate with the widgets and are separate entities.

## 4.3 System functionality

The system functionality is a very important aspect of the planning stages since this is details how the main elements of the system work and communicate with each other. Commonly, flow diagrams are used in this situation since they can detail an entire program without adding too much complication. Once the main system architecture had been decided the author used a flow chart to plan out the systems functionality and how it would come together in order to provide the user with a simple functional interface:



Figure 4.3 – Simple flow chart of Able

Figure 4.3 illustrates the functional flow of Able. Firstly, the program will begin by displaying the user with a simple welcome screen, which provides the user with the ability to either open or create a file. Once a file has been successfully obtained, the code-editing interface is initialised and loaded. By looking at figure 4.3 it is obvious that the flow of the program evolves around the code editor and most interaction leads back to the editing interface. All further functionality once the code editor has been initialised is initiated through user interaction; this can be either a key press or the selection of a button. If the interaction leads to the code being changed then the autocomplete and syntax highlighting algorithms are prompted to begin adding new keywords to dictionaries and re-highlighting the current block of code. Once these tasks have finished, the editing interface returns back to its original waiting state. If the user selects to save a file, the file system is contacted and the editor is re-initialised.

## 

Figure 4.3.1 – Auto completer (left) and syntax highlighter (right) flow charts

Figure 4.3.1 provides a more in depth plan of the “Re-highlight syntax” and “Add/suggest auto correction” processes that are displayed in figure 4.3. The autocompletes functionality is fairly simple, since it decides to either provide a completion suggestion or add a new entry into the dictionary. Whereas, the syntax highlighter is slightly more complex. Here, the flow chart shows that the syntax highlighter performs match detection on the input string with an array of regular expression. If a match is found then the colour token for that expression is applied to the matched area. The process then loops back to the beginning if there are still expressions to apply.

## 4.3 User interface

As stated previously, the user interface design was especially important. This is because, naturally, code editors are often used for extreme periods of time, which can have a huge effect on both performance and motivation of the user if the visual elements do not suffice. In order to combat this the author has decided to adopt a “Minimalist” approach, meaning that the UI will be designed to be as least complex as possible. Research into the effectiveness of simple design stated:

“We started with the obvious notion that in order for the CE devices to be usable to everyday users, the UI must be simple. To be simple, we believe that the UI must diligently an consistently adhere to three principles: minimum, intuitiveness and consistency” (Won Kim, 2007).

In order to produce a successful user interface, the author will design the interface around the principles of it being minimum, intuitive and consistent. In this section it will be discussed how these principles were implemented into the design of this project.

Initial concept artwork for the software was developed:



Figure 4.3.1 – Initial design concept of Able (10th October 2015)

Figure 4.3.1 shows the initial plan for the main window of the application. These drawings were creating inside of a simple digital art program. Both the colour scheme and the layout attempt to replicate a clean and minimalistic design style. This is to try and direct as much user eye focus towards the code editing area as possible. The simple colour scheme of blacks, greys, blues and greens create and very relaxing environment for the user.

As seen in the concept art, the main window is separated into three main widgets. The file tree, tab bar and code editor. Further concept art was generated to further plan these individual elements:



Figure 4.3.2 – Initial design concept of file tree widget

Figure 4.3.2 shows multiple concept images of the ‘fileTree’ widget that were created during the first draft. The furthest left sticks to a very minimal design and removes the complications of any icons. The middle design attempts to illustrate open/closed files with the use of small arrows, which tell the user if the file has been expanded, or not. The furthest right draft also uses this approach. However, instead of using small arrows, it conveys this message through the use of colours. Orange means that the folder has been expanded, whereas yellow means it is not.



Figure 4.3.3 – second design concepts for the file tree

Figure 4.3.3 displays the adaptations that were made in order to produce the second lot of concept images. Here, you can see that the author combined all of the elements for figure 4.3.2. The arrow icons were added but kept to a very minimalistic style. Since Able is being designed to accommodate multiple themes, the author created concept images for both a dark theme and a light theme.



Figure 4.3.3 – Initial design concept for the tab widget

Figure 4.3.3 shows the very early stages of design for the ‘tab’ widget. Although this design conformed with the minimalistic art style that Able, as being designed around it was not very functional user. It was not obvious which tab was in focus and the un-focused tabs drew too much attention to themselves. For the second iteration of concept images, the author created a more simple design, which drew more attention to the tab in focus:





Figure 4.3.4 – Second design concepts for the tab widget

In figure 4.3.4, which displays the second design concept for the tab widget. The focused tab now has a brighter font and a sleek underline, which attracts user eye focus towards that element. Both the light and the dark theme use a more simplistic font style as well as cantered alignment.



Figure 4.3.5 – Design concepts for code editor widget

Once the main widgets had been planned, the author began decided how to display the code to the user in the most effective way. Figure 4.3.5 displays two different syntax highlighting colour schemes being applied to both, the dark and the light theme. A study by David Beymer (David Beymer, 2008) states “Using the overall speed metric, the serif font, Georgia, was read 7.9% faster than the san serif font”, taking this into account, the initial concept design of the syntax highlighting used the Source Code pro (SourceCode Pro, 2016) which was designed especially to display code text.

Figure 4.3.2 – Able’s main window (light and dark theme)

Figure 4.3.2 shows the final interface for Able. By default, able comes with both a light theme and dark theme pre-installed and can be changed using a simple button in the settings menu. Colour pallets that generate a chilled environment were chosen.

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 4.3.2 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.

Although the UI is designed to be as minimal as possible, all of the desired functionality is accessible by the user. A good example of this is the intuitive footer bar, which is located at the bottom of the code-editing interface. This widget houses useful code editing functions such as ‘search and replace’, quick access to settings, the ability to change highlighting colour schemes and useful information such as the cursors position within the editor.



Figure 4.3.3 – Able’s footer bar (collapsed)

By default the bar remains collapsed and uses only a small margin of the windows view space, this makes the bar less distracting. This can be seen in Figure 4.3.3 which shows the collapsed bar. When the bar is in its ‘collapsed’ state, the user has access to a search function, syntax style changing and cursor information. However, by simply clicking the small arrow (located on the right of the bar) will cause for the bar to enter is ‘expanded state’:



Figure 4.3.4 – Able’s footer bar (expanded)

Now the user is able to access more features, such as a replace function, quick access to settings and further information on the currently open file. Clicking the small arrow will return the bar back to its collapsed state.

## 4.4 System implementation

## 4.4.1 Text editor



Figure 4.4.1.1 – Able’s text editor highlighting a html file

Figure 4.4.1 shows the fully functioning text editor that has been implemented into Able. In this section, the author will discuss the stages of implementations that led to this finished product.

As compared in section 3.2.2 “Comparison of technologies”, the QT (QT, 2015) framework was found to be the most relevant to this project. One of the advantages of using QT was the widget system that it provided; all graphical elements provided by the framework are used as widget object. Developers can create/manipulate any widget to fit their needs. In this case, the text-editing interface shown in figure 4.4.1 is derived from the core widget “QPlainTextEdit”. However, apart from the ability to enter text, the “QPlainTextEdit” widget does not provide much functionality in terms of code editing. Below is an example of how a widget might be created:



Figure 4.4.1.2 – Example of custom widget creation

Figure 4.4.2 shows an example of how a developer might create their own QT widget and override an existing one. In this illustration, the displayed code is an extremely shortened version of the CodeEditor widget created by the author for use in Able. Widgets communicate themselves through the use of ‘hooks’ and ‘sockets’ and this can been seen inside of the “protected” claim in figure 4.4.2. A hook for a key-press event that is inherited form the QPlainTextEdit widget has been overridden and adopted by the CodeEditor widget. In the case of Able, this key press event will later house the functionality for updating the syntax highlighter and firing the autocorrect.

## 4.4.1.1 Syntax highlighting

As discussed above, the syntax highlighting functionality is fired whenever the overridden keyPressEvent method is triggered. This causes the syntax highlighter to re-highlight the code. The syntax highlighter provides the effect by matching any tokens within the changed the code and applying a foreground colour change using the built in “SetFormat” function. This applies a defined format to piece of text. Figure 4.4.1.3 illustrates an example of this.



Figure 4.4.1.3 – Example of custom widget creation

Figure 4.4.1.3 shows the algorithm that is implemented in Able to highlight syntax. As can be seen in the above illustration, a “HighlightingRule” object contains both a regular expression and an associated syntax colour hex value. The object “ruleSet” contains a list of “HighligtingRules” which collectively describe the grammar of the selected programming language. The displayed algorithm increments through the list of grammar rules and matches the contained regular expressions against the block of code. If there is a match then the syntax colour stored in that rule is applied to the foreground of that text item and thus causing for the syntax highlighting effect shown in figure 4.4.1.4.

Figure 4.4.1.4 – Two different rule-sets being applied to a C++ file (Able)

A “ruleSet” may contain as many as 18 rules and, each rule is dedicated to matching a different aspect of the code. For example, the default rule-set for C++ contains a rule dedicated to locating C++ operators and its associated expression looks like this “[-+/\*><?!=&|%]”. When this rule is reached by the highlighting algorithm then all text items that are matched with this expression will be applied with the hex colour “#CF5C51”. To the entire language support rule set for C++ please view appendix A.

Block changed optimisation

When working on large files it can be extremely taxing on the system if the syntax highlighting algorithm is forced to re-highlight the entire code file every time the users triggers the kreyPressEvent. In order to reduce the time taken to re-highlight, able is built to understand that code is written in blocks, so instead of re-highlighting the entire file able re-highlights the single individual block (Unless a multi-lined comment is matched, then all blocks within the comment are re-highlighted). Figure 4.4.1.5 shows how the algorithm interprets a code file in terms of ‘blocks’.



Figure 4.4.1.5 –Able’s text block feature example

## 4.4.1.2 Auto completion

The auto completion function that has been implemented into Able works hand in hand with the syntax highlighting mechanism, and even functions similarly. Initially, if the editing interface is initiated with an already existing file then the auto-completer is required to scan the loaded content in order to pick out future suggestions. This could be variable names, function names, keywords, operators and more. Once the loaded document has built its dictionary it waits for user interaction. Similarly to the syntax highlighter, the auto-completer is triggered via the overridden keyPressEvent discussed in section 4.4.1.

Once the auto-completer has been triggered via the keyPressEvent, If the pressed key is a new line initiator (Enter key), the algorithm assesses the changed block of code in order to locate any new potential suggestions. Again, these could be either variable names or function name. If the pressed key is not a new line initiator then the algorithm locates all strings within its dictionary that relate to what the user is typing. This functionality looks like this:



Figure 4.4.1.6 –Able’s auto-completer function (Python)

Figure 4.4.1.6 shows the auto-completer in full working order. The suggested values “del” and “def” are keywords reserved by python. The final suggestion “descriptionVar” is a dynamic variable created by the user. The auto-completer builds its dictionary by scanning the inputting strings with a regular expression, which can be found inside of the language support file. The auto-completer also shares the same optimization advantages as the syntax highlighter since it massively benefits from the “Block change” implementation. The auto-completer only performs dictionary update queries on the blocks that have been recently changed instead of scanning the entire document every time the user makes a change.

## 4.4.1.3 Editing functionality

Apart from the already existing tools that are pre-built in to QT’s plain text editor (QT, 2015) such as copy/paste and undo/redo. Able’s code editing interface has been equipped with more useful functionality, which caters for programmers more specifically. These features include:

* Regular expression search and replace
* Auto-indenting
* Automatic syntax recognition

The search mechanism can be found in the footer bar of the code editor. Clicking the ‘expand’ arrow will reveal the replace bar. If the user inputs a string into the search-box and selects the ‘search’ button, an algorithm that highlights any items that match the search string is triggered. Originally, the search mechanism fired every time the user changed the text within the search box, however this proved to very taxing when typing in larger words. Below is an example of the search method:



Figure 4.4.1.7 –Able’s search function (Html)

The algorithm that performs this action is similar to the syntax-highlighting algorithm. The user inputted expression is tested against the editors contained code, If a match is found then the background and text colour of that item is changed (The colour values are set in the language support file, see appendix A). This is repeated until all of the matches have been found. An example of this code is shown below in figure 4.4.1.7.



Figure 4.4.1.7 –Able’s highlight algorithm

Once the algorithm locates all of the matches, the user is able to enter a replace string, which will cause for all of the highlighted objects to be replaced with the replace string.



Figure 4.4.1.8 –Able’s replace function (Html)

Figure 4.4.1.8 displays the result of replace function being used. The code that in figure 4.4.1.7 has been processed by the algorithm using the regular expression “<[/a-zA-Z]+>?” all of the matches have then been replaced by the text “<test>”.

Auto-indentation is another useful feature added to the code editor. This function automatically adjusts the users code to the indent margin whenever the user presses the new line key.



Figure 4.4.1.9 –Able’s auto-indentation function (C++)

Figure 4.4.1.9 shows the effect of the auto indenter working when the user creates a new line. The dotted lines visualise the indent margins set by the position of each code line and the thick red bar shows the size of the indentation that the editor has automatically inserted. Again, this method is triggered by the keyPressEvent and whenever the presses the new line key and algorithm searches through the text above the newly created line in order to find its indentation margin. The below flow chart (figure 4.4.1.10) describes this process in more detail:



Figure 4.4.1.10 –Able’s auto-indentation getMargin() flowchart

The text cursor is then overridden and the number of tab indexes is automatically inserted. A tab index by default consists of 4 spaces, however this value can be changed inside of the settings section.

Able is designed to accommodate multiple languages and it can be time consuming if user has to change the highlighting rules to their desired language every time they open a new file. For this reason, Able has been equipped with a simple function that locates the language support plug-in that is associated with the syntax of the currently typed language. This works by taxing the details of the file extension (example “.txt”, ”.php”) and matching it against the plug in configurations in its database.

## 4.4.2 Plug-in system

Able comes equipped with an extremely easy to use plug-in system which was designed to work from its file directory system. Able is capable of providing plug-in support for both language-support files and custom theme CSS files. To import a new plug-in, the user simply has to navigate to their able install directory then to “libs” and add their new plug-in to either the “language\_support” file or the “CSS” file for themes. Once these files have been loaded they will automatically be recognized by the able asset-manager system.

Figure 4.4.2.1 –Able’s plug-ins being loaded

Figure 4.4.2.1 shows how the asset manager automatically loads the files from the language\_support directory. These files are then available to be applied from the footer bar of the code-editing interface. By simply clicking one of the plug-in in the combo box will cause for the editor to instantly switch to that support file. This functionality is the same for the theme CSS files, however the user is able to access these from within the able settings window instead of the footer bar.

The asset-manager is a static class that is used by all objects throughout software and acts as a resource object. It is the connection link to all external resources such as fonts, images and plug-ins. When able is first loaded, the asset manager pulls all of the necessary files from the specified locations in its configuration. More specifically for plug-ins, language support configuration files are then parsed into a SyntaxHighlightingRuleSet (as discussed in section 4.4.1.1 syntax highlighting) by the asset-manager and all themes are loaded into CSS objects.

## 4.4.2.1 Language support plug-ins

Language-support files are a collection of regular expression rules which describe a grammar and provide syntax-highlighting colour that were previously mentioned in section 4.4.1.1 (Syntax highlighting). As well expressions, these files contain other useful information such as the file types that they support which is later used by features such as auto-syntax recognition and the code editing interface. Every support file contains up to 16 grammar rules and each rule is dedicated to a different function. For example, a rule may be declared for matching single line comments, in which case its expression would be “[#][^\n]\*” for python comments and its hex colour value might be “#CF5C51”. Below is collection of rules taken from multiple language support files and compared:

|  |  |  |
| --- | --- | --- |
| 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]\* |

Table 4.4.2.1 – Sample expressions taken from language support files

Table 4.4.2.1 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. Since a support file merely consists of some regular expressions and colour values, creating them is very simple and quick, below the table 4.4.2.1 rules are being applied (In Able):



Figure 4.4.2.2 – 4 different languages written with able

Themes

The QT framework (QT, 2015) comes pre-packed with its own “qcss” parsing system. This means that specially equipped “.css” files can be used on any application created with QT. Able has harnessed this function and utilised it to create its own plug-in theme system, users can write their own (or download) style sheets and apply them to their versions of able with ease. Below is example code taken from the light-theme CSS style-sheet:



Figure 4.4.2.3 –Part of Able’s light-theme CSS

Figure 4.4.2.3 shows some example code taken from Able’s default light theme. This CSS code is being applied to the buttons on the footer bar of the code editor. Whenever the code-editing interface is initiated, this code is automatically applied to its elements. The user has complete control over the interface elements of Able through the use of the theme system, below are two examples, a light theme and a dark theme that come pre packed with Able:

Figure 4.4.2.1 –Able’s plug-ins being loaded

## 4.4.3 File tree

The implementation of Able’s file tree system presented a few challenges. Firstly, the UI dimensions of the widget had to be responsive since it was designed to collapse whenever it was unpopulated and therefore providing more window space for more important widgets such as the code editor. The figure below displays a window with the file tree expanded and collapsed:

Figure 4.4.3.1 –Able’s file tree collapsed (left) and expanded (right)

Figure 4.4.3.1 shows how the file tree widget collapses and causes for the entire UI to resize. Another challenge that the file tree presented was the difficulty of gathering and populating the tree widget with data scraped from the selected directory. In order to solve this, a recursive algorithm needed to be developed that scanned through the given directory and returned any files. If the algorithm came to a folder then it would need to execute the algorithm on that file, this is called recursion. A simple flow chart was created to explain this more simply:



Figure 4.4.3.2 –Able’s file tree getFiles() algorithm in flow chart form

With this algorithm, the able file tree widget is able to locate all files that are related to a given file path. For example, if a user were to open their documents folder then all of the containing folders and files would be displayed in the file tree widget. See below figure:



Figure 4.4.3.3 –Able’s file tree populated with documents folder

## 4.5 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 4.6 – 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 functionality.

## 4.5.1 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**

%FILE\_FORMATS

cpp

h

%comment\_Start\_Expression:

/\\*

%comment\_End\_Expression:

\\*/

%operator\_Format:

#CF5C51

[-+/\*><?!=&|%]

%number\_Format:

#708D44

\b[0-9]+\b

%keyword\_Format:

#ED7A6F

alignas

alignof

and

and\_eq

asm

auto

bitand

bitor

bool

break

case

catch

char

char16\_t

char32\_t

class

compl

concept

const

constexpr

const\_cast

continue

decltype

default

delete

do

double

dynamic\_cast

else

enum

explicit

export

extern

false

float

for

friend

goto

if

inline

int

long

mutable

namespace

new

noexcept

not

not\_eq

nullptr

operator

or

or\_eq

private

protected

public

register

reinterpret\_cast

requires

return

short

signed

sizeof

static

static\_assert

static\_cast

struct

switch

template

this

thread\_local

throw

true

try

typedef

typeid

typename

union

unsigned

using

virtual

void

volatile

wchar\_t

while

xor

xor\_eq

%function\_Format:

#515E8C

\b[A-Za-z0-9\_]+(?=\()

%class\_Format:

#99A6BC

<.\*>

%other\_Format:

#E69133

[A-Za-z]+::

%single\_Line\_Comment\_Format:

#9FACB3

//[^\n]\*

%multiLine\_Comment\_Format:

#9FACB3

%quotation\_Format:

#708D44

(\"(\\.|[^\"])\*\")|(\'(\\.|[^\'])\*\')

%autocomplete\_Format

[a-zA-Z]+[ ]\*[=;]

%autocompleteTrim\_Format

[^=; ]+

%CurrentLineHighlight\_Format

#EEE

%SearchHighlightBackground\_Format

#F3F709

%SearchHighlightBackground\_Foreground

#000000

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]

David Beymer, Daniel Russell, Peter Orton, 2008. *An Eye Tracking Study of How Font Size and Type Influence Online Reading.*[Online]. Available at: *http://www.bcs.org/upload/pdf/ewic\_hc08\_v2\_paper4.pdf* [Accessed: 2nd April 2016].

FLTK, 2014. Fast light toolkit C++ framework [online]

Available at: <http://www.fltk.org> [Accessed 2016 March 10

Laxmi Joshi. (2014). Case study: Java is secure programming language. *International Journal of Computer Networking*. 4 (2), p6-8.

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]

Phillip A. Laplante, 2007. *What every engineer should know about software engineering*.

CRC press. ISBN 0849372283.

Ultimate++, 2016. C++ GUI framework [online]

Available at: http://www.ultimatepp.org/ [Accessed 2016 March 10]

Yourdictionary, 2015. Software engineering definition [online]

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

SourceCode Pro, 2016. Code editor [online]

Available at: https://github.com/adobe-fonts/source-code-pro [Accessed 2016 April 2nd]

Sublime 2, 2015. Code editor [online]

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

QT, 2015. C++ development framework [online]

Available at: <http://www.qt.io> [Accessed 2016 March 10]

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.

Wangping Sun, Xin Wang and Xian Sun, 2012. *USING MODULAR PROGRAMMING STRATEGY TO PRACTICE COMPUTER PROGRAMMING: A CASE STUDY.*[Online].

Available at: *https://www.asee.org/public/conferences/8/papers/3155/download* [Accessed: 1st April 2016].

Won Kim, Seonghoon Kang (2007) *Minimalist and Intuitive User Interface Design Guidelines for Consumer Electronics Devices .*[Online]. Available at:*http://www.jot.fm/issues/issue\_2007\_03/column5.pdf* [Accessed: 1st April 2016].