

Code editor with syntax highlighting & autocomplete

Final Report

Submitted for the BSc in
Computer science

May 2016

by

Jacob Plaster

Table of Contents

1. Introduction	5
1.1 Initial brief	6
1.2 Project context	6
2. Aims and Objectives	8
2.1 Primary objectives	8
2.2 Secondary objectives	8
3. Background	10
3.1 General context	10
3.1.1 Grammar analysis	10
3.1.2 Syntax highlighting	11
3.1.3 Auto complete	13
3.1.4 Software functionality	14
3.2 Comparison of technologies	17
3.2.1 Programming language	17
3.2.2 Frameworks and GUI API's	18
3.3 Comparison of algorithms	22
3.3.1 Syntax highlighting techniques	22
3.3.2 Lexical analysis	23
3.3.3 Autocomplete matching	24
3.4 Alternative solutions	25
3.4.1 Critical appraisal of Atom	25
3.4.2 Critical appraisal of Sublime	26
4. Technical development	27
4.1 System Design	27
4.1.1 Specifications	28
4.2 System architecture	29
4.2 Modular design	30
4.3 System functionality	30
4.3 User interface	33
4.4 System implementation	38
4.4.1 Text editor	38
4.4.1.1 Syntax highlighting	39
4.4.1.2 Auto completion	40

4.4.1.3 Editing functionality	42
4.4.2 Plug-in system	44
4.4.2.1 Language support plug-ins	45
4.4.3 File tree	47
4.4.4 Workspace automatic saving	48
4.5 Testing	50
4.5.1 Formal and platform testing	50
4.5.2 Performance testing	50
4.5.2 Functional testing	52
5. Critical analysis	53
5.1 Project Achievements	53
5.2 Further development	54
5.3 Personal reflection	55
5.3.1 Past Technical Knowledge	55
5.3.2 Past Similar Projects	55
5.3.3 Management Abilities	55
5.3.4 Encountered Problems	56
5.3.5 Production of the report	56
5.3.6 Advice for future students	56
6. Ethical issues and risk analysis	57
6.1 Risk analysis	57
6.2 Ethical issues	57
7. Appendices	58
7.1 Appendix A – Language support file	58
7.2 Appendix B – Risk analysis	61
7.3 Appendix C – Personal task list	64
7.4 Appendix D– Project task plan	65
7.5 Appendix E – Planned Time plan	66
7.6 Appendix F – Planned Code implementation Task list and grant chart	67
7.7 Appendix G – Actual Time plan	69
7.8 Appendix H – Actual Code implementation Task list and grant chart	70
7.9 Appendix I– Able source code commit graphs	73
7.10 Appendix J– Able user manual	76
7.11 Appendix k– Able user manual	76
8. References	85

Acknowledgements

Firstly, I would like to thank Dr. Septavera Sharvia who supervised me throughout the production of this report. Without her advice and guidance I would not have been able to produce a project anywhere near the standard that it is today. Also, I would like to thank Dr. Darryl Davis who provided assistance during my 3 years at The University of Hull.

My friends, particularly those who I lived with, helped me stay both motivated and sane throughout my time studying. Our regular trips to the Gardner's pub and Brynmor Jones library helped tremendously. I wish all of you the best of luck in both your time here at the university and in the future.

Last but not least, I would like to express my sincerest gratitude to my family: Kenneth, Diane, Luke and Annabelle. For encouraging and supporting me through my education and helping me to get where I am today.

1. Introduction

Software engineering is:

“A discipline for solving business problems by designing and developing software-based systems. As with any engineering activity, a software engineer starts with problem definition and applies tools of the trade to obtain a problem solution. However, unlike any other engineering, software engineering seems to require great emphasis on methodology or method for managing the development process, in addition to great skill with tools and techniques”

(Ivan Marsic, 2012)

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 and auto-complete.

Code editors became possible when computers were able to utilise a graphical terminal. Systems before this were incapable of accommodating such software since they relied on the usage of punch cards and flow charts to operate (Fisk, 2005). As soon as computer systems were able of supporting software such as code editors they were quickly implemented due to the difficulty of reading and comprehending machine code. Hofstadter famously said that the difficulty of reading machine code is similar to looking at a DNA molecule atom by atom (Hofstadter, 2000). Originally code editors existing purely within the terminal interface and only provided simple functionality such as syntax highlighting. However, modern code editors are now able to utilise a full graphical interface and provide much more complex functionality such as advanced syntax highlighting, auto-completion, error handling, code reformation and more.

Syntax highlighting is one of the core functions of a standard code editor. It colour in specific keywords within the code in order to quickly draw the users attention/eye focus towards the more important elements. Research into the speed of program comprehension and syntax highlighting showed that “The presence of syntax highlighting significantly reduces context switches” (Sarkar, 2015).

Autocompleting aids the speed and accuracy of the user by providing suggestions that have been calculated whilst the user types. This function’s efficiency improves depending on the size of the sequence that the user attempting to type. Theoretically, auto-completion reduces the amount of keystrokes to complete a word by 50% (Herold, 2008). 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 code 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 functions such as compilers, smarter grammar understanding and more.

This report will aim to cover:

- The initial brief and project description
- The aims and objectives of the project
- The background on existing code editors and their features
- The design and implementation of technologies within development of this project
- The task list with detailed timelines and progression
- Ethics and risk analysis
- A personal reflection from the author

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 desired outcome of this project would be a code editor that 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 (Muslu, et al., 2012):

- 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 that 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

To develop a clean and minimalistic interface that 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:

- a. Create files
- b. Rename files
- c. Remove files
- d. Edit 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:

- a. User generated customization
- b. Machine learning autocomplete
- c. Multi-language support
- d. Compilers

1. User generated customization

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

2. Machine learning

Code editor with syntax highlighting and autocomplete

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

3. Multi-language support

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

3. Background

3.1 General context

A code editor is an essential piece of equipment that all software engineers and computer programmers would struggle to work without. They provide tools that automate common tasks such as auto-complete, text manipulation, syntax highlighting and more (Muslu, et al., 2012). 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 software should be especially easy for users to navigate around (Minelli, et al., 2016). 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. According to Paul R. Kroeger, in order to understand any language, you first need to analyse it lexically and then syntactically (Kroeger, 2005).

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 sequence of 'lexemes' and 'tokens' (Trim, 2016). A lexeme is the literal piece of source code and the token is its corresponding label. This is massively important because it means that the program is split up into understandable tokens such as 'keywords', 'symbols', 'numbers' and more. Once the program is broken down, the software has better knowledge of the grammar. This means functions such as auto-complete can locate individual words in the code such as identifiers in order to use them for suggestions later on and syntax highlighters can locate and visually manipulate the codes tokens.

For example, the equation "foo = y * 3" would be split up into its corresponding token categories, which in this case, are: "Identifier", "Assignment operator", "identifier", "multiply operator", "integer" and finally an "End of statement symbol" (Aho, 2006). To make this simpler, a graph has been formed which can be seen in table 1.2 below:

Lexeme	Token
foo	Identifier
=	Assignment operator
y	Identifier
*	Multiply operator
3	Integer
;	End of statement symbol

Table 1.2 – Lexical analysis of assignment statement

Syntax analysis

The syntax analyser is a lot more complex than the lexical analyser since this process must decide on what statements have been used depending on both the syntax and the tokens that have been used within the statement and if the statement is appropriate for the language (Klint, 2007). The program will then generate a data structure called a “parse tree” or “syntax tree”, using the tokens provided by the lexical analyser, which gives the code syntactic meaning. This can be seen in figure 1.2 below:

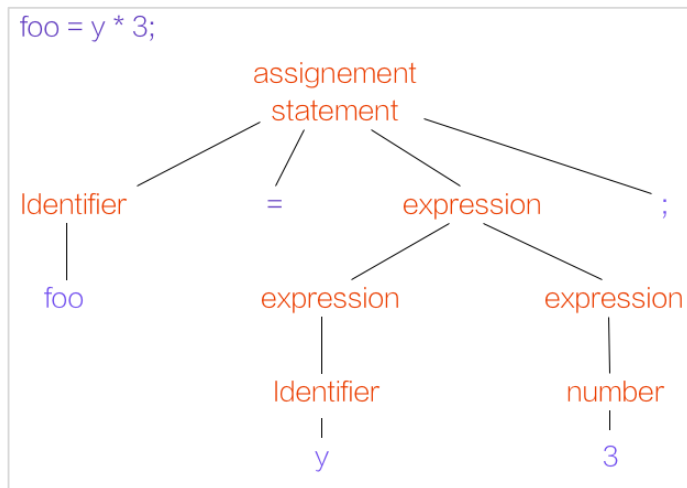


Figure 1.2 – Lexical analysis of a ‘C’ styled assignment statement

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 (Grune, et al., 2008). To describe a grammar, rules are defined in the form of a list. BNF is especially popular due to its ability to easily nest statements. Here is an example a BNF rule

```

<assignment_statement> ::= <expression> "=" <identifier>

<if_statement> ::= IF <conditional> THEN <statement_list> ENDIF
                | IF <conditional> THEN <statement_list> ELSE <statement_list> ENDIF
  
```

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 utilise the easiness of using regular expressions in order to match the tokens provided in the input stream.

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 (D’Anjou, et al., 2004). 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 (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:

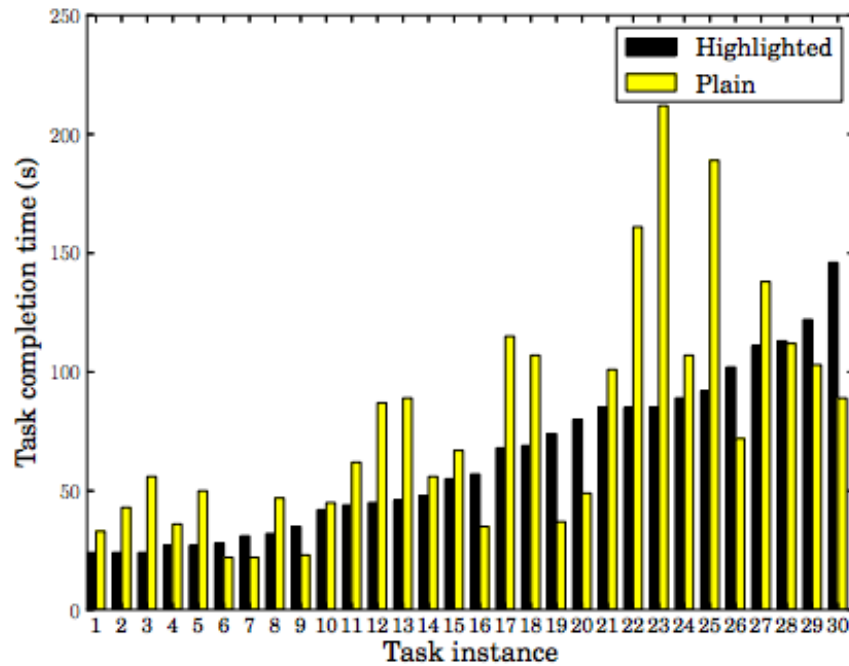


Figure 3.1.1 – Program comprehension times with and without syntax highlighting (Advait Sarkar, 2015)

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:

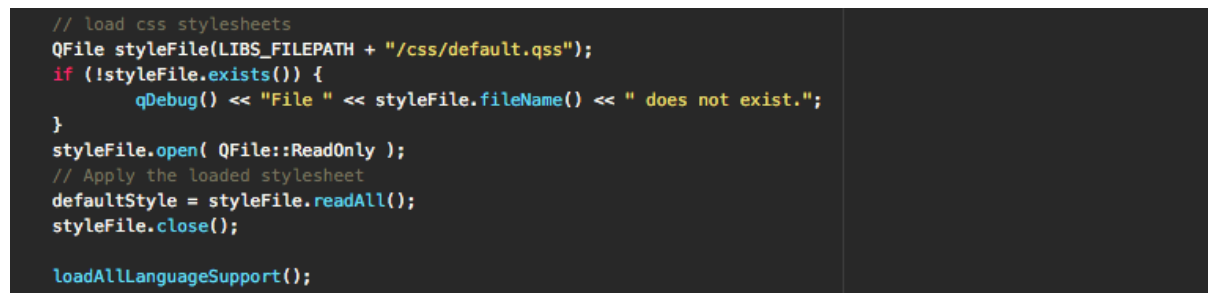
```
def loadVocab(self, index, inNormal, inResult):
    self._Vocabulary[index].append((inNormal, inResult))
def _getFromVocab(self, inIdentifier, inNormal):
    for index, i in enumerate(NaturalLanguageObject._Identifiers):
        # get the index
        if(inIdentifier == i):
            # locate normal
            for index2, val in enumerate(self._Vocabulary[index]):
                if(val[0] == inNormal):
                    return val[1]

def loadVocab(self, index, inNormal, inResult):
    self._Vocabulary[index].append((inNormal, inResult))
def _getFromVocab(self, inIdentifier, inNormal):
    for index, i in enumerate(NaturalLanguageObject._Identifiers):
        # get the index
        if(inIdentifier == i):
            # locate normal
            for index2, val in enumerate(self._Vocabulary[index]):
                if(val[0] == inNormal):
                    return val[1]
```

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

Figure3.1.2 outlines Notepad++’syntax highlighting the difference and the difference it makes when trying to understand the code. Notice how the keywords and identifiers (such as “IF”, “FOR” and “_getFromVocab”) have been visually enhanced.

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:



```
// load css stylesheets
QFile styleFile(LIBS_FILEPATH + "/css/default.qss");
if (!styleFile.exists()) {
    qDebug() << "File " << styleFile.fileName() << " does not exist.";
}
styleFile.open( QFile::ReadOnly );
// Apply the loaded stylesheet
defaultStyle = styleFile.readAll();
styleFile.close();

loadAllLanguageSupport();
```

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

This helps to improve the understanding and readability of the code. Knuth's literal programming notion, viewed software readability as the key element to being able to maintain and write code (Knuth, 1983). 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 (University of Virginia, 2007), this method makes it extremely hard to accommodate different programming languages due to the time consuming task of deeply defining a grammar in order to generate a parse tree. Some code editors adopt a regular expression finite state machine approach, and remove the concept of supporting a parse tree such as Notepad++ (Notepad++, 2016). This allows the software to still provide functions such as basic autocomplete and syntax highlighting but not very complex functions such as compiler integration. However, with a more simplistic approach these code editors are able to support a wider variety of languages.

3.1.3 Auto complete

Originally auto-completion functionality was invented to aid people with disabilities, but was implemented in other text editing tools such as code editors soon after (Tam, et al., 2009).

'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. Theoretically, word-prediction software should reduce the amount of keystrokes to complete a word by 50% (Herold, 2008).

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 that are capable of supporting a large array of languages, as discussed above in 3.1.2. 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 can perform extra functions (Vu, et al., 2005). 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:

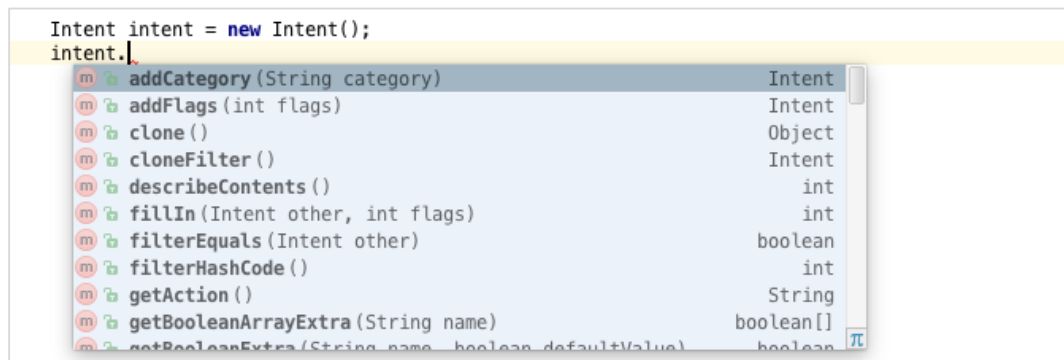


Figure 3.1.3 – Android studios IntelliJ autocomplete (Android Studio, 2016)

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:

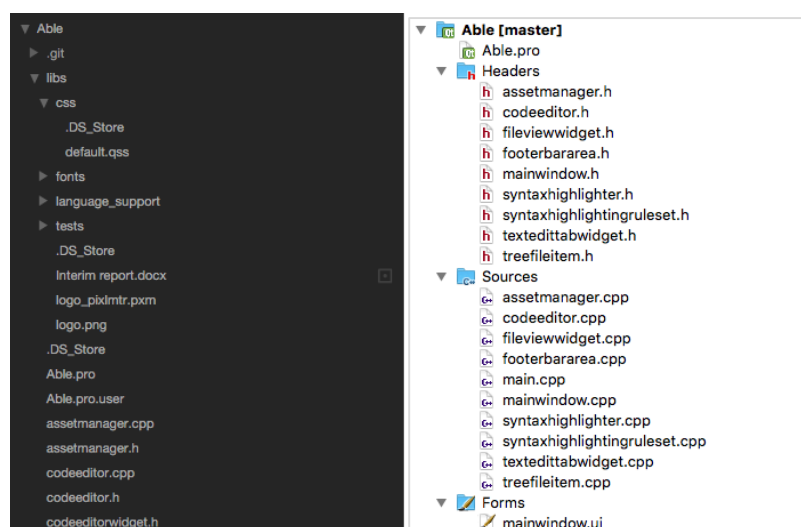


Figure 3.1.5 – Comparison of project management systems (Atom (Atom, 2015) on the left and QT Creator (QT Creator, 2015) on the right)

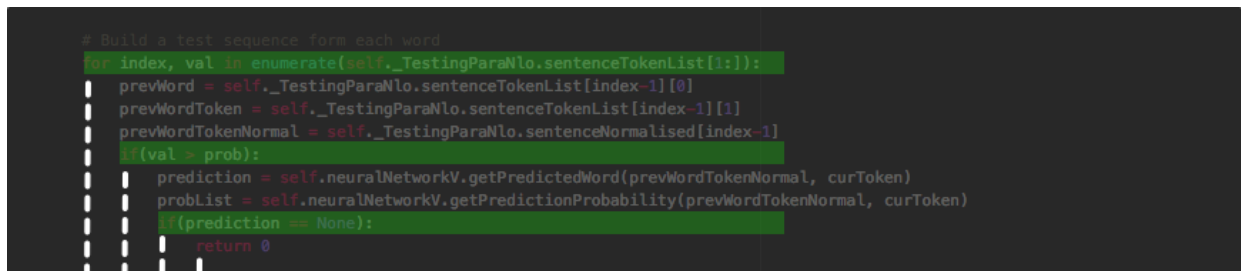
Figure 3.1.5 gives an example of how differently typical IDE's and code editors handle their file systems. IDE's have a more complex file system since they are built with firm knowledge of the language they support.

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. These are:

- Auto-indentation
- Code refactoring
- RegEx search and replace
- Multi-lined editing

Firstly, auto-indentation simply formats the 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:

A screenshot of a code editor showing a Python function. The code is as follows:

```
# Build a test sequence from each word
for index, val in enumerate(self._TestingParaNlo.sentenceTokenList[i:]):
    prevWord = self._TestingParaNlo.sentenceTokenList[index-1][0]
    prevWordToken = self._TestingParaNlo.sentenceTokenList[index-1][1]
    prevWordTokenNormal = self._TestingParaNlo.sentenceNormalised[index-1]
    if (val > prob):
        prediction = self.neuralNetworkV.getPredictedWord(prevWordTokenNormal, curToken)
        probList = self.neuralNetworkV.getPredictionProbability(prevWordTokenNormal, curToken)
        if (prediction == None):
            return 0
```

The code is syntax-highlighted. The indentation margins are marked with vertical dotted lines. The lines of code are highlighted in green, and the indentation margins are highlighted in a lighter green.

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.

Code refactoring is the process of re-modelling certain aspects of the code automatically in order to reduce the program complexity. This is done in a certain way that does not affect the external usage of the code; it still runs the same (Fowler, 1999). However, this functionality is only found in very powerful IDE's (Such as Visual Studios (Microsoft, 2013)) and is often accessed by the user right clicking on a certain function and selecting this "Refactor" button. This will cause for the selected function to be broken down into smaller functions and therefore reducing the complexity 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:

Code editor with syntax highlighting and autocomplete

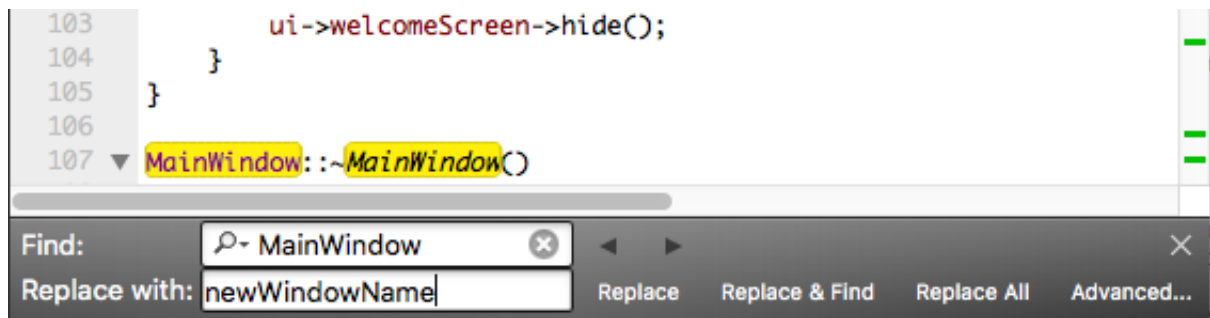


Figure 3.1.7 – Example of QT's search and replace function (QT Creator, 2015)

Multi-lined code editing is a feature that allows for users to create multiple text cursors. This gives them the ability to input text in multiple locations with a single key press, therefore making it much easier to create tedious templates such as HTML or XML.

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’” (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. Python’s dynamic typing system and minimal syntax means that you can do more stuff with less code. Therefore, the speed of production is drastically improved. Also, python dominates a huge array of programming areas from games, robotics, web, and more (Nosrati, 2011). But, 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. Python’s 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. When compared to Java and python, C++ has a much faster code execute time (Alnaser, et al., 2012). 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:

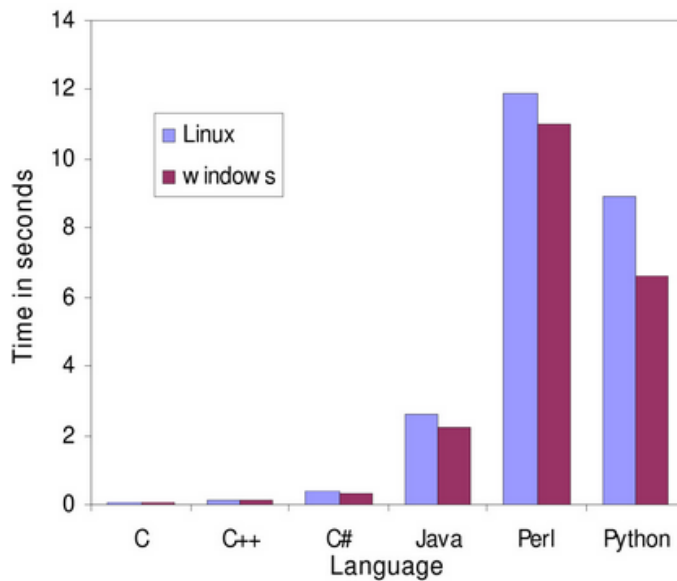


Figure 3.2.1 – Efficiency comparison of languages (Neighbour-joining algorithm) (Biomedcentral, 2015)

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:

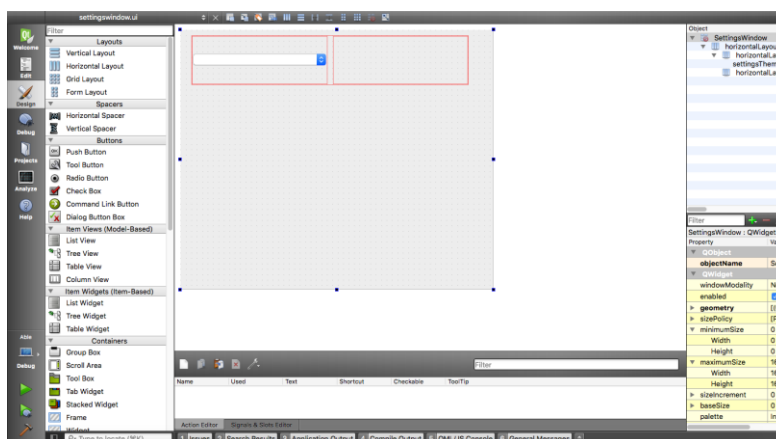


Figure 3.2.2 – QT's GUI tool (QT Creator, 2015)

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 the user would be able to create their own style-sheet and

completely change the look of the application with out re-compiling the source code of the code-editor. A sample of QCSS (QT's adapted version of CSS):

```
FileViewWidget
{
    background: #242424;
    border: none;
    color: #8C8D8C;
    padding-top: 5px;
    border-right: 1px solid #44474A;
    selection-background-color: #44474A;
    alternate-background-color: white;
}
```

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:

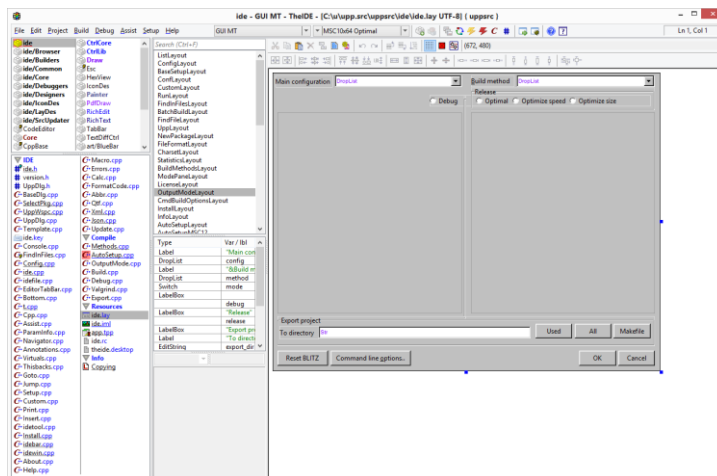


Figure 3.2.4 – Ultimate++’s IDE (Ultimatepp, 2016)

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.

Text editing features

As discussed in section 3.1.4 software functionality, there are 4 common features that exist in various code editors that improve the users ability to write code. Due to the planned timing constraints of this project (see appendix E) the author only has enough time to implement only two functionalities. So it is important that the author chooses the most effective functions. The author produced a questionnaire that was posted on the popular computer science forum "Hacker news" (Hacker news, 2016) in order to gain a better understanding of the target users wants. This questionnaire asked common programmers which feature they use the most out of the 4:

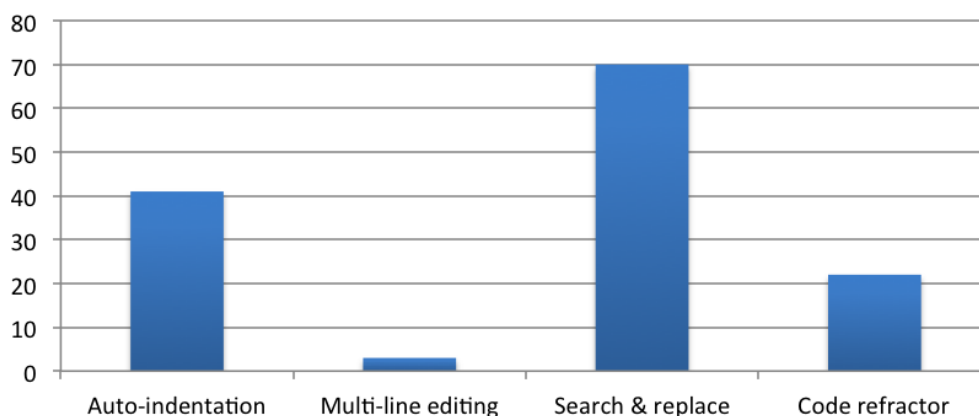


Figure 3.2.6 – Results for questionnaire

As can be seen in Figure 3.2.5, 136 applicants completed the questionnaire. The results showed that there was an extremely high demand for the search and replace functionality which received a total of 70 votes. Second came the auto-indentation feature, which received 41 votes and was closely followed by code refactoring (22 votes). Multi-lined editing came in last with a mere 3 votes.

This graph shows us that the typical programmers that would be using the Able software, if were to only pick 2 features, would choose to use the “search and replace” functionality along with the auto-indentation feature and therefore making these the most suitable for the project.

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 (Johnson, 2008). 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 (University of Virginia, 2007).

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 utilise 'look-ahead' without serious complication and do not support nesting, so larger expressions can look extremely complicated and messy.

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 as the efficiency is not as drastically impacted as expected.

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

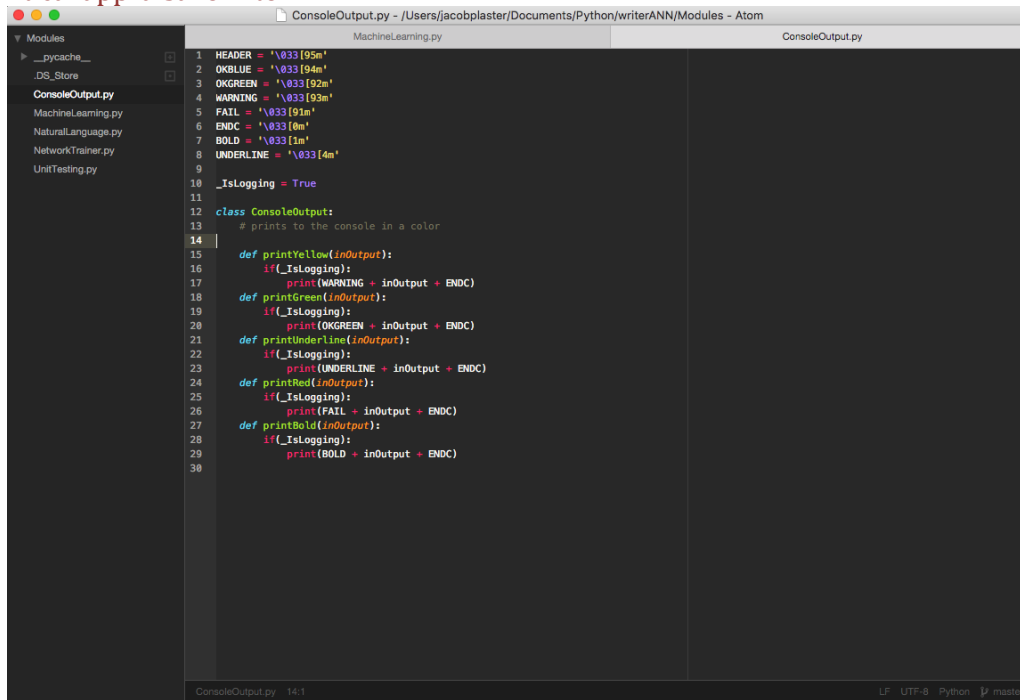


Figure 3.4.1 – Atom editing window containing python code (Atom, 2013)

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

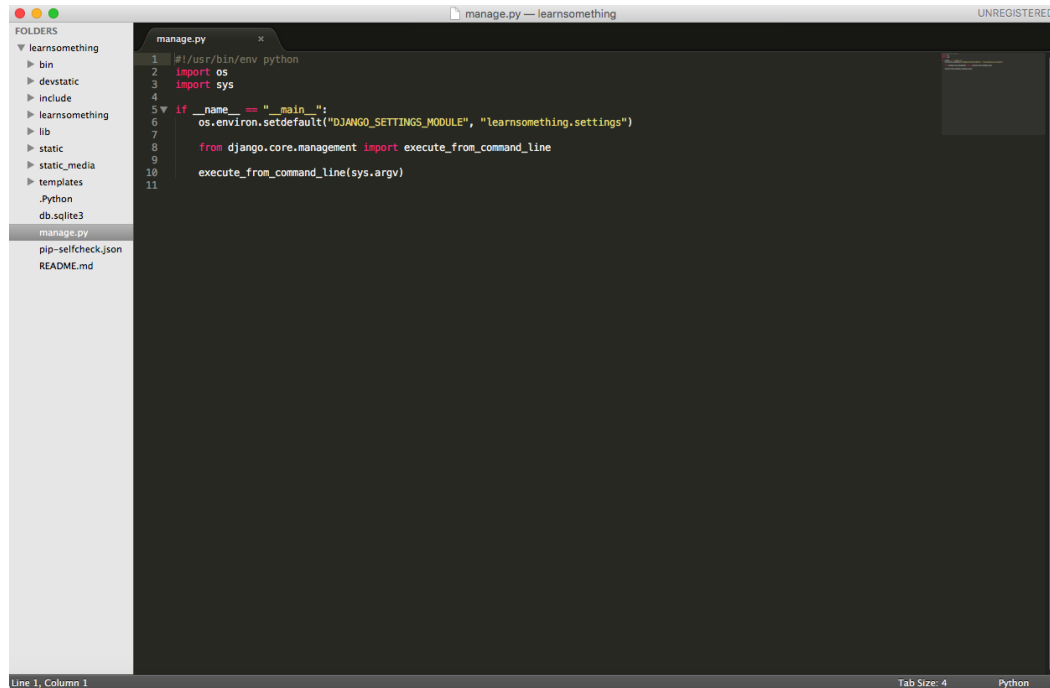


Figure 3.4.2 – Sublime editing window containing python code (Sublime, 2015)

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:

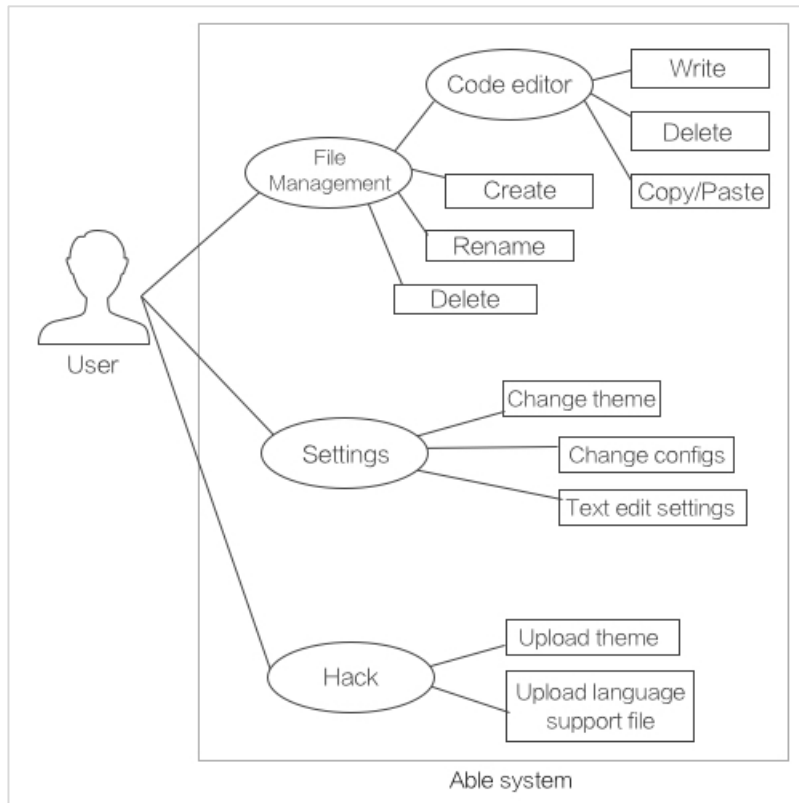


Figure 4.1 – Use case diagram

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 (.cfg) 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

- 1.2 Load
- 1.3 Save
- 1.4 Rename
- 1.5 View

2. Text editing

- 2.2 Manipulate code
 - 2.2.1 Paste/Cut/Copy
 - 2.2.2 Undo/Redo
 - 2.2.3 RegExp Find/Replace
- 2.3 Syntax-highlighting
 - 2.3.1 Change syntax support
- 2.4 Auto-completing
 - 2.4.1 Change auto-completion support
- 2.5 Indentation/auto-indentation

3. User Interface

- 3.2 Change theme
- 3.3 Control file management
- 3.4 Control text editor
- 3.5 Clean/Minimalistic

4. Customization

- 4.2 Plug-ins
 - 4.2.1 *Theme*
 - 4.2.2 *Language support*
- 4.3 Load plugin
- 4.4 Change editor preferences
 - 4.4.1 *Font*
 - 4.4.2 *Indentation*
 - 4.4.3 *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.

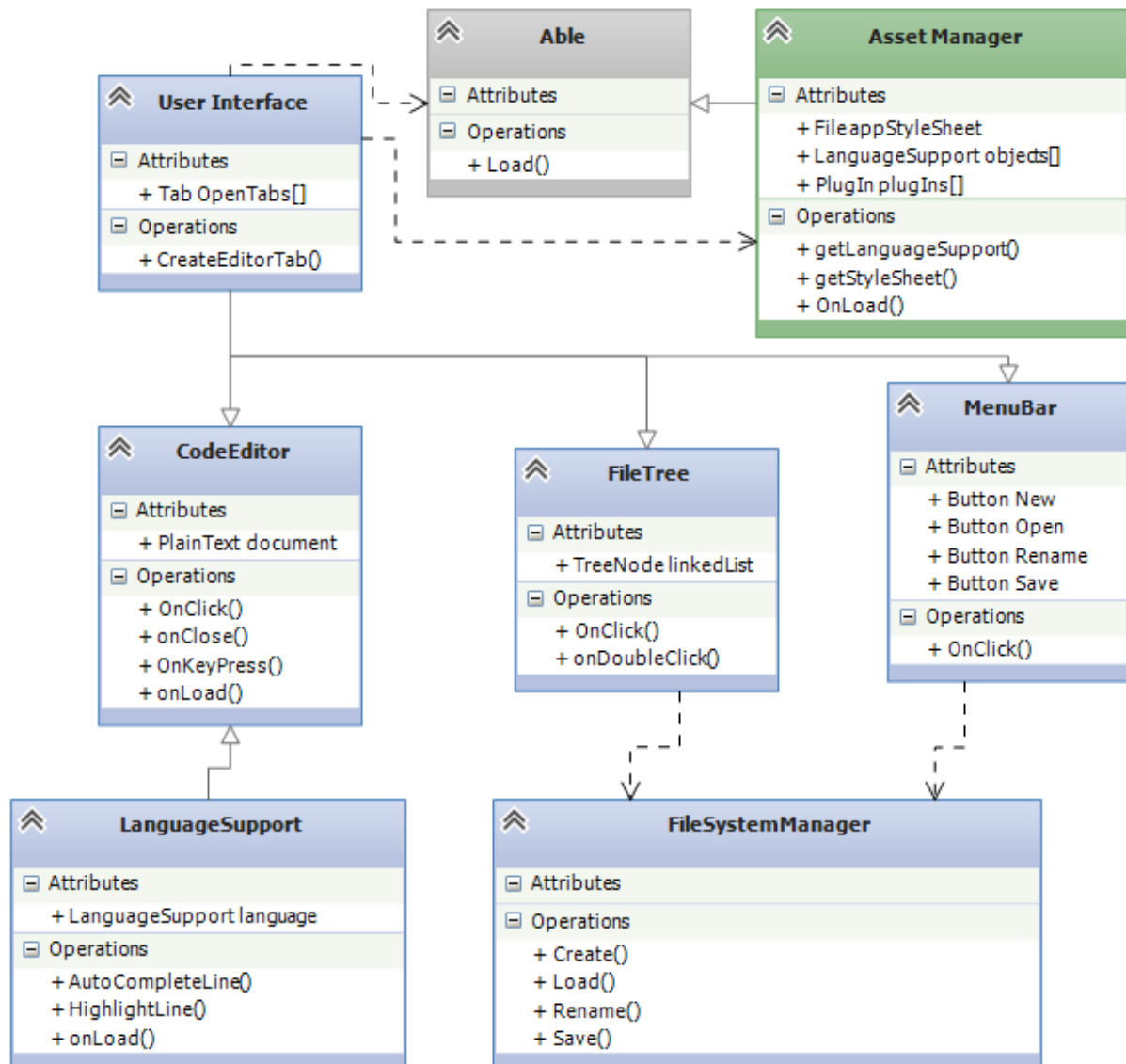


Figure 4.2.1 – A simplified class diagram of Able's system architecture

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 but will be discussed in section 4.4. 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 locate auto-complete suggestions and to highlight certain tokens 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" (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" (Sun, 2012). The Figure below illustrates how Able's core functionality will be separated in to a series of widgets and modules:

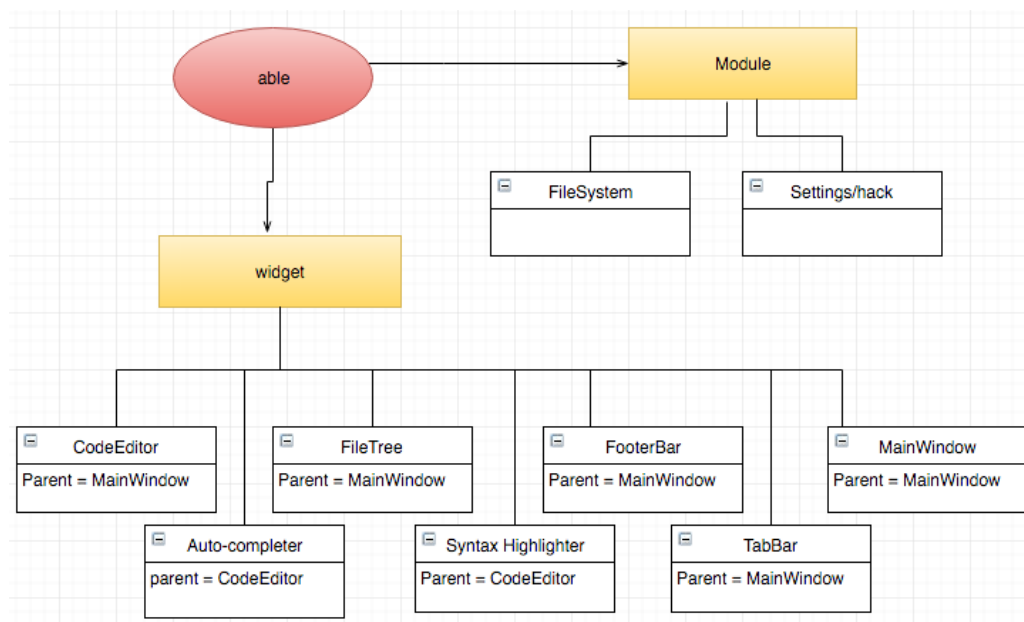


Figure 4.2.2 – Simple diagram of Able's widgets and modules

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 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:

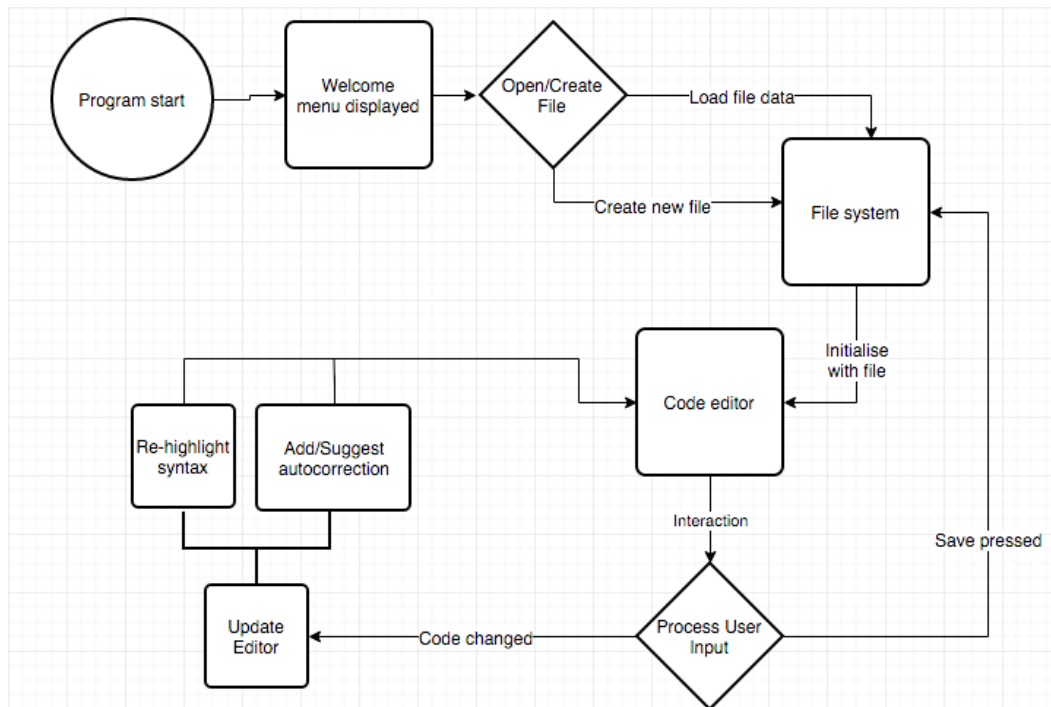


Figure 4.3 – Simple flow chart of Able

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. 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 triggered 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.

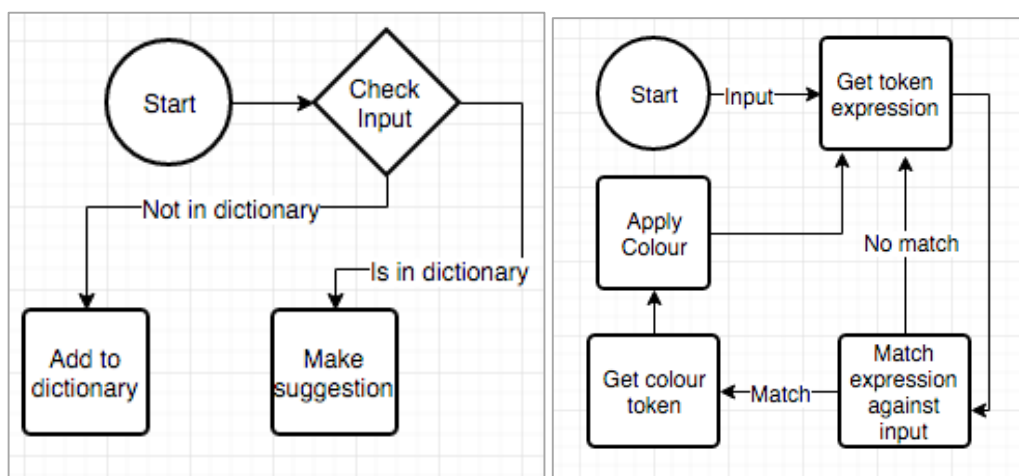


Figure 4.3.1 – Auto completer (left) and syntax highlighter (right) flow charts

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 and consistently adhere to three principles: minimum, intuitiveness and consistency” (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:

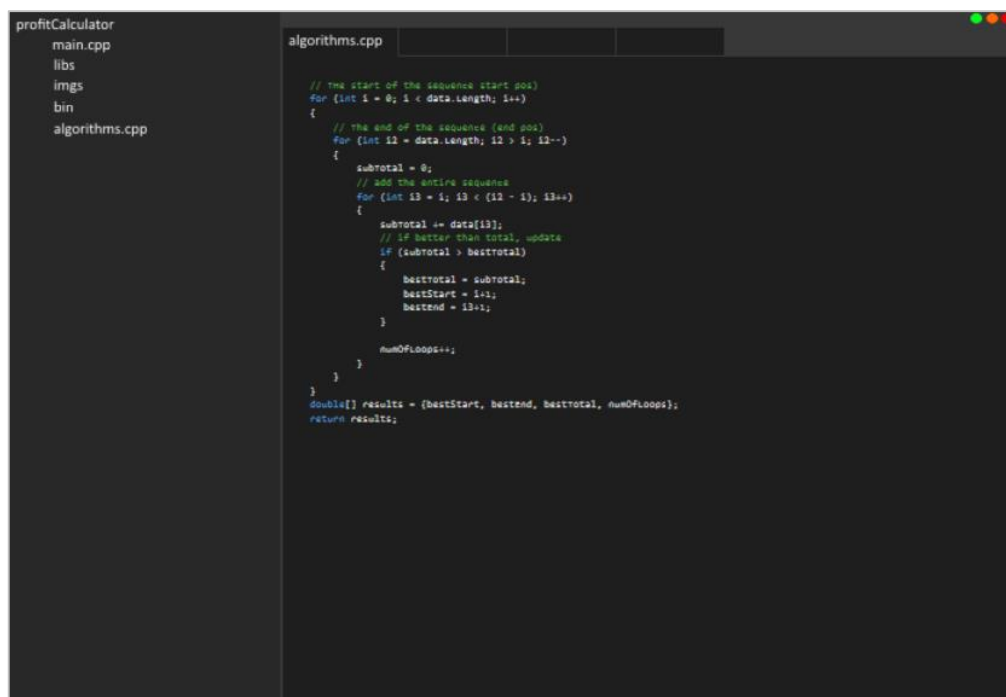


Figure 4.3.1 – Initial design concept of Able (10th October 2015)

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 a 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:

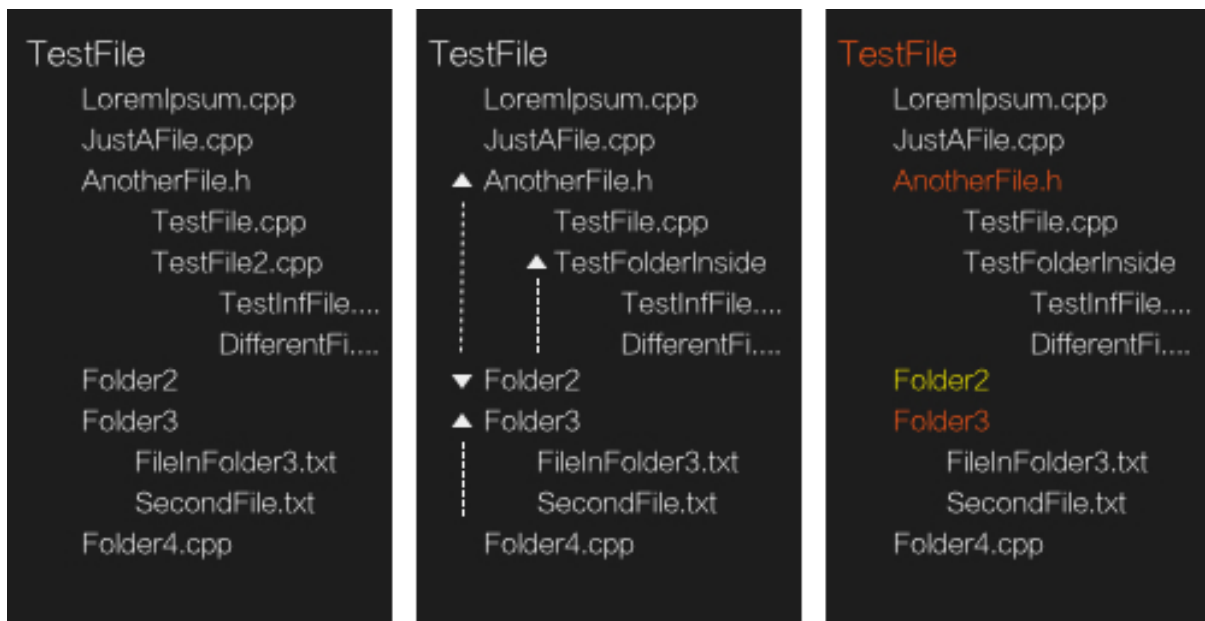


Figure 4.3.2 – Initial design concept of file tree widget

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.

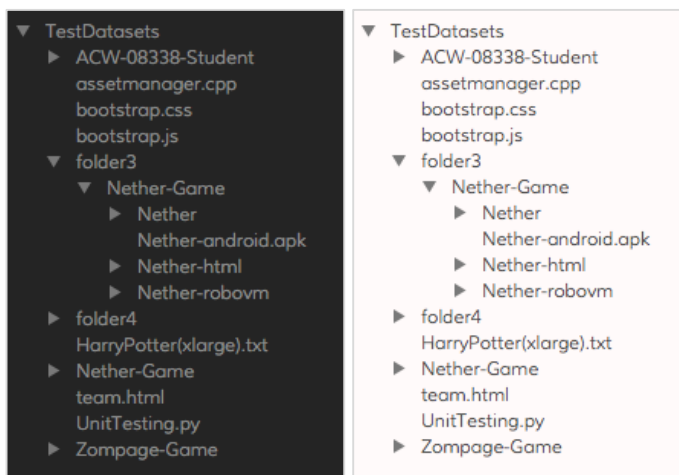


Figure 4.3.3 – second design concepts for the file tree

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.

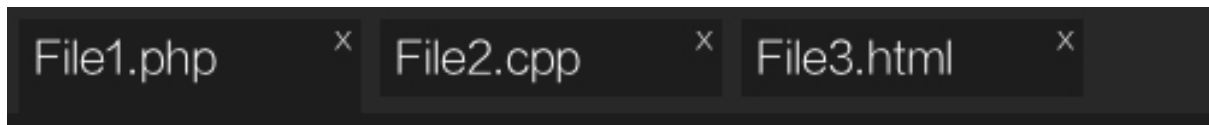


Figure 4.3.3 – Initial design concept for the tab widget

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:

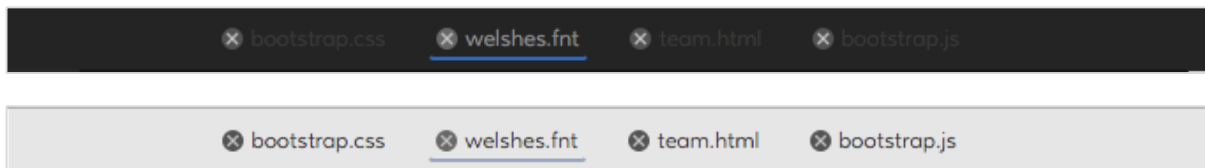


Figure 4.3.4 – Second design concepts for the tab widget

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.

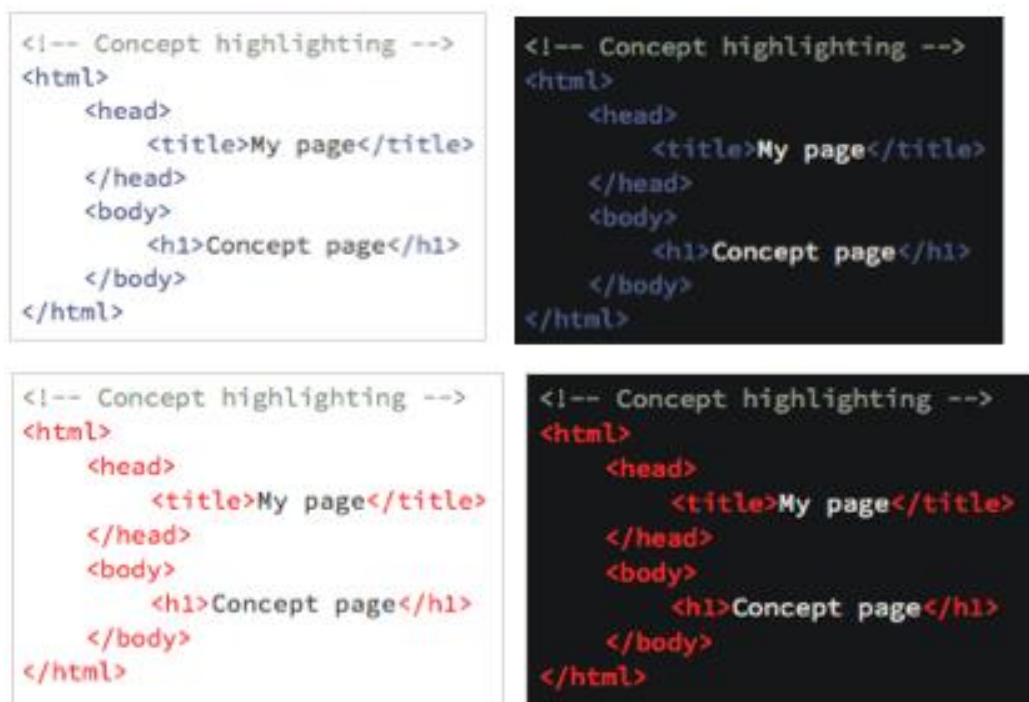


Figure 4.3.5 – Design concepts for code editor widget

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 (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.

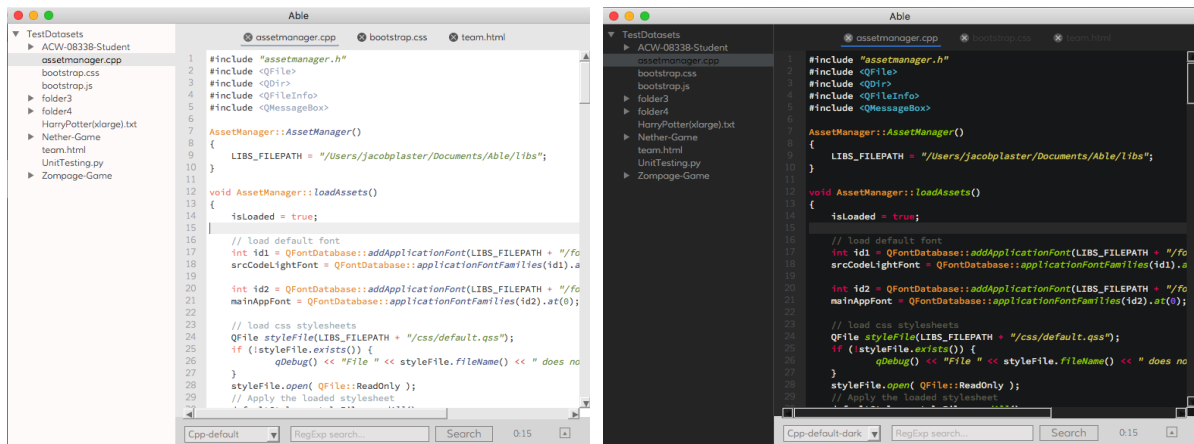


Figure 4.3.2 – Able’s main window (light and dark theme)

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 palettes 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.

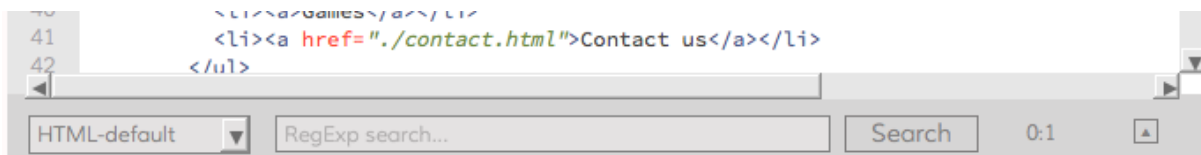


Figure 4.3.3 – Able’s footer bar (collapsed)

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 its ‘expanded state’:

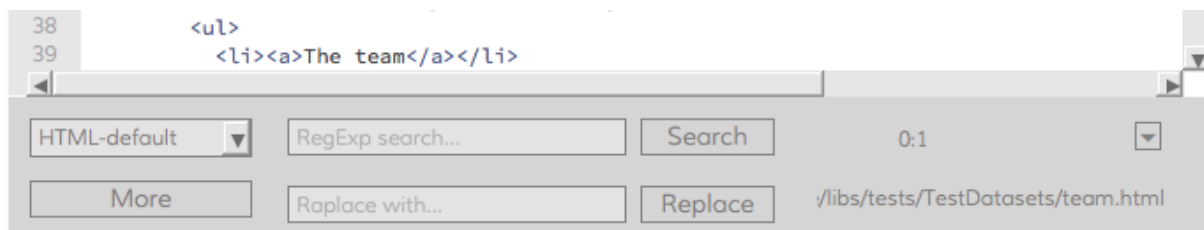


Figure 4.3.4 – Able’s footer bar (expanded)

Code editor with syntax highlighting and autocomplete

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

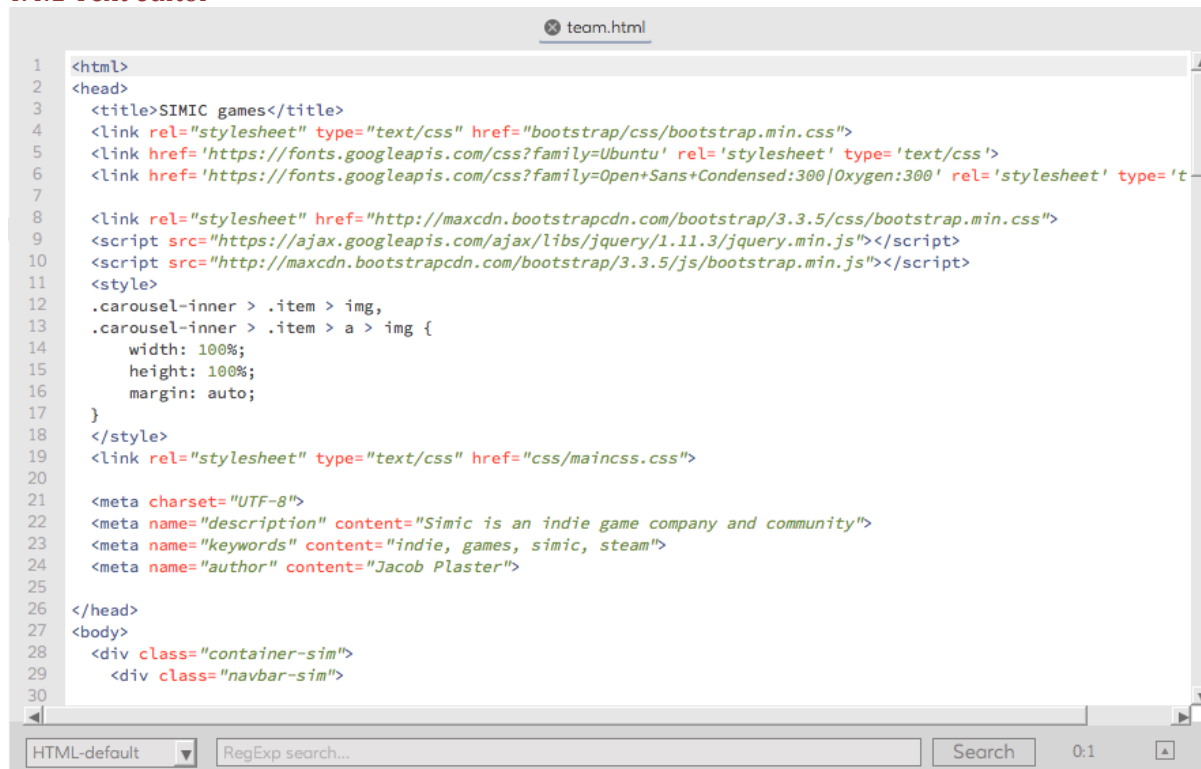


Figure 4.4.1.1 – Able code editor, highlighting a html file

Figure 4.4.1 shows the fully functioning text editor widget 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:

```
class CodeEditor : public QPlainTextEdit
{
    Q_OBJECT
public:
    CodeEditor(QWidget *parent = 0);
    ~CodeEditor();
    void myCodeEditorFuntion();
protected:
    void keyPressEvent(QKeyEvent *e) Q_DECL_OVERRIDE;
};
```

Figure 4.4.1.2 – Example of custom widget creation

Figure 4.4.2 shows an example of how a developer might create a custom 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 be seen inside of the “protected” claim in figure 4.4.2. A hook for a key-press event that is inherited from 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.

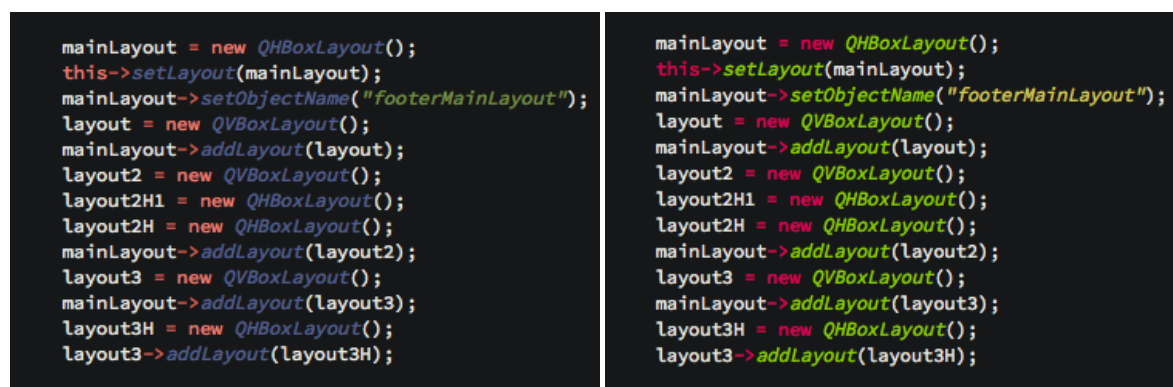
4.4.1.1 Syntax highlighting

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 block of 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.

```
foreach (const SyntaxHighlightingRuleSet::HighlightingRule rule, ruleSet->highlightingRules) {
    QRegExp expression(rule.pattern);
    int index = expression.indexIn(text);
    while (index >= 0) {
        int length = expression.matchedLength();
        setFormat(index, length, rule.format);
        index = expression.indexIn(text, index + length);
    }
}
```

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 “HighlightingRules” 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.



```
mainLayout = new QHBoxLayout();
this->setLayout(mainLayout);
mainLayout->setObjectName("footerMainLayout");
layout = new QVBoxLayout();
mainLayout->addLayout(layout);
layout2 = new QVBoxLayout();
layout2H1 = new QHBoxLayout();
layout2H = new QHBoxLayout();
mainLayout->addLayout(layout2);
layout3 = new QVBoxLayout();
mainLayout->addLayout(layout3);
layout3H = new QHBoxLayout();
layout3->addLayout(layout3H);
```

```
mainLayout = new QHBoxLayout();
this->setLayout(mainLayout);
mainLayout->setObjectName("footerMainLayout");
layout = new QVBoxLayout();
mainLayout->addLayout(layout);
layout2 = new QVBoxLayout();
layout2H1 = new QHBoxLayout();
layout2H = new QHBoxLayout();
mainLayout->addLayout(layout2);
layout3 = new QVBoxLayout();
mainLayout->addLayout(layout3);
layout3H = new QHBoxLayout();
layout3->addLayout(layout3H);
```

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 see the entire language support rule set for C++ please view appendix A). This functionality is displayed in the form of a flow diagram:

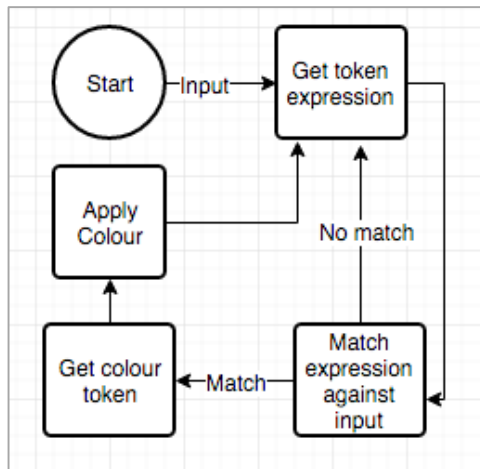


Figure 4.4.1.5 – Two different rule-sets being applied to a C++ file (Able)

Figure 4.4.1.5 shows how the syntax highlighter module grabs the tokens, which are supplied by the language support file and attempts to match them against a block of code.

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 `keyPressEvent`. 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 that has been changed (Unless a multi-lined comment is matched, then all blocks within the comment are re-highlighted). Figure 4.4.1.6 shows how the algorithm interprets a code file in terms of ‘blocks’.

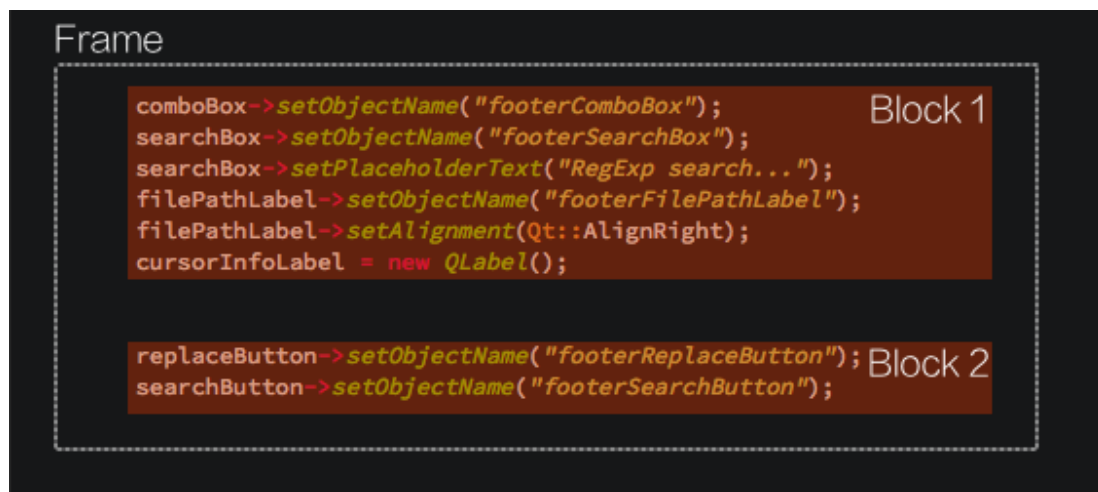


Figure 4.4.1.6 –Able’s text block feature example

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. When created, 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 has been visualised in the form of a flow chart (previously displayed in section 4.2):

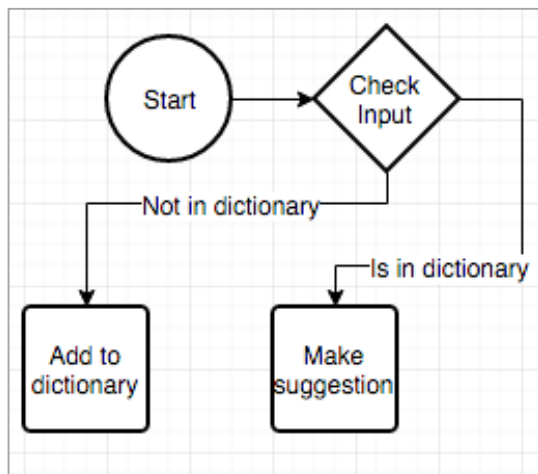


Figure 4.4.1.6–Able’s auto-completer function (Python)

When implemented into the software, this functionality looks like this:

```

helloWorldVar = "Hello world!"
nameVar = "My name is Jacob."
descriptionVar = "Im testing out ables auto-completer"

print(helloWorldVar + " " + nameVar + " " + de
  
```

del
def
descriptionVar

Figure 4.4.1.7 –Able’s auto-completer function (Python)

Figure 4.4.1.7 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.

```
fmt.setBackground(ruleSet->searchHighlightColor);
fmt.setForeground(ruleSet->searchHighlightColorForeground);
if(expr.pattern() != "")
{
    int index = expr.indexIn(text);
    while(index >= 0)
    {
        int length = expr.matchedLength();
        this->setFormat(index, length, fmt);
        index = expr.indexIn(text, index+length);
    }
}
```

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.

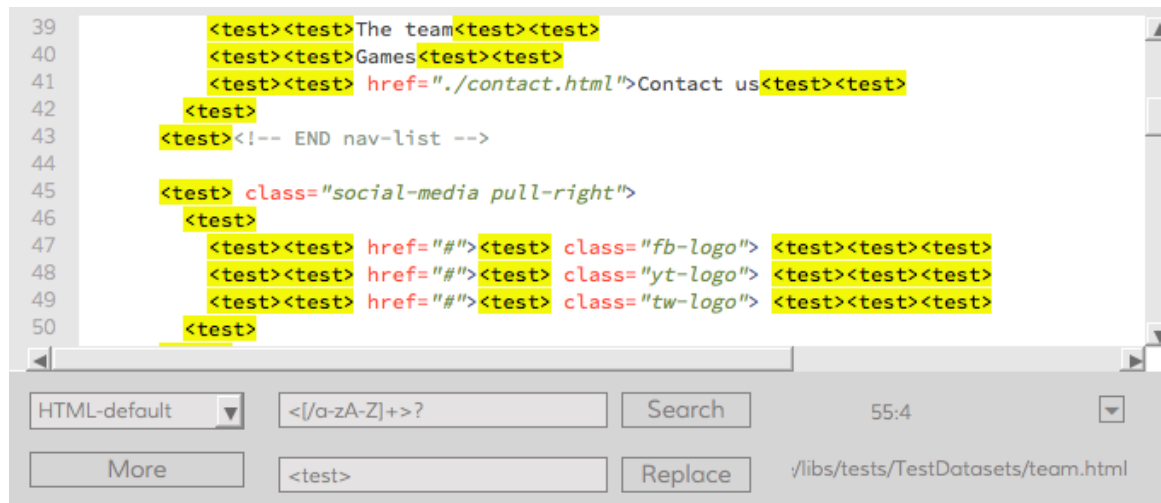


Figure 4.4.1.8 –Able’s replace function (Html)

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.

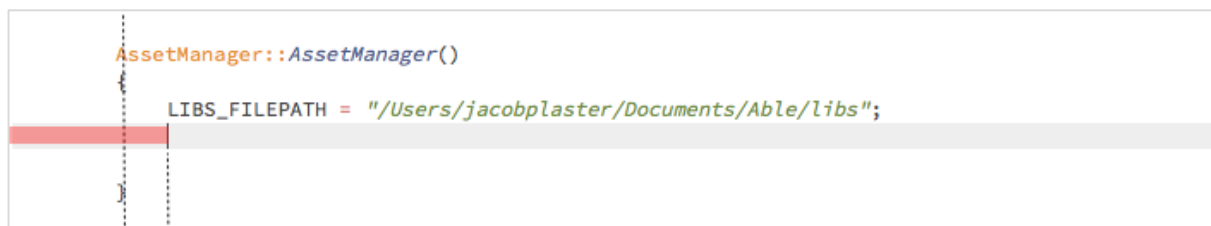


Figure 4.4.1.9 –Able’s auto-indentation function (C++)

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:

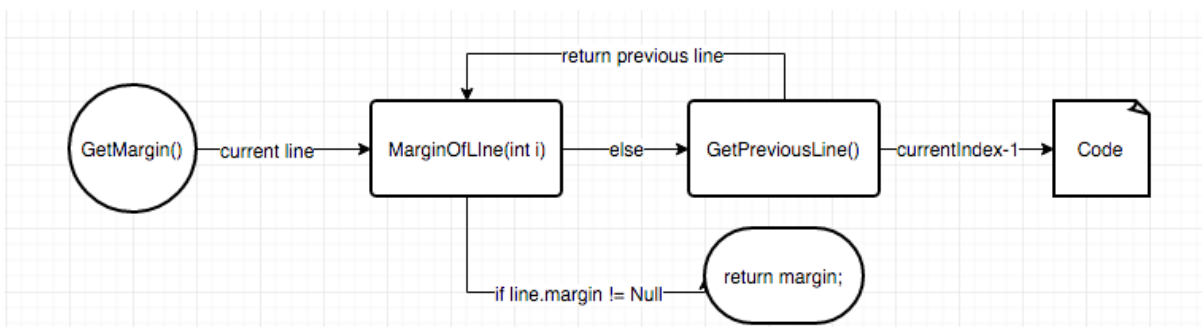


Figure 4.4.1.10 –Able’s auto-indentation getMargin() flowchart

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 tacking 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.

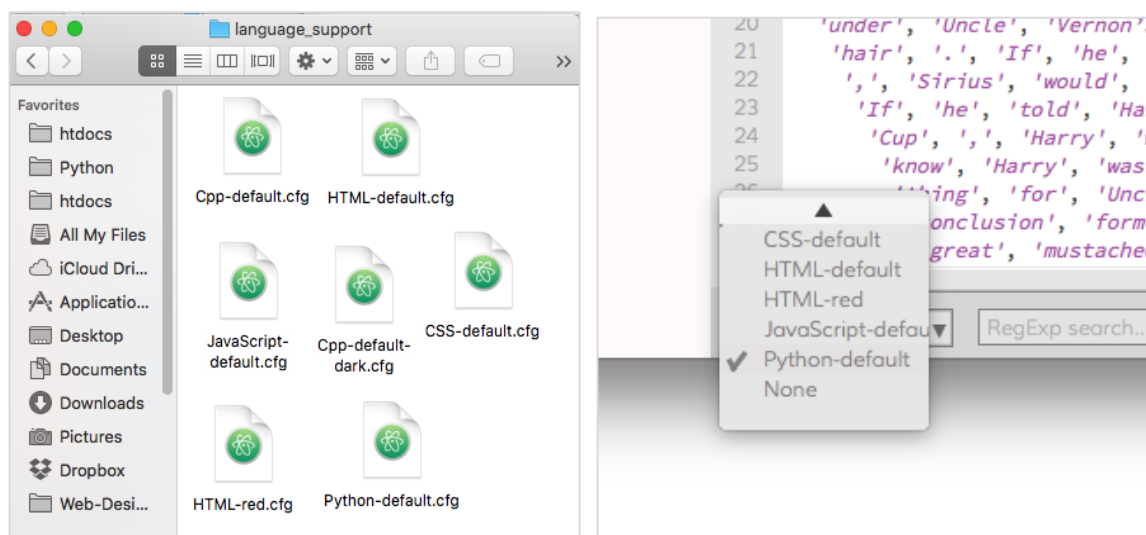


Figure 4.4.2.1 –Able’s plug-ins being loaded

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

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):

<pre>#PYTHON #Simple program that counts bottles of beer def simpleBottleProgram(name, age, location): string = " Bottles of beer on the wall" for val in range(100, 0): print(val + string) print("No more bottles on the wall")</pre>	<pre>//C++ //Simple program that counts bottles of beer public void bottles() { for(int i = 100; i > 0; i--) { cout << String(i) cout << " Bottles of beer on the wall"; } cout << "No more bottles of beer on the wall"; }</pre>
<pre><!-- Multi-line comment here --> <meta charset="UTF-8"> <meta name="description" content="S"> <meta name="keywords" content="j"> <meta name="author" content=""> <div> <h1>Title</h1> </div></pre>	<pre>.input-block-level { display: block; width: 100%; min-height: 30px; -webkit-box-sizing: border-box; -moz-box-sizing: border-box; box-sizing: border-box; }</pre>

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:

```
QPushButton#footerMoreButton
{
    color: #828080;
    padding: 2px 7px;
    border: 1px solid #828080;
}
QPushButton#footerReplaceButton
{
    color: #828080;
    border: 1px solid #828080;
    padding: 2px 7px;
    min-height: 15px;
    min-width: 60px;
}
```

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:

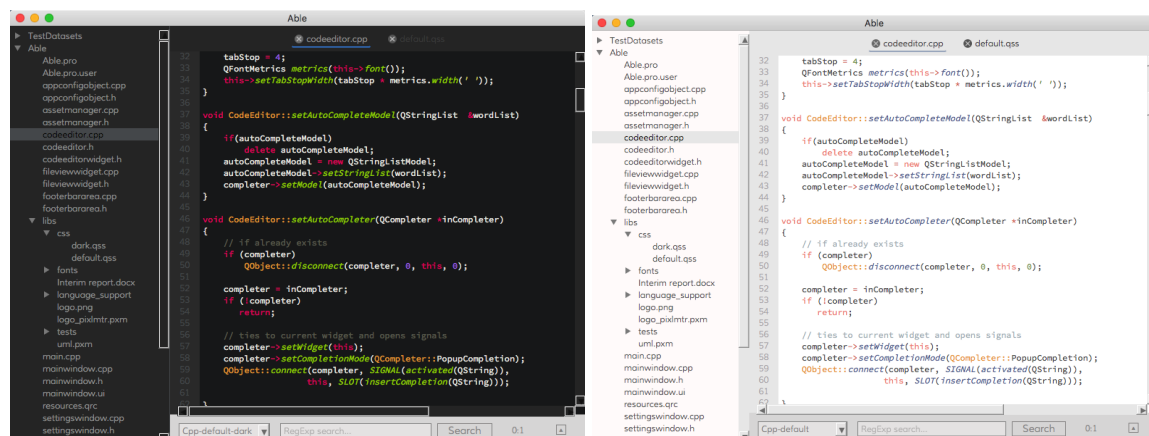


Figure 4.4.2.1 –Able’s plug-ins being loaded

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:

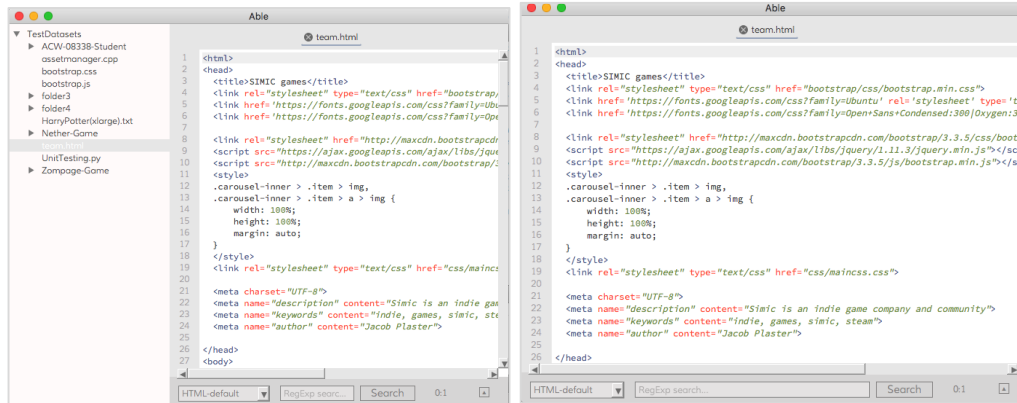


Figure 4.4.3.1 –Able's file tree collapsed (left) and expanded (right)

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:

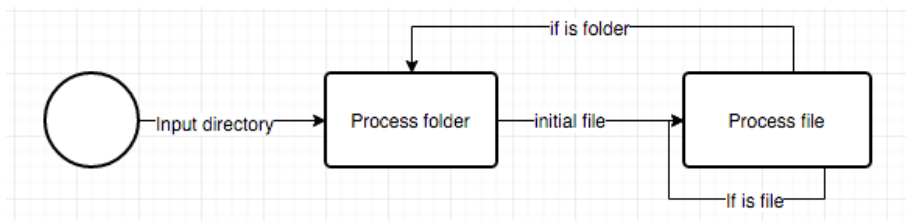


Figure 4.4.3.2 –Able's file tree getFiles() algorithm in flow chart form

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:

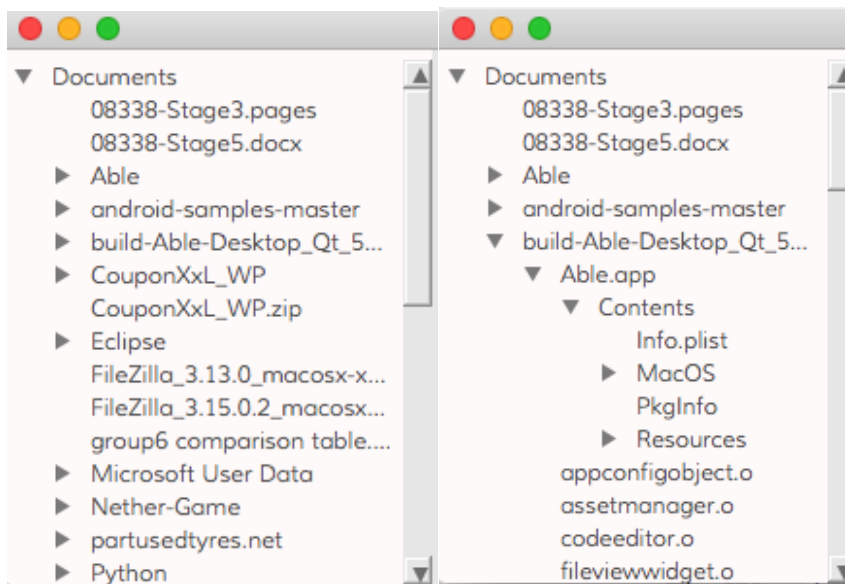


Figure 4.4.3.3 –Able’s file tree populated with documents folder

4.4.4 Workspace automatic saving

Whenever a user closes the Able client, an automatic function is initiated which saves the workspace of the user. The workspace consists of any currently loaded tab widgets and any folders loaded into the file-tree widget. The data is stored in a user configuration file within the Able install directory as a series of commands. When Able is re-loaded the asset-manager locates the user configuration file and runs each command, which causes for the workspace to be loaded. This means that whatever tabs/files were open when the user closed the application are remembered so they do not have to waste time setting up their workspace again. The examples below show an example “OpenCodeTab” command and a “LoadFileTree” command, which may be found in a typical user configuration file:

```
CT:
/Users/jacobplaster/Documents/Able/main.cpp
PD:
/Users/jacobplaster/Desktop
PD:
/Users/jacobplaster/Documents/Able
```

Figure 4.4.4.1 –Able’s file tree populated with documents folder

Figure 4.4.4.1 shows an example of how a user configuration file may look. The “CT:” string defines that the next file-path string loaded will trigger the Able software to load it into a code tab and initiate an editing interface for that file. The “PD:” string defines that the next loaded file-path will be passed into the file-tree directory-loading algorithm. Below is an example of how the Able software would create and interpret these commands in the form of a flow chart:

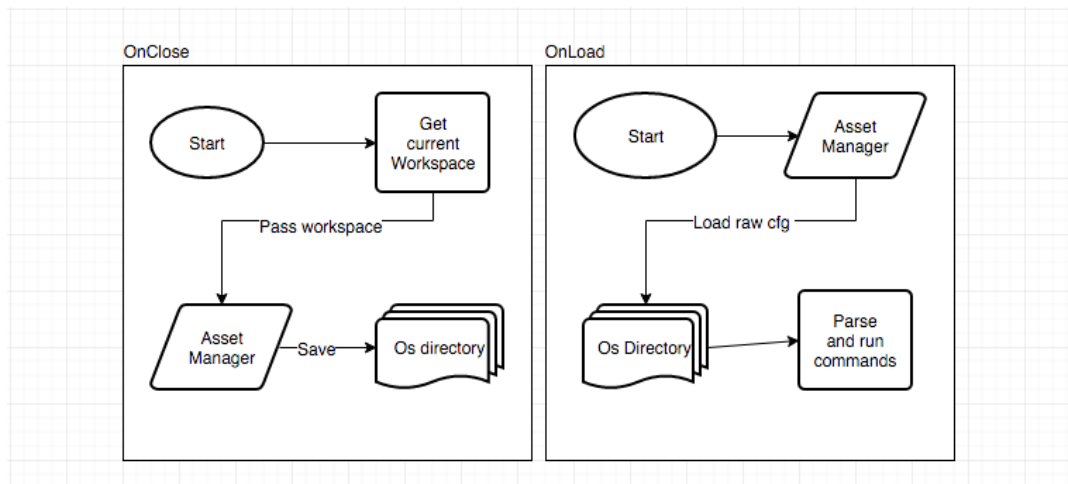


Figure 4.4.4.2 –Able’s automatic workspace loader (flow-diagram)

Figure 4.4.4.2 illustrates how the automatic workspace loading functions is split into two calls. When the program is loaded and when the program is closed. OnClose saves and updates the user configuration file whereas the OnLoad loads the user configuration file and processes its contents.

4.5 Testing

4.5.1 Formal and platform testing

Parasoft (Parasoft, 2016) 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 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.

4.5.2 Performance testing

Syntax highlighting performance

All unit tests were handled on the MacBook air machine with the El Capitan operating system (see Table 4.6), It is important that unit tests are carried out to ensure that the software does not deviate off of the planned design route during the development period. Able aims to cope under high amounts of stress 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)
26/12/15 (Full syntax highlighting)	C++	Full	7208	143
	Css	Full	133614	854
	Js	Full	61884	1018
	Text	None	56450	154
	Html	Full	7372	96
	Python	Full	10732	87
05/11/15 (Basic syntax highlighting)	C++	Med	7208	122
	Css	None	133614	733
	Js	None	61884	245
	Text	None	56450	169
	Html	None	7372	32
	Python	None	10732	65
29/10/15 (Basic syntax highlighting)	C++	Med	7208	137
	Text	None	56450	167
	Html	None	7372	36
	Python	None	10732	38

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

In table 4.4.1, 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 table 4.4.1 you can easily see that once the syntax highlighter was fully implemented it had a large impact on efficiency, 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 to pinpoint problem areas and optimise them.

File tree load directory algorithm

Another important process that is required by Able is the ability to load entire project directories into the project manager tree view.

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

Table 4.4.2 shows how 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. The algorithm became slightly less efficient during the tests of 26/12/15 since it was edited to retrieve more information of the containing files. This was due to the development of the code editor footer bar, which required the knowledge of the absolute file path of the located file to be displayed.

4.5.2 Functional testing

Certain testing tasks were undertaken throughout the development stages of the project to ensure that the correct functionality was maintained during development. Whenever a drastic change was made to the software, these tasks were re-ran to test if the system still worked as specified.

Procedure	Expected output	Result
Load Able application	Theme to be automatically loaded and applied to the software. Welcome screen to be displayed with logo and buttons	Expected
Load File directory	All items including: files, folders, sub-files and sub-folders. To be available for interaction.	Expected
Load Code file	Tab added to tab-bar widget. Code displayed in editing window. Syntax-highlighting applied (If language support available).	Expected
Save code file	File located in system directory and updated with the exact results taken from the editing window.	Expected
Change theme	Load theme from AssetManager class and instantly apply to all widgets and interfaces.	Expected
Change editor language	Instantly update all contents contained in editing window with the newly loaded syntax highlighting rules.	Expected

Table 4.4.3 – Procedures and expected outcomes (see appendix J for more)

Table 4.4.2 displays the core procedures detailed in the software's specifications and their expected output. The results column shows the result produced by the final Able software once these procedures had taken place. All of the procedures produced the expected result showing that Able had successfully achieved its core functionality.

5. Critical analysis

5.1 Project Achievements

Section 2.1 of this report outlined the planned aims and objectives of this project and detailed both primary and secondary objectives. In this section I will compare the final results against those aims and objectives.

Aims/Objective	Type	Comment	Result
Create clean minimalist code editor	Primary	The GUI should be responsive, simple and comfortable for users to use for long periods of time.	Success
Add ability to handle files	Primary	The software should give the user the ability to manipulate file structures and allow them to: <ul style="list-style-type: none"> • Create files • Rename files • Remove files 	Success
Add additional core features such as syntax-highlighting and autocorrect	Primary	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.	Success
Ensure software is efficient and reliable	Primary	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.	Success
User generated customization	Secondary	Develop a system, which allows 3 rd party plug-ins to be created and implemented into the software. These plug-ins should have the capability to change both the functionality and aesthetics of the software.	Success
Machine learning auto-complete	Secondary	Develop and provide the auto-complete algorithm with the ability to utilize machine learning in order to predict and complete the user's word.	Failure

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

Table 5.1 – Speed tests of Able’s load folder function

As displayed in table 5.1 most of the aims and objectives declared at the beginning of the project have been completed with success. However, “Machine learning auto-complete” was not. When initially planning the project, the machine learning auto-complete seemed like a very useful feature, which would add a lot of value to the users experience. However, during the development stages, the author realised that the machine learning system would not make a large difference to the prediction of autocomplete suggestions. This is because the amount of time and effort required to implement a system such as this was far to great when compared to the impact it have on the usability of the software, so it was decided that this element would be left out.

Although the effect of leaving out the machine learning technology would have nearly an unnoticeable effect, it still is a shortfall to the software since the original specifications of the code editor advertised this element.

5.2 Further development

The Able code editor managed to meet all of its aims and objection. However, there are many other features that could be implemented into the software.

Firstly, although the plug-in system allows users to manipulate both, the aesthetics and language support of the software. This still remains very limited. Able would massively benefit from a system, which allows users to implement their own code directly into the framework. This would give the users the ability to create their own functional elements like text editing add-ons or efficiency optimisations.

In relation to plug-ins, it would be useful to provide the community of Able plug-in developers with an easy way of publishing their own add-ons. An online market place styled system would greatly improve the usefulness of Able since it would both allow developers to advertise their products and would help able users install official plug-ins. This marketplace could be directly linked to the Able software and made available from within the code editing software.

As well as extra systems being implemented, Able’s core objectives could also be improved. Since Able at heart, is a code editor, it would benefit from having more text editing functionality implemented inside of its editing interface, such as:

- Multi-line cursors. This is a tool that implemented in most popular code editors and it allows the user to create multiple text cursors in different placing meaning the user can write input characters onto multiple lines with a single key press.
- Image viewing capability. Programming projects often include the need to use images and Able would be a lot more functional if it could provide the ability for users to view any image files from within the software.
- GUI content building tools. These tools are often implemented to help users build layouts and utilise a drag and drop interface. Tools such as this would make able a more powerful editor by providing the ability to quickly create code.

Finally, the common practices of programming software require the heavy use of source control software. These tools provide an efficient way of synchronising and storing developing source code. As these tools are so popular amongst developers, it would be a wise choice to develop a function, which allows user to link source control directly to their project. Then, whenever the user saves their code, their source control system is automatically notified.

5.3 Personal reflection

The production of this project has been a rewarding, enlightening yet also a demanding Experience. In this section the author will discuss how they managed to produce this project, including:

- Past technical knowledge
- Past similar projects
- Management abilities
- Encountered problems
- Production of the report
- Advice for future students

5.3.1 Past Technical Knowledge

Although much technical knowledge was researched in order to produce the software, some of it was already known. The author had worked with many multi-platform frameworks prior to the development of Able such as Google's 'PlayN' (PlayN, 2016) and 'LibGDX' (LibGDX, 2016). This prior knowledge of how cross-platform frameworks commonly work helped the author to quickly comprehend how the QT (QT, 2015) framework could be implemented into the project.

Due to the size of this project, it was required that the author adopted a method of source control in order to aid development. Although methods of subversion were recommended it was decided that GitHub (GitHub, 2016) would be used for this project as it provided better log support. The author has used tools such as GitHub on multiple occasions when developing past projects so there was no research needed for the adoption of source control tools.

5.3.2 Past Similar Projects

Prior to the development of this project, the author had worked on projects similar to this, the experienced gained through this was crucial. 'DinoSprint' was a mobile game that incorporated complex algorithms to randomly generate 2D terrain. It also incorporated the LibGDX cross-platform framework (Discussed above in section 6.3.1) in order to provide a clean responsive design that could be exported to any device. It also adopted the same UI principles of Able since it attempted to create a clean minimalistic art style.

The author has also worked on multiple websites such as 'jacobplaster.net' and 'Cloudplayer.io' which aided in the use various technologies, such as UI frameworks, regular expression and the use of CSS. CSS was specifically important since this knowledge was applied to Able's CSS plug-in theme system. The author was able to quickly understand and implement CSS capabilities. All of the projects mentioned in this section were of substantial size and required the use of source control. This meant that the author was provided with experience of using technologies such as GitHub.

5.3.3 Management Abilities

In order to keep on track with a project of this size, the author was required to plan out the entire development process. For this, Gantt charts and task list tables were created which can be seen in Appendices B, C, D and E. These were a very useful tools since it helped the author too meet specific deadlines and to keep within the timeframe of the project. To aid this, the author ensured that they

constantly re-assessed the performance of the project in order to make sure that the project plan effectively matched their current progress. Below is a chart taken from GitHub's contributions graph:

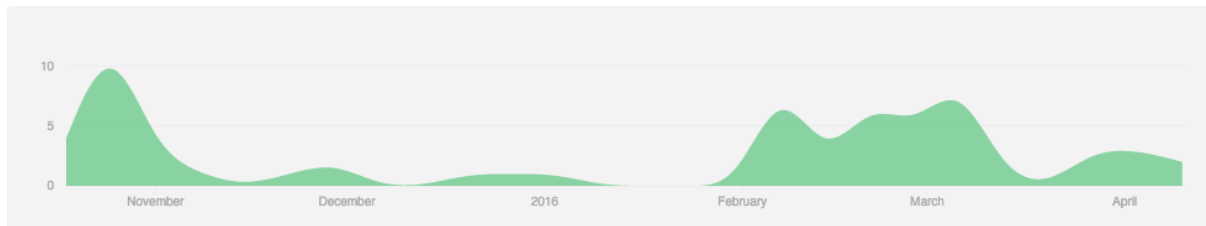


Figure 5.1 – Number of contributions per month by the author (GitHub, 2016)

Figure 5.1 shows how many contributions they made from November to April. As you can see, this graph correlates with the time plan in appendix E. As well as charts, the author conducted weekly meetings with their assigned project supervisor. The allocated project supervised helped greatly with this project, by providing advice for both the production of the software and the generation of this report. See appendix I for more source code commit graphs.

5.3.4 Encountered Problems

Although this project was considered a success, which was discussed in section 5.1 “project achievements”, there were many problems that were encountered by the author.

Firstly, the base requirements of this software (To develop a code editor with syntax highlighting and autocorrect) meant that the author had to create a very large piece of software. Furthermore, extra components such as multi-language support, cross-platform capability and more meant that the size of the project grew to an even greater amount. Towards the end of development it was found that the author had been slightly too ambitious.

In relation to the size of the project, the author found it particularly hard to manage a project of this size. This was because there were so many different modules of the software that had to reach a certain standard. The author also found difficulty finding motivation to work on certain aspects of this project since it required so much work.

5.3.5 Production of the report

The author had no prior experience of writing academic reports such as this. This caused for great difficulty when attempting to construct the structure of the document. As mentioned before, help from the allocated supervisor and the interim report demo helped greatly since it provided a benchmark to improve upon.

5.3.6 Advice for future students

During the development of this report, the author took note of any experiences that they wished they had been warned about before the production of this report. Firstly, ensure that the developer has a 100% fully planned out specification of the software, accompanied with both grant chars and task plans. Secondly, make sure that the time allocated to the development process is realistic and also accommodates outside events such as other module ACW's.

When designing the software, make sure that the developer is not too ambitious with extra implementation since the base requirements of the software is already a lot of work

6. Ethical issues and risk analysis

6.1 Risk analysis

Please see appendix B for the full risk analysis. This table displays a collection of risks that may have an effect of the result of the project. It then gives an example of how to avoid such as risk and how to recover. The risks are ranked by there 'Severity', 'Likelihood' and 'Significance'.

6.2 Ethical issues

Typical software such as Able may cause some ethical issues, in this section will address all of the possible issues.

Use by others

Due to the nature of common code editors, users may use software such as Able in order to work on extremely sensitive data that may carry a large amount of significance. For example, a typical user could be using Able in order to program some documents for their company's server, if Able was to fail and cause problems it could lead to serious consequences. This is a major ethical issue for Able, in order to combat this; the software is made to be as reliable as possible.

Malicious users could harness Able in an effort to program their own malicious code such as viruses. Able cannot directly combat a problem such as this. However, Able does not provide any functionality to aid endeavours such as this.

Use of external libraries

Able harnesses technologies from lots of different libraries. Although, all of the licenses of these libraries allows for the implementation within able, a special effort has been made to ensure that the source code is directly referenced to the original author.

Participants of user testing

Special considerations need to take place in order to accommodate human testers. Firstly, the user should not feel as if it is them being tested. They should be regularly reminded that it is the software that is being tested. The user should not be humiliated in any way, especially when under observation. It should also be considered that the test users anxiety levels may rise whilst being observed.

Software Licensing

The licensing of the software is fairly un-restricted and has been issued an MIT license, meaning that any one can download and modify the contained code as long as they reference the author of the software. This is because the author wants to use the source code of the software to be used as educational research for any future developers which aspire to create similar products.

7. Appendices

7.1 Appendix A – Language support file

%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:
```

Code editor with syntax highlighting and autocomplete

```
#708D44
(\\"\\.[^"])*\\)(\\'\\.[^']*')*\'
%autocomplete_Format
[a-zA-Z]+[ ]*[=;]
%autocompleteTrim_Format
[^=; ]+
%CurrentLineHighlight_Format
#EEE
%SearchHighlightBackground_Format
#F3F709
%SearchHighlightBackground_Foreground
#000000
```

Code editor with syntax highlighting and autocomplete

7.2 Appendix B – 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 3rd party technologies	M	M	M	Allow for more time allocated to research	Switch to a technology which the author has more experience with

Risk	Severity (L/M/H)	Likelihood (L/M/H)	Significance (Sev. x Like.)	How to Avoid	How to Recover
Slow program performance	M	M	M	Ensure that the developed code is being written as efficient as possible	Perform rigorous optimisations
Lack of understanding of how to develop software	H	M	H	Perform significant amount of research prior to development	Focus efforts on aspects that are understood and seek help from external sources such as supervisor
Failure to develop software in timeframe	H	H	H	Regularly compare progress to planned task lists and grant charts	Focus efforts on primary requirements. Leave out secondary requirements to gain time.
Failure to implement secondary requirements	H	M	M	Ensure that primary requirements are completed early	Ensure that the main primary objectives are finished to a good quality
Lack of motivation for the project	M	H	M	Focus and follow planned task lists and grant charts	Switch between tasks to create a more diverse mode of work
Insufficient amount of software testing	M	H	M	Ensure that requirements are finished early allowing for extra testing time	Ensure that core tests (performance and usage) are completed to a satisfactory level.

Risk	Severity (L/M/H)	Likelihood (L/M/H)	Significance (Sev. x Like.)	How to Avoid	How to Recover
Other ACW's taking up final project time	M	H	M	Constantly re-evaluate time necessary to complete the final project and allocate effectively	Focus on final report since it carries more weight
Unable to fix bugs	M	M	M	Pre-plan all algorithms and features to minimise the amount of potential bugs	Perform research on the bug related areas
Damage to equipment	L	L	L	Ensure that all work is constantly updated via cloud and source control software	Quickly seek alternative equipment to continue development on
Planned aims/objectives changing during development	M	H	M	Ensure that the developed software closely relates to the planned designs	Re-work any features that don't agree with the plans
Legal dispute (Claims of copyright)	H	L	H	Ensure an appropriate amount of research has been done before using any 3 rd party tools	Remove from project until a replacement has been found
Software too complex to implement	M	M	M	Plan and follow software designs	Focus on primary requirements and see research

Code editor with syntax highlighting and autocomplete

7.3 Appendix C – 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

Code editor with syntax highlighting and autocomplete

7.4 Appendix D– Project task plan

#	Task name	Duration (days)	Start date	Finish date
1	initial report	15	01/10/2015	15/10/2015
2	Product development	150	31/10/2015	28/03/2016
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 (1 st draft)	13	03/04/2016	15/04/2016
	Final report (2 nd draft)	10	15/04/2016	25/04/2016
	Final report (final draft)	11	25/04/2016	05/05/2016

7.5 Appendix E – Planned Time plan

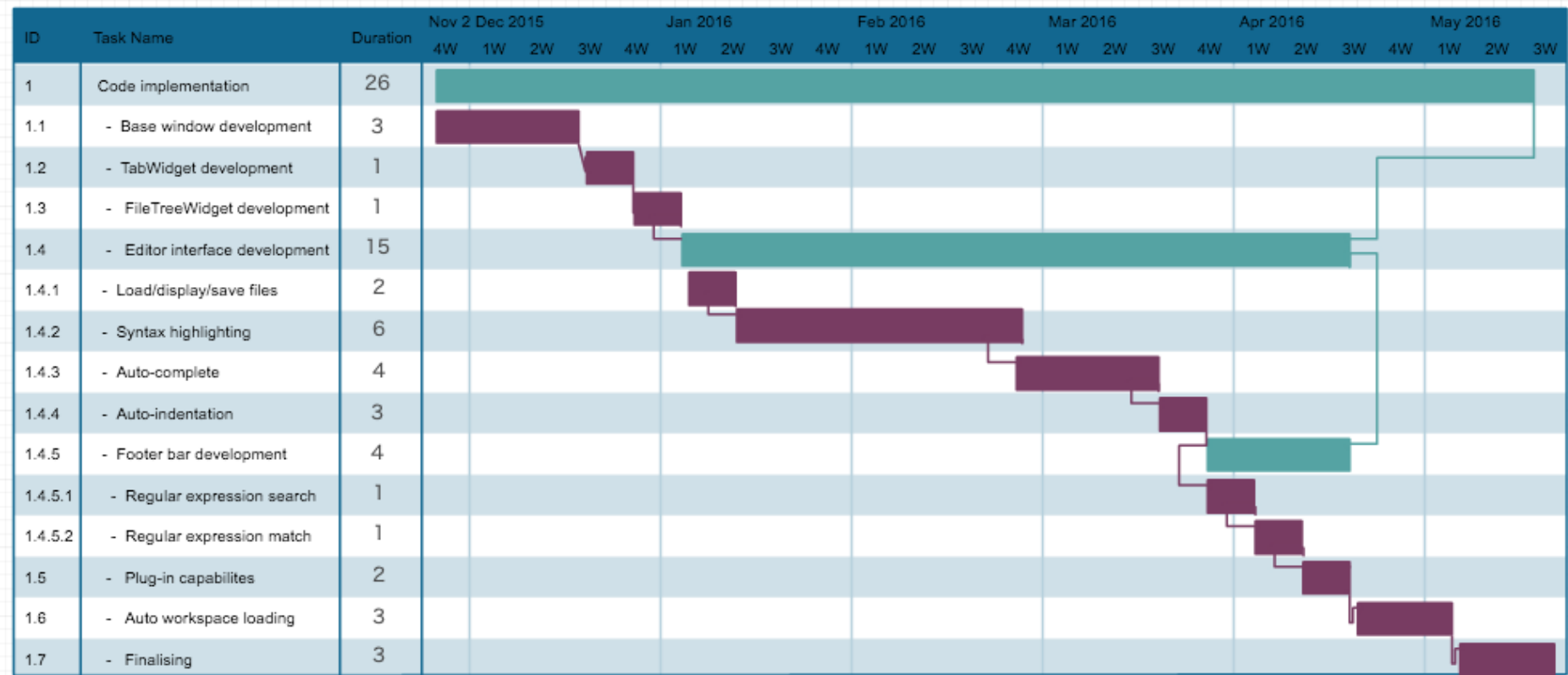
		University Calendar Weeks																																		
#	Task Name	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36		
1	Initial report				D																															
2	Development of project product																																			
	Project planning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

Code editor with syntax highlighting and autocomplete

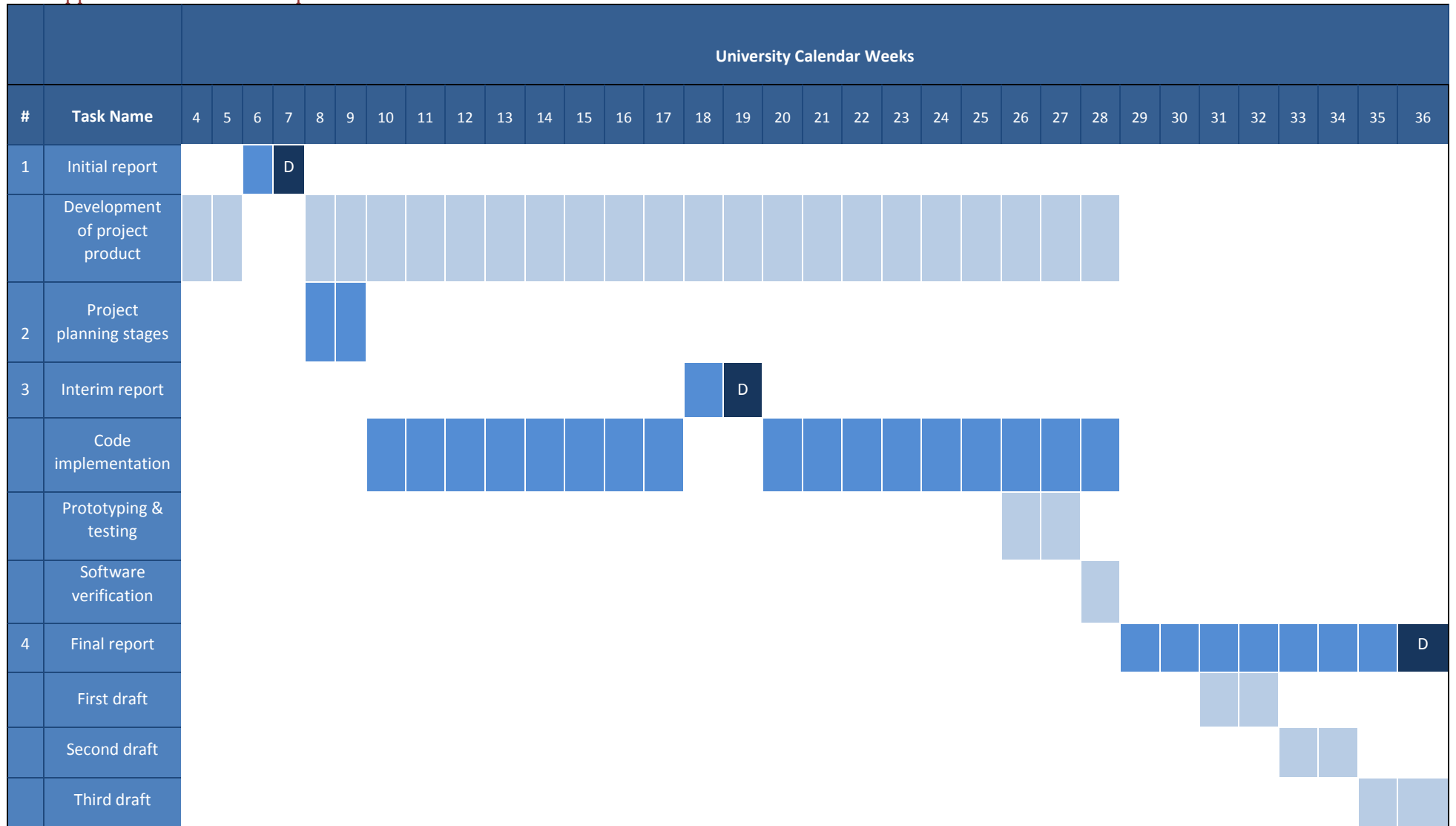
7.6 Appendix F – Planned Code implementation Task list and grant chart

#	Task name	Duration (Weeks)	Start date	Finish date	Notes
	Code implementation	26	28/10/2015	11/05/2016	
1	Base window development	3	28/11/2015	19/12/2015	Colours, layout and responsiveness
2	TabWidget development	1	19/12/2015	26/12/2015	Design for multiple themes
3	FileTreeWidget development	1	26/12/2015	02/01/2016	Directory crawl algorithm
	Editor interface development	15	02/01/2016	16/04/2016	
4	Load/display/save files	2	02/01/2016	16/01/2016	
5	Syntax highlighting	6	16/01/2016	27/02/2016	Load rules from language support
6	Auto-complete	4	27/02/2016	26/03/2016	//
7	Auto-indentation	3	26/03/2016	16/04/2016	Algorithms to set margin
	Footer bar development	4	16/04/2016	16/04/2016	
8	Regular expression search	1	16/04/2016	23/04/2016	
9	Regular expression match	1	23/04/2016	07/05/2016	
10	Plug-in capabilities	2	07/05/2016	21/05/2016	
11	Auto-workspace saving/loading	3	21/05/2016	11/06/2016	

Code editor with syntax highlighting and autocomplete



7.7 Appendix G – Actual Time plan



Code editor with syntax highlighting and autocomplete

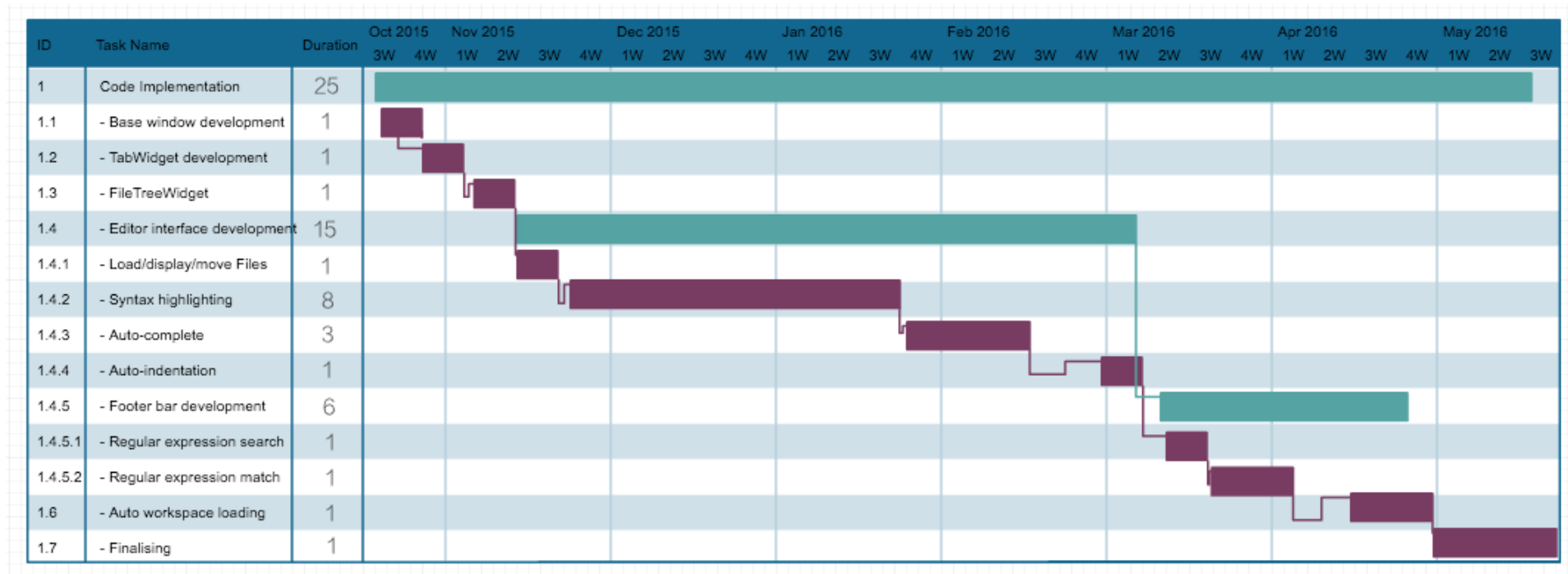
7.8 Appendix H – Actual Code implementation Task list and grant chart

#	Task name	Duration (Weeks)	Start date	Finish date	Notes
	Code implementation	25	23/10/2015	17/05/2016	Does not include finalisation which lasted until deadline
1	Base window development	1	23/10/2015	30/10/2015	Partial development continues throughout entire design process
2	TabWidget development	1	30/10/2015	02/11/2015	Extra development later on due to themes
3	FileTreeWidget development	1	02/11/2015	08/11/2016	Initial widget development, algorithm developed last
	Editor interface development	15	08/11/2016	26/04/2016	
4	Load/display/save files	1	02/01/2016	16/01/2016	Process was a lot more simpler than originally thought
5	Syntax highlighting	8	16/01/2016	12/03/2016	Initial development done within 5 weeks, however needed to be linked to plug in system
6	Auto-complete	3	12/03/2016	20/04/2016	Again, needed to be linked to plug in system (last week)
7	Auto-indentation	1	20/04/2016	26/04/2016	Algorithm developed much faster than originally planned

Code editor with syntax highlighting and autocomplete

Footer bar development		6	26/04/2016	17/05/2016	Dependant on the code editing interface to be completed
8	Regular expression search	1	26/04/2016	02/05/2016	Same as originally planned
9	Regular expression match	1	02/05/2016	10/05/2016	Also had to implement to work with language support.
10	Plug-in capabilities	1	10/05/2016	15/05/2016	Depended on syntax-highlighting and autocomplete
11	Auto-workspace saving/loading	1	15/05/2016	17/05/2016	Last main element that was developed. However finalisation continued until deadline

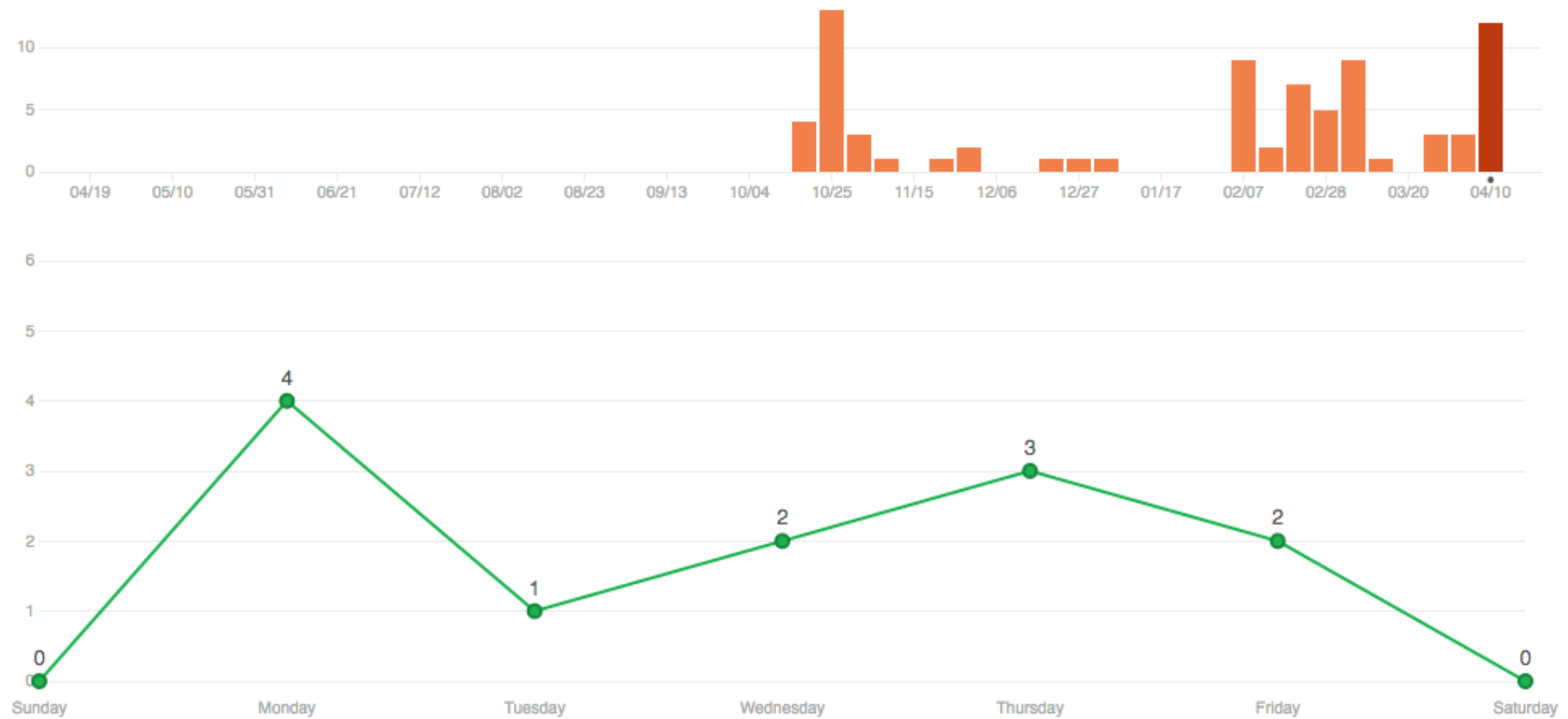
Code editor with syntax highlighting and autocomplete



Code editor with syntax highlighting and autocomplete

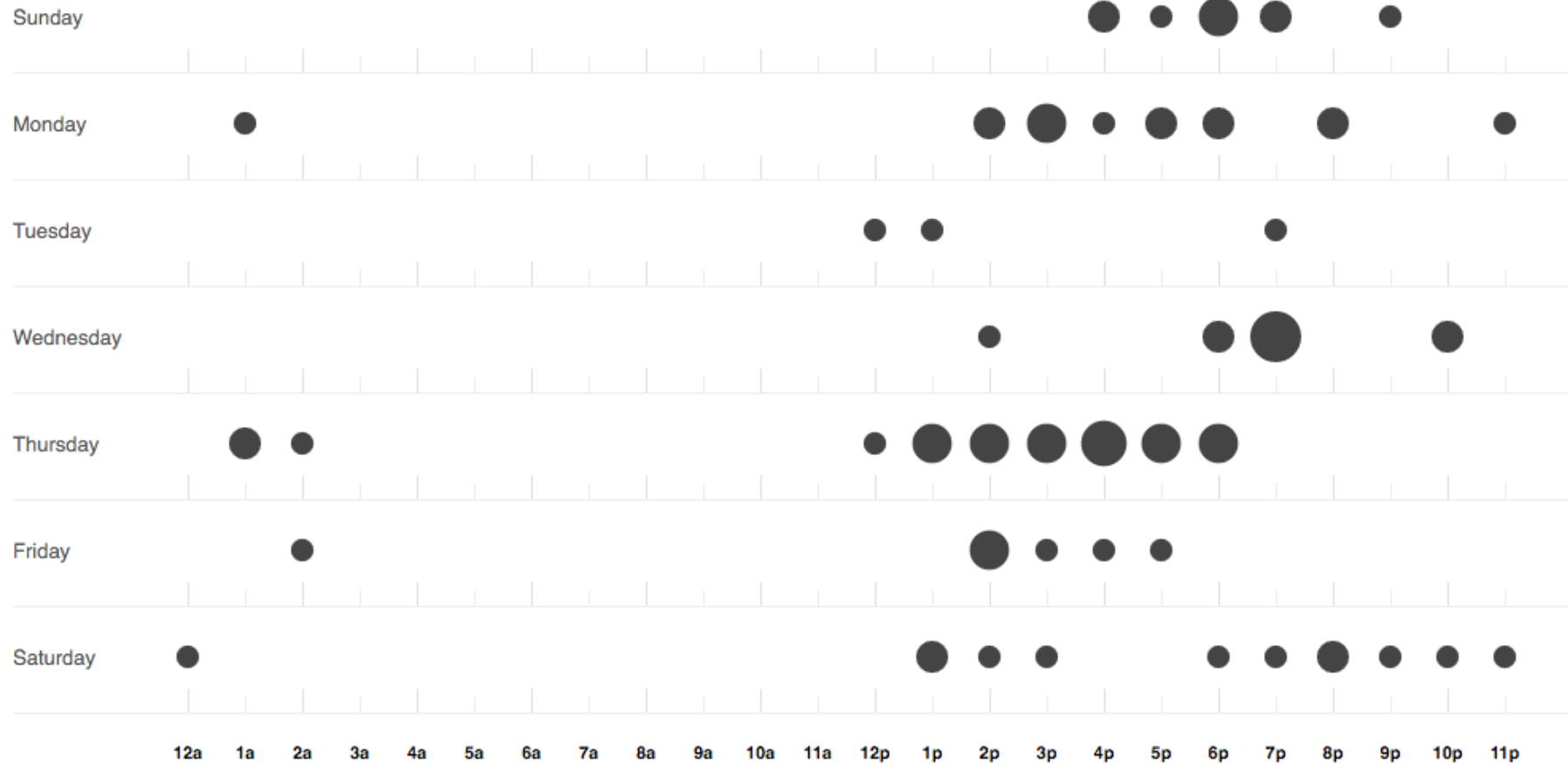
7.9 Appendix I– Able source code commit graphs

(generated by (GitHub, 2016)) – These graphs show the frequency, dates and times of the source code commits by the author



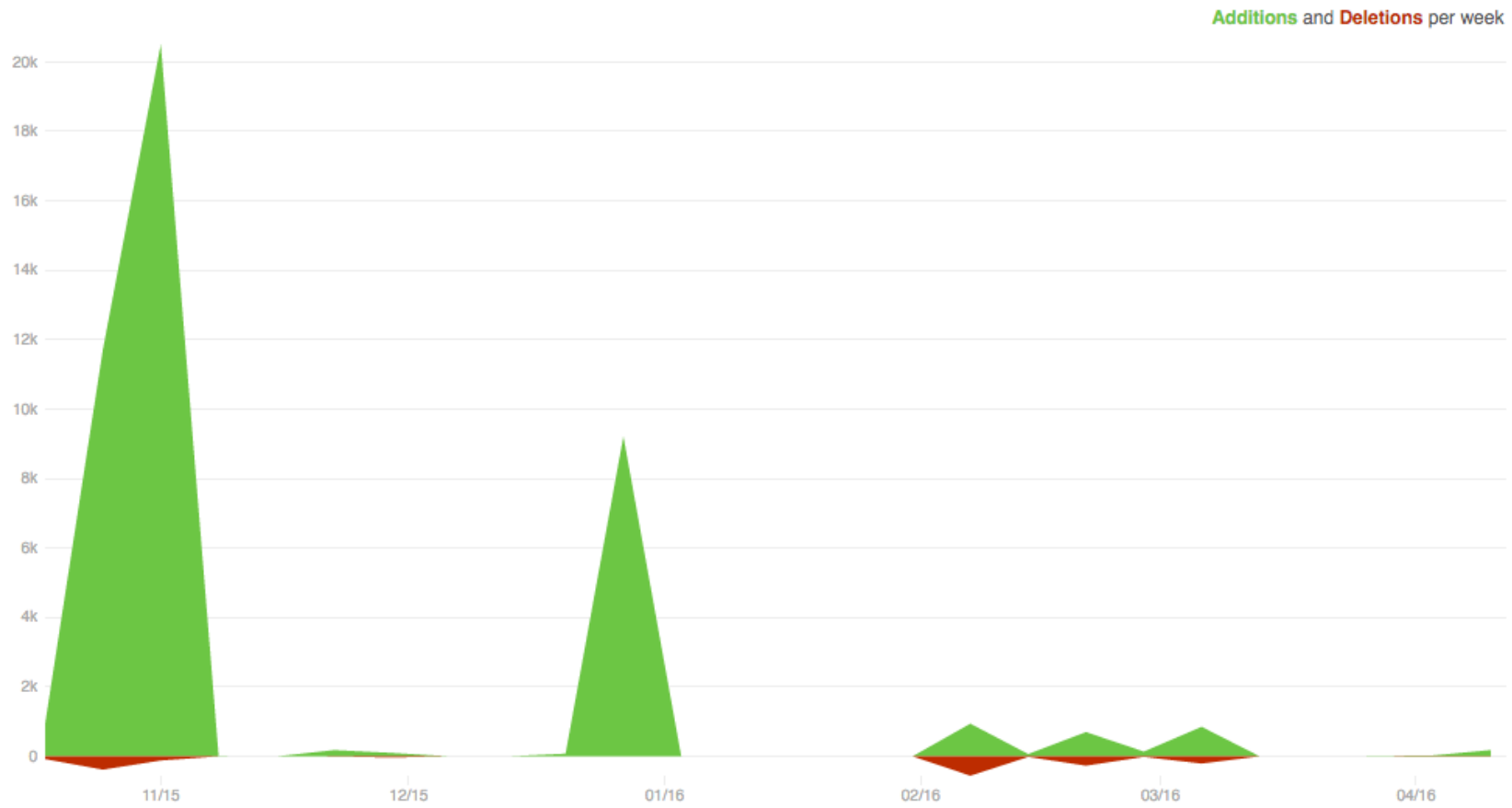
Code editor with syntax highlighting and autocomplete

Punch card for commits



Code editor with syntax highlighting and autocomplete

Code frequency



7.10 Appendix J– Able user manual

Procedure	Expected output	Result
Match regular expression via the code editor footer bar	All elements in code editing interface that are matched to the inputted regular expression are instantly highlighted. All matched items should be made as obvious as possible.	Expected
Replace regular expression via the code editor footer bar	All items that are matched with the defined regular expression are instantly replaced with the defined string within the replace input box.	Expected
Open the Able application (1 st time use)	Welcome screen displayed giving the option to either load or create a new file.	Expected
Open the Able application	The original workspace of the user to be loaded. All tabs, files and themes to remembered from the last time the user used Able and loaded into the system.	Expected
Installed own language support config	File to be located (within the able directory) by the asset manager and displayed as an option with the code editors “language support” combo box.	Expected
Installed own theme	File to be located (Within the able directory) by the asset manager and displayed as an option inside of the settings “themes” combo box.	Expected

7.11 Appendix k– Able user manual (See next page)



Able

User guide

Welcome to Able,

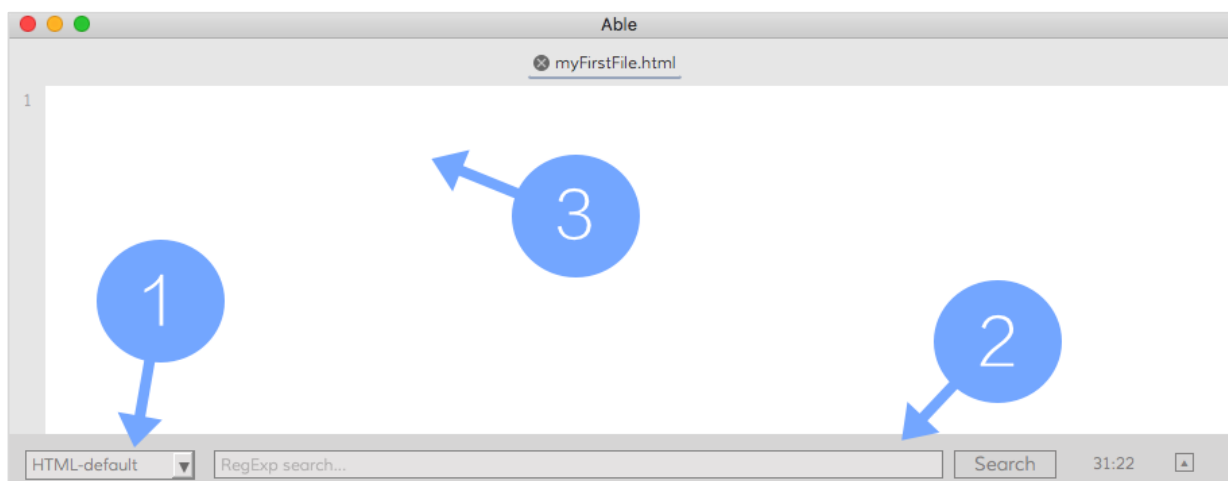
The Fast, lightweight and customisable code editor. Thank you for downloading, we can't wait to hear what you have done with it. But first, lets install a few things.

Installing Able

Fortunately able doesn't come with many bags. All you have to do is locate the downloaded able package, and run the Able application file. Make sure the Able application file stays within the same directory as the "Libs" folder as this contains all of the dynamic resources such as themes and language support files.

Able 'Hello Word'

Since this is your first time using Able, you should be presented with the 'welcome screen'. Once you have started using able more, you will notice that this screen will not be presented as much and this is because the Able system is saving and reloading your original workspace instead of starting from scratch. Anyway, select the 'create new' button in order to create your very first file. Once you have specified the path in which you want to store your first file, Able will generate it for you. Now you should have a screen, which looks like this:



Now you are inside of the editing interface. There are 3 key elements of the editing interface that we need to quickly run through before we start writing any code. These are:

1. The language selector combo box
 - This is where you can change both the syntax-highlighting and auto-complete capabilities. Usually, Able automatically recognises the language of the code that you are writing. But if you wish to select a different language, then simply click the combo box and choose one of the elements.
2. The regular expression search box
 - Here you can type a regular expression and have it instantly matched against any code that you have written after simply pressing the 'search' button. Any code that is matched will be highlighted with a unique colour, which is dependant on the selected language support.
3. The editor text box
 - This is the main element of Able since this is where the magic happens. By clicking on the text area (If not already in focus) begin to type in order to start writing code. Able should begin to perform its syntax highlighting, auto-complete and auto-indentation function. All of these will be explained later in the 'Using Able section'.

Using Able

Able is packed with lots of fun features, but we will continue from where we left off, the editing interface.

[Code-editing interface](#)

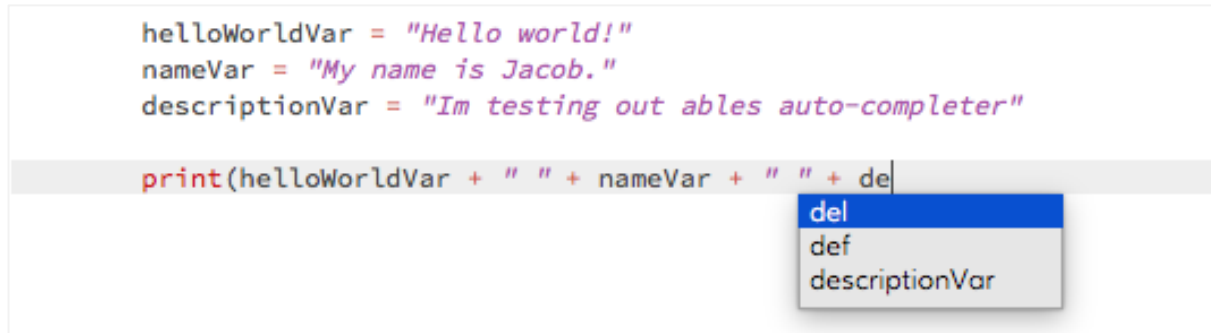
This is the core of Able. This allows users to write code in any language with the assistance of features such as syntax highlighting, auto-complete and auto indentation. Firstly, Syntax highlighting is a common feature that is provided by IDE's and code editors. It manipulates the aesthetic values of the source code in order to draw users attention to more important parts of the code.

```
#PYTHON
#Simple program that counts bottles of beer
def simpleBottleProgram(name, age, location):
    string = " Bottles of beer on the wall"
    for val in range(100, 0):
        print(val + string)
    print("No more bottles on the wall")
```

```
//C++
//Simple program that counts bottles of beer
public void bottles()
{
    for(int i = 100; i > 0; i--)
    {
        cout << String(i)
        cout << " Bottles of beer on the wall";
    }
    cout << "No more bottles of beer on the wall";
}
```

To use syntax highlighting you just have to write in the code editing area and make sure that you have selected the correct language support rules in the footer. Able will automatically apply the highlighting for you.

Another important feature that allows users to write code quicker is the 'auto-complete' function. As you type, this function will produce suggestions of your target word. Theoretically, this process will reduce the amount of keystrokes that you are required to make in order to complete a word by 50%.

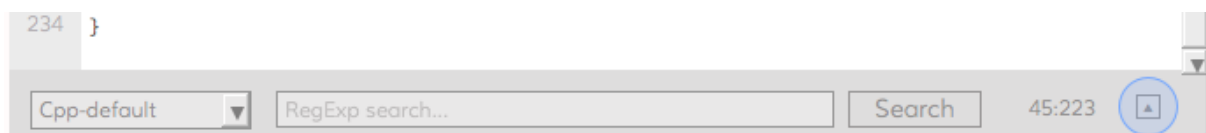


The screenshot shows a code editor with three lines of C++ code: `helloWorldVar = "Hello world!";`, `nameVar = "My name is Jacob.";`, and `descriptionVar = "Im testing out ables auto-completer";`. The code is syntax-highlighted. Below the code, a line of code is partially visible: `print(helloWorldVar + " " + nameVar + " " + de`. A dropdown menu is open, showing suggestions: `del`, `def`, and `descriptionVar`.

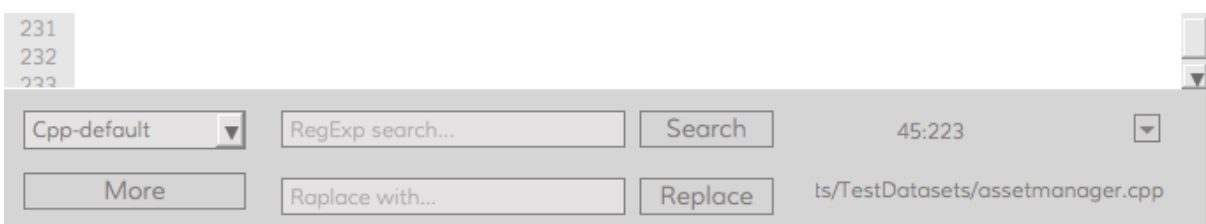
Similarly to the syntax highlighter, in order to use this function, you just need to make sure that you have the correct language selected in the footer combo box.

To further increase the writing speed of Able users, there is an auto-indenter function added to the interface. Since common syntax of code uses indentation as a form of formatting it can be a tedious task to keep pressing tab in order to ensure that your code all stays nicely aligned. Able does this all for you without you even having to do anything. Just type.

The bottom of the editing interface lies a sleek footer bar, this bar contains various useful information such as the absolute file path of the code file that is being edited, The character and row position of the text editing cursor and a function, which allows you to search and replace text. To reveal all of these functions, simply click the expansion arrow:

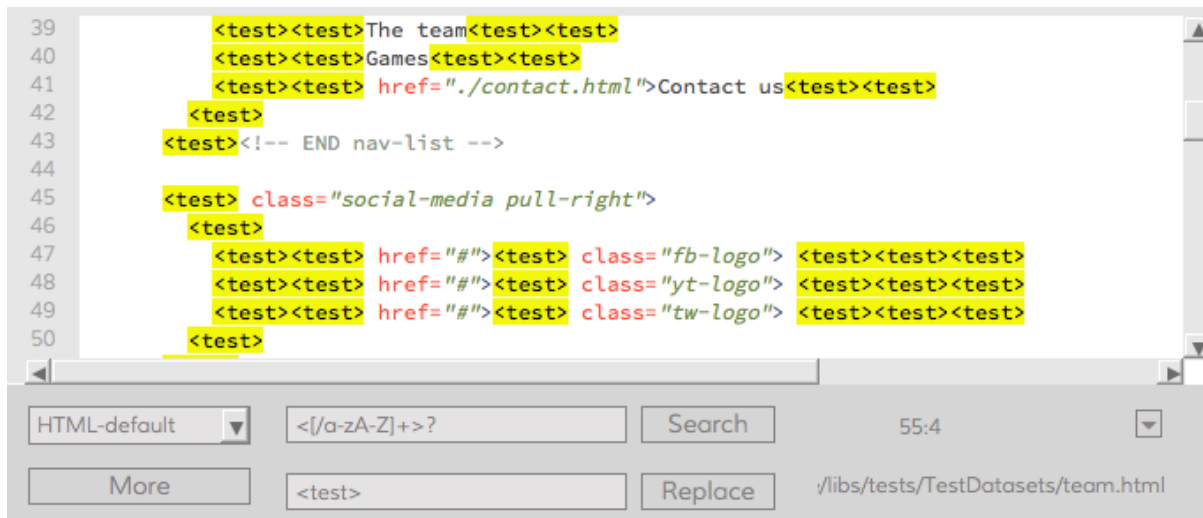


This will cause for the area to expand:



Selecting the button again will cause for the bar to collapse into its default position. The search bar matches an inputted regular expression against the code contained within the editing interface. All matches will be highlighted with a unique colour that is set inside of the language support. Simply write an expression and hit the search button:

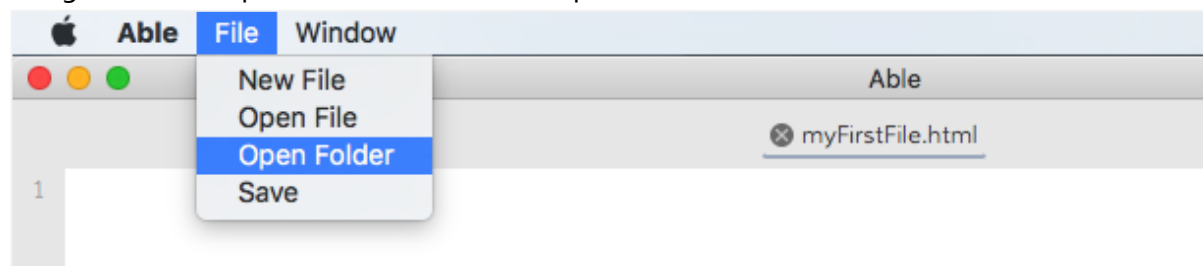
Code editor with syntax highlighting and autocomplete



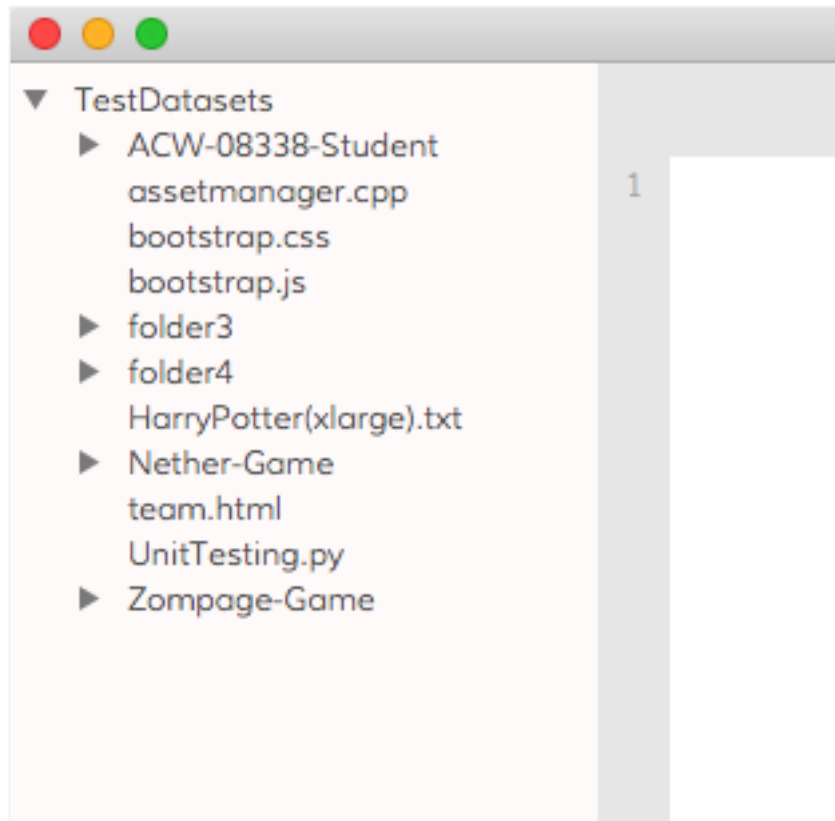
Once the matches have been found, you now have the ability to instantly replace all the matches with a set string. To do this, write your replacement string inside of the replace text input box and hit the replace button. The above image shows some HTML tags being replaced with the string "<test>".

[File system](#)

Able aims to make writing code easier. One way of doing this, able provides a way for the user to manage entire directories of code from within the editor. To use this function, navigate to the top menu bar and select "Open folder":



This will cause for Able to locate the selected directory and display all of its contents inside of the Able file tree widget. From here, you have the ability to quickly access any file displayed.



[Workplace memory](#)

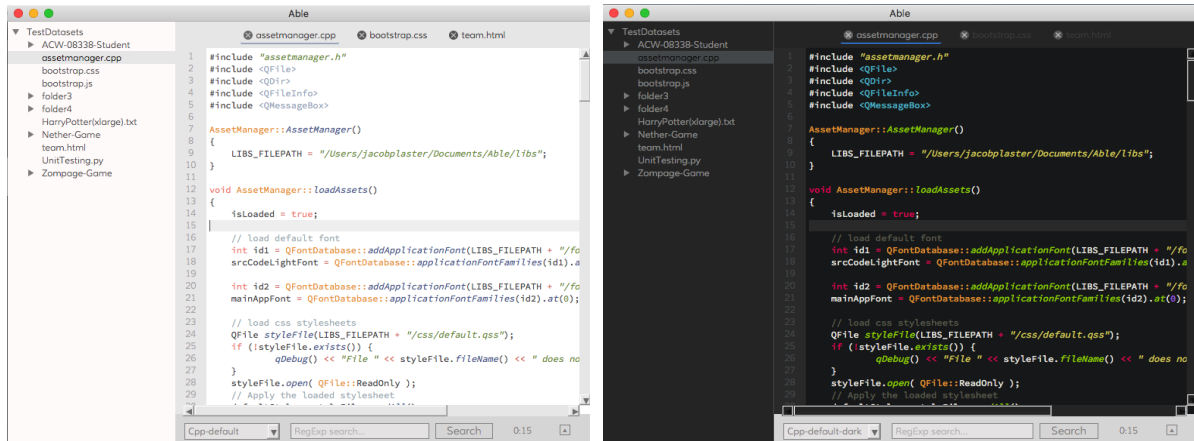
Whenever you close the Able application, your entire workplace will be instantly and automatically saved. Then, the next time you open Able it will be re-loaded. Because we know it's annoying when you have to locate multiple files in order to set up your workspace. To use this feature, you don't have to do anything.

Code editor with syntax highlighting and autocomplete

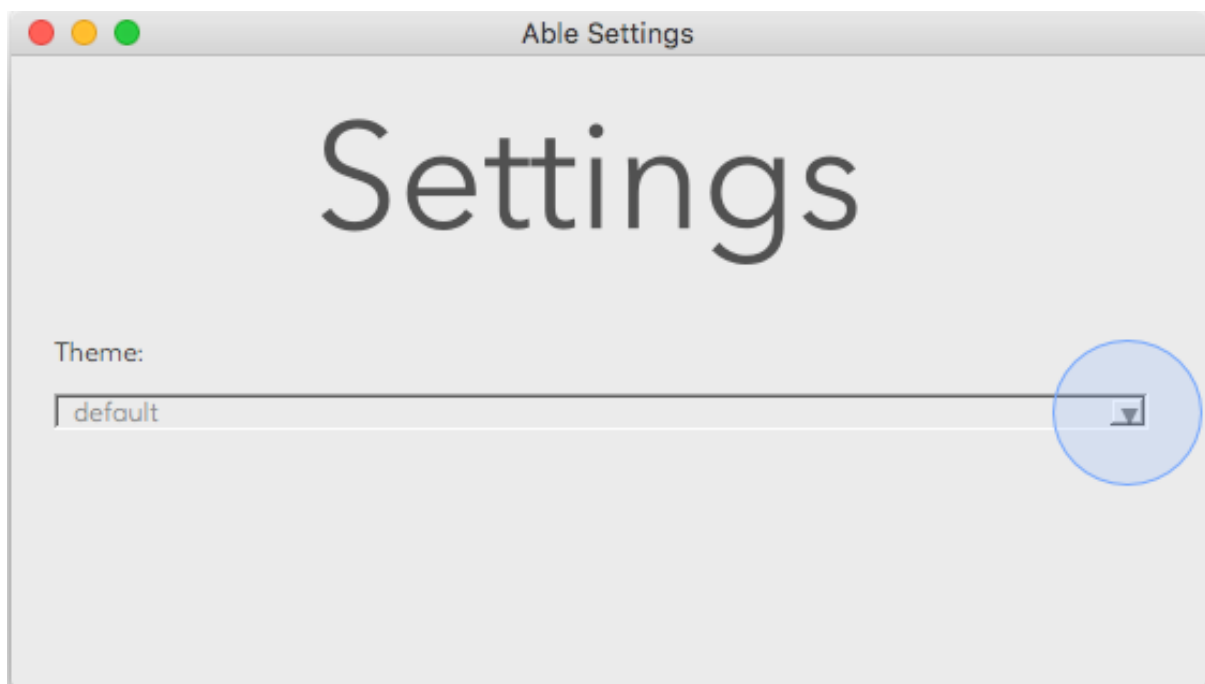
Customising Able

Themes

Able comes pre-packed with two different themes: The light theme and the dark theme.



These themes have been specially developed to follow a common design style called “Minimalistic”. This means that the default themes use very simple colour pallets and sleek UI design in order to provide the user with a calm environment. To change the theme within able, simply navigate to the settings menu by selecting Window>More. You should be presented with a window such as this:



By simply clicking the drop down arrow (displayed inside of the blue circle) you will be presented with a selection of installed themes. Click any one and Able will instantly begin using that theme.

To install your own theme, you need to write your own QCSS style sheet, which conforms to Able's widget guidelines. For examples please see the default light and dark themes that are provided. Once you have done this, simply place the new theme inside of the able directory "Able/libs/css".

[Language support](#)

Able automatically selects the appropriate language support file for the document, which you are editing. So usually, you don't have to worry about it. However, if you feel the need to use a different language support file then simply change it from within the code editor footer bar.

To create your own language support file (Users may want to do this to support a language that isn't supported by default by Able). Firstly, you need to define the file pre-fixes that this file applies to. For example the C++ file uses ".h" and ".cpp". Once you have defined these you can begin writing your expressions that will be used within the syntax highlighter. Able allows for 16 expressions to be used. It doesn't really matter what order the expressions are added as long as the first two are preserved for comments and the last two are reserved for auto-completion. For more detail on this matter please look inside one of the default packed language support files.

Thank you for reading. We can't wait to see what you have made.

8. References

Advait Sarkar., 2015. *The impact of syntax colouring on program comprehension*. Available at: https://www.cl.cam.ac.uk/~as2006/files/sarkar_2015_syntax_colouring.pdf. [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]

Alfred V. Aho., 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison Wesley.

As'ad Mahmoud Alnaser, Omar AlHeyasat , Ashraf Abdel-Karim Abu-Ein, Hazem (Moh'd Said) Hatamleh and Ahmed A. M. Sharadqeh., 2012. Time Comparing between Java and C++ Software, *Journal of Software Engineering and Applications*, 5, pp. 630-633 [Online]. Available at: http://file.scirp.org/pdf/JSEA20120800010_60440877.pdf [Accessed: 14 April 2016].

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://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]

Craig Trim., 2016. *The Art of Tokenization*, Available at: <https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en> [Accessed: 2016 April 13].

Dale Fisk., 2005. *Programming with Punched Cards*, Available at: <http://www.columbia.edu/cu/computinghistory/fisk.pdf> [Accessed: 20th April 2016].

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].

Dick Grune and Criel Jacobs., 2008. *Parsing Techniques: A Practical Guide*, 2nd edn. US: Springer.

Donald E. Knuth (1983) 'Literate Programming', *The computer journal*, 27(2), pp. 97-111 [Online]. Available at: <http://www.literateprogramming.com/knuthweb.pdf> (Accessed: 17 April 2016).

Douglas R. Hofstadter., 2000. *Gödel, Escher, Bach: an eternal golden braid*, London: Penguin.

FLTK., 2014. Fast light toolkit C++ framework [online]
Available at: <http://www.fltk.org> [Accessed 2016 March 10]

Github., 2016. Source control tool [online]
Available at : <https://github.com/> [Accessed 2016 April 11]

Hacker news., 2016. *Computer science community forum*, Available at:
<https://news.ycombinator.com/> [Accessed: 17 April 2016].

Ivan Marsic., 2012. *Software engineering*, Rutgers University, New Brunswick, New Jersey: Rutgers.

Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, Pat McCarthy., 2004. *The Java Developer's Guide to Eclipse*, 2nd edn., Boston: Addison-Wesley.

Kevanc Muslu, Yuriy Brun , Reid Holmes , Michael D. Ernst and David Notkin., 2012. *Speculative Analysis of Integrated Development Environment Recommendations*.
Available at: <https://people.cs.umass.edu/~brun/pubs/pubs/Muslu12oopsla.pdf> [Accessed: 14 April 2016].

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

LibGDX., 2016. Cross-platform framework [online]
Available at: <https://libgdx.badlogicgames.com/> [Accessed 2016 April 11]

Maggie Johnson and Julie Zelenski., 2008. Lexical Analysis , Available at:
<http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/03-Lexical-Analysis.pdf>
[Accessed: 14 April 2016].

Marina Herold., 2008. Typing speed, spelling accuracy and the use of word-prediction, *South African Journal of Education*, 28, pp. 117-134.

Martin Fowler and Kent Beck., 1999. *Refactoring: Improving the Design of Existing Code*, : Addison-Wesley Professional.

Masoud Nosrati., 2011. Python: An appropriate language for real world programming, *World Applied Programming*, 1(2), pp. 110-117 [Online]. Available at:
<http://waprogramming.com/download.php?download=50ae4a1125d607.12866725.pdf>
[Accessed: 14 April 2016].

Microsoft., 2013. Visual studios 2013 [online]
Available at: <http://www.visualstudios.com> [Accessed 2016 March 3]

Minh Vu and Craig Thompson., 2005. *E2 Agent Plugin Architecture* ,
Available at:<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1427047&tag=1>
[Accessed: 17 April 2016].

Notepad+., 2016. Code/Rich text editor [online]
Available at: <https://notepad-plus-plus.org/> [Accessed 2016 March 3]

Parasoft., 2016. Available at: <https://www.parasoft.com/product/jtest> [Accessed: 7th April 2016]

Paul Klint., 2007. *Syntax Analysis*, Available at: <http://homepages.cwi.nl/~daybuild/daily-books/learning-about/syntax-analysis/syntax-analysis.html> [Accessed: 2016 April 13].

Paul R. Kroeger., 2005. *Analyzing Grammar: An Introduction (Cambridge Textbooks in Linguistics)*, Cambridge University Press.

Phillip A. Laplante., 2007. *What every engineer should know about software engineering*. CRC press. ISBN 0849372283.

PlayN., 2016. Google cross-platform framework [online]
Available at: <https://github.com/playn/playn> [Accessed: 2016 April 11]

Roberto Minelli, Andrea Mocci and Michele Lanza., 2016. *Measuring Navigation Efficiency in the IDE*, Available at:
<http://www.inf.usi.ch/phd/minelli/downloads/Mine2016a.pdf> [Accessed: 14 April 2016].

Ultimate++, 2016. C++ GUI framework [online]
Available at: <http://www.ultimatepp.org/> [Accessed 2016 March 10]

University of Virginia., 2007. *Lexical Analysis: Regular expression*, Available at:
<http://www.cs.virginia.edu/kim/courses/cs471/lec/cs471-02-lex.pdf>
[Accessed: 14 April 2016].

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://www.qt.io/download/> [Accessed 2016 March 7]

Tam C and Wells D., 2009. Evaluating the Benefits of Displaying Word Prediction Lists on a Personal Digital Assistant at the Keyboard Level, *Assistive Technology: The Official Journal of RESNA*, 21(3), pp. 105-114

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].

Code editor with syntax highlighting and autocomplete

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].