|  |
| --- |
|  |
| Word Processor – The Text Generation  Jamie McGrath |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  |  |  |  | |

Contents

[Table of Figures 2](#_Toc90155338)

[1. Executive Summary 3](#_Toc90155339)

[2. Introduction 4](#_Toc90155340)

[3. Requirements 5](#_Toc90155341)

[4. Design 6](#_Toc90155342)

[4.1 Purpose of Object-Oriented Design 6](#_Toc90155343)

[4.2 Design Methodology 6](#_Toc90155344)

[4.3 Software Design 7](#_Toc90155345)

[4.3.1 Use Case Diagram 7](#_Toc90155346)

[4.3.2 Activity Diagram 8](#_Toc90155347)

[4.3.3 Save Activity Diagram 9](#_Toc90155348)

[4.3.4 Class Diagram 10](#_Toc90155349)

[4.3.5 Sequence Diagrams 11](#_Toc90155350)

[4.4 Summary of Design 15](#_Toc90155351)

[4.5 Object-Oriented Principles 19](#_Toc90155352)

[5. Summary 21](#_Toc90155353)

[6. References 22](#_Toc90155354)

[7. Appendices 23](#_Toc90155355)

[7.1 Pseudocode 23](#_Toc90155356)

[7.2 Initial GUI Draft Wireframe 28](#_Toc90155357)

[7.3 How to Execute the Code 29](#_Toc90155358)

# Table of Figures

[Figure 1: Initial GUI Draft 6](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155374)

[Figure 2: Use Case Diagram 7](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155375)

[Figure 3: Activity Diagram 8](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155376)

[Figure 4: Save Activity 9](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155377)

[Figure 5: Class Diagram 10](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155378)

[Figure 6: OpenFile Sequence 11](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155379)

[Figure 7: SaveFile Sequence 12](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155380)

[Figure 8: CopyPaste Sequence 12](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155381)

[Figure 9: Undo/Redo Sequence 13](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155382)

[Figure 10: Templates Sequence 13](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155383)

[Figure 11: SaveOnExit Sequence 14](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155384)

[Figure 12: SRP 15](#_Toc90155385)

[Figure 13: Save Abstract Class 15](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155386)

[Figure 14: Open Abstract Class 15](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155387)

[Figure 15: OpenFile Extends Method 15](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155388)

[Figure 16: Private Variables 16](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155389)

[Figure 17: Getters and Setters 16](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155390)

[Figure 18: Liskov Principle in Operation 16](#_Toc90155391)

[Figure 19: Main Method with Thread 17](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155392)

[Figure 20: Implements Runnable Interface 17](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155393)

[Figure 21: ActionListener Interface 17](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155394)

[Figure 22: UserTemplates Interface 17](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155395)

[Figure 23: AlterText Interface 18](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155396)

[Figure 24: KeyListeners Interface 18](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155397)

[Figure 25: Templates Interface 18](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155398)

[Figure 26: Inheritance 19](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155399)

[Figure 27: Encapsulation Getters and Setters 19](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155400)

[Figure 28: Polymorphism Example 20](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155401)

[Figure 29: Show File Wireframe 28](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155402)

[Figure 30: Initial Wireframe Draft 28](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155403)

[Figure 31: Show Template Wireframe 28](https://d.docs.live.net/9ef314dae1738a85/Desktop/CIS2149%20OOP/CW1_01/report/CW1_01.docx#_Toc90155404)

[Figure 32: Source Path 29](#_Toc90155405)

[Figure 33: Set JDK to Path 29](#_Toc90155406)

[Figure 34: Compilation of Java Files in src Folder 29](#_Toc90155407)

[Figure 35: Application Executed 29](#_Toc90155408)

# Executive Summary

The objective of this report is to explain, in detail, the design and implementation of a Word Processing application.

The simple Word Processor application will allow users to create their own notes, open existing text files, save current files and manipulate the text within the application.

The application will allow users to manipulate the text by implementing undo and redo actions in the form of buttons, change the font style from regular to **bold**, *italic* and underlined using JToggleButtons, as well as allow them to cut, copy and paste text using the keyboard actions or right clicking from within the notepad area and selecting the appropriate options, and finally allow the user to select the Font Family they wish to use within the text area.

The application will also allow users to add pre-defined templates, such as a CV template, Cover Letter template and a standard letter template.

Finally, the application will prompt the user to save any existing notes that have been created before exiting the application.

# 2. Introduction

This report will show and detail how an application has been created using Java, which will allow a user to utilise a word processor for every day needs.

The program will consist of multiple factors, including a Graphical User Interface (GUI). The programme will also allow users to perform all the functions that a word processor should allow, such as cut, copy and paste; undo and redo functionality; text styling such as **bold**, *italic* and underline; allow the user to insert ready made templates and also allow the user to save the current document – as well as prompting the user if they are overwriting a current existing file, and also prompting the user to save the data if they exit the application without saving.

Using Java to create the program; the facilities involved in the creation were successfully designed and implemented utilising various classes such as FileWriter, FileReader, JFileChooser and BufferedReader; as well as utilising error catching code such as if/else statements, try/catch and more, to make the application streamlined and fault-free where possible. The code also utilises various class constructors and methods to achieve the result of a fully functioning Word Processor.

The report will initially detail the low level requirements of the application, which were decomposed from the high level requirements created during the design phase.

Moving on to the design portion - the design process will explain the purpose of Object-Oriented Design and why it is useful. The explain the design methodology; how the design came together, including the initial drafts, UML diagrams and how the code was implemented from the design.

The design section of the report will also detail the Object-Orientated Principles that were used during the design and why they were used.

The report will then summarise what has been achieved during the design and implementation of the program. The summary will also reflect on the design features were met and how the application could be improved in the future.

# 3. Requirements

Table : High & Low-Level Requirements

|  |
| --- |
| HLR1: Must include a Graphical User Interface (GUI). |
| LLR1.1: GUI shall include options for a user to interact. Buttons shall include:  File | Templates | Bold | Italic | Underline | Undo | Redo |
| LLR1.2: Buttons must be intuitive, utilising toggles and text tips to make the user aware of what they do. |
| HLR2: Must allow users to type in to a text area. |
| LLR2.: Text within text area must be editable, using cut, copy, paste, bold, italic and underline. |
| HLR3: Must allow user the ability to open existing files. |
| LLR3.1: Must open within a specified file path in order for the user to select the existing file. |
| HLR4: Must allow users the ability to use pre-defined templates. |
| LLR4.1: Must open within a specified file path in order for the user to select the pre-defined template. |
| HLR5: Must allow user the ability to save their current document. |
| LLR5.1: Must open within a specified file path in order for the user to save their document. |
| LLR5.2: Must prompt user if they are attempting to overwrite an existing file. |
| HLR6: Must prompt user to save existing file if they attempt to exit the application. |
| HLR6.1: Must open within a specified file path in order for the user to save their document. |

# 4. Design

## 4.1 Purpose of Object-Oriented Design

The purpose of Object Oriented Design (OOD) is to create a structure for building a programme that is streamlined, fault free and easy to develop from the design principles – this usually comes after the Object Oriented Analysis (OOA). With good OOA, it makes it easier to enter the design phase, in which the architecture for the code is developed, Graphical User Interfaces (GUI’s) are designed and the implementation enables the programming phase to be completed efficiently.

Elgabry (2017) argues “The design phase refines the analysis model and applies the needed technology and other implementation constrains. It focuses on describing the objects, their attributes, behavior, and interactions. The design model should have all the details required so that programmers can implement the design in code.”.

Essentially, OOD is the most useful in terms of ensuring high quality design and code. Making it easier for the programmers to build the application.

## 4.2 Design Methodology

As part of the design process for the Word Processor application, a Rapid Application Development (RAD) model was used.

Prototyping and developing the design of program, alongside building the application side by side, making the entire process efficient and effective.

During the OOD of the design phase, an initial draft of the GUI (Graphical User Interface) (Figure.1) was created, with emphasis on the high level requirements found during the OOA phase. Ensuring the user has the ability to utilise every feature of the application in an intuitive manner.

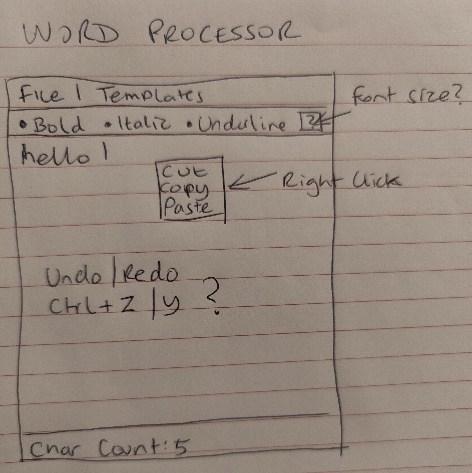
Once the initial draft of the GUI was created, creation of the UML (Unified Modeling Language) was started. The UML being the next phase of the design process, in which it will create the architecture to build the code from.

Figure 1: Initial GUI Draft

## 4.3 Software Design

### 4.3.1 Use Case Diagram

The Use Case Diagram (Figure.2) is built from the requirements of the program, as below:

*Basic -* User requires an application that can allow them to write notes in to some kind of text box. The application must have the ability to open and save text files and must also tell the user how many characters are in the text box.

*Intermediate -* The application must warn the user if they try to overwrite a file when saving, and must allow the ability to overwrite if they accept. The application must warn users if they try to exit without saving. The application must also allow users to insert pre-defined templates and must allow them to do so using a menu or button.

*Advanced* - The application also requires the ability to copy and paste text within the text box, provide an undo/redo facility and finally, the application must give the ability to stylise the text (bold, italic, underline).

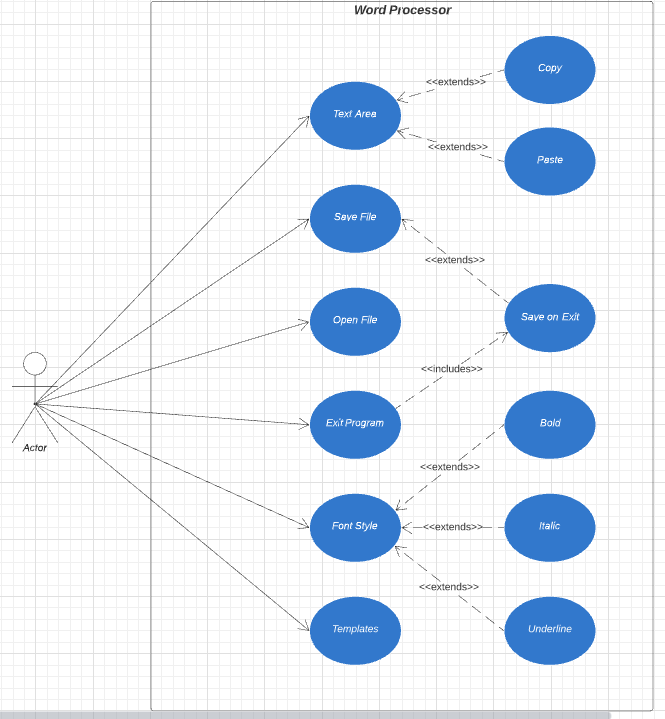


Figure 2: Use Case Diagram

### 4.3.2 Activity Diagram

Figure.3 is the Activity Diagram, which shows the flow from the user opening the Word Processor application, going through the basic features with the main activity as follows:

* Open application.
* [Optional] Open file from location.
* Type/Edit document.
* Save file -> Select location of save.
* Exit application -> Save on exit.

Diagram

Description automatically generated

Figure 3: Activity Diagram

### 4.3.3 Save Activity Diagram

Figure.4 shows the flow of the save function; if the user selects a location that does not exist, or if the file cannot be saved due to conflict or system error, the application will throw an error and display to the user.

Diagram

Description automatically generated

Figure 4: Save Activity

### 4.3.4 Class Diagram

Once the Use Case and Flow Diagrams were created, the next step was to create the code architecture with Class Diagrams.

The Class Diagram (Figure.5) is intended to show the relationships and classes required in order to build the code from. Contained within the Class Diagram are the methods and variables that will be used to create the Word Processor application. Athuralyia (2021) states “A class diagram is a UML diagram type that describes a system by visualizing the different types of objects within a system and the kinds of static relationships that exist among them. It also illustrates the operations and attributes of the classes.”.

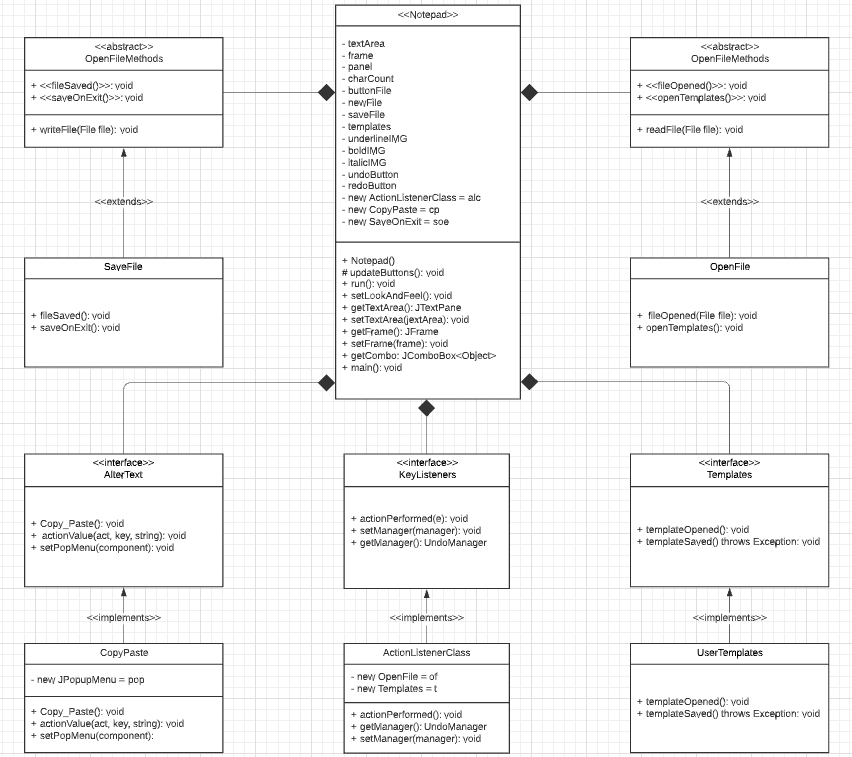


Figure 5: Class Diagram

### 4.3.5 Sequence Diagrams

Sequence diagrams are designed for testing purpose; they represent the flow of messages the system would send in order to ensure the user has a clear idea of any errors or successful actions they have encountered or taken. Pedamkar (2020) states “UML Sequence Diagrams are designed so that they can depict a timeline. On the top, you can see the beginning, and then the diagram flow descends downwards to mark the sequence of all interactions in the system.”, which will show in the following Sequence Diagrams:

#### 4.3.5.1 Open File Sequence

Figure.6 is the Sequence the user would take when opening a text file.

The first thing the user would do is select ‘Open File’ from the ‘File’ Menu Button, which would open up a JFileChooser dialog box; in which the user would then be able to select the specified file and the interface would display the contents of the file in the text area.

Diagram

Description automatically generatedShould there be any errors with opening the text file, such as the file path no longer exists, or the file is corrupt; the application would display a “File Cannot be Opened” error message box.

Figure 6: OpenFile Sequence

#### 4.3.5.2 Save File Sequence

Figure.7 shows the SaveFile Sequence, in which the user would select the ‘Save’ option from the Menu Button. Once the ‘Save’ option is chosen, the application will open the JFileChooser in which the user will be able to save the document to. If an existing file is chosen to be saved over, a dialog box will ask the user if they wish to overwrite the existing file with the new one.

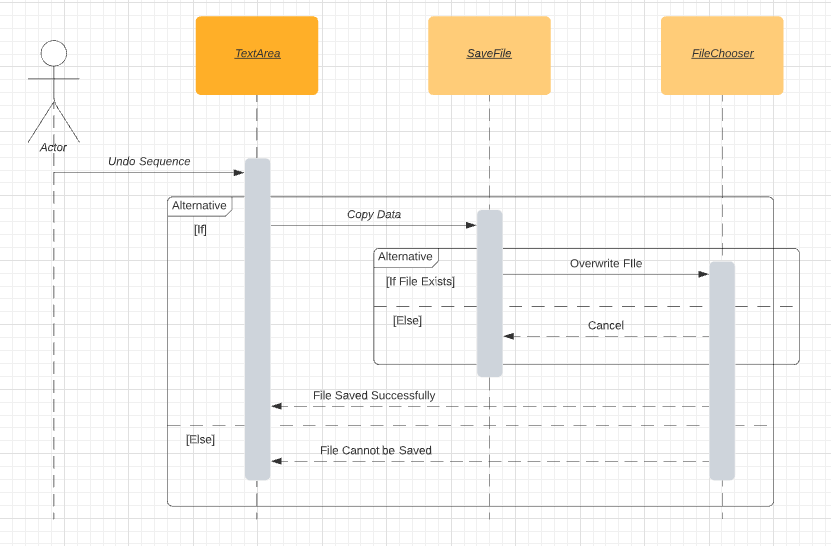


Figure 7: SaveFile Sequence

#### 4.3.5.3 Copy/Paste Sequence

Figure.8 is the Copy and Paste sequence. This shows the user has the ability to copy and paste. By right clicking the mouse and selecting cut, copy, or by using the keyboard actions (Ctrl + X, C), the user selected text within the text area of the application copies to a clipboard contained within the library class DefaultEditorKit which will then paste the selected text back to the text area, once the paste action has been utilised (Keyboard action: Ctrl + V, or right click + Paste).

Diagram

Description automatically generated

Figure 8: CopyPaste Sequence

#### 4.3.5.4 Undo/Redo Sequence

Diagram

Description automatically generatedFigure.9 shows the undo and redo sequence. In this sequence, the user would click the ‘Undo’ button and the text that is undone will copy to a clipboard contained within the UndoManager library, ready to be pasted back to the text area once the ‘Redo’ button is clicked.

Figure 9: Undo/Redo Sequence

#### 4.3.5.5 Templates

Figure.10 shows the relationship between the text area and the pre-defined templates when the user wants to insert a template. The user would click the ‘Templates’ button in the application, which will then open the JFileChooser dialog box where the user can select the template they wish to use. The template will then open in the text area for the user to edit how how they choose.

Diagram

Description automatically generatedIf any errors occur, such as a corrupted file, for any reason, the application will notify the user that the file could not be opened.

Figure 10: Templates Sequence

#### 4.3.5.6 Save on Exit Sequence

Figure.11 is the SaveOnExit sequence. In this sequence it shows the relationship between the TextArea, SaveOnExit and FileChooser objects.

When the user clicks to exit the application, a dialog box will appear asking the user if they would like to save the file they are currently working on.

* If the user selects ‘Yes’, the application will open the File Chooser where they will be able to specifiy the save.
* If user selects ‘No’, the application will close.
* If user selects ‘Cancel’, the application will stay open and will give user access to the text area again to continue typing.

If the user selects ‘Yes’, then continues to save the file but an error exists within the system, such as a corrupted file, the application will show a dialog box that warns the user that the file could not be saved.

If the user attempts to overwrite an existsing file, the application will also prompt the user to accept, reject or cancel the save.

Chart, box and whisker chart

Description automatically generated

Figure 11: SaveOnExit Sequence

## 4.4 Summary of Design

For this application, Object-Oriented principles were adhered to as closely as possible. The main focus was to create as many objects as possible within their own class, and have the classes interact with each other, in order to maintain an effective and effiencent program.

**S**ingle Responsibility Principle: When creating the classes for the application, SOLID principles were adhered to where possible. Attempting to keep each function in it’s own class, in order to utilise the “Single Responsibility Principle” (SRP).



Figure 12: SRP

Figure.12 shows the classes that were created in order to maintain the Single Responsibility Principle. Each class focuses on one particular task, ranging from opening the files for the user to edit, or saving on exit if the user chooses.

**O**pen/Close Principle: The Open/Close Prinicple (OCP) is a way of ensuring your code is open to extension, but closed to modification. This essentially means that the main body of the code is left untouched, but future methods can be added utilising that code, but without modifying it – this forces programmers to create new classes that can plug in to the extension points created by the abstract classes and interfaces (McLean Hall, 2017).

Figure.13, 14 & 15 show two abstract classes and an abstract method that were created utilising the file manager processes within the application. The purpose of these classes are to provide the opportunity to add any extra functionality that may be required for expansion, without having to modify the main body of the code; meaning any methods implemented in the future can access the main body of code without having to change it to suit the method.

**Text

Description automatically generatedText

Description automatically generated**

Figure : Save Abstract Class

Figure : Open Abstract Class

**A screenshot of a computer

Description automatically generated with medium confidence**

Figure : OpenFile Extends Method

Figure.16 & 17 show getters and setters, as well as private variables, all maintaining the structure of the classes and variables, with the purpose of maintaining the Open/Close principles.

**Text

Description automatically generated**Text

Description automatically generated

Figure 16: Private Variables

Figure 17: Getters and Setters

**L**iskov Substitution Principle: Similar to the OCP, the Liskov Subsitution Principle extends from OCP, with Janssen (2018) arguing “It extends the Open/Closed Principle by focusing on the behavior of a superclass and its subtypes.”.

With that in mind, an effort was made to adhere to this rule, by creating subclasses that could be altered that would in turn, not affect the application running and working. An example of this can be seen below:

Figure.18 shows the Liskov Principle in effect. If one class is removed from the application, the application still works as intended. Essentially allowing for new subclasses to be used to replace the current subclasses for greater efficiency, where possible.

Text

Description automatically generated

Figure : Liskov Principle in Operation

**I**nterface Segregation Principle: The Interface Segregation Principle (ISP) determines that clients should not be forced to implement Interfaces that they will not use. ISP is essentially intended to split large Interfaces in to smaller, more specific interfaces so that clients will only know about the Interfaces they intend to use (Fadatare, 2018).

In the program used for the application, two types of Interface were utilised – library interfaces and a class interface. Figure.19 & 20 show the Runnable Interface being utilised. The Runnable Interface is contained within the Notepad main class, due to a Thread being used to check the character count is being constantly updated and is never empty when the user is typing.

Text

Description automatically generated

Figure : Main Method with Thread

Figure : Implements Runnable Interface

Figure.21 shows the ActionListener Interface being used. This ensure the application receives the action events utilised within the methods in that class.

Text

Description automatically generated

Figure 21: ActionListener Interface

Figure.22 shows a class interface. This interface is the intended implementation of the ISP. Here, anything related to the User Templates class is contained within the Templates interface.

Text

Description automatically generated

Figure : UserTemplates Interface

Each class within the application created utilises either an interface or an abstract class. Keeping in line with the ISP as intended.

**D**ependency Inversion Principle: DIP is a strategy of each class depending upon interfaces or abstract functions and class, rather than depending upon concrete functions. DIP states that high level modules should not have any dependency on low level modules, with both depending on abstraction, and abstractions should not depend upon details; details should depend on abstractions (Arora and Chauhan, 2014).

Whilst every effort was made to understand how to implement the Dependency Inversion Principle (DIP), it proved difficult to implement. However, an attempt was made to ensure that every class was contained within abstraction or an interface as shown in Figures.23, 24 & 25, with an effort towards maintaining the core principles of DIP.

Text

Description automatically generatedText

Description automatically generated

Figure : AlterText Interface

Figure : KeyListeners Interface

Text

Description automatically generated

Figure : Templates Interface

With the SOLID design principle in mind, the hours put in to understanding the SOLID principles shaped the design of the code during the OOA phase, and helped with understanding many features that could be implemented in to the program to make it more streamlined and error free.

## 4.5 Object-Oriented Principles

The design for the Word Processor application features instances of Inheritance, Encapsulation, Abstraction and Polymorphism at its core principles.

Inheritance: This is when a class inherits the values of another class, in order to prevent duplication of code. Thanoshan (2020) states “Inheritance is a mechanism in which one object acquires all the states and behaviors of a parent object.”.

As seen in Figure.26, the OpenFile class extends from OpenFileMethods, as when the user wants to open any type of file, be it the templates or a saved file, the OpenFile class utilises the code within the body of the OpenFileMethods abstract class in order to give the user access to the files where they will find the intended document, and then open said document; this is a direct extension of the OpenFileMethods class, as such, Inheritance is utilised in order to maintain unduplicated code.

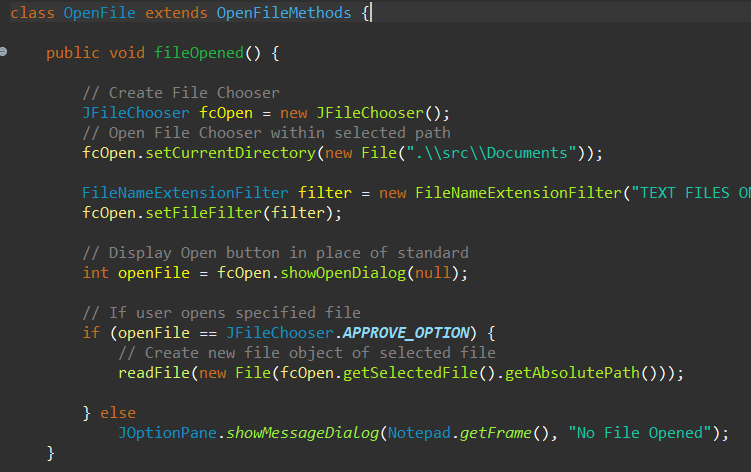


Figure 26: Inheritance

Text

Description automatically generatedEncapsulation: This is intended to hide the implementation details from the user. Essentially if a method/variable is set to private then it can only be accessed within the same class, meaning no outside classes can access the implementation details of the method/variable that has been set to private. (Singh, 2013). In order to access these methods/variables inside another class in order to gain access to the implementation details, getters and setters are set up within the containing “private” class. The getter is then called on from the outside class in order to access the method/variable.

Figure 27: Encapsulation Getters and Setters

Figure.27 shows one instance within the Word Processor classes where getters and setters were created in order to access the methods from within other, outside classes; helping to maintain the encapsulation principles.

Abstraction: As shown on page 16 (Liskov Substitution Principle), 17 & 18 (Dependency Inversion Principle), abstract methods and interfaces were utilised in order to maintain a heirarcy between superclasses and subclasses.

The purpose of abstraction is to hide details from a user, intending to only show the information required for the particular task that the user intends to utilise:

“When you design a method as part of a solution to a problem, each method begins as a box that states what it does but not how it does it. No one box may “know” how any other box performs its task – it may know only what the task is” (Prichard and Carrano, 2013).

Polymorphism: This is when a program has many classes that are related to each other through inheritance, only polymorphism allows the program to use methods for different tasks.

Text

Description automatically generated

Figure : Polymorphism Example

As shown in Figure.28, Notepad.getTextArea() is utilising polymorphism in order to set the text within an existing file that has been opened by the user, to the text area for view and editing by the user – this is just one example of polymorphism being used within the Word Processor application.

Kendal (2009: 90) states that, “Polymorphism allows objects to be handled without regard for their precise class. This can assist in making systems extensible without compromising encapsulation of the existing design.”. This basically allows the programmer to use objects from other methods without worrying about non compliance with SOLID principles and such.

# 5. Summary

To conclude, as seen in the report the Word Processor application was completed with the main goal of adhering to the SOLID design principles, as well as maintaining Object-Oriented Programmings’ core values.

The aim of the report was to show the importance of the analysis and design before the implementation of the code to create the application; including the design methodology used in the creation of the program.

Starting with the purpose of Object-Oriented Design, then going through the stages of UML diagrams, which shows the flow from start to finish beginning with Use Cases all the way through to Sequence Diagrams, as all part of the analysis phase of the design.

Once the analysis phase was over, it was then on to the design phase, which began with the design of the graphical user interface through to creating the classes required and then finally the implementation of the code, as shown within Chapter 4.4 Summary of Design.

As for what was achieved during the design and implementation process; SOLID design principles were mostly adhered to, including Single Responsibility Principle, Open/Close Principle, Liskov Substitution Principle, Interface Segregation Principle and Dependency Inversion Principle. However, with the Dependency Inversion Principle in mind, it felt this was not implemented in an efficient way, due to the complexities that were involved, it proved tough to adhere to.

Another achievement was the use of the core principles of Object-Oriented Programming; In that Inheritance, Encapsulation, Abstraction and Polymorphism were used throughout the implementation of the code.

There are many ways this application could be improved in the future, the main one would be utilising DIP 100%. Then there are extra features, such as allowing the importation of PDFs and giving the user the ability to alter the font size.

Another improvement that could be made would be to utilise Multithreading properly. Currently the application utilises one Thread, which checks the character count is being updated at all times during the process of the application. However; using a Thread for the undo and redo libraries could also have been implemented, in order to keep the buttons from throwing any errors if the text field is empty – something that can be done by pausing the “character count” thread whilst the “undo/redo” thread is in operation. However, again; implementing this proved difficult, but not without many hours trying.

All in all, the application runs well, and as intended; with scope for improvement in more ways than one, and also with the ability to add features, such as font size and font family in the future.

# 6. References

Arora, A., Chauhan, N., 2014. An Object Oriented Design Approach for Modification of Rotten Code using Regression Testing & Refactoring [eBook]. 4th ed. India: International Research Publication House. Available from: [ijictv4n7spl\_05.pdf (ripublication.com)](http://www.ripublication.com/irph/ijict_spl/ijictv4n7spl_05.pdf) [Accessed 11 December 2021].

Athuraliya, A., 2021. *The Easy Guide to UML Class Diagrams*. [Blog online]. 10 September. Available from: [The Ultimate Class Diagram Tutorial to Help Model Your Systems Easily (creately.com)](https://creately.com/blog/diagrams/class-diagram-tutorial/) [Accessed 17 November 2021].

Carrano, F., and Prichard, J.J., 2013. *Data Abstraction and Problem Solving with Java, Walls and Mirrors.* 3rd ed. Boston: Addison-Wesley.

Elgarby, O., 2017. *Introduction to object-oriented analysis and design.* [Blog online]. 19 March. Available from: [Object-Oriented Analysis And Design — Introduction (Part 1) | by Omar Elgabry | OmarElgabry's Blog | Medium](https://medium.com/omarelgabrys-blog/object-oriented-analysis-and-design-introduction-part-1-a93b0ca69d36) [Accessed 16 November 2021].

Fadatare, R., 2018. *Interface Segregation Principle in Java with Example* [Blog online]. Available from: [Interface Segregation Principle in Java with Example (javaguides.net)](https://www.javaguides.net/2018/02/interface-segregation-principle.html) [Accessed 20 November 2021].

Janssen, T., 2018. *SOLID Design Principles Explained: The Liskov Substitution Principle with Code Examples* [Blog online]. 11 April. Available from: [SOLID Design Principles Explained: The Liskov Substitution Principle with Code Examples – Stackify](https://stackify.com/solid-design-liskov-substitution-principle/) [Accessed 20 November 2021].

Kendal, S., 2009. *Object-Oriented Programming Using Java* [eBook]. 1st ed. bookboon.com. Available from: [Object Oriented Programming using Java (irdtuttarakhand.org.in)](https://irdtuttarakhand.org.in/Upload/object-oriented-programming-using-java.pdf) [Accessed 21 November 2021].

McLean Hall, G., 2017. *Adaptive Code: Agile coding with design patterns and SOLID principles.* 2nd ed. Redmond, Washington: Microsoft Press.

Pedamkar, P., 2020. *UML Sequence Diagram*. [Blog online]. Available From: [UML Sequence Diagram | Symbol and Components of Sequence Diagram (educba.com)](https://www.educba.com/uml-sequence-diagram/) [Accessed 17 November 2021].

Singh, C., 2013. *Encapsulation in Java with example* [Blog online]. Available from: [Encapsulation in Java with example (beginnersbook.com)](https://beginnersbook.com/2013/05/encapsulation-in-java/) [Accessed 21 November 2021].

Thanoshan, MV, 2020. *Object-Oriented Programming Principles  in Java:  OOP Concepts for Beginners* [Online blog]. 1 May. Available from: [Object-Oriented Programming Principles  in Java:  OOP Concepts for Beginners (freecodecamp.org)](https://www.freecodecamp.org/news/java-object-oriented-programming-system-principles-oops-concepts-for-beginners/) [Accessed 20 November 2021].

# 7. Appendices

## 7.1 Pseudocode

**Notepad Class extends JFrame implements Runnable**

*New Object of ActionListener Class;*

*New Object of CopyPaste Class;*

*New Object of SaveOnExit Class;*

*Notepad()*

*Set Name App Title (“Word Processor”);*

*Create Text Area;*

*Create Menu Bar;*

*Create Tool Bar;*

*Create Menu Option (“File”);*

*Create Menu Option (“Template”);*

*Create Dropdown Menu Option (“New”, add Image Icon);*

*Create Dropdown Menu Option (“Open”, add Image Icon);*

*Create Dropdown Menu Option (“Save” add Image Icon);*

*Create Dropdown Menu Option (“Open Template” add Image Icon);*

*Create EditListener for the Text Area, Undo Function;*

*Add Action Listener for (“New”) Menu Option;*

*Add Action Listener for (“Open”) Menu Option;*

*Add Action Listener for (“Save”) Menu Option;*

*Add Action Listener for (“Open Template”) Menu Option;*

*Add (“File”) to Menu Bar;*

*Add (“Template”) to Menu Bar;*

*Create new Toggle Button (“Bold”);*

*Create margin for Bold Button;*

*Create new Toggle Button (Italic);*

*Create margin for Italic Button;*

*Create new Toggle Button (“Underline”);*

*Create margin for Underline Button;*

*Create new Action Lisntener, using StyledEditorKit library for Bold Button;*

*Create new Action Lisntener, using StyledEditorKit library for Italic Button;*

*Create new Action Lisntener, using StyledEditorKit library for Underline Button;*

*Create new Button (“Undo”);*

*Create new Button (“Redo”);*

*Add Action Listener to Undo Button;*

*Create margin for Undo Button;*

*Add Action Listener to Redo Button;*

*Create margin for Redo Button;*

*Add (“Bold”) to Tool Bar;*

*Add (“Italic”) to Tool Bar;*

*Add (“Underline”) to Tool Bar;*

*Create space between buttons;*

*Add Undo Button to Tool Bar;*

*Add Redo Button to Tool Bar;*

*Create Popup Menu for Text Area;*

*Add Character Count to Application;*

*Add new Window Listener for SaveOnExit Class;*

*Add ScrollPane to Text Area;*

*Add JPanel to SOUTH;*

*Add Tool Bar to NORTH;*

*Set Menu Bar;*

*Add Icon Image;*

*Set Size (600, 600);*

*Set Look and Feel;*

*Do Nothing when User Closes app;*

*Set Visible to show Application;*

*Run()*

*While Text Area is the focus owner;*

*Get the length of text in Text Area;*

*If character is white space decrement from count;*

*Set count to Character Count;*

*setLookAndFeel()*

*Setting new theme using UIManager;*

*getTextArea()*

*return Text Area;*

*setTextArea()*

*Set Notepad Text Area to “jextArea”;*

*getFrame()*

*return Frame;*

*setFrame()*

*Set Notepad Frame to Frame;*

*getCombo() JComboBox<>*

*return combo;*

*Main()*

*Create new Thread instance of Notepad;*

*Start Threat;*

*end*

**ActionListenerClass implements ActionListener**

*KeyListener <<interface>>*

*New Object of OpenFile Class;*

*New Onject of Templates Class;*

*Create new Undo Manager;*

*actionPerformed()*

*Set Action Command to “onClick”;*

*If onClick = (“Save”);*

*Run SaveFile’s fileSaved Method;*

*If onClick = (“Open”);*

*Run OpenFile’s fileOpened Method;*

*If onClick = (“New”);*

*Run Notepads getTextArea Method and create empty Text Area;*

*If onClick = (“Open Template”);*

*Run Templates openTemplates Method;*

*If onClick = (“Undo”);*

*Try to undo appropriate edits;*

*Catch UndoException error;*

*If onClick = (“Redo”);*

*Try to redo appropriate edits;*

*Catch RedoException error;*

*getManager()*

*return manager;*

*setManager()*

*Set ActionListenerClass’ manager to manager;*

**Abstract SaveFileMethods Class**

*Abstract method fileSaved();*

*Abstract method saveOnExit();*

*writeFile(file)*

*Try;*

*Create New FileWriter;*

*Create New BufferedWriter;*

*Overwrite existing text in Text Area to new File;*

*Display (“File Saved Successfully”);*

*Close BufferedWriter;*

*Catch IOException;*

*Display (“No File Saved”);*

*fileSaved() extends SaveFileMethods;*

*Create New FileChooser;*

*Set FileChooser to Current Directory (“.\\Source\\Documents”);*

*Set FileChooser to display Save button;*

*If user chooses Save Option, Display Selected File;*

*Try;*

*If File already exists;*

*Prompt user to overwrite existing File;*

*If User selects YES;*

*Call writeFile() method;*

*Else;*

*Call writeFile() method;*

*Catch IOException;*

*Display (“No File Saved”);*

*saveOnExit() extends SaveFileMethods;*

*If Text Area is not empty;*

*When user exits Application, warn of unsaved changes;*

*If user selects YES when prompted to save changes;*

*Call writeFile() method;*

*Exit system;*

*If user selects No when prompted to save changes;*

*Exit system;*

*Else;*

*Return to Text Area;*

*Else;*

*Exit system;*

*end*

**Abstract OpenFileMethods Class**

*Abstract method fileOpened();*

*Abstract method openTemplates()*

*readFile(file);*

*Try;*

*Create new String Object 1 (Exiting Text in Text Area);*

*Create new String Object 2 (Text within Selected file);*

*Create New FileReader;*

*Create New BufferedReader;*

*Read each line from String Object 2;*

*While String Object 1 has no empty lines;*

*Maintain existing newline characters when opening String Object 2 (selected file);*

*Overwrite String Object 1 (existing text in Text Area) with Stirng Object 2 (selected file);*

*Close BufferedReader;*

*Catch IOException;*

*Display (“No File Opened”);*

*fileOpened() extends OpenFileMethods;*

*Create New FileChooser;*

*Set FileChooser to Current Directory (“.\\Source\\Documents”);*

*Set FileChooser to display Open button;*

*If user chooses Open on selected file;*

*Create new File from user selection;*

*Call readFile() method;*

*Else;*

*Display (“No File Opened”);*

*openTemplate() extends OpenFileMethods;*

*Create New FileChooser;*

*Set FileChooser to Current Directory (“.\\Source\\Templates”);*

*Set FileChooser to display Open button;*

*If user chooses Open on selected file;*

*Create new File from user selection;*

*Call readFile() method;*

*end*

**CopyPaste Class**

*AlterText <<interface>>*

*copyPaste();*

*actionValue(act, key, string);*

*setPopMenu(component);*

*Create new Popup Menu;*

*copyPaste()*

*Create new Action Value using DefaultEditorKit.Cut, bind to Key Event Ctrl+X, set to String (“Cut”);*

*Create new Action Value using DefaultEditorKit.Copy, bind to Key Event Ctrl+C, set to String (“Copy”);*

*Create new Action Value using DefaultEditorKit.Paste, bind to Key Event Ctrl+V, set to String (“Paste”);*

*actionValue(action (TextAction), key (Accelerator Key), string (“Copy”, “Cut”, “Paste”)*

*Create Abstract Action of Accelerator Key, get users KeyStroke, set key to Ctrl;*

*Create Abstract Action of Name, set to string (“Copy”, “Cut”, “Paste”);*

*Add action to Popup Menu;*

*setPopupMenu(component)*

*If component = empty, return component;*

*For text in component;*

*Set text to Popup Menu;*

*end*

## Text Description automatically generated with medium confidence7.2 Initial GUI Draft Wireframe

Figure 29: Show File Wireframe

Graphical user interface, text, application

Description automatically generatedGraphical user interface, application

Description automatically generated

Figure 30: Initial Wireframe Draft

Figure 31: Show Template Wireframe

## 7.3 How to Execute the Code

Opening code using cmd (Command Prompt):

Open cmd > find the source path to the program that requires compilation (Fig.28):



Figure : Source Path

Set the source path to the Java Development Kit (Fig.29):



Figure : Set JDK to Path

Compile Java Class files (Fig.30):



Figure : Compilation of Java Files in src Folder

Execute the main Java Class file for the program (Fig.31):

Graphical user interface, text, application

Description automatically generated

Figure : Application Executed