Software Engineering Group Project 20
Maintenance Manual

Author:	Kain Bryan-Jones[kab74],
	Luke Wybar[law39],
	Tom Perry[top19]

Config Ref:	MaintenanceManualGroup20
Date:	06/05/2020
Version:	1.0
Status:	Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

Clear, consistently followed, document standards[3] are essential in software engineering.

1.1 Purpose of this Document

This document describes an in-depth view of the Welsh Vocabulary Tutor program. This document should be read by any developers or 'maintainers' who are looking for specific answers to questions they have regarding how the Welsh Vocabulary Tutor program works. It will be useful for the user to be familiar with the Design Specification Documentation [1] as it will be a reference a lot in this document. This is to save time in this document where the Design Specification [1] has already provided a sufficient description.

1.2 Scope

This document describes an in-depth view of the Welsh Vocabulary Tutor program. This document should be read by any developers or 'maintainers' who are looking for specific answers to questions they have regarding how the Welsh Vocabulary Tutor program works. It will be useful for the user to be familiar with the Design Specification Documentation [1] as it will be a reference a lot in this document. This is to save time in this document where the Design Specification [1] has already provided a sufficient description.

1.3 Objectives

The objective of this document is to:

Provide maintainers with the resources they need to understand the workings of the Welsh Vocabulary Tutor program.

Provide maintainers with a detailed description of the program so that they can use it to solve any issues they may be encountering.

Provide maintainers with the knowledge on how to safely implement features into the program. I.e So that the maintainer does not break the current working code.

2. PROGRAM DESCRIPTION

The Welsh Vocabulary Tutor (WVT) program's purpose is to provide a user interface where users can practice their Welsh ability. WVT provides the ability to lookup dictionary entries in a table, add these entries to a practice list, lookup these practice words in a table, run assessments with these practice words, and finally provide flashcards based on the practice words to assist with learning and revision.
WVT uses JSON files to store dictionary information. The user will need to provide the JSON file to be loaded/saved from/to.

3. PROGRAM STRUCTURE

The Design Specification document [1] outlines in detail the program structure.
Section 2.1 of the Design Specification document [1] provides a description of the different functionalities provided by the Welsh Vocabulary Tutor program. A list of all modules(packages) is provided along with a description of the purpose of each module.

Section 2.2 of the Design Specification document [1] provides a more detailed look into each of the modules of the program. Every class in the modules is displayed along with a description of what each class provides.
Section 4 of the Design Specification document [1] provides a list of all methods in each of the classes. These classes are grouped by which module they lie in. This section also provides a description of each method as well as the signature of that method, i.e the name of the method, its return type, its access modifier, as well as the parameters it takes as input.

4. ALGORITHMS

The Design Specification document [1] outlines the significant algorithms within the program. Descriptions of these algorithms are also provided within the document as well as where these algorithms take place. That is to say what must happen before this algorithm runs and what happens as a result of said algorithms running.

Section 5.2 of the Design Specification document [1] provides a list of all significant algorithms and their descriptions

5. THE MAIN DATA AREAS

The Design Specification document [1] details where important information is stored in the program. The main area of data is the storage of DictionaryEntry objects within a linked list. This is done within the Application Class in the package 'uk.ac.aber.cs221.group20.javafx'. The practice list stored within this class is a subset of the dictionary list, this is also a linked list.

Section 5.3 of the Design Specification document [1] provides detail on significant data structures.

6. FILES

6.1 Json files

The program stores data in a standard JSON file, formatted to the schema defined in the specification document 'SE.QA.CSRS DC3'[2]. This is requested from the user at runtime, and then loading into the Application's internal data structures. When the program is closed, the file is automatically written to the location it was loaded from, providing a seamless experience for the user. This file is fundamental to the program's functionality.

6.2 FXML files

The program uses FXML to store the layout and elements displayed on the Screen to fulfil each functional requirement in the original specification[2]. These files and the screen they display are listed below:

1. Add Word scene - addword.fxml
2. Dictionary scene - dictionary.fxml
3. Flashcard scene - flashcard.fxml
4. Practice List scene - practicelist.fxml
5. Match Meaning scene - matchthemeaning.fxml
6. Six Meaning scene - sixmeanings.fxml
7. Translation scene - translation.fxml

These files are stored in the resources folder in the package 'uk.ac.aber.cs221.group20', it is possible to add more scenes by including a new FXML file here and adding the scene in the SceneEnum Enumeration in the Class 'uk.ac.aber.cs221.group20.javafx.ScreenSwitch' in the format addWordScene("addword.fxml") for the Add Word scene above for example.

6.3 CSS files

A single CSS file was used to produce the program to the original specification[2], this was stored along with the FXML files in the resources folder in the package 'uk.ac.aber.cs221.group20' with the filename styles.css. This is used to add further styling to the JavaFX scenes outside of the FXML files.

6.4 Image files

A number of image files were used to make the GUI more intuitive, allowing actions and screens to be simplified into simple icons.

These files are stored in the resources folder in the package 'assets.icons' with packages under this layer breaking the icons up into white and black, assets.icons.white_icons and assets.icons.black_icons respectively, with one further layer of packages breaking them up into resolution.
Black icons are unselected, white icons when selected, this is intuitive to the user experience.

6.5 Flashcard file

There is a single image file in the resources folder in the package 'assets.flashcard'.
This contains an image which is used as the frame on the Flashcard Screen.

7. INTERFACES

The program is written in Java 12, using a number of libraries.
We used IntelliJ IDEA as our Java IDE.
Jackson version 2.9.4 is used as the json IO library.
JUnit Jupiter version 5.4.2 is used as the unit testing library.
JavaFX version 11 is used as the GUI library.

Library management is performed by Maven, if you wish to add more libraries or change the version of the library used, you will need to make the appropriate modifications to the pom.xml file stored at the root of the project. Maven will download the required libraries at compile time, this means library files will not otherwise be visible to the developer.

Json files are assumed to be formatted in the schema defined in 'SE.QA.CSRS DC3'[2].

8. SUGGESTIONS FOR IMPROVEMENT

If the program is to be improved, there are four primary areas that can be changed in order to make the program better. These improvements would have been added to the program if given more development time. The four suggestions for potential improvement are as described below:

8.1 Self Assessment Tests

A suggestion for improving the Self Assessment Tests would be to increase the number of types of test that the user can do to add a greater variety of questions. Currently, the program implements the three question types as specified in SE.QA.CSRS[2] with support added for additional tests. A possible idea for one of these additional tests could include a 'spot the odd one out' tests on a number of dictionary definitions and translations to find the one that is incorrect.

The implementation for adding extra questions would be relatively simple with any new tests extending the Question class and being added to the AssessmentGenerator class's generate assessment so that they would be included in the list of questions that are generated. An additional FXML file and controller class would be needed so that the test has a GUI for the user to interact, with the FXML for this being very similar to the existing self assessment tests.

8.2 JUnit Tests

A suggestion for improving the JUnit tests would be to add more tests to the JUnit test classes detailed in the Design Specification Document[1] 'test' package, to improve the general robustness of the system by testing for multiple scenarios. Examples could include adding additional JUnit tests into 'JSONTest' to see what happens when you try and load a file with fields missing.

As well as expanding the existing classes, additional tests could be added for the Controllers to test the robustness of the program's JavaFX within the 'test' package. These tests can be implemented using the TestFX library which specializes in testing JavaFX with these tests originally planned but were not implemented due to time constraints.

8.3 FXML

Another suggestion for improvement would be the programs FXML. In its current state, the program's pane cannot be decreased past a certain point due to some of the FXML not scaling properly leading it to overlap with the screens side bar. This was due to the screens not being made consistently meaning some could scale well whilst others couldn't. This was planned to be fixed however it couldn't be done due to time constraints and would make the program more user friendly.

In order to implement these changes, it is recommended that most of the pages are redesigned within scene builder which has built in functionality to get panes to automatically scale relative to their parent.

8.4 Program's 'dictionary' list

One final suggestion for improving the program would be to add automatic alphabetical sorting to the program's 'dictionary' list in Application, whenever a new DictionaryEntry object is added to it. In its current state, the program is not re-sorted when a new definition is added, meaning the 'DictionaryController' has to sort the words alphabetically by its chosen language whenever the dictionary is displayed. Adding automatic resorting would mean that the controller no longer has to do the sorting manually, moving away functionality that isn't relevant to the class.

A way to put this into practice could be by implementing a *compareTo* method within the DictionaryEntry class. This method would be used to compare the alphabetical ordering of the words English or Welsh definition depending on the current language ordering. With this method implemented, you could use a Collections.Sort method to re-sort the dictionary every time a new word is added.

9. THINGS TO WATCH OUT FOR WHEN MAKING CHANGES

When making changes to the program there are a number of things that you must watch out for in order to avoid the system breaking. Possible issues deriving from changes could include errors at runtime such NullPointerExceptions or UI issues such as LoadExceptions. Listed below are various things to watch out for when making any changes to the system:

9.1 Changes to variables

When making changes to the programs variables including adding, removing or renaming variables, it is advised to take great care when doing so to avoid errors. Variables being changed should have their scope checked as well as their instances to see how the variable is used in the program. It is important to fully refactor any changes to variables so that all instances of the variable are updated, with IDE's usually offering a way of doing this automatically. If changes are not fully refactored throughout the system, the program could fail to compile with other parts of the program that reference the old variable, throwing an ElementNotFoundException.

Changes to the variable's type should also be watched closely, as whilst it is possible to refactor the type automatically through an IDE, it doesn't refactor instances where the variable is utilised in parts of the code. For example, the variable could be equated to a variable of a different type, leading to a TypeMismatchException.

9.2 Changes to FXML & Controllers

Alterations to the program's FXML & GUI such as adding or removing elements should also be taken with care as it is easy to get a LoadException. This is because every FXML file is linked to a Controller class, where each @FXML element in the Controller points to a corresponding element in the FXML file. Due to this, any elements that are removed from the FXML file with a fx:id should also be removed from the Controller as this could cause NullPointerExceptions.

Similarly, whenever a element is added that has an event e.g. 'onAction= #event', a corresponding event should be created within the FXML file's controller handling the event otherwise a LoadException will be thrown as it cannot find the event it refers to.

9.3 Changes to JSON file

When altering the format of the program's JSON file care should also be taken as the program's Jackson library deserializes the files by mapping the JSON directly onto the DictionaryEntry using the objects getters and setters. As a result of this, any additional fields added to the JSON file should also have matching getters and setters in the DictionaryEntry class with matching names of the same type to avoid any errors.

In addition to these precautions, any variables added to the DictionaryEntry class will be automatically mapped onto the JSON file when saved, so it is important to mark them as ignored using @JSONIgnore if they aren't supposed to be mapped to the file, avoiding potential errors.

10. PHYSICAL LIMITATIONS OF PROGRAM

The program has some physical limitations that the user is required to meet in order to run properly. These limitations are as described below:

10.1 Screen Size

Limitations in the programs UI mean that users must have a screen size of at least 1100 X 680 in order to be able to see the program properly on their screens. This due to the UI struggling to scale below this set size, therefore the program's window is given this size as a minimum that cannot be made smaller.

10.2 Memory

The program's memory usage is dependent on the number of dictionary definitions that are loaded into the program as each definition is loaded into the program. The default dictionary.json file contains roughly 1300 definitions and uses around 200-300 MB of memory. As a result of this, it is advised to take note that the memory required will increase when more definitions are added.

11. REBUILDING & TESTING

It should be noted that these instructions are specific to our IDE, IntelliJ IDEA (v2020.1.1), and your experience may vary.

11.1 How to Rebuild the project

In order to rebuild the program it is recommended to open up the project within the IntelliJ in order to gain an overview. Once opened, find the 'Build' dropdown tab located at the top of the IDE. This section can then be used to build the program by clicking the drop down menu, selecting 'Build Project' and doing so will cause the project to be built.

11.2 How to test the project

In order to run the program's tests it is also recommended to open up the project within an IDE. After doing so, look for the list of JUnit tests which are contained within the package 'uk.ac.aber.cs221.group20.test'. This package will include the tests listed within the 'test' package in the Design Specification Document[1] and each test can be run by clicking the class and selecting 'Run *X*' where *X* is one of the four tests specified. After selecting run, the JUnit class will run with the result of each test being displayed with a green tick for a pass, yellow caution symbol if the test runs but doesn't get the expected outcome and a red cross if the test fails to run.

REFERENCES

[1] Software Engineering Group Project 20: Design Specification. H. Dugmore [hjd3], K. Bryan-Jones [kab74], L. Wybar [law39],M. Jakob [maj83],O. Pocock [osp1],T. Perry [top19],W. Watts [ncw]. DesignSpecGroup20 1.7 Release

[2] Software Engineering Group Projects: Welsh Vocabulary Tutor Requirements Specification. C. J. Price. SE.QA.CSRS 1.1 Release

[3] Software Engineering Group Projects: General Documentation Standards. C. J. Price, N. W. Hardy, B.P. Tiddeman. SE.QA.03. 1.8 Release

DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
0.1	N/A	05/05/20	Initial document writeup	TP LW KB
0.2	73,74,75, 76,77	06/05/20	Updated document to bring up to standard and fix issues detailed in CCFs	TP LW
1.0	N/A	06/05/20	Changed Document to release	TP LW OP WW