Software Engineering Group Project
Design Specification

Author:	Shane Waters [shw30]
Config Ref:	Des-Spec_GP15
Date:	1st May 2020
Version:	1.2
Status:	Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1. INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to provide a detailed description and analysis of the design for the "Welsh Vocabulary Tutor" program. This document will allow anyone without prior knowledge of how the program works to have an in-depth understanding of how all the components and modules work and interreact with one another. It is also to ensure the creation of the program will follow a design which was agreed upon by all the developers.

1.2 Scope

This document specifies all the required information any future developer may need for the program in accordance to SE.QA.05. It will outline all the classes, components, interfaces, algorithms, data structures and sequences.

1.3 Objectives

The objective of this document is to provide a resource in which future developers can refer to for an in-depth understanding of the following:

- Significant classes in the program
- Component overviews
- Interface descriptions
- Detailed explanation of significant algorithms, data structures and sequences.

This document will provide developers a formal design to follow when the program is created to ensure everything will work and the correct design is implemented.

2. DECOMPOSITION DESCRIPTION

2.1 Programs in the system

2.1.1 Desktop Application

The single program being created is called the "Welsh Vocabulary Tutor". The program will act as a language tutoring service with the premise of teaching the user Welsh words. Upon the user starting the desktop app, the program will display the user a dictionary of all the words provided in the dictionary JSON file. The user can sort this dictionary in either Welsh or English with the press of a button. The dictionary can be searched if the user types a prefix of a word into the search bar, it will then present all the words starting with the matching prefix. The search function works for both English and Welsh.

The user also has a practice list which will show any words from the dictionary the user has marked as a word they wish to learn. This practice list acts like the dictionary as it can also be sorted by English and Welsh and searched in either language. The user can remove words from their practice list at any time. Flashcards can be generated from the practice list which will prompt the user with a random Welsh word and when clicked will display the English translation.

The user can also access a tests page which will allow them to generate a test of 3 varying types. This will generate a random question of the type selected from their dictionary and prompt them to answer it. After answering a question, they will be given feedback on how they did. Finally, the user also has the option to generate a full test which is a series of tests which the user will have to answer.

2.2 Significant classes in the system

2.2.1 Desktop Application

- **BaseApplication**: The class which is launched on program start. Responsible for initialising the other significant classes and communicates between the backend and UI classes.
- **Dictionary**: Contains the data structures responsible for holding all the words loaded from the JSON file. Has all the crucial functions required for the user to interact with the dictionary.
- **Word**: Contains all the information for a word loaded from the JSON file.
- **Parser:** Serializes and deserializes the JSON file.
- **Test:** An interface which varying types of tests implement for consistency. Holds a list of questions and answers.
- **TestBuilder:** Generates tests for the user based upon the contents of the dictionary. Uses the Test interface to create varying types of tests.
- **UserInterface:** Responsible for generating the UI and handling user events.

2.3 Mapping from requirements to classes

Requirement	**Classes providing requirement**
FR1	Dictionary, BaseApplication, Parser
FR2	Dictionary
FR3	Dictionary
FR4	Dictionary, Word
FR5	Dictionary, Word
FR6	Dictionary
FR7	Dictionary
FR8	Dictionary, TestBuilder
FR9	Dictionary, TestBuilder
FR10	TestBuilder

3. DEPENDENCY DESCRIPTION

3.1 Component Diagrams

3.1.1 Component Diagram for desktop application

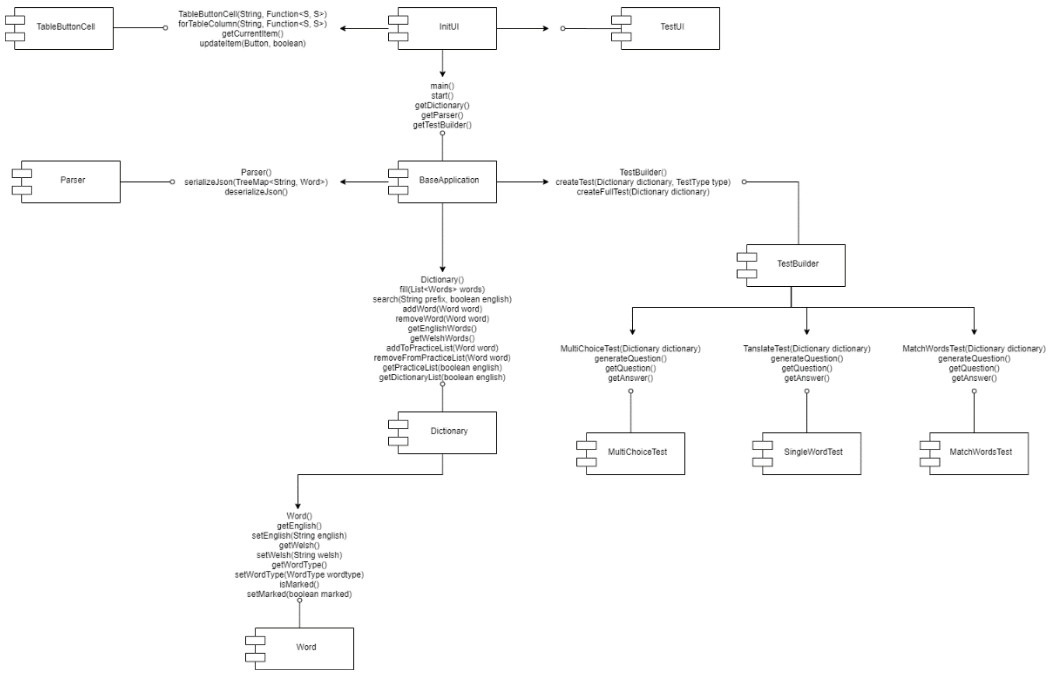
TableButtonCell
TableButtonCell(String, Function<S, S>)
forTableColumn(String, Function<S, S>)
getCurrentItem()
updateItem(Button, boolean)
InitUI
TestUI
main()
start()
getDictionary()
getParser()
getTestBuilder()
Parser
Parser()
serializeJson(TreeMap<String, Word>)
deserializeJson()
BaseApplication
TestBuilder()
createTest(Dictionary dictionary, TestType type)
createFullTest(Dictionary dictionary)
TestBuilder
Dictionary()
fill(List<Words> words)
search(String prefix, boolean english)
addWord(Word word)
removeWord(Word word)
getEnglishWords()
getWelshWords()
addToPracticeList(Word word)
removeFromPracticeList(Word word)
getPracticeList(boolean english)
getDictionaryList(boolean english)
Dictionary
MultiChoiceTest(Dictionary dictionary)
generateQuestion()
getQuestion()
getAnswer()
TanslateTest(Dictionary dictionary)
generateQuestion()
getQuestion()
getAnswer()
MatchWordsTest(Dictionary dictionary)
generateQuestion()
getQuestion()
getAnswer()
MultiChoiceTest
SingleWordTest
MatchWordsTest
Word()
getEnglish()
setEnglish(String english)
getWelsh()
setWelsh(String welsh)
getWordType()
setWordType(WordType wordtype)
isMarked()
setMarked(boolean marked)
Word

Figure 1: Component Diagram

Figure 1 shows the component diagram for our program. This diagram shows what classes can see methods contained within other classes or what methods the classes depend on from those classes. One of the first main dependencies within the program is between the test builder and the BaseApplication. This dependency simply allows the BaseApplication UI's to call the test builder, and from there, it handles creating all the tests in the program, as it has access to all the methods located in each of the test classes. We can see this through how the arrows show that the BaseApplication depends on the TestBuilder class, and the TestBuilder class depending on the four test classes to complete what it is assigned by the BaseApplication class.

Another dependency you notice from the diagram would be the dependency between the BaseApplication class, the Dictionary class and the Word class. Without the dependency between the dictionary class and the word class, the program would not be able to see the words read in from the JSON file, nor would it be able to manipulate the inputted word in any way. The dependency between the dictionary and BaseApplication class allows the UI portion of the BaseApplication class to complete its functionality. Without it, the user would be unable to simply add new words to the program, get the English or welsh version of the word, or even add the word to their practice list.

The main aspect that this image shows is how the BaseApplication depends on all aspects of the program, in order to complete the various tasks and objectives set out to us by the client. Without the listed dependencies, the program would be unable to complete the basic of tasks set to it, and ultimately the program would not work as intended.

The other main aspect of the component diagram would be the dependency between the BaseApplication class and InitUI class. The dependency between these classes is the backbone of our program, with the BaseApplication initialising the program, and the UI taking the initiation, cascading and starting the rest of the application. As you may realise, without the dependency between these classes, the program would cease to function entirely sections of the program would fail to initialise, with such sections being things like updating the dictionary table or showing the AddWordMenu form.

And finally, one final aspect of the component diagram would be the dependency between the TesUI and the InitUI classes. The dependencies between these classes are the core of the testing functionality within our program, with the InitUi containing the content for the TestUI to generate the tests for the user. The InItUI contains the content required display anything within the Test tab of the UI when TestUI is initialised.

4. INTERFACE DESCRIPTION

4.1 Desktop application Use Case

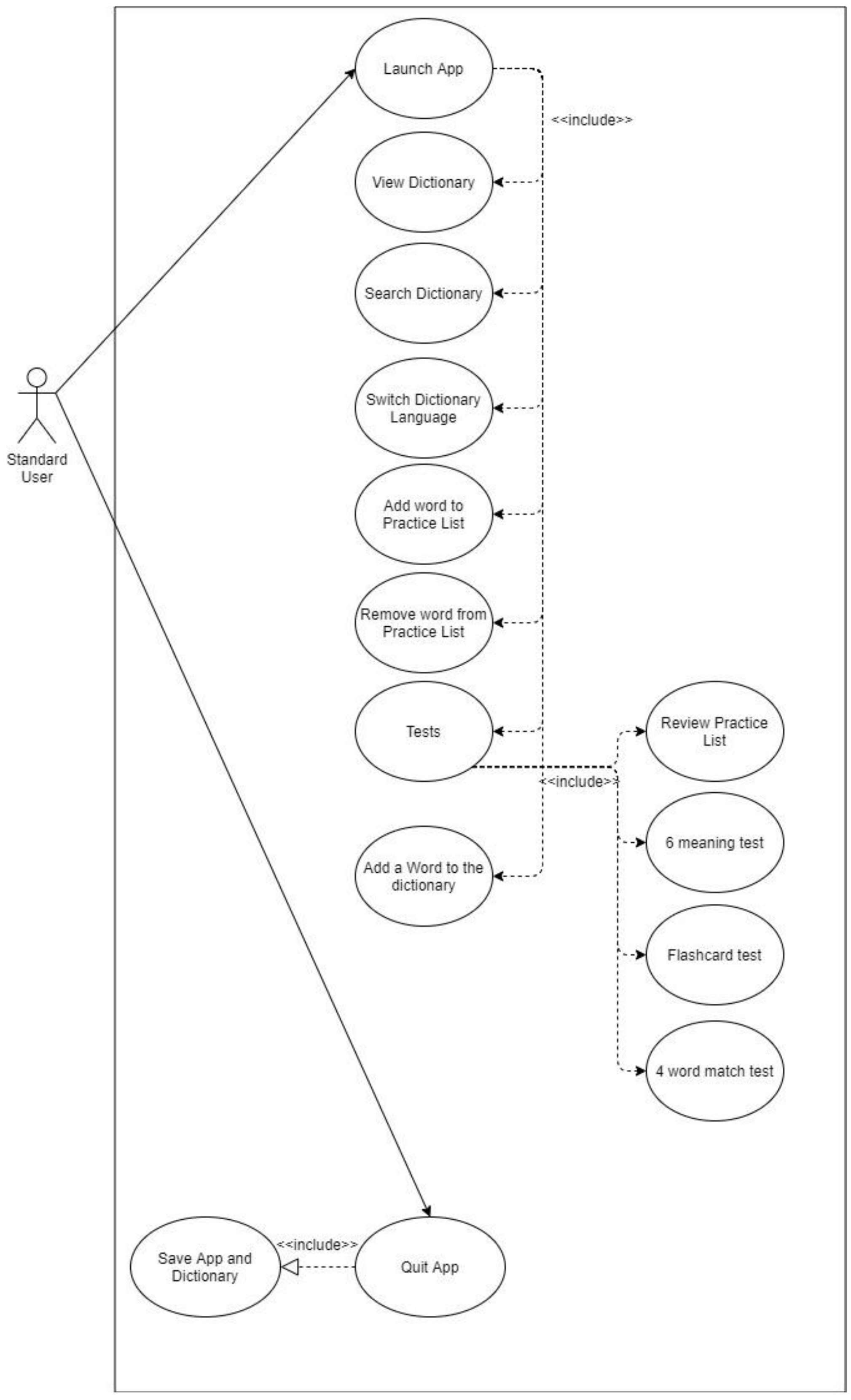

Figure 2: Desktop Use Case

Figure 2 above shows the relationship between methods which would need to be included but at a user level.

4.2 Class interface specification

The following interface specifications use UML formatting to represent public, private etc...

4.2.1 BaseApplication interface specification

This class was originally called "Application" but due to JavaFX extending from its own Application the two names conflicted so it was renamed to BaseApplication.

BaseApplication: This is a public class. BaseApplication is the main class of the program, is houses the main() method and in initialises other major classes within the program. Shown in figure 4, BaseApplication directly accesses 3 other classes:

- Dictionary(1..1): BaseApplication needs a Dictionary on creation so it can store the words loaded from the JSON file. As BaseApplication communicates with the UI, holding the dictionary object here means the whole program can see the dictionary object if it's passed as a parameter.
- Parser(1..1): BaseApplication needs a Parser so it can serialize and deserialize the JSON file which holds all the words needed for the dictionary.
- TestBuilder(1..1): BaseApplication needs a TestBuilder so the UI can generate tests without having any backend code in the UI classes. BaseApplication acts as a bridge between the UI and backend.

Instance Variables:

- -Dictionary: Dictionary: A dictionary which holds all the words loaded from the JSON file and has all the functions to interact with the words.
- -TestBuilder: TestBuilder: A test builder which allows the UI to create tests without any overhead.
- -Parser: Parser: Serializes and deserializes the JSON file so the dictionary can be filled and saved.

Methods:

- <u>+Main(String[] args)</u>: void: The main method of the program. Launches application for the JavaFX class which starts the UI.
- +Start(): void: To keep the main method small all start-up methods are run here.
- +getDictionary(): Dictionary: Returns the dictionary so it is visible to the UI.
- +getParser(): Parser: Returns the parser so it is visible to the UI.
- +getTestBuilder(): TestBuilder: Returns the test builder so it is visible to the UI.

4.2.2 Dictionary interface specification

Dictionary: This is a public class. Dictionary is responsible for holding all the words within the program and performing any functionality on the words. Shown in figure 4, Dictionary directly accesses 2 other classes:

- BaseApplication(1..1): A dictionary cannot exist without an application to use it.
- Word(0..*): A dictionary will store 0 or many words within its data structures.

Instance Variables:

- -EnglishWords: TreeMap<String, Word>: A sorted tree which holds all the words ordered in English.
- -WelshWords: TreeMap<String, Word>: A sorted tree which holds all the words ordered in Welsh.

Methods:

- +Dictionary(): The default constructor for the class.
- +Fill(List<Word> words): void: Fills the dictionary with all the words deserialized by the parser which was passed as a list.
- +Search(String prefix, Boolean English, Boolean practiceList): Map<String, Word>: Uses the FindPrefix method to search the dictionary, the English Boolean determines whether to search the English or welsh list (what TreeMap to pass to the FindPrefix). The practiceList Boolean determines whether to search the full dictionary or the users practice list.
- +AddWord(Word word): void: Adds a word to the dictionary.
- +RemoveWord(Word word): void: Removes a word from the dictionary.
- +GetEnglishWords(): TreeMap<String, Word>: Returns all the words ordered in English as a TreeMap.

- +GetWelshWords(): TreeMap<String, Word>: Returns all the words ordered in Welsh as a TreeMap.
- +addToPracticelist(Word word): void: Marks a word which means the word is part of the users practice list.
- +removeFromPracticeList(Word word): void: Unmarks a word which means the word is removed from the users practice list.
- +getPracticeList(): TreeMap<String, Word>: Returns a TreeMap of all the words that's in the users practice list.
- +getDictionaryList(): TreeMap<String, Word>: Returns a TreeMap of all the words that's in the dictionary.
- -FindPrefix(SortedMap<String, Word> baseMap, String prefix): SortedMap<String, Word>: Searches the dictionary (baseMap) for all the words matching the prefix. Returns a sorted map containing the matching words.

4.2.3 Word interface specification

Word: This is a public class. Word contains all the data that each word has within the JSON file. Shown in figure 4, Word directly accesses 1 other class and 1 enumeration:

- Dictionary(1..1): A word cannot exist without a dictionary to hold it.
- WordType<<enumeration>>: There is only a limited amount of wordtypes so an ENUM stores these unique types.

Instance Variables:

- -English: String: Holds the English version of the word.
- -Welsh: String: Holds the Welsh version of the word.
- -WordType: WordType: Holds the type of word the word is.
- -Marked: Boolean: Represents whether the word is part of the users practice list.

Methods:

- +Word(String English, String Welsh, String WordType): The constructor for the class, its parameters set the instance variables as they are filled when the JSON is loaded automatically (by GSON).
- +getEnglish(): String: Returns the English version of the word.
- +setEnglish(String English): void: Sets the English version of the word.
- +getWelsh(): String: Returns the Welsh version of the word.
- +setWelsh(String Welsh): void: Sets the Welsh version of the word.
- +getWordType(): String: Returns the wordtype of the word.
- +setWordType(WordType wordtype): void: Sets the wordtype of the word.
- +isMarked(Boolean): void: Returns if the word is in the users practice list.
- +setMarked(Boolean): void: Sets the word as marked meaning it will be part of the users practice list.

4.2.4 Parser interface specification

Parser: This is a public class. Parser is used to serialize and deserialize the JSON file. Shown in figure 4, Parser directly accesses 1 other class:

- BaseApplication(1..1): A parser cannot exist without a dictionary to take its return value.

Methods:

- +Parser(): Default constructor for the class.
- +SerializeJson(TreeMap<String, Word> words_map): void: Saves the TreeMap within the dictionary into the JSON file.
- +DeserializeJson(): List<Word>: Returns a list of words which contains Word objects for each word within the JSON file, the dictionary will turn the list into a TreeMap.

4.2.5 TestBuilder interface specification

TestBuilder: This is a public class. TestBuilder creates tests for the user to do so there is less overhead in the UI for picking our questions from the dictionary. Shown in figure 4, TestBuilder directly accesses 2 classes and 1 enumeration:

- BaseApplication(1..1): A TestBuilder cannot exist without a dictionary to build tests from.

- Test(0..*): A TestBuilder may make 0 or many tests.
- TestType<<enumeration>>: There is only a limited amount of test variants so an ENUM stores their unique types.

Methods:
- +TestBuilder(): Default constructor for the class.
- +CreateTest(Dictionary dictionary, TestType type): Test: Creates a test object which will be of the test type specified. Returns the test so the UI can easily use it.
- +CreateFullTest(Dictionary dictionary, TestType type): List<Test> tests: creates a list of test objects of random test types. Returns a list of tests so the UI can easily use one after the other to represent a full test.

4.2.6 Test interface specification

Test<<interface>>: This is a public interface. There is a new class for each test which implements this interface as all tests follow the same exact structure but will require polymorphism for their methods. Inheritance could have also been used but without the need for any class to call its super() method an interface would be more appropriate. Shown in figure 4, Test directly accesses 1 other class and 4 other classes implement it:
- TestBuilder(1..1): A Test cannot exist without being used by the test builder.
- MultiChoiceTest, Translate Test, MatchWordsTest, FlashcardTest<<implements>>: All these test types follow the same structure but require polymorphism for they implement Test.

Instance Variables:
- -Question: List<String>: A list containing all the questions for the test, may be just one word or many.
- -Answer: List<String>: A list containing all the answers for the test, may be just one answer or many.

Methods:
- +GenerateQuestion(): void: Sets the instance variables of the test, uses getRandomWord().
- +GetQuestion(): List<String> question: Returns the questions on the test.
- +GetAnswer(): List<String> answer: Returns the answers on the test.
- +GetRandomWord(TreeMap<String, Word> words): Word: Using the dictionary, a random word will be selected and returned.

4.2.7 InitUI interface specification

InitUI: This is a public class. It extends from the JavaFX application class. This class contains all the UI which needs to be initialised on startup. Shown in figure 4, InitUI directly accesses 3 classes:
- BaseApplication(1..1): InitUI is launched from BaseApplications main method.
- TestUI(1..1): InitUI uses TestUI to load all the test UI, it is separated for readability and to separate the logic of the dictionary from the tests.
- TableButtonCell(0..*): InitUI needs TableButtonCell to create custom cells within the tables which display the dictionary. Depending on the amount of entries determines the number of objects needed.

Instance Variables:
- -app: BaseApplication: Reference to the base application.
- -dictionarySortByEnglish: Boolean: Remembering whether the dictionary was sorted by English or Welsh so when the user changes tab it does not reset. Also needs to be used by multiple methods.
- -practiceSortByEnglish: Boolean: Remembering whether the practice list was sorted by English or Welsh so when the user changes tab it does not reset. Also needs to be used by multiple methods.
- -dictionaryTable: TableView<Word>: Holds the dictionary table data so it is visible to everything in the class.
- -practiceListTable: TableView<Word>: Holds the practice list table data so it is visible to everything in the class.
- -prefix: String: Stores the prefix in the search bar as is needs to be visible to whole class and make sure it is remembered upon changing tabs.
- -screen: Rectangle2D: Holds the users screen information (resolution).
- -stage: Stage: Reference to the window which the program is ran in.
- -maxWordLength: short: The maximum number of characters you can have for English or Welsh.

Methods:

- +Start(Stage): void: Starts the JavaFX application, initialises elements.
- -dictionaryTabElements(): VBox: Sets up elements for dictionary tab.
- -practiceTabElements(): VBox: Sets up elements for the practice tab.
- -initDictionaryMenuBar(): HBox: Initialises search bar, language sorting and add word button.
- -initPracticeListMenuBar(): HBox: Initialises search bar, language sorting and create flashcard button.
- +showAddWordMenu(): void: Displays the add word menu.
- +showFlashcard(): void: Displays the flashcard menu.
- +updateTable(String prefix, Boolean English, Boolean practice): void: Displays the updated table with new data.
- -checkNewWord(String English, String welsh, String wordType): Boolean: Checks to see if the new word is valid.
- +showErrorMessage(String messageText): void: Displays a window with an error message.

4.2.8 TableButtonCell interface specficiation

TableButtonCell: This is a public class. It extends from TabelCell class. This class contains overloading methods so a table cell can have button functionality. Shown in figure 4, TableButtonCell accesses 1 class:

- InitUI(1..1): TableButtonCell is used in the tables within InitUI to display the “add word” and “remove word” button within each table cell.

Instance Variables:

- actionButton: Button: The button to be added to the cell.

Methods:

- TableButtonCell(String label, Function<S, S> function): Constructor for a table cell.
- forTableColumn(String label, Function<S, S> function): TableCell<S, Button>: Overloads the callback of a table cell with a button call which uses the function passed in.
- getCurrentItem(): S: Gets the data of the row the button is in.
- updateItem(Button item, Boolean empty): Allow update of button graphics.

4.2.9 TestUI interface specification

TestUI: This is a public class. It is responsible for displaying the test UI and handling full tests. Show in figure 4, TestUI accesses 1 class:

- InitUI(1..1): TestUI cannot exist without the InitUI.

Instance Variables:

- -app: BaseApplication: Reference to the base application.
- -stage: Stage: Reference to the window which the program is ran in.
- -correctTally: float: Tracks the users score during a full test.
- -numQuestions: AtomicInteger: The amount of questions to be displayed inside a full test.
- -currentQuestionNumber: Boolean: The current question number the user is on.
- -testScreenVBox: VBox: Reference the test screen.
- -currentTest: VBox[]: Reference to the current test window which is to be displayed.
- -tests: List<Test>: A list of all the randomly generated tests ready to be displayed.
- -mainWindowTabPane: TabPane: Reference to the tabs so they can be modified.

Methods:

- -multipleChoiceTabElements(Tests test): VBox: Creates the UI for the multiple choice test.
- -genMultiChoice(Test test, Button button, Button... Answers): Boolean: Uses the test given to display it as a multichoice test.
- -checkMultiAnswers(Button question, Button chosenAnswer, Test test, Button... answer): boolean: Checks if the user chose the correct answer for multi choice tests.

- -matchWordsTabElements(Test test): VBox: Creates the UI for the match words test.
- -genMatchWords(Test test, List<String> userAnswers, Button checkAnswers, List<Button> questionButtons, List<Button> answerButtons, List<Button> buttonsSubmitted): boolean: Uses the test given to check the answers given.
- -setUpMatchWords(List<Button> questionButtons, List<Button> answerButtons, Test test): void: Displays the test on the question and answer buttons.
- -checkMatchAnswers(List<String> userAnswers, Test test, List<Button> buttonsSubmitted): int: Checks the users answers given for a match words test and returns the amount of correct answers.
- -translateTabElements(Test test): VBox: Creates the UI for the translate test.
- -genTranslate(Test test, Button question, TextField answer, Button checkAnswer, Label result): boolean: Uses the test given to display it as a translate test.
- -checkTranslateAnswer(TextField answer, Test test): boolean: Checks the users answer if it matches and returns if it was correct or not.
- -snapButton(Button socket, AtomicInteger lastSelected, List<String> answers, List<Button> answerButtons, List<Button> buttonsSubmitted): Handles the match word functionality of words snapping onto the empty sockets and adds it to the list of answers.
- -mainTabElements(): VBox: Initialises the UI for the test screen.
- -runFullTest(int questionCount): void: Sets the program into full test mode and displays all the corresponding UI.
- -nextQuestion(): void: Shows the next question within the full test or display the result screen if no more questions are left.
- -displayScore(): void: Displays the users score for a full test.
- -showNextQuestionButton(): void: Shows the next question button when a user has finished a question and initialises the score readout UI.
- -fullTestMenuScreen(): VBox: Shows the menu which allows the user to select how many questions they want in the full test.

5. DETAILED DESIGN

5.1 Sequence Diagrams

The following sections of the program are too complicated to be understood from just observation so by using decomposition each complex module can be broken down into its lowest level of classes and calls.

5.1.1 Desktop application start

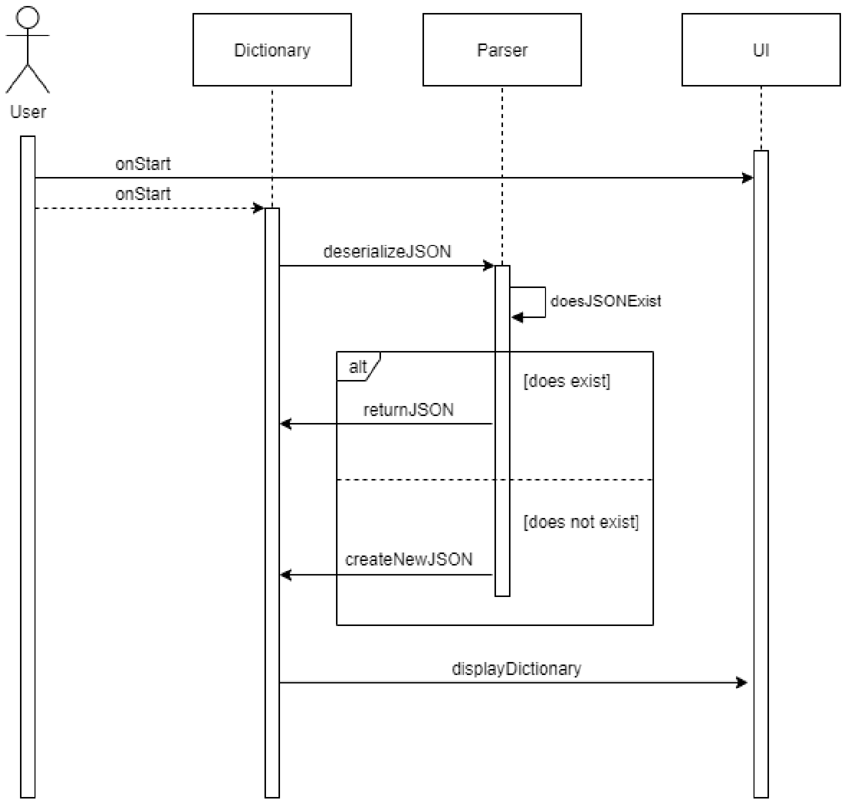

Figure 3.1

Upon launching the application, the program must prompt the user with the UI and start to load the dictionary. Figure 3.1 shows on start a Dictionary object is created which instantly calls the parser to deserialize the JSON file. From here the parser must determine if there is a file to parse, if not then it must create one instead of returning null. Once the JSON has been parsed and stored within the Dictionary the UI can then display its contents.

5.1.2 Language switch

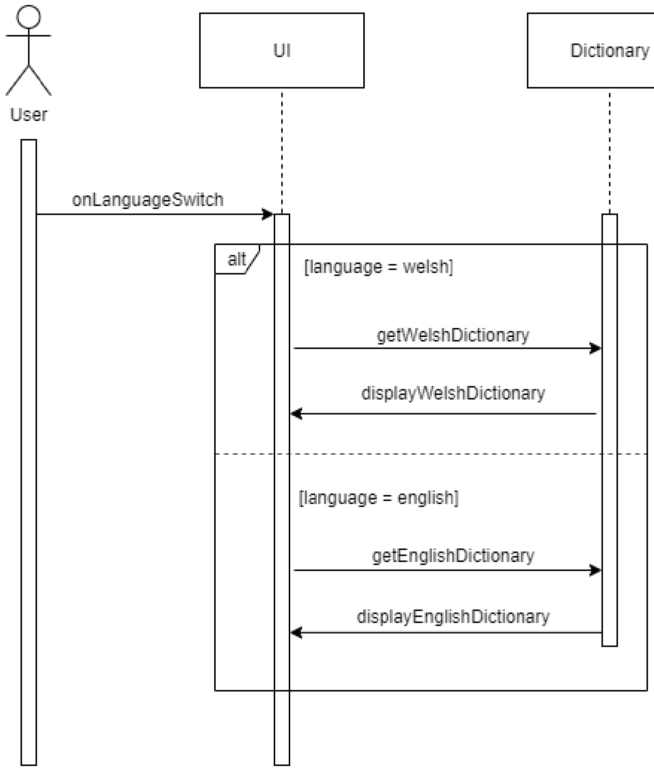

Figure 3.2

Upon the user switching language by pressing the button on the UI, the dictionary must swap out the current language TreeMap for the other languages TreeMap. It does this by calling the Dictionary to return the other TreeMap. This is done because to maintain a fast search speed there is a TreeMap ordered in each language.

5.1.3 Adding a new word

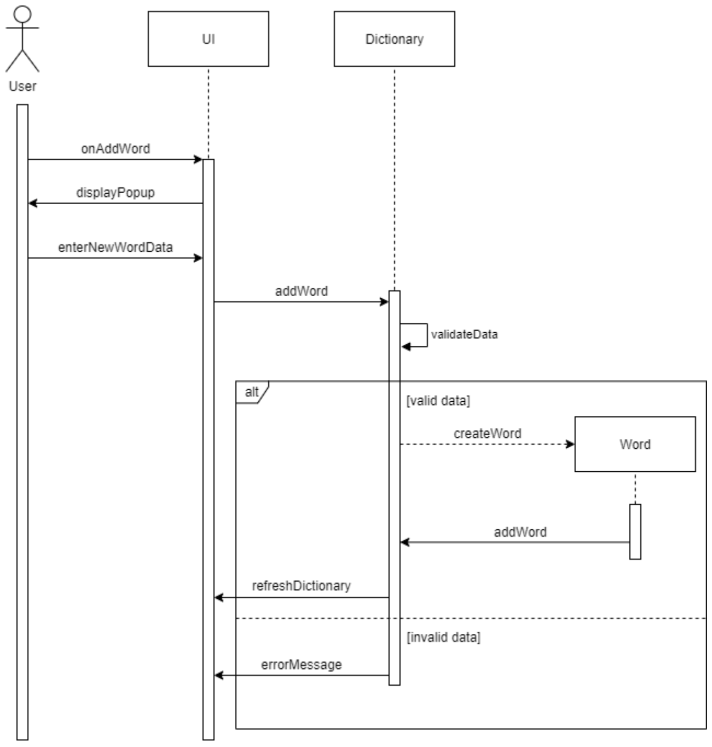

Figure 3.3

Upon the user pressing the “add word” button on the UI the UI will display a popup window for the user to enter their words new data. Once entered, the UI sends this data to the Dictionary to add it as a new word. Before the word is added their data will be validated to see if all the fields are valid and everything is appropriate. If the data input is valid then a new Word is created and added into the Dictionaries data structures. The dictionary will then need to call the UI to refresh so the new word appears. If the data is invalid the Dictionary will tell the UI it failed to add and why.

5.1.4 Starting a new test

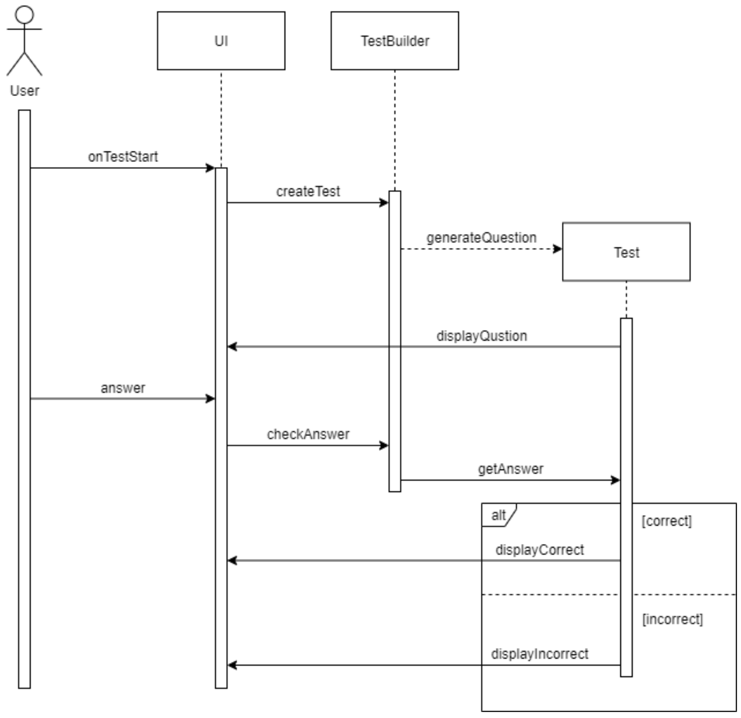

Figure 3.4

Upon the user selecting a new test to start on the UI, it will call the TestBuilder which handles creating random tests and organising all the outputs it would need so there is no overhead in the UI. Once a test has been generated it will be displayed on the UI and await the user’s response. Once the user’s response has been received, the answer will be checked, and the corresponding feedback will be displayed whether it was correct or not.

5.3 State charts

Some complex modules within the program are better explained using a state chart rather than a sequence diagram. State charts also provide even more details on what's happening for modules which already have a sequence diagram.

5.3.1 Searching the dictionary

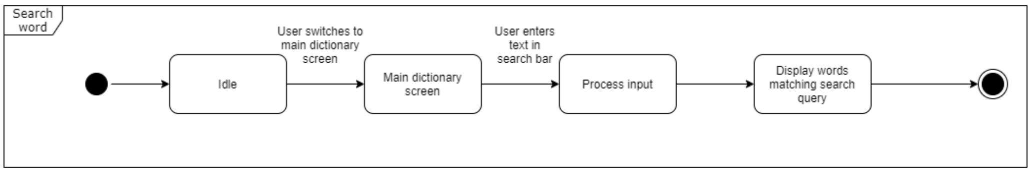
Search word
Idle
User switches to main dictionary screen
Main dictionary screen
User enters text in search bar
Process input
Display words matching search query

Figure 3.5

The dictionary remains idle until the user enters something into its search bar. When an input is entered it is sent to the Dictionary and the matching words will be returned. This is repeated every time a character is inputted into the search bar.

5.3.2 Adding a new word

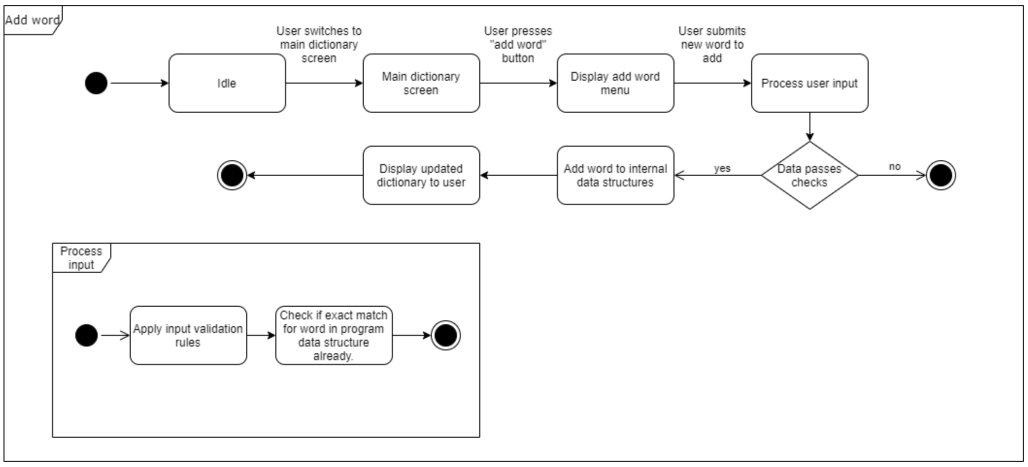
Add word
Idle
User switches to main dictionary screen
Main dictionary screen
User presses "add word" button
Display add word menu
User submits new word to add
Process user input
Data passes checks
yes
no
Add word to internal data structures
Display updated dictionary to user
Process input
Apply input validation rules
Check if exact match for word in program data structure already.

Figure 3.6

5.3.3 Creating and answer a test

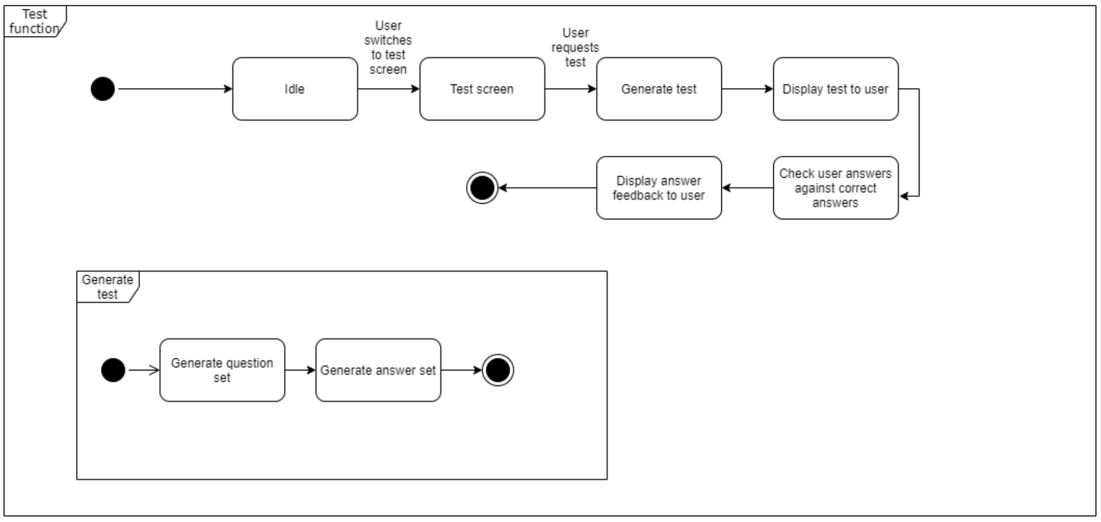

Figure 3.7

5.4 Significant algorithms

5.4.1 Storing words

During the analysis of the software, it was determined that after parsing the JSON file, the objects created must be stored within the program. This would be the biggest decision to decide how the words are going to be stored as the whole program revolves around the dictionary. The datatype used to store these objects must meet the following requirements:

- Store as many objects the user wants to store
- Support partial searching with a prefix
- Any result returned from a search must be within 1 second of the initial search (This will have to be much quicker as there must be time remaining for the UI to update)
- Support for adding and removing objects
- Must be compatible with GSON so it can be exported as a JSON file

5.4.2 Candidate abstract data types

In accordance with the requirements, there are 2 suitable data types:

- Sorted Array List
- Tree Map

5.4.3 Sorted Array List Analysis

One possible solution is to use a sorted Array List. As it's a list, it can be expanded when new objects are appended so it can store as many objects as required. The space complexity would be O(n) where n is the number of objects within the list.

If the Array List is sorted, a partial search returning all the words matching a prefix is possible. Binary search would be used to find the first word with the required prefix and then a simple iteration is used until a new prefix is found. The time complexity for this operation is O(logn) where n is the number of objects within the list.

Adding objects to an Array List would require it to be sorted each time otherwise searching would be O(n) which would not be suitable for meeting the time efficiency requirement. The time complexity for adding an object would be o(nlogn) where n is the number of objects within the list. Removing objects from the Array List would require the use of binary search to find the object. The time complexity for removing an object would be o(logn) where n is the number of objects within the list.

Array Lists are fully supported by GSON, they are the preferred way of loading and saving JSON as its already in a suitable format for writing.

5.4.4 Tree Map Analysis

Another possible solution is to use a Tree Map. As it's a tree structure, new objects can be appended to empty nodes so it can store as many objects as required. The space complexity would be O(n) where n is the number of nodes within the list.

As Tree Maps are already sorted, searching can be implemented with the use of the submap method as it will return a submap of the words starting with the prefix. Tree Maps use the Red-Black tree implementation which means any operations performed on the tree require rotations to keep it balanced. Submap works by taking the prefix provided and mapping it to the start of a 'branch' on the tree. As the start and end are provided, this 'branch' can be retrieved from the list, however, the submap will need to be rotated when being built. The time complexity for this operation is O(logn) where n is the number of nodes within the tree.

Adding objects to the Tree Map follow the Red-Black tree implementation so rotations are required to keep the tree balanced after adding new nodes. The same applies to removing nodes, the tree must be rebalanced each time. The time complexity for this operation is O(logn) where n is the number of nodes within the tree.

Regarding Tree Maps implementation of Red-Black tree, it would be more suitable to use an AVL implementation as it's better suited for searching where Red-Black trees are better at adding and removing objects. Java defaults to using the Red-Black tree as Oracle always choose implementations which are more general, considering the scope of the requirements, it can be assumed using Red-Black trees will have negligible effects on the program.

Saving a Tree Map using GSON will require the Tree Map to be put into a list first as it's easier to read and write.

5.4.5 Conclusion

To conclude, both solutions would meet the requirements. The advantage Tree Maps have over sorted Array Lists are:

- Tree Maps add objects faster than Array Lists. This is because adding to a tree map on average, requires some rotations whereas Array Lists must be fully sorted each time.
- Tree Maps are faster to search than Array Lists. Although they share the same search time, building a submap of a tree and performing some rotations is faster than using binary search and iterating to fill a list constrained to the scope of a small dictionary.
- Tree Maps are easier to implement partial searching as the method to do so already exists.

It's clear that Tree Maps are the most suitable solution. Although both can be used due to them sharing similar time complexities and having the same space complexity, Tree Maps are superior in more areas and therefore will be used.

5.5 Significant data structures

5.5.1 Desktop class Diagram

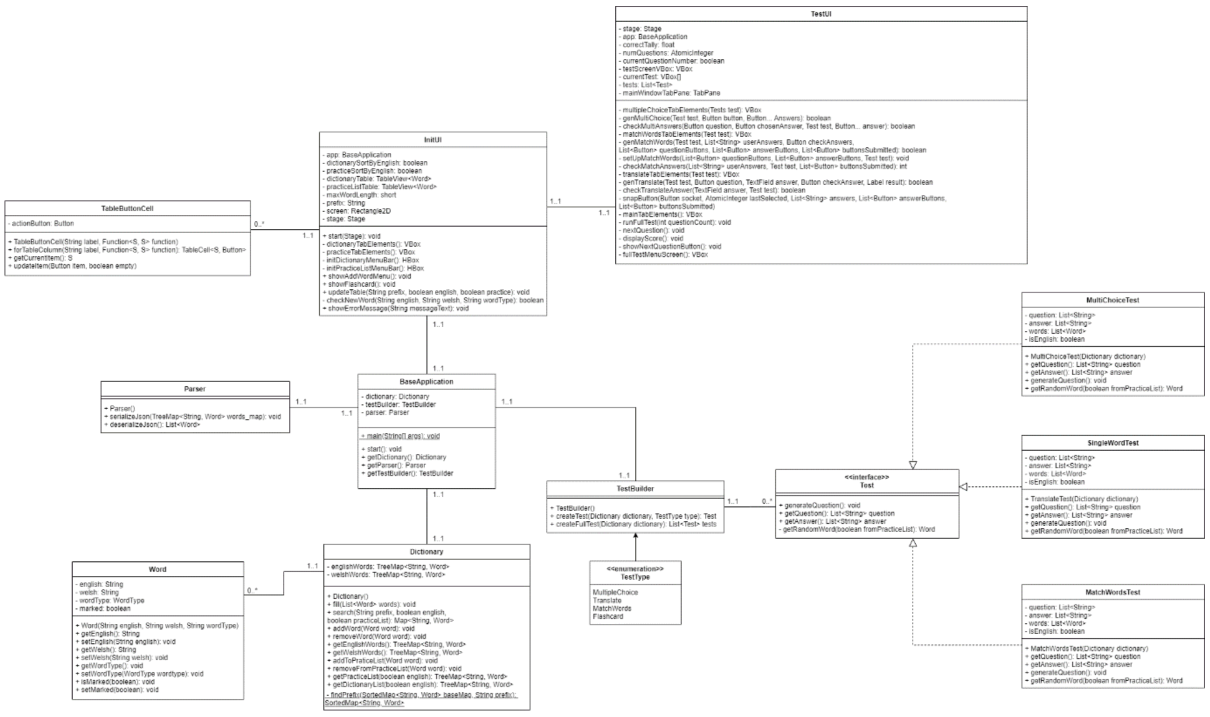
TestUI
- stage: Stage
- app: BaseApplication
- correctTally: float
- numQuestions: AtomicInteger
- currentQuestionNumber: boolean
- testScreenVBox: VBox
- currentTest: VBox[]
- tests: List<Test>
- mainWindowTabPane: TabPane
- multipleChoiceTabElements(Tests test): VBox
- genMultiChoice(Test test, Button button, Button... Answers): boolean
- checkMultiAnswers(Button question, Button chosenAnswer, Test test, Button... answer): boolean
- matchWordsTabElements(Test test): VBox
- genMatchWords(Test test, List<String> userAnswers, Button checkAnswers, List<Button> questionButtons, List<Button> answerButtons, List<Button> buttonsSubmitted): boolean
- setUpMatchWords(List<Button> questionButtons, List<Button> answerButtons, Test test): void
- checkMatchAnswers(List<String> userAnswers, Test test, List<Button> buttonsSubmitted): int
- translateTabElements(Test test): VBox
- genTranslate(Test test, Button question, TextField answer, Button checkAnswer, Label result): boolean
- checkTranslateAnswer(TextField answer, Test test): boolean
- snapButton(Button socket, AtomicInteger lastSelected, List<String> answers, List<Button> answerButtons, List<Button> buttonsSubmitted)
- mainTabElements(): VBox
- runFullTest(int questionCount): void
- nextQuestion(): void
- displayScore(): void
- showNextQuestionButton(): void
- fullTestMenuScreen(): VBox
InitUI
- app: BaseApplication
- dictionarySortByEnglish: boolean
- practiceSortByEnglish: boolean
- dictionaryTable: TableView<Word>
- practiceListTable: TableView<Word>
- maxWordLength: short
- prefix: String
- screen: Rectangle2D
- stage: Stage
+ start(Stage): void
- dictionaryTabElements(): VBox
- practiceTabElements(): VBox
- initDictionaryMenuBar(): HBox
- initPracticeListMenuBar(): HBox
+ showAddWordMenu(): void
+ showFlashcard(): void
+ updateTable(String prefix, boolean english, boolean practice): void
- checkNewWord(String english, String welsh, String wordType): boolean
+ showErrorMessage(String massageText): void
TableButtonCell
- actionButton: Button
+ TableButtonCell(String label, Function<S, S> function)
+ forTableColumn(String label, Function<S, S> function): TableCell<S, Button>
+ getCurrentItem(): S
+ updateItem(Button item, boolean empty)
Parser
+ Parser()
+ serializeJson(TreeMap<String, Word> words_map): void
+ deserializeJson(): List<Word>
BaseApplication
- dictionary: Dictionary
- testBuilder: TestBuilder
- parser: Parser
+ main(String[] args): void
+ start(): void
+ getDictionary(): Dictionary
+ getParser(): Parser
+ getTestBuilder(): TestBuilder
TestBuilder
+ TestBuilder()
+ createTest(Dictionary dictionary, TestType type): Test
+ createFullTest(Dictionary dictionary): List<Test> tests
<<enumeration>>
TestType
MultipleChoice
Translate
MatchWords
Flashcard
Dictionary
- englishWords: TreeMap<String, Word>
- welshWords: TreeMap<String, Word>
+ Dictionary()
+ fill(List<Word> words): void
+ search(String prefix, boolean english, boolean practiceList): Map<String, Word>
+ addWord(Word word): void
+ removeWord(Word word): void
+ getEnglishWords(): TreeMap<String, Word>
+ getWelshWords(): TreeMap<String, Word>
+ addToPraticeList(Word word): void
+ removeFromPracticeList(Word word): void
+ getPracticeList(boolean english): TreeMap<String, Word>
+ getDictionaryList(boolean english): TreeMap<String, Word>
- findPrefix(SortedMap<String, Word> baseMap, String prefix): SortedMap<String, Word>
Word
- english: String
- welsh: String
- wordType: WordType
- marked: boolean
+ Word(String english, String welsh, String wordType)
+ getEnglish(): String
+ setEnglish(String english): void
+ getWelsh(): String
+ setWelsh(String welsh): void
+ getWordType(): void
+ setWordType(WordType wordtype): void
+ isMarked(boolean): void
+ setMarked(boolean): void
<<interface>>
Test
+ generateQuestion(): void
+ getQuestion(): List<String> question
+ getAnswer(): List<String> answer
- getRandomWord(boolean fromPracticeList): Word
MultiChoiceTest
- question: List<String>
- answer: List<String>
- words: List<Word>
- isEnglish: boolean
+ MultiChoiceTest(Dictionary dictionary)
+ getQuestion(): List<String> question
+ getAnswer(): List<String> answer
+ generateQuestion(): void
+ getRandomWord(boolean fromPracticeList): Word
SingleWordTest
- question: List<String>
- answer: List<String>
- words: List<Word>
- isEnglish: boolean
+ TranslateTest(Dictionary dictionary)
+ getQuestion(): List<String> question
+ getAnswer(): List<String> answer
+ generateQuestion(): void
+ getRandomWord(boolean fromPracticeList): Word
MatchWordsTest
- question: List<String>
- answer: List<String>
- words: List<Word>
- isEnglish: boolean
+ MatchWordsTest(Dictionary dictionary)
+ getQuestion(): List<String> question
+ getAnswer(): List<String> answer
+ generateQuestion(): void
+ getRandomWord(boolean fromPracticeList): Word
1..1
0..*

Figure 4: Desktop Class Diagram

5.5.2 Desktop object diagram

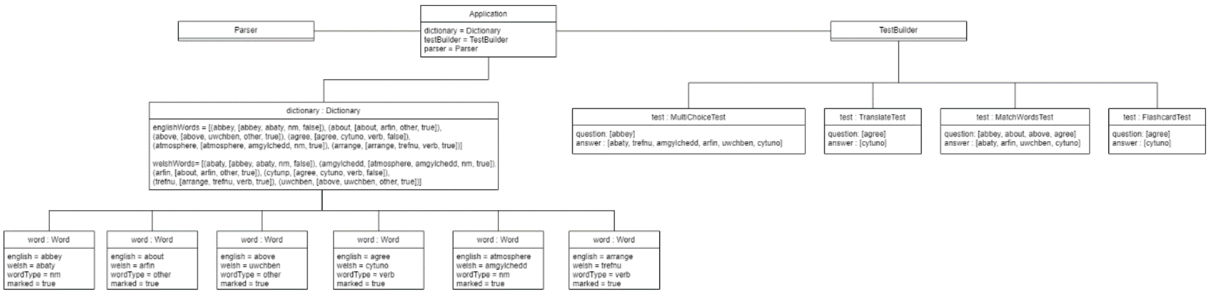
Parser
Application
dictionary = Dictionary
testBuilder = TestBuilder
parser = Parser
TestBuilder
dictionary : Dictionary
englishWords = [(abbey, [abbey, abaty, nm, false]), (about, [about, arfin, other, true]), (above, [above, uwchben, other, true]), (agree, [agree, cytuno, verb, false]), (atmosphere, [atmosphere, amgylchedd, nm, true]), (arrange, [arrange, trefnu, verb, true])]
welshWords= [(abaty, [abbey, abaty, nm, false]), (amgylchedd, [atmosphere, amgylchedd, nm, true]), (arfin, [about, arfin, other, true]), (cytunp, [agree, cytuno, verb, false]), (trefnu, [arrange, trefnu, verb, true]), (uwchben, [above, uwchben, other, true])]
test : MultiChoiceTest
question: [abbey]
answer : [abaty, trefnu, amgylchedd, arfin, uwchben, cytuno]
test : TranslateTest
question: [agree]
answer : [cytuno]
test : MatchWordsTest
question: [abbey, about, above, agree]
answer : [abaty, arfin, uwchben, cytuno]
test : FlashcardTest
question: [agree]
answer : [cytuno]
word : Word
english = abbey
welsh = abaty
wordType = nm
marked = true
word : Word
english = about
welsh = arfin
wordType = other
marked = true
word : Word
english = above
welsh = uwchben
wordType = other
marked = true
word : Word
english = agree
welsh = cytuno
wordType = verb
marked = true
word : Word
english = atmosphere
welsh = amgylchedd
wordType = nm
marked = true
word : Word
english = arrange
welsh = trefnu
wordType = verb
marked = true

Figure 5: Desktop Object Diagram

5.5.3 Additional word fields

The word class within the program will include an extra field called marked. This will determine whether a word is part of the users practice list or not. This will be useful for when the program wishes to save and load the users practice list.

5.5.4 Loading and saving

The program will save and load its data using a JSON file. It will be one list of many objects in the file with each object having 4 fields. To parse this format into objects within the program the assistance of a parser will be required. The parser we will use will be GSON, this is because:

- Does everything which is required.
- Creates the objects automatically and appends them to a list.
- Serializes the JSON in the exact format is was deserialized.
- Fast enough for our program to save on closing instantly.
- Very easy to use for inexperience JSON users.

It will be included in the lib folder within the desktop program.

5.5.5 Reducing User Interface overhead

Ensuring the UI has as little overhead as possible the issue with how tests will be provided to the UI will be an issue. The dictionary is responsible for holding and searching words but also including the ability to hold even more test structures would make this one object enormous. A separate object called the TestBuilder will be created to separate out the code for all the test creation.

5.5.6 Test Interface

Tests can be implemented in two ways:

- Interface
- Inheritance

Both methods offer polymorphism which is the only reason why they need to be used. An interface will allow more tests to be created by extending the base test and altering the question generation to suit the kind of question being made. Inheritance will allow more tests to be created by inheriting the super class test and overriding the methods which require changing. An interface will be used as there is no need for the use of super() which inheritance provides and overriding the methods is default with an interface as well.

REFERENCES

[1] Software Engineering Group Projects: Quality Assurance Plan. C. J. Price. SE.QA.01. 2.2 Release.

[2] Software Engineering Group Projects: General Documentation Standards. C. J. Price. SE.QA.02. 2.3 Release.

[3] Software Engineering Group Projects: Design Specification Standards. C. J. Price. SE.QA.05. 2.2 Release.

[4] Software Engineering Group Projects: Java Coding Standards. C. J. Price, A. McManus. SE.QA.09 2.0 Release.

DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
0.1	N/A	15/03/20	Initial creation of document.	SHW30
0.2	N/A	22/03/20	Inclusion of the class diagram, component diagram, object diagram and sequence diagrams.	SHW30
0.3	N/A	23/03/20	Section 3 finished & finalised.	SEC26
0.4	N/A	24/03/20	Section 2 finished & finalised.	SHW30
0.5	N/A	25/03/20	Inclusion of Use Case diagram.	MAC127
0.6	N/A	26/02/20	Section 4 finished & finalised.	SHW30
0.7	N/A	27/02/20	Inclusion of state charts.	IEB7
0.8	N/A	29/02/20	Section 5 finished & finalised.	SHW30
1.0	N/A	30/02/20	Finalisation of document.	SHW30
1.1	#17, #18	20/04/20	Updated dependencies and included UI modules.	SHW30, SEC26
1.2	#18	01/05/20	Finalised diagrams to include UI modules, updated specifications to include UI modules.	SHW30