WordUp - Developer Guide

1. Setting up . . . 2
    1.1. Prerequisites . . . 2
    1.2. Setting up the project in your computer . . . 2
    1.3. Verifying the setup . . . 2
    1.4. Configurations to do before writing code . . . 2
2. Design . . . 3
    2.1. Architecture . . . 4
    2.2. UI component . . . 5
    2.3. Logic component . . . 5
    2.4. Model component . . . 6
    2.5. Storage component . . . 7
    2.6. Common classes . . . 7
3. Implementation . . . 7
    3.1. Word Search feature . . . 7
    3.2. Usage Statistics feature . . . 8
    3.3. Word Synonyms feature . . . 12
    3.4. Quiz feature . . . 12
    3.5. Logging . . . 14
    3.6. Configuration . . . 15
4. Documentation . . . 15
5. Testing . . . 15
6. Dev Ops . . . 15
Appendix A: Product Scope . . . 15
Appendix B: User Stories . . . 16
Appendix C: Use Cases . . . 17
    C.1. Use case: Finding the meaning of a word . . . 18
    C.2. Use case: Making quizzes . . . 18
    C.3. Use case: Checking recently added words . . . 18
    C.4. Use case: Scheduling reminders . . . 19
Appendix D: Non Functional Requirements . . . 19
Appendix E: Glossary . . . 19
Appendix F: Instructions for Manual Testing . . . 20
    F.1. Launch and Shutdown . . . 20
    F.2. Deleting a person . . . 20
    F.3. Saving data . . . 20

By: `AY1920S1-CS2113T-T14-3`   Since: `Aug 2019`

1. Setting up

1.1. Prerequisites

1. **JDK** `11` or above
2. **IntelliJ** IDE

NOTE IntelliJ by default has Gradle and JavaFx plugins installed.
Do not disable them. If you have disabled them, go to `File` > `Settings` > `Plugins` to re-enable them.

1.2. Setting up the project in your computer

1. Fork this repo, and clone the fork to your computer
2. Open IntelliJ (if you are not in the welcome screen, click `File` > `Close Project` to close the existing project dialog first)
3. Set up the correct JDK version for Gradle
   a. Click `Configure` > `Project Defaults` > `Project Structure`
   b. Click `New…` and find the directory of the JDK
4. Click `Import Project`
5. Locate the `build.gradle` file and select it. Click `OK`
6. Click `Open as Project`
7. Click `OK` to accept the default settings
8. Open a console and run the command `gradlew processResources` (Mac/Linux: `./gradlew processResources`). It should finish with the `BUILD SUCCESSFUL` message.
   This will generate all resources required by the application and tests.

1.3. Verifying the setup

1. Run the `Launcher` class and try a few commands
2. Run the tests under `src/test` to ensure they all pass. To run all tests, right-click on the `test` folder and select `Run 'All Tests'`.

1.4. Configurations to do before writing code

1.4.1. Configuring the coding style

This project follows oss-generic coding standards. IntelliJ's default style is mostly compliant with ours but it uses a different import order from ours. To rectify,

1. Go to `File` > `Settings…` (Windows/Linux), or `IntelliJ IDEA` > `Preferences…` (macOS)

2. Select `Editor` > `Code Style` > `Java`
3. Click on the `Imports` tab to set the order
    - For `Class count to use import with '*'` and `Names count to use static import with '*'`: Set to `999` to prevent IntelliJ from contracting the import statements
    - For `Import Layout`: The order is `import static all other imports`, `import java.*`, `import javax.*`, `import org.*`, `import com.*`, `import all other imports`. Add a `<blank line>` between each `import`

1.4.2. Updating documentation to match your fork

After forking the repo, the documentation will still have the `AY1920S1-CS2113-T14-3` branding and refer to the `AY1920S1-CS2113-T14-3/main` repo.

If you plan to develop this fork as a separate product (i.e. instead of contributing to `AY1920S1-CS2113-T14-3/main`), you should do the following:

1. Configure the site-wide documentation settings in `build.gradle`, such as the `site-name`, to suit your own project.
2. Replace the URL in the attribute `repoURL` in `DeveloperGuide.adoc` and `UserGuide.adoc` with the URL of your fork.

1.4.3. Setting up CI

Set up Travis to perform Continuous Integration (CI) for your fork.

After setting up Travis, you can optionally set up coverage reporting for your team fork.

NOTE Coverage reporting could be useful for a team repository that hosts the final version but it is not that useful for your personal fork.

Optionally, you can set up AppVeyor as a second CI.

NOTE Having both Travis and AppVeyor ensures your App works on both Unix-based platforms and Windows-based platforms (Travis is Unix-based and AppVeyor is Windows-based)

1.4.4. Getting started with coding

When you are ready to start coding, we recommend that you get some sense of the overall design by reading about WordUp's architecture.

2. Design

2.1. Architecture

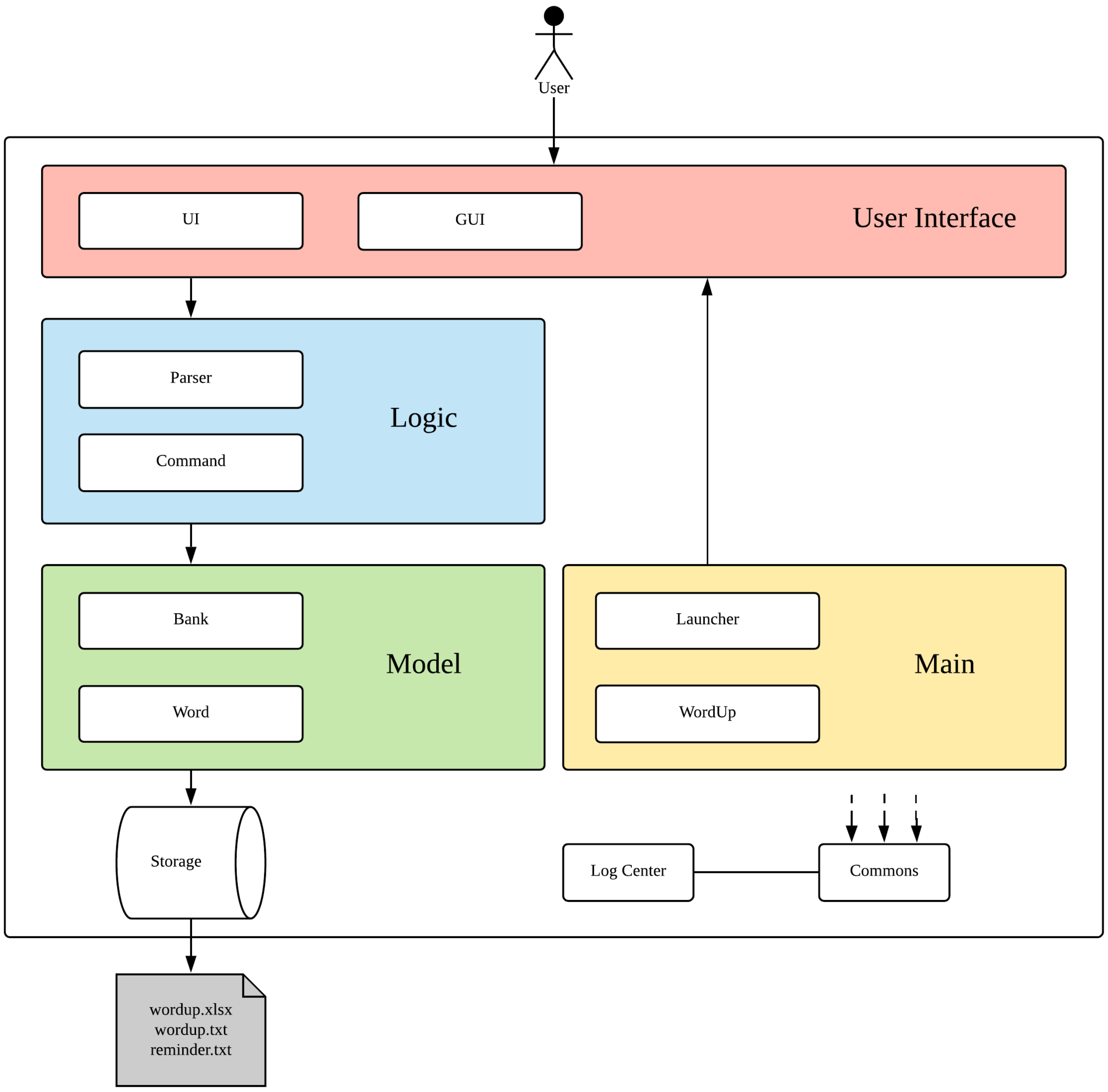


Figure 1. Architecture Diagram

The ***Architecture diagram*** above gives a high-level overview of the design of the WordUp application. The app adopts an n-tier style architecture diagram, where higher layers make use of services provided by lower layers. The following is a quick overview of each component.

`Main` has two classes called `WordUp` and `Launcher`. It is responsible for,

- At app launch: Initialises the application components in the correct sequence, and connects them up with each other. During this process the GUI is also setup and then launched as a JavaFX application.
- At shut down: Shuts down the components and invokes cleanup method where necessary.

`Commons` represents a collection of classes used by multiple other components. The following class plays an important role at the architecture level:

- `LogsCenter` : Used by many classes to write log messages that will be stored in the application’s log file. Allows developers to trace any errors and have a clearer overview of the system flow during run for easier maintenance of the application.

The rest of the App consists of four components.

- **`User Interface`**: The UI of the App.
- **`Commons`**: A collection of classes used by multiple other components.
- **`Logic`**: The main controller of the entire application.
- **`Model`**: Holds the data of the application in-memory.
- **`Storage`**: Reads from and writes data to the hard disk, via text files and excel files.

How the architecture components interact with each other

The *Sequence Diagram* below shows how the components interact with each other for the scenario where the user issues the command `delete w/kiwi`.

[ArchitectureSequenceDiagram] | *ArchitectureSequenceDiagram.png*

Figure 2. Component interactions for `delete 1` *command*

The sections below give more details of each component.

2.2. UI component

[UiClassDiagram] | *UiClassDiagram.png*

Figure 3. Structure of the UI Component

API : `Ui.java`

The UI consists of a `MainWindow` that is made up of parts e.g.`CommandBox`, `ResultDisplay`, `PersonListPanel`, `StatusBarFooter` etc. All these, including the `MainWindow`, inherit from the abstract `UiPart` class.

The `UI` component uses JavaFx UI framework. The layout of these UI parts are defined in matching `.fxml` files that are in the `src/main/resources/view` folder. For example, the layout of the `MainWindow` is specified in `MainWindow.fxml`

The `UI` component,

- Executes user commands using the `Logic` component.
- Listens for changes to `Model` data so that the UI can be updated with the modified data.

2.3. Logic component

[LogicClassDiagram] | *LogicClassDiagram.png*

Figure 4. Structure of the Logic Component

API : `Command.java` `Parser.java`

[CommandClassDiagram] | *CommandClassDiagram.png*

Figure 5. Structure of the Command Class

1. `Logic` uses the `AddressBookParser` class to parse the user command.
2. This results in a `Command` object which is executed by the `LogicManager`.
3. The command execution can affect the `Model` (e.g. adding a person).
4. The result of the command execution is encapsulated as a `CommandResult` object which is passed back to the `Ui`.
5. In addition, the `CommandResult` object can also instruct the `Ui` to perform certain actions, such as displaying help to the user.

Given below is the Sequence Diagram for interactions within the `Logic` component for the `execute("delete 1")` API call.

[DeleteSequenceDiagram] | *DeleteSequenceDiagram.png*

Figure 6. Interactions Inside the Logic Component for the `delete 1` Command

NOTE The lifeline for `DeleteCommandParser` should end at the destroy marker (X) but due to a limitation of PlantUML, the lifeline reaches the end of diagram.

2.4. Model component

[ModelClassDiagram] | *ModelClassDiagram.png*

Figure 7. Structure of the Model Component

API : `Model.java`

The `Model`,

- stores a `UserPref` object that represents the user’s preferences.
- stores the Address Book data.
- exposes an unmodifiable `ObservableList<Person>` that can be 'observed' e.g. the UI can be bound to this list so that the UI automatically updates when the data in the list change.
- does not depend on any of the other three components.

NOTE As a more OOP model, we can store a `Tag` list in `Address Book`, which `Person` can reference. This would allow `Address Book` to only require one `Tag` object per unique `Tag`, instead of each `Person` needing their own `Tag` object. An example of how such a model may look like is given below.

[BetterModelClassDiagram]

2.5. Storage component

[StorageClassDiagram] | *StorageClassDiagram.png*

Figure 8. Structure of the Storage Component

API : `Storage.java`

The `Storage` component,

- can save `UserPref` objects in json format and read it back.
- can save the Address Book data in json format and read it back.

2.6. Common classes

Classes used by multiple components are in the `seedu.addressbook.commons` package.

3. Implementation

This section describes some noteworthy details on how certain features are implemented.

3.1. Word Search feature

Search word feature allows user to look for the word that they have added to the word bank. There are 2 types of searches in our WordUp: Search using the whole word using “search w/[WORD]”, or search using the beginning substring of the word using “search w/[BEGIN_SUBSTRING]”. These 2 methods are facilitated by WordBank.

3.1.1. Search for Meaning

This allows the user to look for the meaning of a specific word that he/she has added to the bank. It is implemented as its own individual class SearchCommand, which extends class Command.

It contains an attribute searchTerm: string representing the word that user is looking for.

Given below is an example of usage scenario for Search Word feature:

Step 1: User have already added a few words as below. Our word bank use a data structure to store all words as a binary tree.

Step 2: User wants to search for a word, e.g. “one”. It first goes to the word “seven”. We see that “one” appears before “seven”, so it searches on the left subtree of “seven”. Then it reaches “four”, and see that “one” appears after “four”, so it searches to the right. Then it reaches “one” and return it. The words appeared in searching are marked as yellow.

Step 3: (If the word doesn’t appear in the bank): When search pointer reaches the lowest level but still cannot find the word, it will look for the “near” words. A “near” word is defined as the ratio between the edit distance between 2 words and the length of shorter word is less than 60%.

3.1.2. Search with Beginning Substring

This allows the user to look for the word that has a specific start. It is implemented as its own individual class SearchBeginCommand, which extends class Command. Step 1: Similar to Search Word, word bank loads all of the words in a binary search tree. Step 2: SearchBeginCommand will look to the first word in the dictionary that starts with a specific substring.

In the diagram above, if the user inputs “f”, it will searches in the sequence “seven” → “four” → “five”, and get “five” as the first word starts with “f”. If user inputs “s”, it will searches “seven”. It will see that the predecessor of “seven” is “one”, which doesn’t start with “s”, so it stops searching and get the word “seven”. Step 3: From that word, continuously look for its successor to find the word that starts with the specific substring. When it reaches a word that doesn’t start with that substring, it terminates and returns all the found words.

3.2. Usage Statistics feature

The usage statistics feature includes a collection of commands that can allow the user to obtain information regarding his usage habits of WordUp that can help the user use the word bank more efficiently.

3.2.1. Recently Added

The Recently Added feature allows the user to quickly check back on the words he had recently added to the application. It is facilitated by the RecentlyAddedCommand, which extends the Command class. It contains the following attributes: * numberOfWordsToDisplay: int This represents the number of words the user has requested to be displayed. * wordHistory: Stack<Word> The Word objects in the word bank will be stored in a first-in first-out data-structure of a stack so that the words can be retrieved quickly and in chronological order of addition to the word bank. It implements the following operations: * RecentlyAddedCommand(int) - Assigns the value of words requested to the numberOfWordsToDisplay attribute on the construction of the command object. * execute(Ui, WordBank, Storage, WordCount) - Creates the wordHistory stack and calls Ui to display the recently added words accordingly.

The following is an example usage scenario for the Recently Added feature. Step 1: The user enters history 5 command to see the last 5 words he has added to the word bank. The history command instantiates a RecentlyAddedCommand, which creates the wordHistory Stack. This is done by Storage calling the loadHistoryFromFile() method. A wordHistory stack containing the list of words in order of addition to the word bank is then created.

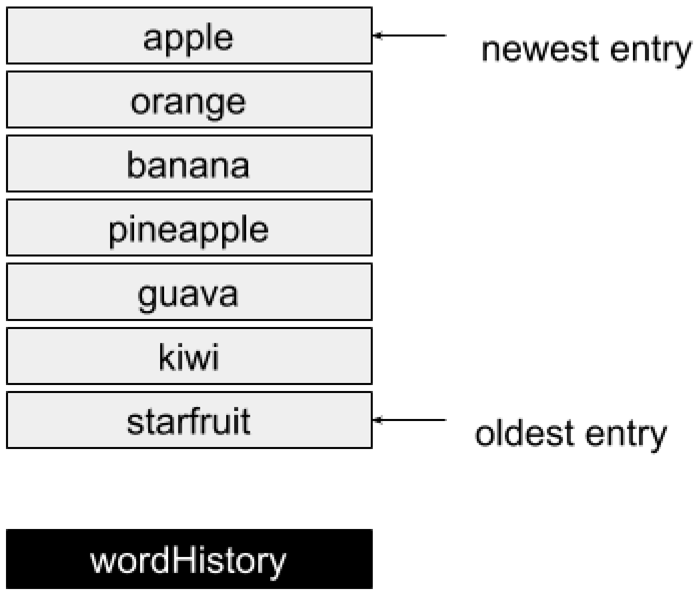


Figure 9. Sample stack containing list of words

Step 2: Ui is then called to display the numberOfWordsToDisplay, which in this case is 5, on the screen to the user as requested. In this case, the words displayed to the user are the top 5 in the wordHistory stack as shown:

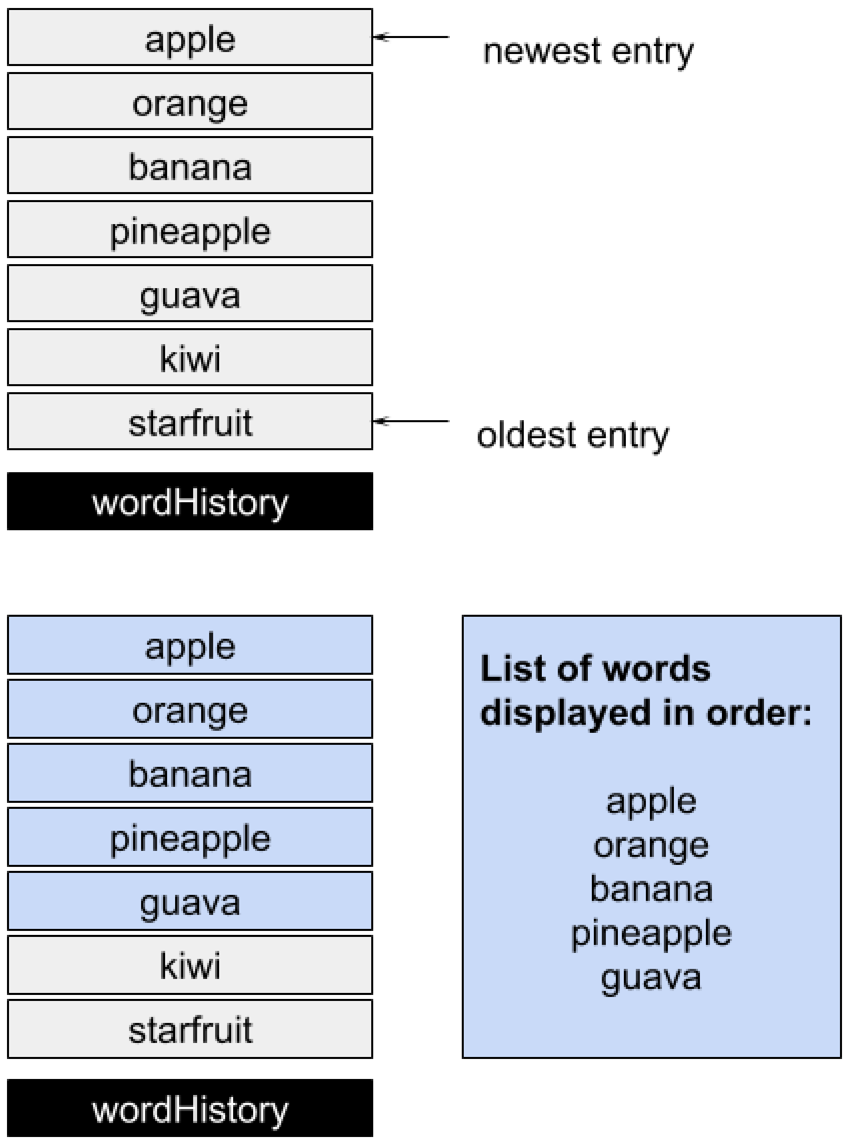


Figure 10. Stack content and corresponding words shown to user

If the wordBank currently contains less than 5 entries, an exception will not be thrown. Instead, the Ui will display the full current list of words (less than 5) on the screen for the user. This is to account for the fact that a user may not recall precisely the number of words in his wordBank. The following sequence diagram shows how the RecentlyAdded feature works:

Search Frequency

The Search Frequency feature allows the user to see the words with the highest/lowest search counts as a reflection of which words he was most unfamiliar with and therefore had to repeatedly search its meaning for. It is facilitated by the SearchFrequencyCommand, which extends the Command class, and the SearchCommand. SearchFrequencyCommand contains the following attributes: * order : String This represents the order the list displayed should be in (i.e. highest search count first or lowest search count first). It implements the following operations: * SearchFrequencyCommand(int) - Assigns the value of the display order to the displayOrder attribute on the construction of the command object. * execute(Ui, WordBank, Storage, WordCount)

- Calls Ui to display the words from wordCount to the user SearchCommand contains the following attributes: * searchTerm : String This represents the word being queried. It implements the following operations: * SearchCommand(String) - Assigns the value of the word being queried to the searchTerm attribute on construction of the command object. * execute(Ui, WordBank, Storage, WordCount) - Obtains the meaning of the word from wordBank and increases the search count in wordCount The following is an example usage scenario for the Search Frequency feature. Step 1: The user enters search w/happy to check the meaning of the word 'happy'. Through the SearchCommand, the meaning of the word is retrieved by the wordBank and wordCount calls the increaseSearchCount method to increase the search count. Ui is called to display the meaning of the word to the user. Step 2: After a few searches of different words, which is carried out following the process described in Step 1, the user enters the command freq o/desc. SearchFrequencyCommand then tells Ui the displayOrder to display the word and their word counts in.

3.2.2. Schedule Revision feature

The schedule revision feature allows the user to schedule words existing in the wordbank for recurring revision notifications up till the deadline set. It is facilitated by ScheduledRevision and Date. When the user enters a word, he is able to set a deadline for the word, such as the test date on which the word will be tested. The system then schedules a series of recurring reminders up to the deadline that reminds him of the word and its meaning. In future versions this reminder could include sample sentences using the word or a small fill-in-the-blanks quiz for the user to enhance his learning. The following is a sample usage case: Step 1: Assume that the current date is 01/01/2019 and the word 'happy' and its meaning is stored in the wordBank. Step 2: The user enters schedule w/happy by/01/02/2019. The system should store the reminder deadline onto permanent storage. It calculates the number of days to the deadline, and schedules the recurring reminder for every 3 days until the deadline. Step 3: On each reminder date, there will be a notification showing the word and its meaning for the user to revise, thereby automating his learning process.

3.2.3. Design Considerations

Aspect: How undo & redo executes

- **Alternative 1 (current choice):** Saves the entire address book.
  - Pros: Easy to implement.
  - Cons: May have performance issues in terms of memory usage.
- **Alternative 2:** Individual command knows how to undo/redo by itself.
  - Pros: Will use less memory (e.g. for `delete`, just save the person being deleted).
  - Cons: We must ensure that the implementation of each individual command are correct.

Aspect: Data structure to support the undo/redo commands

- **Alternative 1 (current choice):** Use a list to store the history of address book states.
  - Pros: Easy for new Computer Science student undergraduates to understand, who are likely to be the new incoming developers of our project.
  - Cons: Logic is duplicated twice. For example, when a new command is executed, we must remember to update both `HistoryManager` and `VersionedAddressBook`.

- **Alternative 2:** Use `HistoryManager` for undo/redo
  - Pros: We do not need to maintain a separate list, and just reuse what is already in the codebase.
  - Cons: Requires dealing with commands that have already been undone: We must remember to skip these commands. Violates Single Responsibility Principle and Separation of Concerns as `HistoryManager` now needs to do two different things.

3.3. Word Synonyms feature

3.3.1. Adding Synonym

The synonym feature allows words or phrases that means exactly or nearly the same as the current word to be added and referenced. When searched for, all the synonyms belonging to the searched word will be reflected and user is free to replace the searched word with any of the synonyms for personal use. The synonym function implements the following operations: * AddSynonym(String) - Inserts the synonyms into the HashSet of the main word. The function can only be used when we have the main word in our dictionary. User is expected to learn a word and meaning before being able to add synonyms to the word.

- execute(Ui, WordBank, Storage, WordCount) - Overwrites the storage file and WordBank while the program is running to append synonyms into their respective data structure.

The synonyms are structured using a Union Find algorithm to group the words together. When word A and word B are synonyms to each other, adding a word C to synonym of word B will automatically classify all three words together as synonyms. They are stored in the same cell within the Excel File under the StorageBank Sheet. You may view the excel file to see storage structure of the words.

The following is a sample usage case: Step 1: Assume that the word “drink” and its meaning has already been saved into the word bank by the functions supported above. Step 2: User learnt a new word “beverage” that has the same meaning as “drink”. User adds “beverage” as a synonym to the main word “drink” Step 3: In doing so, the union find algorithm will group the words together and store them within the same cell. Note that beverage itself does not need to be saved into the dictionary before adding as a synonym to a main word. However the main word “drink” must be added to the dictionary before the usage of this feature.

Searching of synonyms

Since the synonyms are chained together using a Union Find algorithm, words are inherently grouped together. When we look for synonyms of a word, the tree structure essentially returns every node that is reachable from our main word node. This allow us to lookup synonyms in a quick manner.

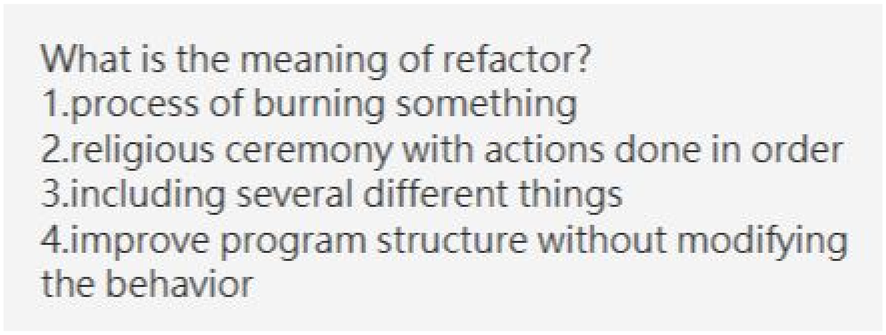
3.4. Quiz feature

- Generate quizzes to test the user’s understanding of a word, with a score at the end of the quiz. Wrongly answered words will be shown at the end of a quiz. See QuizScene.java and

QuizCommand.java for details.

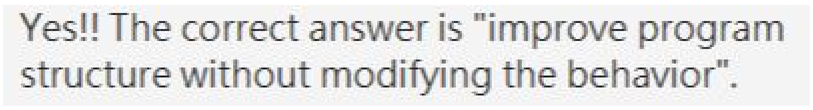
- The quizzes are in the form of 4-option MCQs,4 in a row. The generateQuiz() function generates a quiz if there is at least 4 word object saved into the word bank. It selects 1 word object and retrieve the vocabulary and meaning for the expected answer. It then randomly select 3 other word objects and retrieve their meanings for options of the MCQ.
- The quiz will output a word, and the 4 choices of meanings. Prompting the user to enter between “1 to 4” similar to MCQ picking before informing the user if they have gotten the quiz question correct.

In the following example (Figure 11), if the user inputs “1”, WordUp will response the correctness (Figure 12), and at the end of the quiz it will show wrongly answered words so the user can review the words (Figure 13).



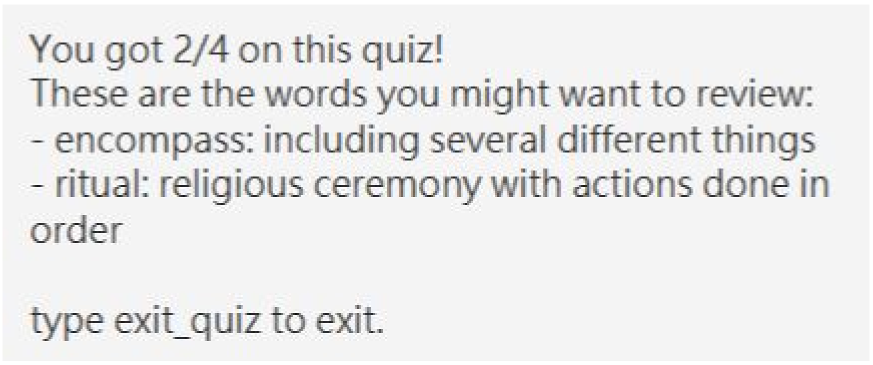
```
What is the meaning of refactor?
1.process of burning something
2.religious ceremony with actions done in order
3.including several different things
4.improve program structure without modifying
the behavior
```

Figure 11. 4-option MCQ



```
Yes!! The correct answer is "improve program
structure without modifying the behavior".
```

Figure 12. Answering response



```
You got 2/4 on this quiz!
These are the words you might want to review:
- encompass: including several different things
- ritual: religious ceremony with actions done in
order

type exit_quiz to exit.
```

Figure 13. Review words

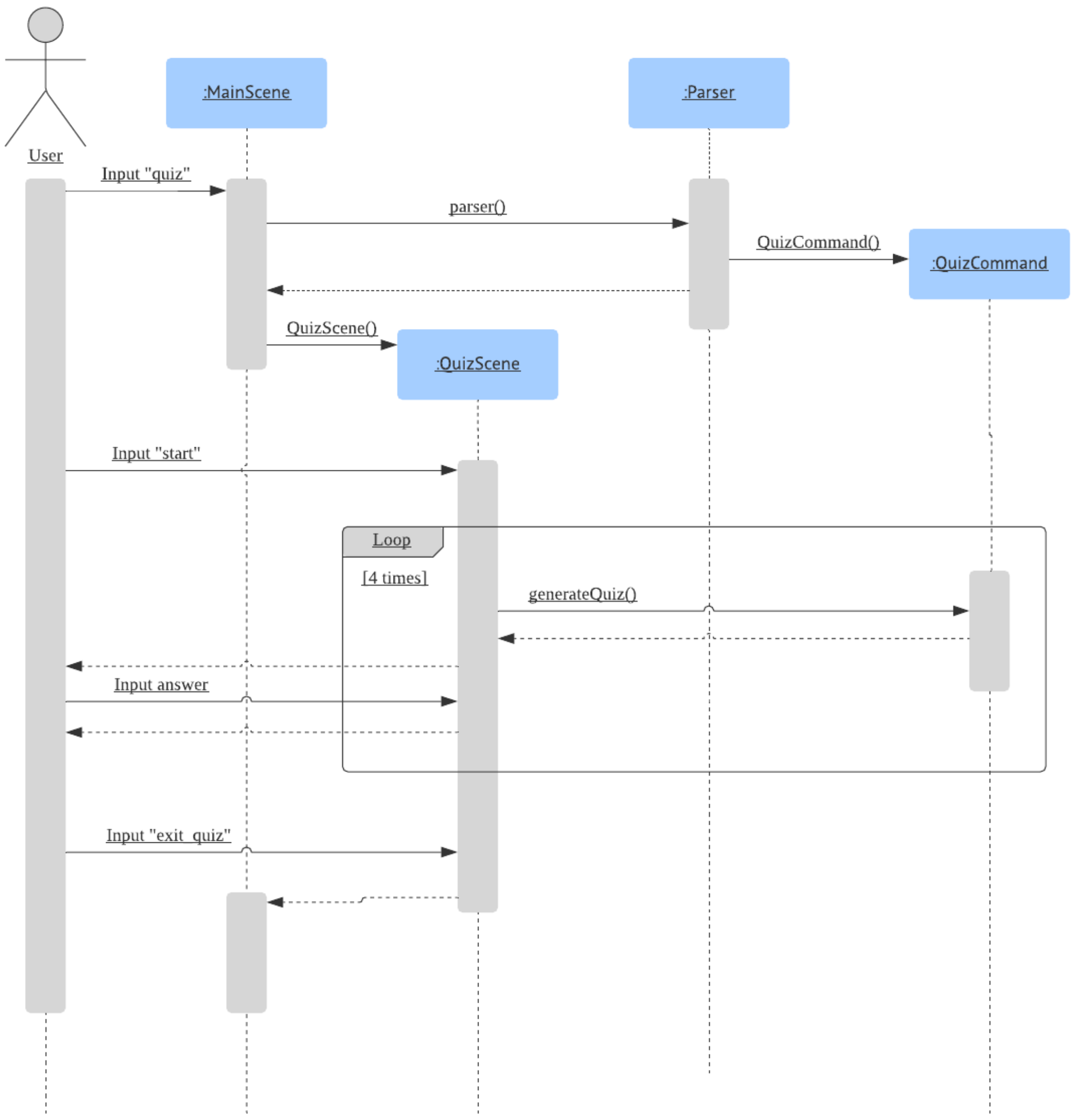


Figure 14. Sequence Diagram of a quiz

3.5. Logging

We are using `java.util.logging` package for logging. The `LogsCenter` class is used to manage the logging levels and logging destinations.

- The logging level can be controlled using the `logLevel` setting in the configuration file (See Section 3.6, “Configuration”)
- The `Logger` for a class can be obtained using `LogsCenter.getLogger(Class)` which will log messages according to the specified logging level
- Currently log messages are output through: `Console` and to a `.log` file.

Logging Levels

- `SEVERE` : Critical problem detected which may possibly cause the termination of the application

- `WARNING` : Can continue, but with caution
- `INFO` : Information showing the noteworthy actions by the App
- `FINE` : Details that is not usually noteworthy but may be useful in debugging e.g. print the actual list instead of just its size

3.6. Configuration

Certain properties of the application can be controlled (e.g user prefs file location, logging level) through the configuration file (default: `config.json`).

4. Documentation

Refer to the guide here.

5. Testing

Refer to the guide here.

6. Dev Ops

Refer to the guide here.

Appendix A: Product Scope

Target user profile: Tech-savvy english language students

- Learns many new words consistently over an extended duration
- Needs a space-efficient way of storing their words
- Prefers digital recording of words instead of writing by hand in notebooks
- Wants to practice spelling and typing words
- Needs to catalog words according to their meaning and/or alphabetical order for better future referencing

Value proposition: manage vocabulary collection, revision and searching faster than a typical handwritten/GUI driven app

Product rationale: Language students usually have a list of vocabulary to learn and master with each chapter of material taught. It is sometimes difficult to track all the words learnt, and even less easy to sort and categorise them by handwritten or analog means.

Our app aims to allow these students to easily store and collate new words learnt easily through a CLI. With a CLI, the word storage process may be much faster compared to handwriting notes for a user who types quickly, especially since the new students may be still unfamiliar with hand-writing the characters in the English alphabet. In addition to recording words, the app also aims to assist students in revising the words in an interactive and automated manner, which is a feature lacking in traditional analog recording methods. This app is developed with the aim of providing a simple, fast and value-adding service for English language students.

Appendix B: User Stories

Priorities: High (must have) - * * *, Medium (nice to have) - * *, Low (unlikely to have) - *

Priority	As a ...	I want to ...	So that I can...
* * *	English learner	store new words I have learnt in one place	refer back to it to refresh my memory
* * *	English learner who prefers interactive learning methods	have vocabulary quizzes	practice how well I can remember the new words
* * *	time-conscious English learner	record the meaning of the word on keying in just the word	do not have to copy and paste it from the net manually
* * *	English learner	categorise the words I have learnt into different subgroups	conveniently find a group of words I need to use (e.g. a subgroup can be all the words from a particular lesson/chapter)
* *	English learner who likes to pace my learning	schedule words for revision and get reminders for them	effectively revise selected words before a test/custom deadline

Priority	As a …	I want to …	So that I can…
* *	statistically oriented English learner	view my search history	check which words I keep needing to review on and put in more effort to learn those words
* *	English learner	enter letter to display words starting with it	type a word correctly even if I am unfamiliar with how to spell it
* *	English learner	see how much I have searched for a word	track the most “forgotten” words and target those words specifically in my learning
*	English learner	export my wordbank to word or pdf	print them out and read them on the go during revision for any tests/just for my own ease of learning

{More to be added}

Appendix C: Use Cases

(For all use cases below: System is defined to be WordUp and User is an English language learner for all following use cases:)

Use case: Adding a word to the word bank

MSS

1. User enters command to add a word.
2. System adds the word and its meaning to the word bank.
3. System displays the added word and its meaning.

   Use case ends.

C.1. Use case: Finding the meaning of a word

MSS

1. User enters command to add a word.
2. System adds the word and its meaning to the word bank.
3. System displays the added word and its meaning.

   Use case ends.

Extensions

5a. There is no such word in the word bank.

- 5a.1. System throws an error to inform user that the word is not in word bank.

  Use case ends.

C.2. Use case: Making quizzes

MSS

1. User enters command to start a quiz.
2. System searches for recorded word and meanings.
3. System generates a question from the search.
4. User answers the question.
5. Repeat step 2 to 5 until all questions are done.

   Use case ends.

Extensions

4a. There is no such word in the word bank.

- 4a.1. System throws an error to inform user that the word is not in word bank.
- 4a.2. System displays the correct meaning.

  Use case ends.

C.3. Use case: Checking recently added words

MSS

1. User enters command to ask for recent words he has added.
2. System checks the wordHistory containing the words in the order they were added in.
3. System displays the words in order of latest added words to the oldest added word.

Use case ends.

Extensions

1a. There were no words added before the command.

- 1a.1. System throws an error to inform user that the wordbank is empty.
- 1a.2. System suggests user to enter new words first and exits the command.

  Use case ends.

C.4. Use case: Scheduling reminders

MSS

1. User enters command to start a schedule reminder.
2. User enters a list of words to be scheduled.
3. System prompts user for the reminder date and time.
4. User enters the reminder date and time.
5. System shows the summary of the reminder details.

   Use case ends.

Appendix D: Non Functional Requirements

1. Should work on any mainstream OS as long as it has Java 11 or above installed.
2. Should be able to hold up to 1000 words without a noticeable sluggishness in performance for typical usage.
3. Users should be able to accomplish each task without typing more than 10 words in a user input line.

Appendix E: Glossary

Mainstream OS
Windows, Linux, Unix, OS-X

MSS
Main Success Scenario

Word Bank
A collection of words the user has added into our program, stored on user's hard disk

Appendix F: Instructions for Manual Testing

Given below are instructions to test the app manually.

NOTE These instructions only provide a starting point for testers to work on; testers are expected to do more *exploratory* testing.

F.1. Launch and Shutdown

1. Initial launch
   a. Download the jar file and copy into an empty folder
   b. Double-click the jar file
      Expected: Shows the GUI with a set of sample contacts. The window size may not be optimum.
2. Saving window preferences
   a. Resize the window to an optimum size. Move the window to a different location. Close the window.
   b. Re-launch the app by double-clicking the jar file.
      Expected: The most recent window size and location is retained.

{ more test cases … }

F.2. Deleting a person

1. Deleting a person while all persons are listed
   a. Prerequisites: List all persons using the `list` command. Multiple persons in the list.
   b. Test case: `delete 1`
      Expected: First contact is deleted from the list. Details of the deleted contact shown in the status message. Timestamp in the status bar is updated.
   c. Test case: `delete 0`
      Expected: No person is deleted. Error details shown in the status message. Status bar remains the same.
   d. Other incorrect delete commands to try: `delete`, `delete x` (where x is larger than the list size) *{give more}*
      Expected: Similar to previous.

{ more test cases … }

F.3. Saving data

1. Dealing with missing/corrupted data files
   a. *{explain how to simulate a missing/corrupted file and the expected behavior}*

{ more test cases … }