**C868 – Software Capstone Project Summary**

**Task 2 – Section C**

(User & Administrator Guide is Separate)

|  |  |
| --- | --- |
| **Capstone Proposal Project Name:** | http://www.idevnews.com/views/images/uploads/general/wgu_logo.png  Text to Phoneme Converter |
| **Student Name:** | Edward Sawyer |

**Table of Contents**

Contents

[Table of Contents 2](#_Toc43140999)

[Task 2 Part C – C868 Software Development Capstone 4](#_Toc43141000)

[Application Design and Testing 4](#_Toc43141001)

[Design Document 4](#_Toc43141002)

[Class Design 4](#_Toc43141003)

[Database Design 9](#_Toc43141004)

[ERD Diagram 10](#_Toc43141005)

[UI Design 11](#_Toc43141006)

[Unit Test Plan 17](#_Toc43141007)

[Introduction 17](#_Toc43141008)

[Purpose 17](#_Toc43141009)

[Overview 17](#_Toc43141010)

[Test Plan 18](#_Toc43141011)

[Items 18](#_Toc43141012)

[Features 18](#_Toc43141013)

[Deliverables 19](#_Toc43141014)

[Tasks 19](#_Toc43141015)

[Needs 19](#_Toc43141016)

[Pass/Fail Criteria 20](#_Toc43141017)

[Specifications 21](#_Toc43141018)

[Procedures 25](#_Toc43141019)

[Results 25](#_Toc43141020)

[C4. Source Code 26](#_Toc43141021)

Task 2 Part C – C868 Software Development Capstone

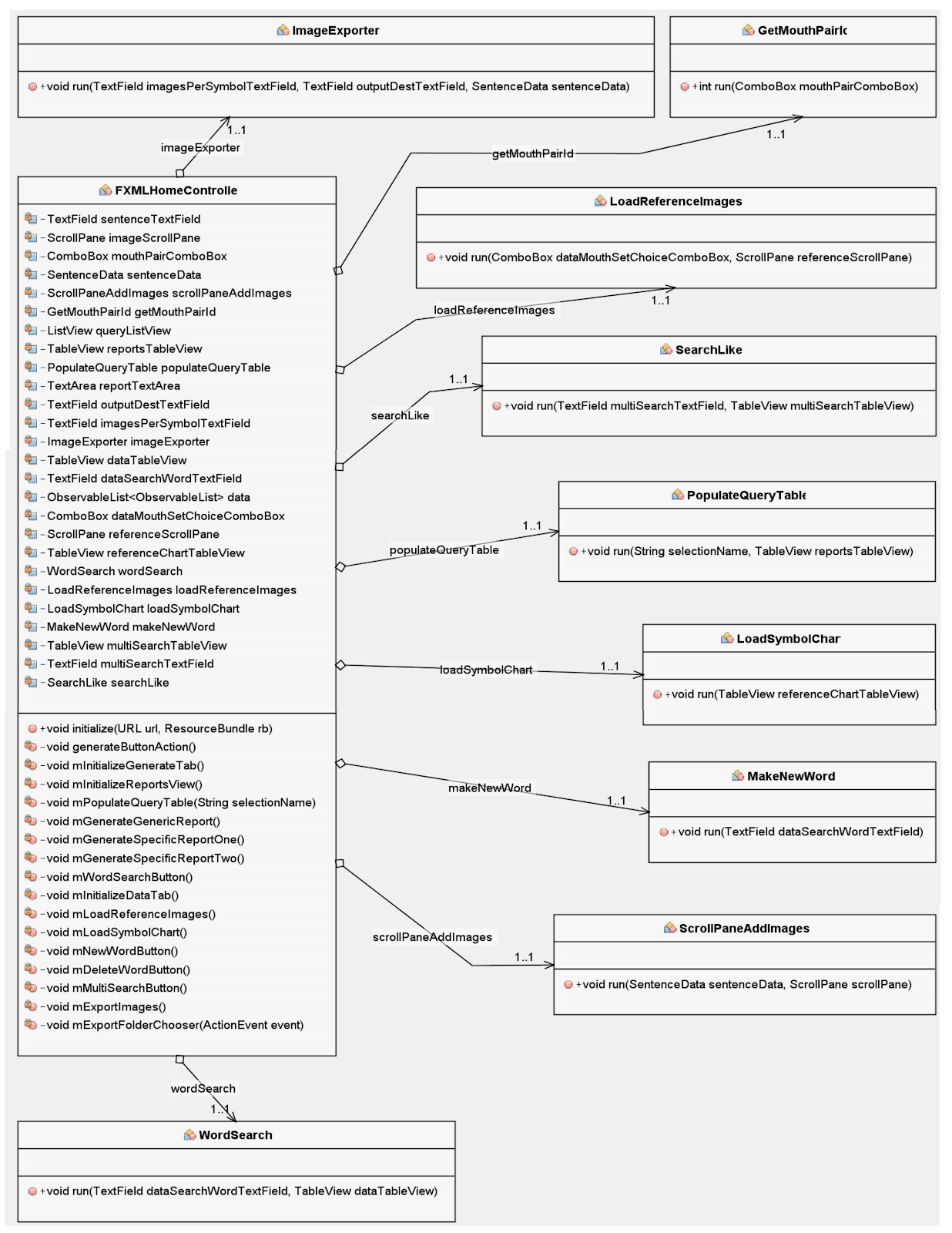
# Application Design and Testing

# Design Document

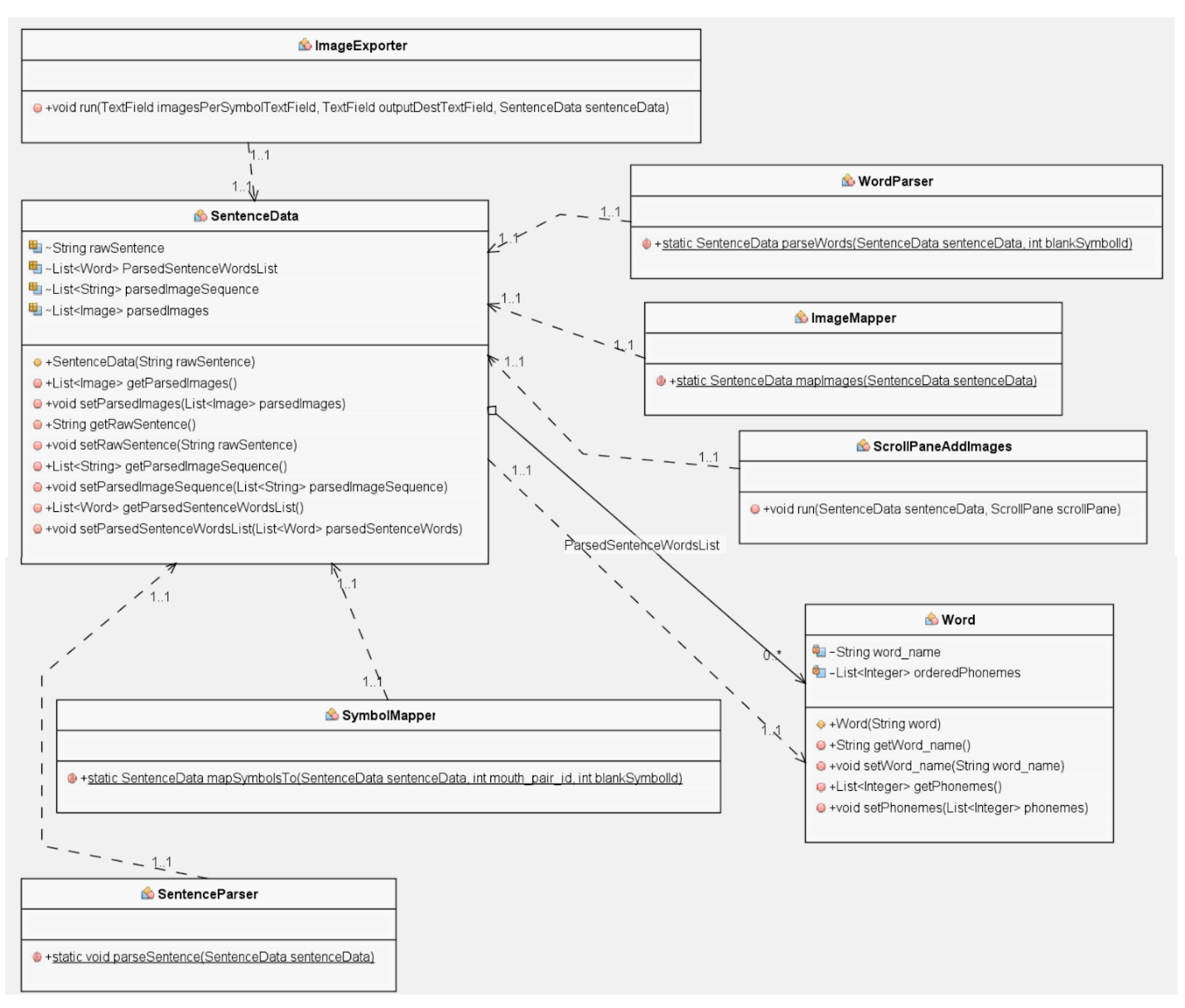
## Class Design

The Class Design for the core of this project focuses on the SentenceData object. This object is first loaded with a raw and unparsed sentence. The SentenceData object is then passed in order through the WordParser, SentenceParser, SymbolMapper, then ImageMapper methods. The SentenceData object is updated with relevant data as it is passed through each method. The final result is a SentenceData object that contains all of its own words and symbols in the original sentence order.

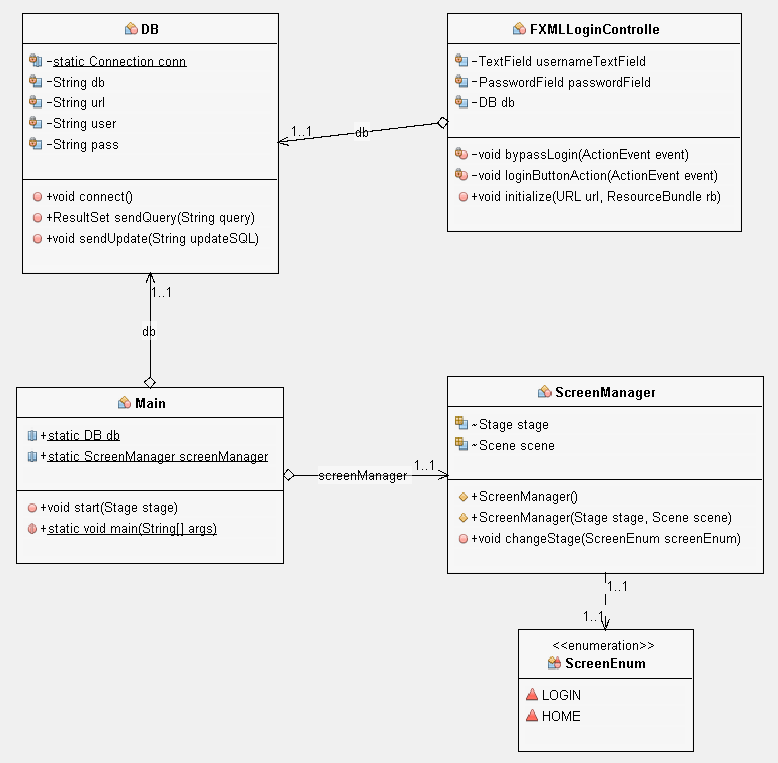
Included below is the Phoneme Converter’s Class Diagram. For organizational purposes, a Tab Pane was used to divide the functions of the application. This Tab Pane is represented by the FXMLHomeController class object. Next, the controller-related functions can be found in this area. The data model is represented by the SentenceData object, and is connected to related classes. Finally, the Database and Login related classes are connected to the DB class object.



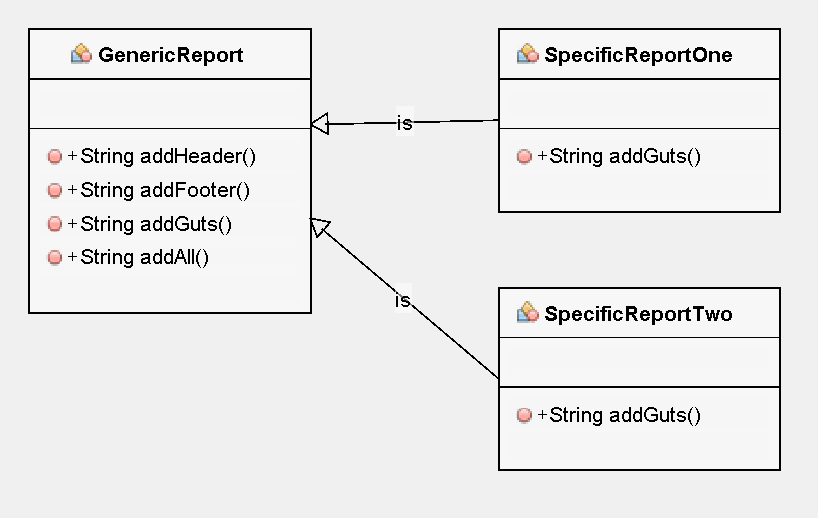
The above diagram the FXMLHomeController class is displayed with its many related methods.



The diagram above illustrates the SentenceData class and the related methods that operate on the SentenceData object.



The above diagram contains the Database and ScreenManager classes. These are used by the Main class to initiate the foundation for the application.



The above diagram illustrates the simple polymorphic objects used in the Reports tab of the application.

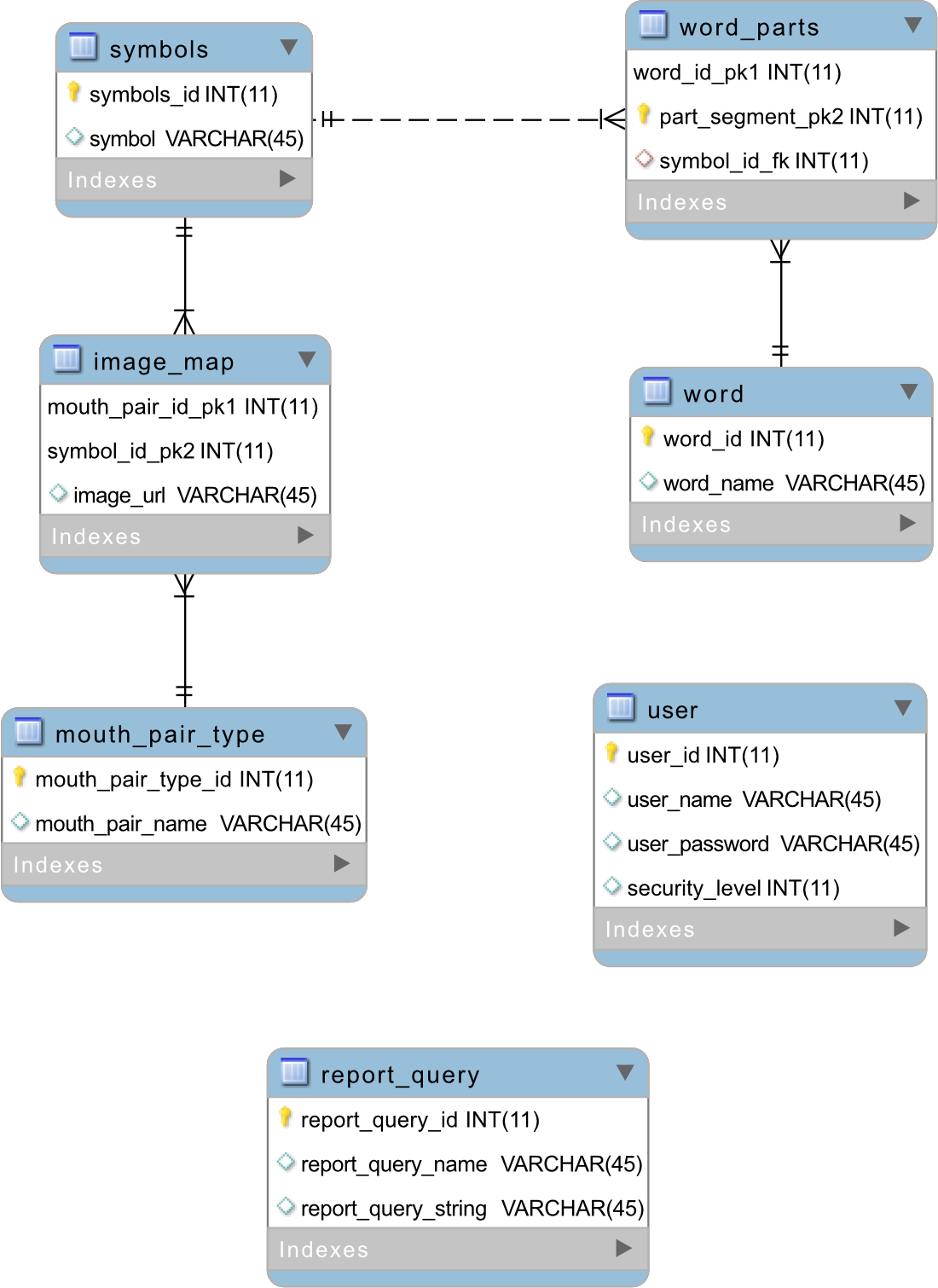
## Database Design

The database is the heart of this text-to-phoneme converter. The structure of the database is designed to maintain data integrity and foster future development. The database adheres to the 3rd Normal Form. Adhering to at least the 3rd Normal Form ensures that each unique type of data only exists once in the database.

The ‘word\_parts’ table holds the bulk of the information derived from the pronunciation dictionary. This table holds a reference to the word name along with the symbols for each part of the word. When the application is parsing the raw sentence into symbols the ‘word\_parts’ table is providing most of the relevant data.

The ‘image\_map’ table is the next most important table. This table holds the mapping between the 84 basic phonemic symbols found in the pronunciation dictionary and their image file keys. The keys are simply the best matching image available in the relevant mouth image set. This table provides the flexibility for an artist to add new images to the image set, as well as adding entirely distinct image sets. For example, a very simple set of only an opened and closed mouth could be added. A much more complex set of 84 individual and unique images could also be added.

### ERD Diagram



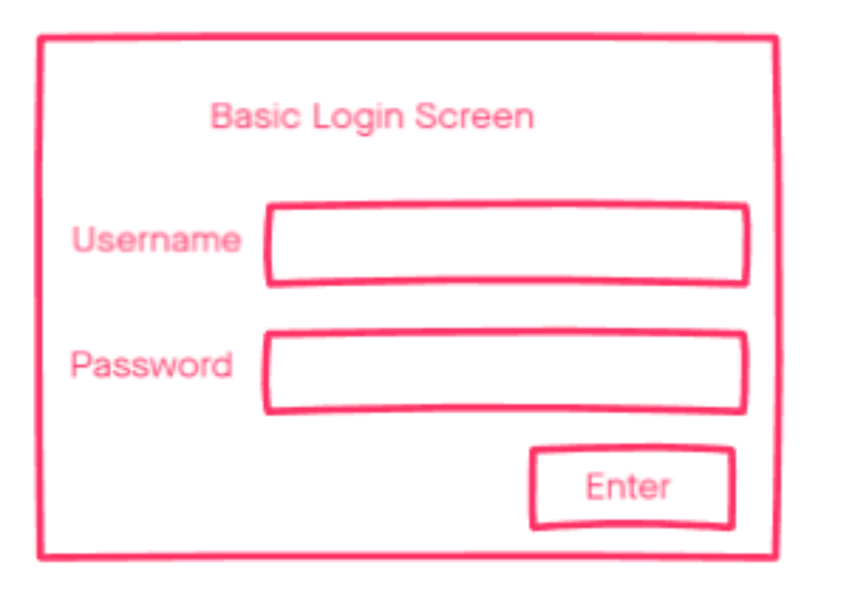
The above Entity Relationship Diagram illustrates the structure of the database.

## UI Design

The User Interface for this application is based upon the Tab Bar. A row of tabs across the top serves as both navigation and a convenient way to separate functionality into manageable pieces. This design approach also provides an easy way to add or remove functionality without any significant design changes.

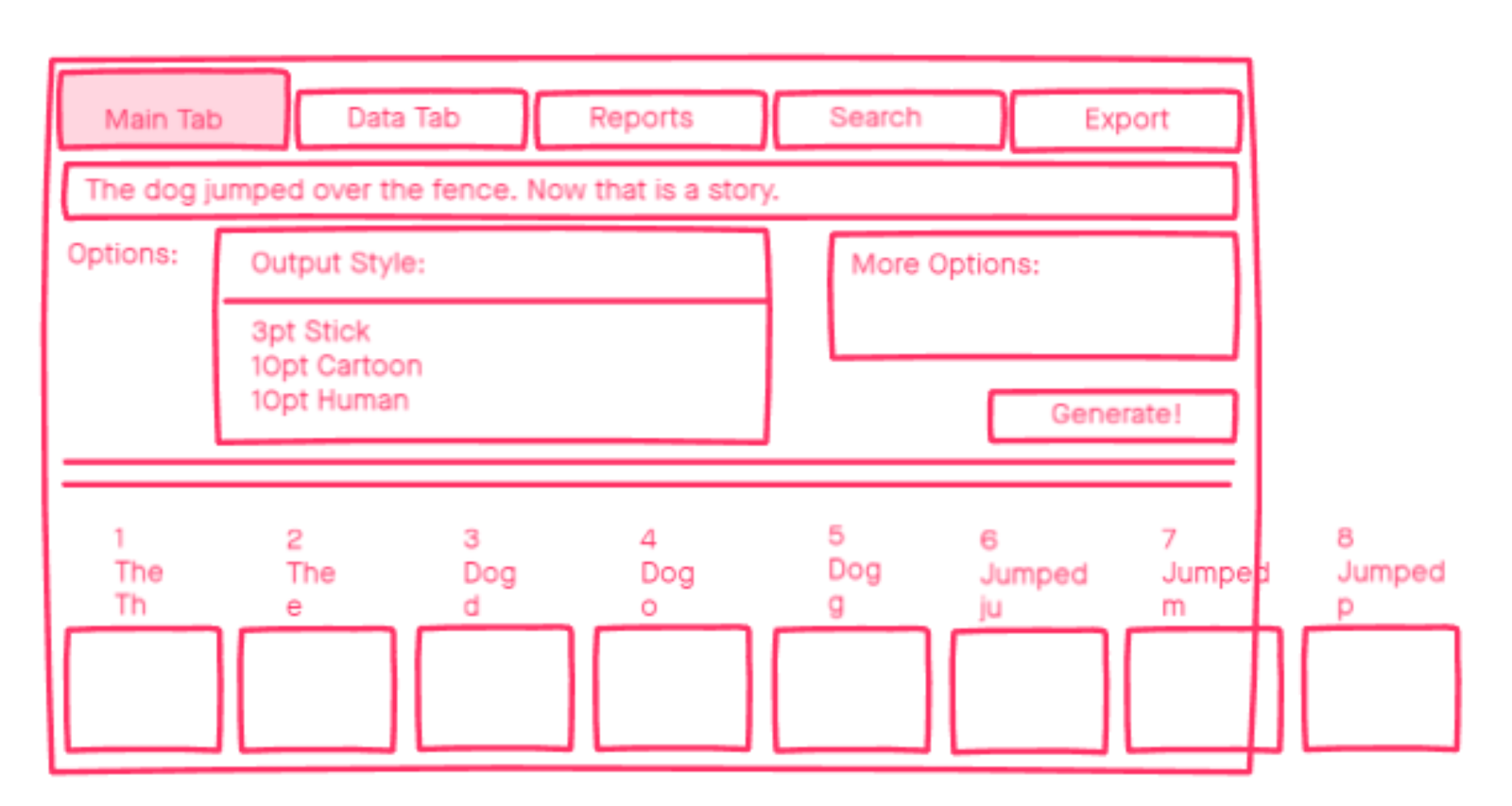
Login Screen.

This is a very basic login screen. Only users with credentials stored in the database can successfully log in. For evaluation purposes only, a ‘Bypass Login’ button has been provided. This button would be removed before any type of public distribution.



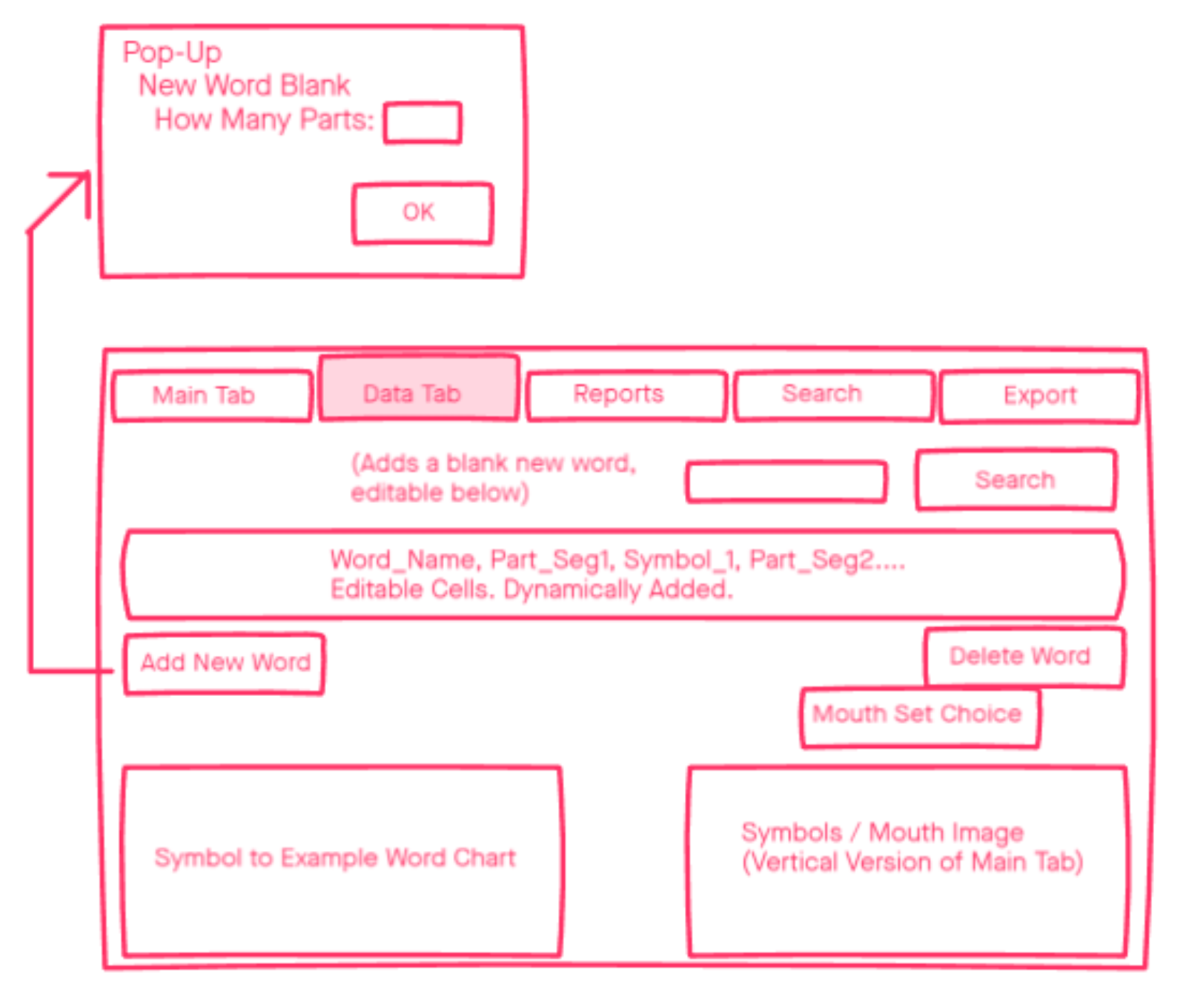
Generate Tab (Main Tab)

The Generate tab represents the core of this utility application. This is the area where the line of dialog will be entered, generated, and displayed. A pulldown menu is provided to allow for a choice in image sets.



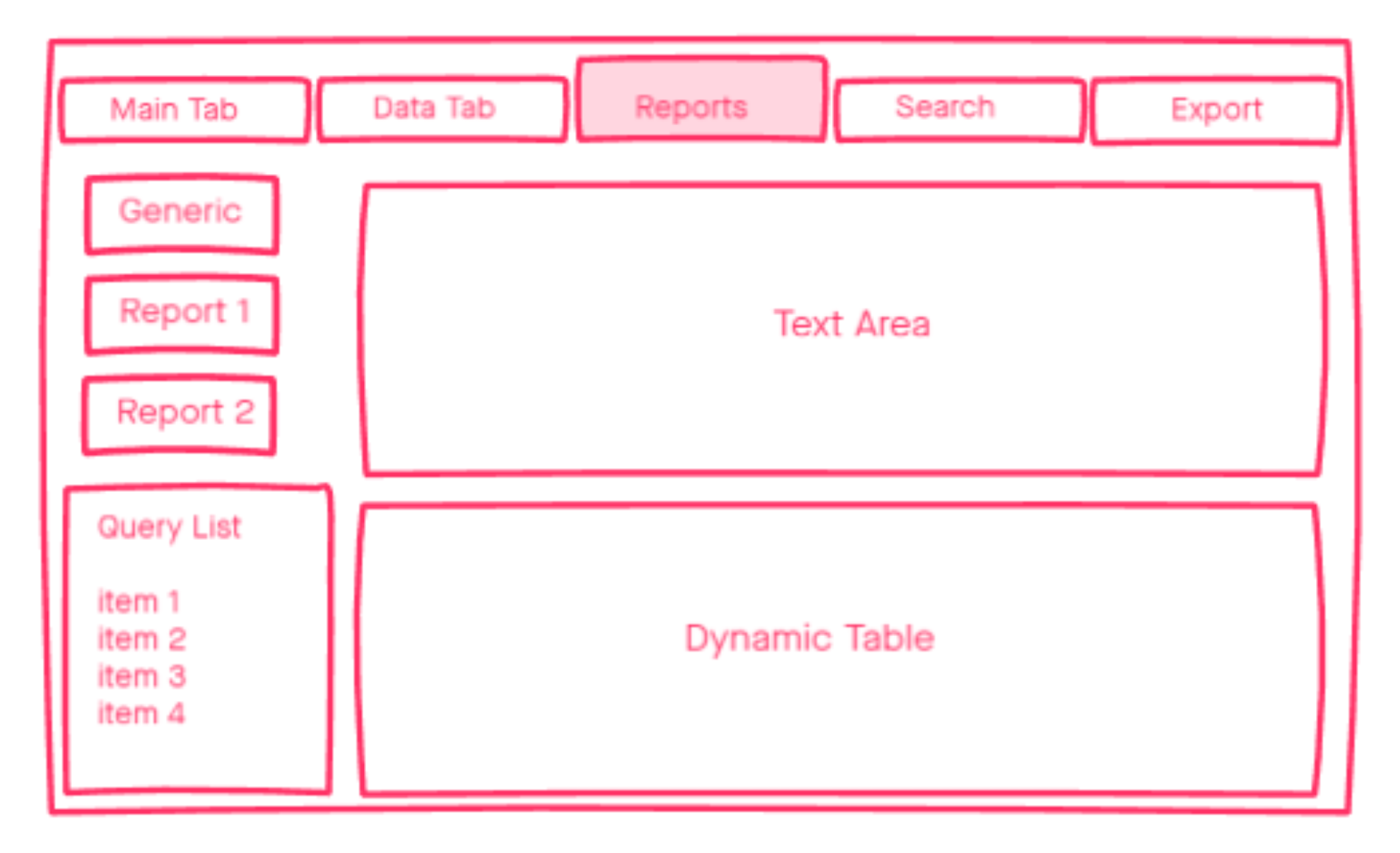
Data Tab

The Data tab is where words in the dictionary can be added, edited, or deleted. When a word is loaded into the table, each cell is individually editable. Additionally, two reference charts are provided to aid in creating a new word.



Reports Tab

The reports tab has two general areas. The upper half is focused on providing an easy way to generate reports using polymorphism to eliminate duplicate code. The lower half provides a list of pre-written queries available to pull from the database.



Search Tab

The search tab provides a more flexible search method than the one found on the data tab. This search page uses the ‘%’ wildcard in the search query to pull related words in the result set.



Export Tab

The export tab provides the user the ability to save the generated image set to disk. A chooser button and the file path text area provide two options for selecting the output directory. The exported files can then be imported into an animation program.



## Unit Test Plan

### Introduction

### Purpose

Unit testing is a critical part of the software development process. Even the smallest applications have many variables and points of interaction that need to be tested to verify functionality. Bugs exist around every corner in software development. Systematic testing is the most effective way to remove these bugs. For the purpose of this project, Unit Tests are provided focusing on the core path from a raw input sentence to a completed sequence of phonemes.

These unit tests proved that the process is working as intended. Each step produces results that match exactly what is expected. These tests are not exhaustive, however. Some inconsistencies may be discovered with continued testing. In the case of a bug being discovered, test cases like the ones provided here would provide a starting point for locating the problem in the source code. Another potential source of errors is the data held in the database. A misbehaving function could be the result of incorrectly entered data, and not broken logic. In the case of a failed test, the source data should be verified as well.

### Overview

The primary purpose of these unit tests is to verify the core functionality of the text-to-image process. The basic process is to first parse a raw input sentence into individual words. The words are then parsed into their phonetic elements. Next, the phonetic symbols are mapped to their respective image counterparts. This core process is the focus of these unit tests, as they are critical to the usefulness of this application.

These unit tests were created and tested inside the IntelliJ IDEA IDE using the JUNIT5 framework. A raw sentence was entered to test the basic requirements of removing punctuation, separating the words, breaking the words down, and mapping the word sub-symbols to images.

Initially several inconsistencies were discovered in the testing process. Fortunately, the problem was never traced all the way back to any logic. The problems were all related to keeping the proper input in line with the expected output.

## Test Plan

### Items

Testing cannot begin until a few elements are in place. The tests are aimed at the core functionality of the application. This implies that the core components of the application should be running before these tests execute. The required elements are as follows:

1. The database derived from the pronunciation dictionary must be active and accessible.
2. The primary data operations need to be in place. This includes the Sentence Parser, Word Parser, and Symbol Mapper classes.
3. The JUnit5 framework needs to be installed in the current IDE.

### Features

The tests look at the methods that create the path from raw text to ordered symbols. The same data is loaded into the SentenceData object in each set of tests for consistency. The specific classes and methods are listed below:

1. SentenceParser.parseSentence(SentenceData sentenceData)
   1. This method takes a raw input sentence and splits it into individual words. This also removes all punctuation.
2. WordParser.parseWords(SentenceData sentenceData, int blankSymbol)
   1. This method uses the database to create an ordered list of symbols for each word.
3. SymbolMapper.mapSymbolsTo(SentenceData sentenceData, int imageSet, int blankSymbol)
   1. This method creates an ordered list of images based on the image set choice and the ordered symbol list created in the parseWords method.

### Deliverables

The IntelliJ IDEA IDE with the installed JUnit5 framework provides console output when the unit tests are run. The assertEquals methods will return errors when the comparison is not true. The error will return the line number inside the test that failed as well as the expected and actual values in the comparison. Tracking down the source of the error can begin from this console report.

### Tasks

In order to run the tests, the full project is loaded into the IntelliJ IDEA IDE. Once the project is loaded, the “Run ‘All Tests’” context option is selected by right-clicking on the project name in Project Tool Window. This will build and run the project and the tests. The output will be available in the ‘Run’ console.

It is also important to note that the unit tests should be performed in order. The tests build upon one another. The first test needs to be the SentenceParser. The second test needs to be the WordParser. Finally, the third test needs to be the SymbolMapper. Each of these tests relies upon valid data produced in the previous step. Running these tests out of order may allow unverified data to move through the processes.

### Needs

The core functionality of the application needs to be completed prior to initiating the unit tests. This includes the Sentence Parser, the Word Parser, and the Symbol Mapper classes. The database also needs to be intact and accessible by the application. The tests use data from the pronunciation dictionary. The JUnit5 framework is also needed for testing in an IDE. JUnit5 is freely available.

### Pass/Fail Criteria

The first step in creating these tests involves manually parsing a sentence using the pronunciation dictionary. A simple sentence is used to both illustrate a typical sentence, and to use odd punctuation and capitalization for validation purposes. This sentence is then reduced to words, then to symbols, and finally mapped to an image set. This manual process mirrors the parsing processes in the application.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Word** | **Symbol 1** | **Symbol 2** | **Symbol 3** | **Symbol 4** |
| Test | 70 | 31 | 68 | 70 |
| One | 81 | 11 | 56 |  |
| Two | 70 | 78 |  |  |
| Three | 71 | 67 | 50 |  |

Test words and their respective phonemic symbols.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Word** | **Symbol 1** | **Symbol 2** | **Symbol 3** | **Symbol 4** |
| Test | LipSet-Layers-Alpha\_08.png | LipSet-Layers-Alpha\_01.png | LipSet-Layers-Alpha\_05.png | LipSet-Layers-Alpha\_08.png |
| One | LipSet-Layers-Alpha\_06.png | LipSet-Layers-Alpha\_03.png | LipSet-Layers-Alpha\_09.png |  |
| Two | LipSet-Layers-Alpha\_08.png | LipSet-Layers-Alpha\_02.png |  |  |
| Three | LipSet-Layers-Alpha\_10.png | LipSet-Layers-Alpha\_02.png | LipSet-Layers-Alpha\_04.png |  |

Test words and their respective image equivalents.

These data sets provide the conditions that are tested in the unit tests. The assertEquals methods will each verify the application is producing output that matches the results in these tables.

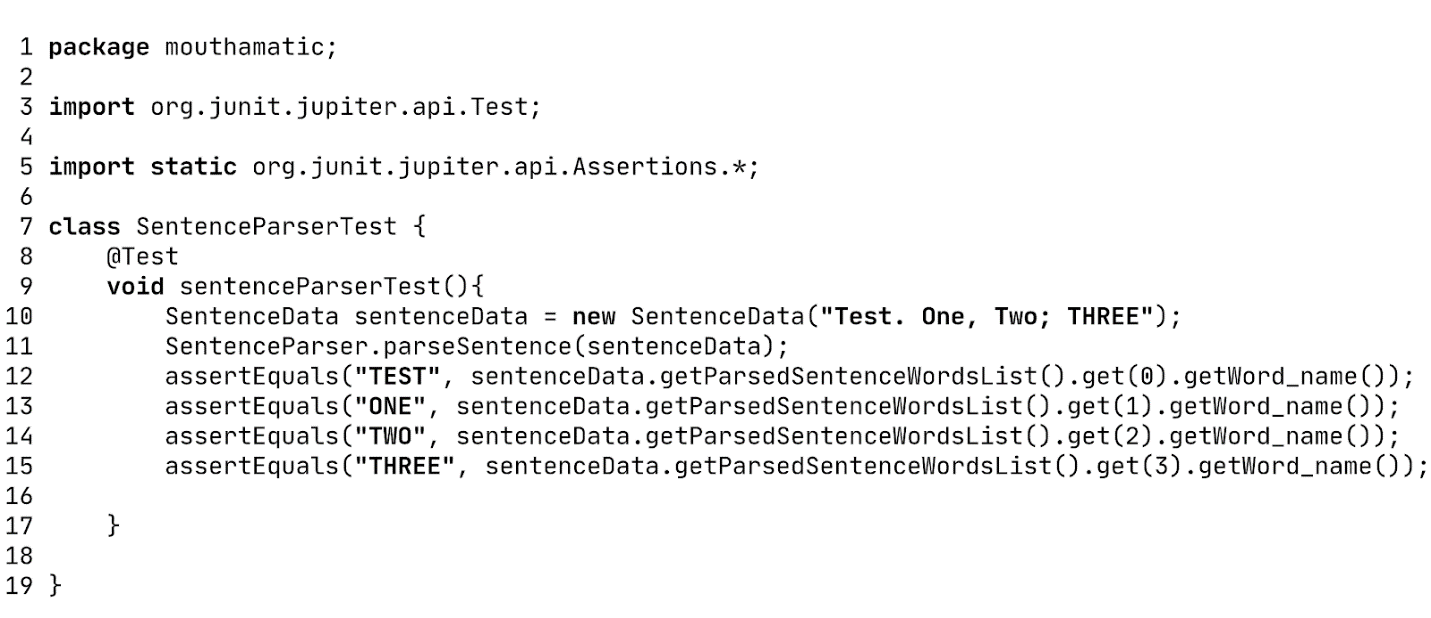
In the case of a discrepancy in one of the assertEquals comparisons the IDE will provide output in the Run console indicating the line in the test that has failed. In the comparison, either the manually derived data is incorrect, or the application’s data is incorrect. After verifying the manual data is correct the IDE can be used to step through the code in debug mode to pinpoint the source of the error. The IDE can be used to ‘watch’ variables as it is running to verify processes are working correctly.

The application also needs to pass the visual verification test. A simple demonstration video will be created to verify the process works in practice. All the unit tests can pass, but if the final generated image sequences do not match well with the recorded dialog, then the project may be deemed a failure. A demonstration video will provide the visual evidence of a successful test.

## Specifications

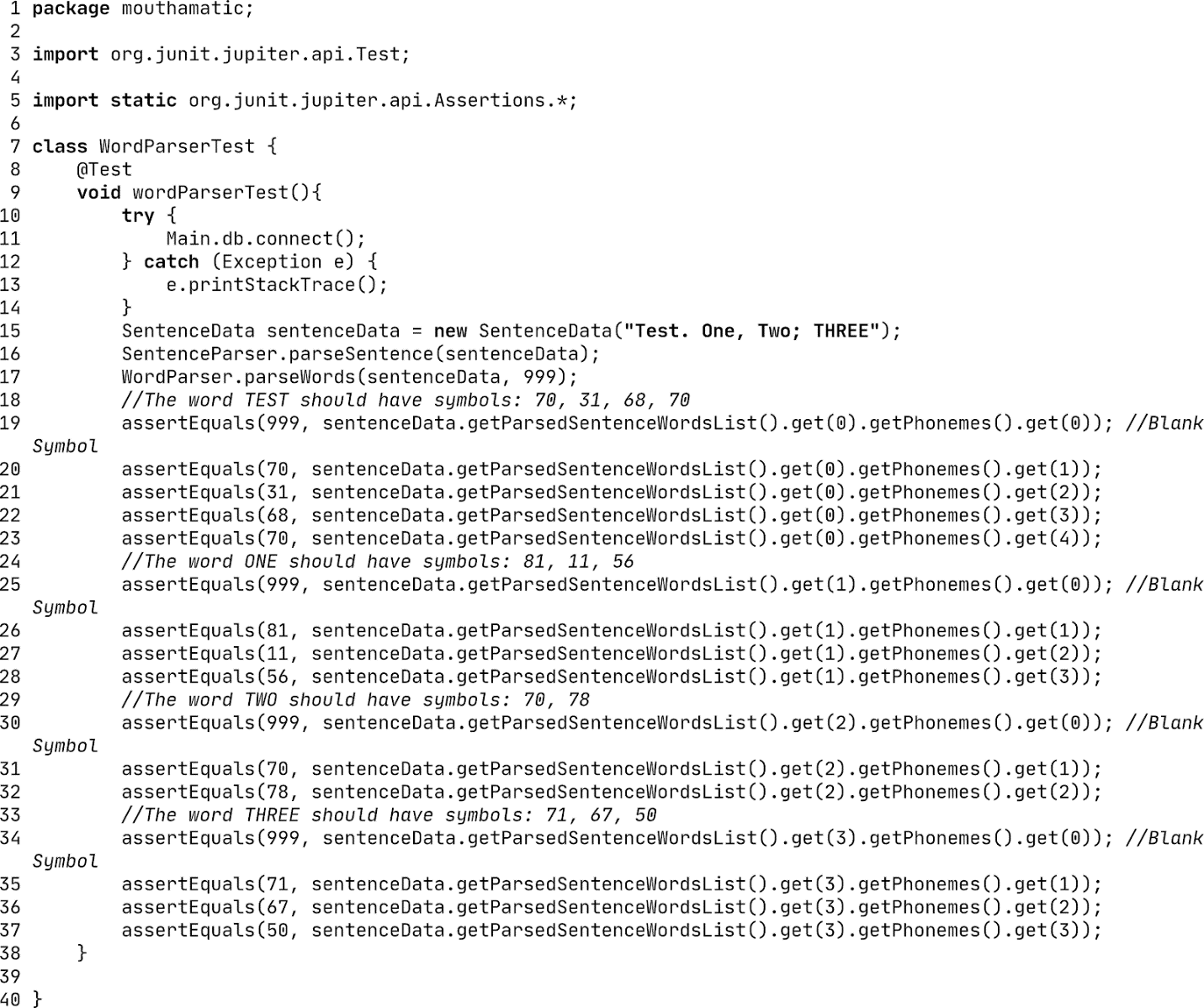
The following sections of code represent the Unit Tests for the core functionality of this application. These tests were written using the Junit5 testing framework. Tests are provided for the Sentence Parser, Word Parser, and Symbol Mapper classes.

Sentence Parser Test



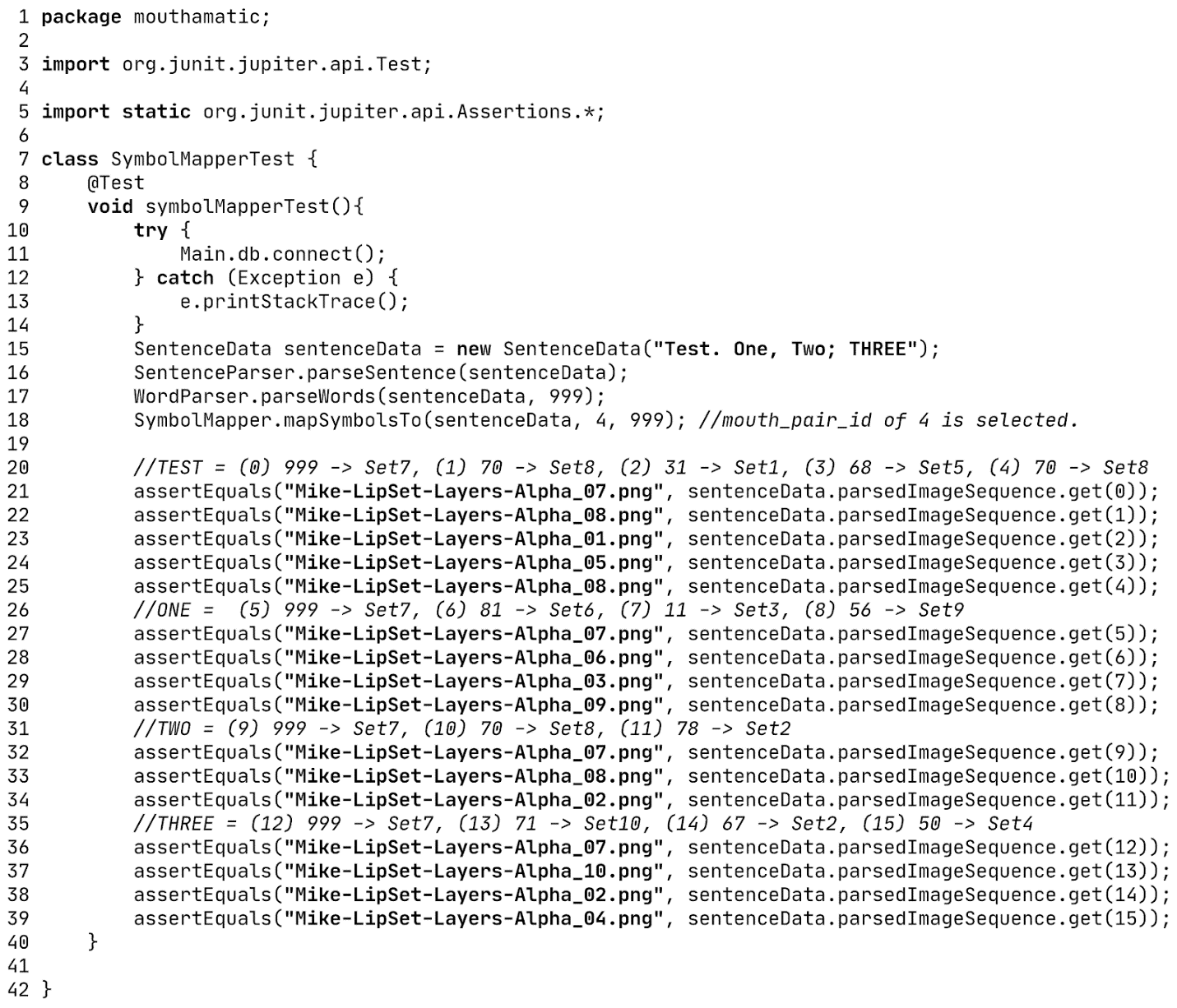
The above code represents a unit test for the SentenceParser class. The test is verifying the parsing of the raw input sentence down to individual words.

Word Parser Test



The above code represents the unit tests for the WordParser class. The tests aim to verify that words are being broken down into constituent phonemic symbols properly.

Symbol Mapper Test



The above code represents the unit tests for the SymbolMapper class. The tests are looking to verify that phonemic symbols are being properly mapped to the chosen image set.

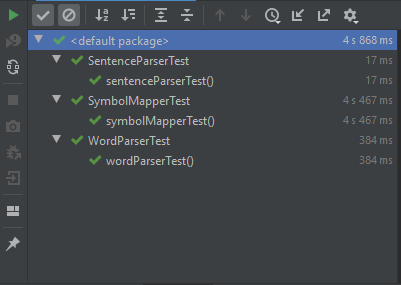
## Procedures

The testing process is conducted with the following steps:

1. Decide on an input sentence. For example, "Test. One, Two; THREE" works well as the input. This input aimed to test the SentenceParser class against punctuation and word capitalization.
2. Manually find the symbols that compose the words in the input sentence. The table with this information is available above in the pass/fail criteria section.
3. Create the tests based on the manually derived data. With the same input sentence, the application should arrive at the same results found manually.
4. For each failed test, view the expected and actual result and begin searching for a cause of discrepancy.

## Results

Provided below is a screenshot of the IDE console after running the unit tests. If all the comparisons made in the assertEquals methods are true, the output should look as it does here. If a comparison were false, information about what failed would be provided in the same area.



# C4. Source Code

The source code for this application has been included with Sections A, C, and User Guide.