Wheel of Fortune
Software Design Description

Author: Kavan Wadhwa
Emerald Games Enterprises™
November, 2017
Version 1.0

Abstract: This document describes the software design for the computer version of Wheel of Fortune, a popular game show rooted in the- classic Hangman game.

Based on IEEE Std 1016™-2009 document format

1 Introduction

This is the Software Design Description (SDD) for the basic Wheel of Fortune computer implementation. This document format is based on the IEEE Standard 1016-2009 recommendation for software design.

1.1 Purpose

This document aims to act as the basis for the development of the Wheel of Fortune application. Using UML class diagrams, it will provide details surrounding the packages, classes, instance and class variables, and method signatures that will be used for the implementation of the various parts of this application. Additionally, UML Sequence diagrams will be used to illustrate the potential interactions between objects once the application is run. This includes handling events.

1.2 Scope

Wheel of Fortune is one of a series of game shows that EGE aims to computerize. Many game shows implement similar features; for example, the wheel spun in Wheel of Fortune imitates the wheel spun on "The Price is Right". The phrase board, and for that matter, general use of phrases, words, and letters, in Wheel of Fortune imitates the use of such elements in shows like Chain Reaction and Lingo. As such, it is important to keep these overlapping similarities in mind to avoid future code duplication. The target language for this software design is Java.

1.3 Definitions, acronyms, and abbreviations

Class Diagram – A UML document format that describes classes graphically. Specifically, it describes their instance variables, method headers, and relationships to other classes.

IEEE – Institute of Electrical and Electronics Engineers, the "world's largest professional association for the advancement of technology".

Framework – In an object-oriented language, a collection of classes and interfaces that collectively provide a service for building applications or additional frameworks all with a common need.

Java – A high-level programming language that uses a virtual machine layer between the Java application and the hardware to provide program portability.

Wheel of Fortune – The SONY Pictures Television game show that will be replicated by our software. The game is founded on Hangman. Contestants spin a wheel and guess consonants or buy vowels and attempt to solve the random phrase displayed on the board. The objective is to guess correct letters and phrases and earn money from doing so. The winner proceeds to a bonus round with a chance for winning a large cash prize or car.

Sequence Diagram – A UML document format that specifies how object methods interact with one another.

UML – Unified Modeling Language, a standard set of document formats for designing software graphically.

Wheel – A circular object that contains wedges with varying text. In Wheel of Fortune this wheel is spun by a contestant before he/she guesses a letter.

Wedges – Parts of the wheel. Most wedges contain price values, which are indicative of the money that will be added to a contestant's total based on the letter he/she guesses and the number of this letter present in the given phrase. Other wedges are "specialty" wedges; instead of a price they present a special opportunity; the "specialty" wedges are defined below.

"Bankrupt" Wedge – Causes a contestant to lose all of his/her money earned during that round and "specialty" tags earned during the entire game.

"Free Play" Wedge – Permits a contestant to guess the phrase or a letter with no penalty.

"Lose a Turn" Wedge – Contestant loses a turn.

"Buying a Vowel" – In Wheel of Fortune, consonants are free but vowels cost $250 each.

Category – Wheel of Fortune has various categories from which phrases are selected. Such categories include places, things, food and beverage etc.

1.4 References

IEEE Std 830^{TM}-1998 (R2009) – IEEE Standard for Information Technology – Systems Design – Software Design Descriptions

1.5 Overview

This Software Design Description document provides a working design for the computerized Wheel of Fortune software application. Section 2 of this document will provide the Package-Level Viewpoint, specifying the packages and frameworks to be designed. Section 3 will provide the Class-Level Viewpoint, using UML Class Diagrams to specify how the classes should be constructed. Section 4 will provide the Method-Level System Viewpoint, describing how methods will interact with one another. Section 5 provides deployment information like file structures and formats to use. Section 6 provides a Table of Contents, an Index, and References. Note that all UML Diagrams in this document were created using the VioletUML editor.

2 Package-Level Design Viewpoint

As mentioned, this design will encompass solely the Wheel of Fortune application. The design will aim to separate the GUI from the logic by allocating different responsibilities to different classes. In building the application we will heavily rely on the Java API to provide services. Following are descriptions of the components to be built, as well as how the Java API will be used to build them.

2.1 Wheel of Fortune overview

The GUI and logic portions of this project will be designed and developed in tandem. Figure 2.1 specifies all the components to be developed and places all classes in home packages.

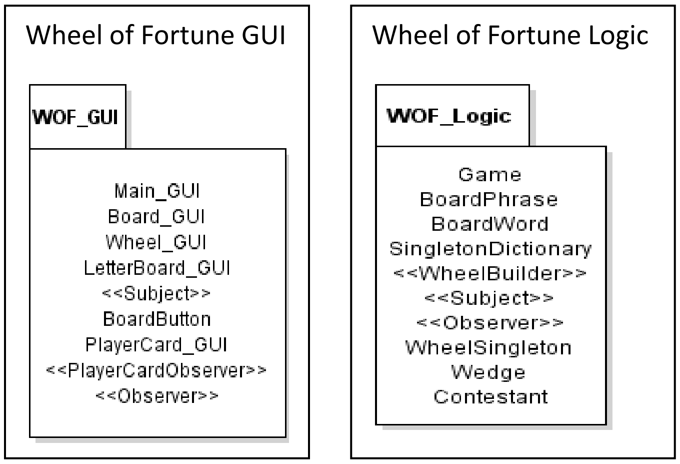


Figure 2.1: Design Packages Overview

Both the logic and GUI portions of the application will be developed using the Java programming languages. As such, this design will make use of the classes specified in Figure 2.2.

2.2 Java API Usage

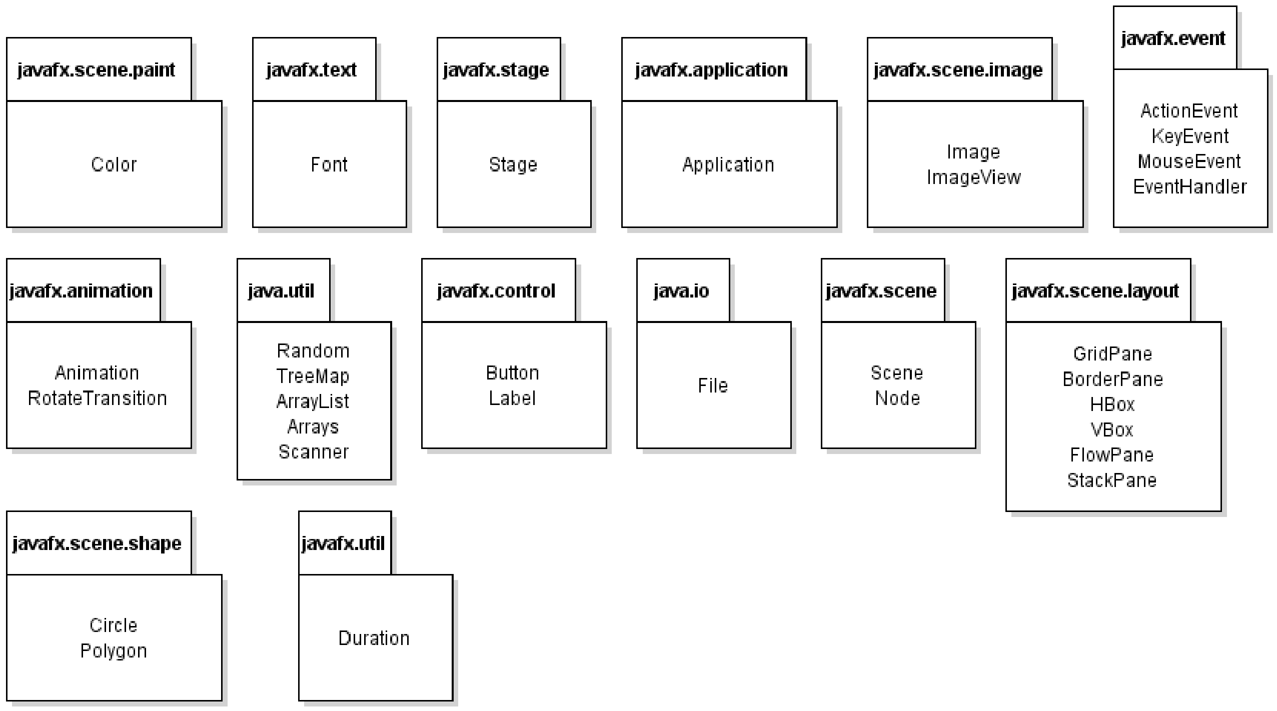


Figure 2.2: Java API Classes and Packages To Be Used

2.3 Java API Usage Descriptions

Tables 2.1-2.12 below summarize how each of these classes will be used.

Class/Interface	Use
Color	For setting the colors for text and for the power bar for spinning wheel

Table 2.1: Uses for classes in the Java API's javafx.scene.paint package

Class/Interface	Use
Font	For setting the fonts for rendered text

Table 2.2: Uses for classes in the Java API's javafx.text package

Class/Interface	Use
Application	Used as the entry point for javafx application; creates an application thread that will process input events and animation timelines

Table 2.3: Uses for classes in the Java API's javafx.application package

Class/Interface	Use
Stage	Used as the top level JavaFX container

Table 2.4: Uses for classes in the Java API's javafx.stage package

Class/Interface	Use
Scene	The container for all content in a scene graph
Node	Base class for scene graph nodes

Table 2.5: Uses for classes in the Java API's javafx.scene package

Class/Interface	Use
Image	For storing image data regarding the wheel to be used and the contestants

ImageView	For appropriately displaying and readjusting images like the wheel and the contestants

Table 2.6: Uses for classes in the Java API's java.scene.image package

Class/Interface	Use
GridPane	Layout to be used for designing the board containing the phrase
BorderPane	Layout to be used for the entire GUI containing the board, contestants, and wheel
HBox	Layout to be used for placing buttons in a row
VBox	Layout to be used for displaying scores of contestants on right side of screen
FlowPane	Layout that will include HBox layouts
StackPane	Layout to be used for wheel and the label that will be in the center of the wheel to show the name of the wedge that has been landed on

Table 2.7: Uses for classes in the Java API's javafx.scene.layout package

Class/Interface	Use
ActionEvent	For getting information about an action event like which button was pressed.
KeyEvent	For getting information about a key event, like which key was pressed.
MouseEvent	For getting information about a mouse event, like where was the mouse pressed?
EventHandler	For responding to an action, key, or mouse event, like a button, key, or mouse button press. We will provide our own custom implementation of this interface.

Table 2.8: Uses for classes in the Java API's javafx.event package

Class/Interface	Use

File	To access File that will then be read using Scanner

Table 2.9: Uses for classes in the Java API's java.io package

Class/Interface	Use
Animation	Need to use Animation class to retrieve status of an animation to disable buttons for a certain period of time
RotateTransition	For giving an object animation. Will be used to provide the wheel with a motion to follow.

Table 2.10: Uses for classes in the Java API's javafx.animation package

Class/Interface	Use
Button	Simplifies user interaction; will need a button for spinning the wheel, for a help button, and for guessing
Label	Will be used for displaying contestant's names and faces and other text in GUI

Table 2.11: Uses for classes in the Java API's javafx.control package

Class/Interface	Use
Random	For generating random values when spinning the wheel
TreeMap	For storing (letter, Map<Word length, words>) key pairs, we'll use it for storing our words in an attempt to provide contestants with an accurate letter probability predictor.
Arrays	For creating an ArrayList from a list using the Arrays.asList method
Scanner	For parsing/going through text files that contain words and random phrases
ArrayList	Perhaps the most used data structure used. Will be used to store everything from Characters to Buttons.

Table 2.12: Uses for classes in the Java API's java.util package

Class/Interface	Use
Duration	Will be used in tandem with RotateTransition.

Table 2.13: Uses for classes in the Java API's javafx.util package

Class/Interface	Use
Circle	Will be used stylistically; the spinning wheel will be put on top of a circle.
Polygon	Will be used to make a triangle that will be placed at the edge of the wheel to act as an indicator for which wedge the wheel has landed on

Table 2.14: Uses for classes in the Java API's javafx.scene.shape package

Class-Level Design Viewpoint

As mentioned, the design presented in this document will encompass the Wheel of Fortune GUI and Logic. The following UML Class Diagrams aim to show this. We will present the class designs using a series of diagrams from an overview to a more detailed perspective.

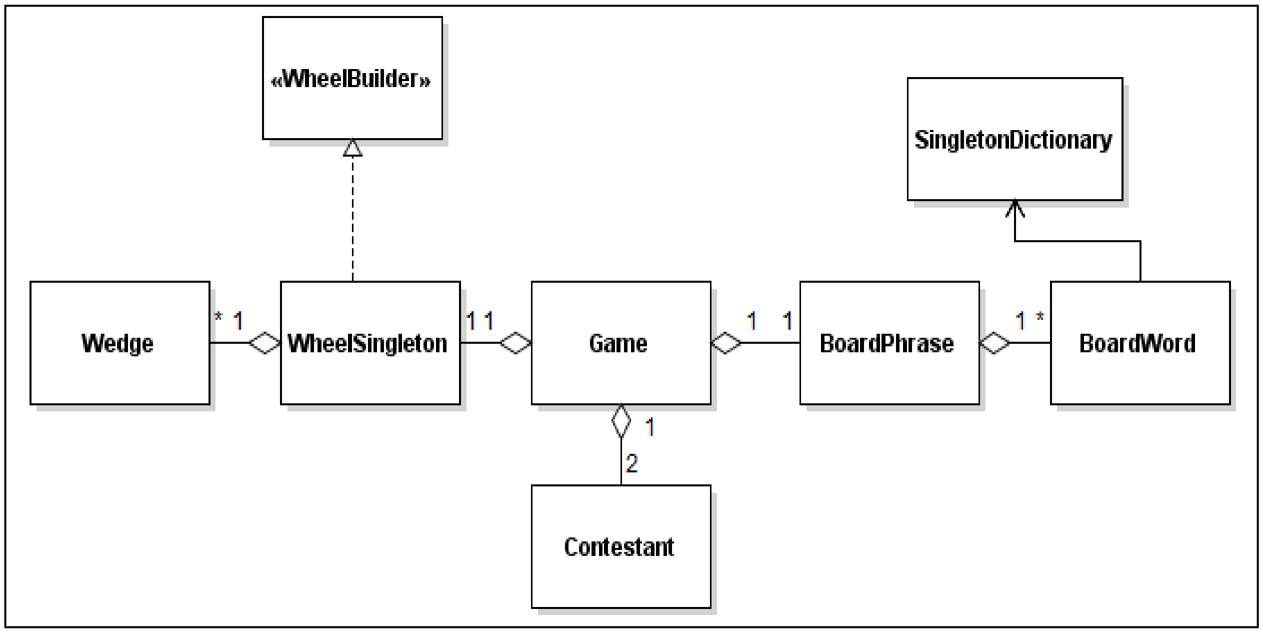


Figure 3.1: Wheel of Fortune Logic overview

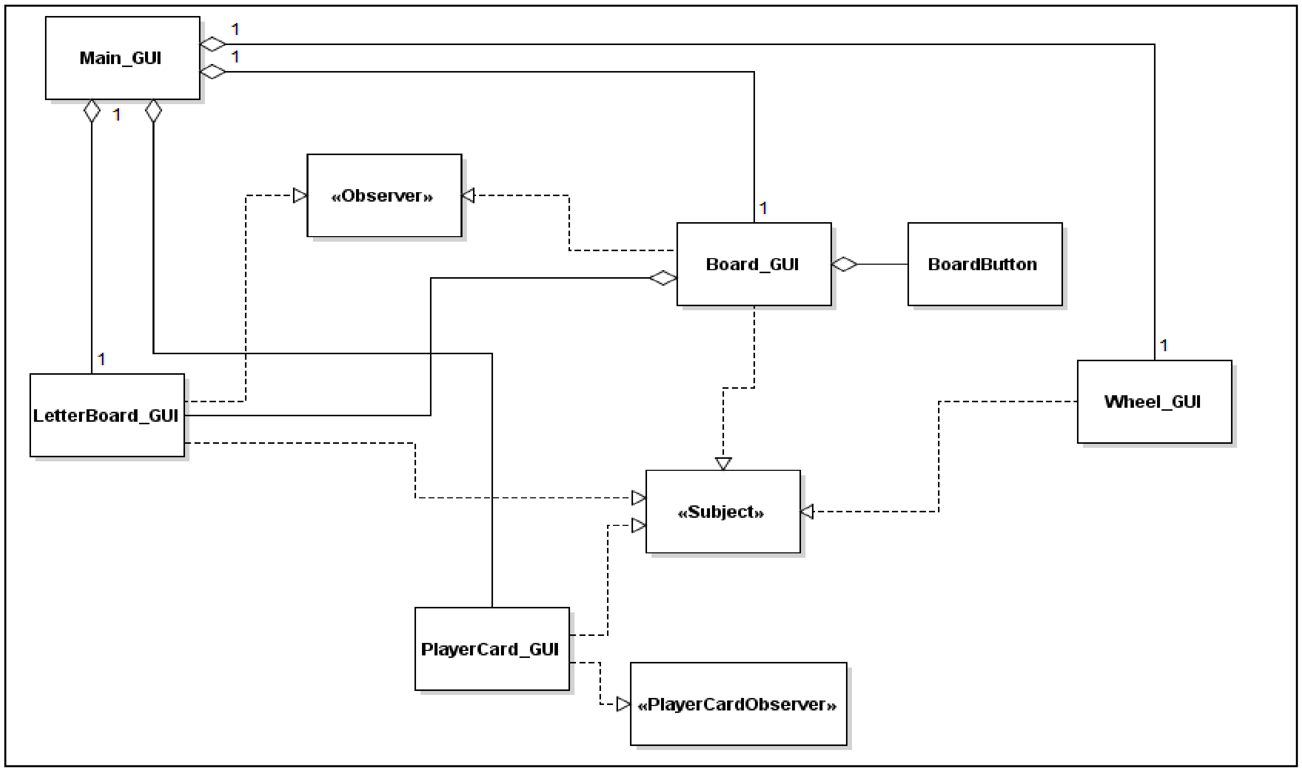


Figure 3.2: Wheel of Fortune GUI Overview UML Class Diagram

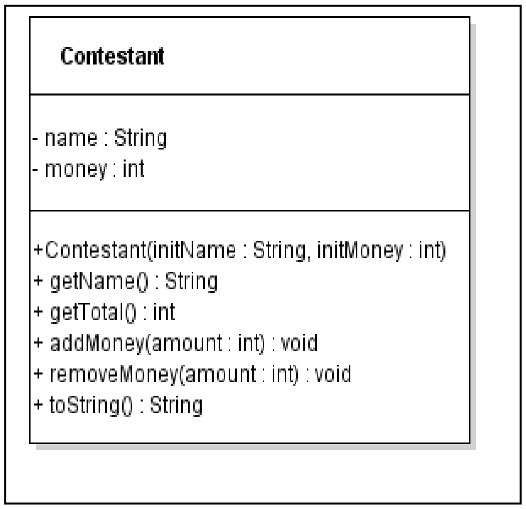

Contestant
- name : String
- money : int
+Contestant(initName : String, initMoney : int)
+ getName() : String
+ getTotal() : int
+ addMoney(amount : int) : void
+ removeMoney(amount : int) : void
+ toString() : String

Figure 3.3: Detailed Contestant UML Class Diagram (Logic section)

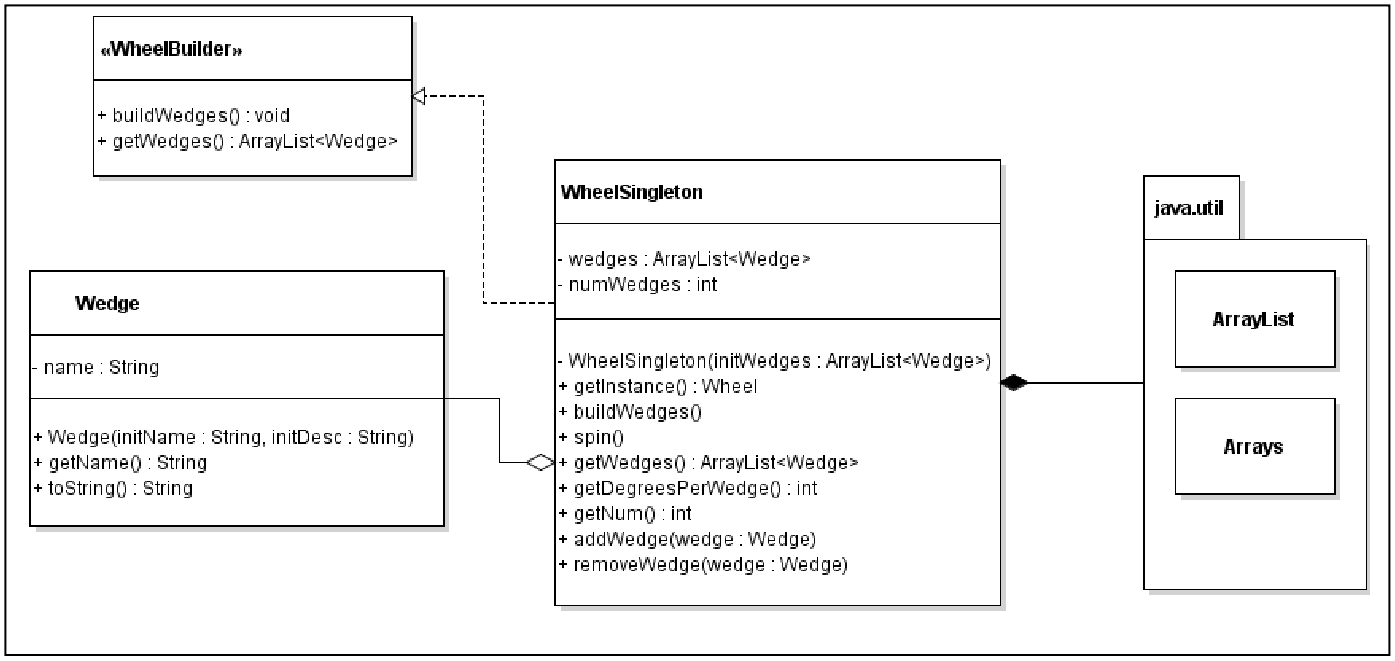

«WheelBuilder»
+ buildWedges() : void
+ getWedges() : ArrayList<Wedge>
WheelSingleton
- wedges : ArrayList<Wedge>
- numWedges : int
- WheelSingleton(initWedges : ArrayList<Wedge>)
+ getInstance() : Wheel
+ buildWedges()
+ spin()
+ getWedges() : ArrayList<Wedge>
+ getDegreesPerWedge() : int
+ getNum() : int
+ addWedge(wedge : Wedge)
+ removeWedge(wedge : Wedge)
Wedge
- name : String
+ Wedge(initName : String, initDesc : String)
+ getName() : String
+ toString() : String
java.util
ArrayList
Arrays

Figure 3.4: Detailed WheelBuilder, WheelSingleton, and Wedge UML Class Diagram

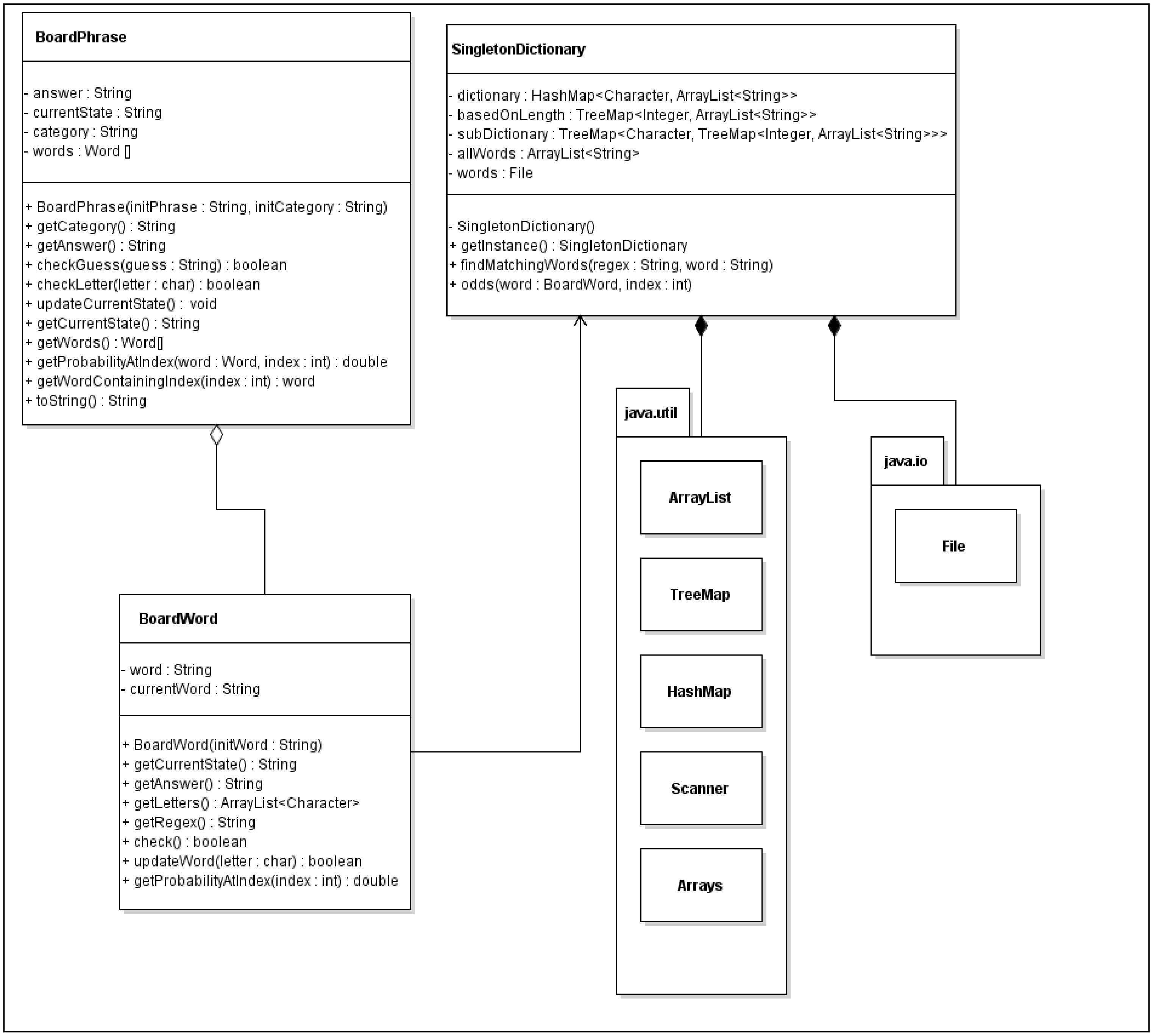

BoardPhrase
- answer : String
- currentState : String
- category : String
- words : Word []
+ BoardPhrase(initPhrase : String, initCategory : String)
+ getCategory() : String
+ getAnswer() : String
+ checkGuess(guess : String) : boolean
+ checkLetter(letter : char) : boolean
+ updateCurrentState() : void
+ getCurrentState() : String
+ getWords() : Word[]
+ getProbabilityAtIndex(word : Word, index : int) : double
+ getWordContainingIndex(index : int) : word
+ toString() : String
SingletonDictionary
- dictionary : HashMap<Character, ArrayList<String>>
- basedOnLength : TreeMap<Integer, ArrayList<String>>
- subDictionary : TreeMap<Character, TreeMap<Integer, ArrayList<String>>>
- allWords : ArrayList<String>
- words : File
- SingletonDictionary()
+ getInstance() : SingletonDictionary
+ findMatchingWords(regex : String, word : String)
+ odds(word : BoardWord, index : int)
BoardWord
- word : String
- currentWord : String
+ BoardWord(initWord : String)
+ getCurrentState() : String
+ getAnswer() : String
+ getLetters() : ArrayList<Character>
+ getRegex() : String
+ check() : boolean
+ updateWord(letter : char) : boolean
+ getProbabilityAtIndex(index : int) : double
java.util
ArrayList
TreeMap
HashMap
Scanner
Arrays
java.io
File

Figure 3.5: Detailed BoardPhrase, BoardWord, and SingletonDictionary UML Class Diagram

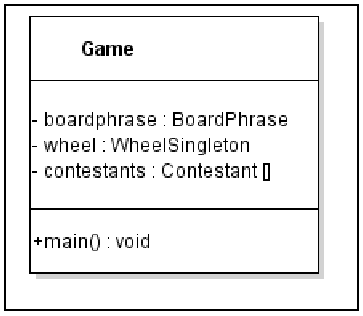
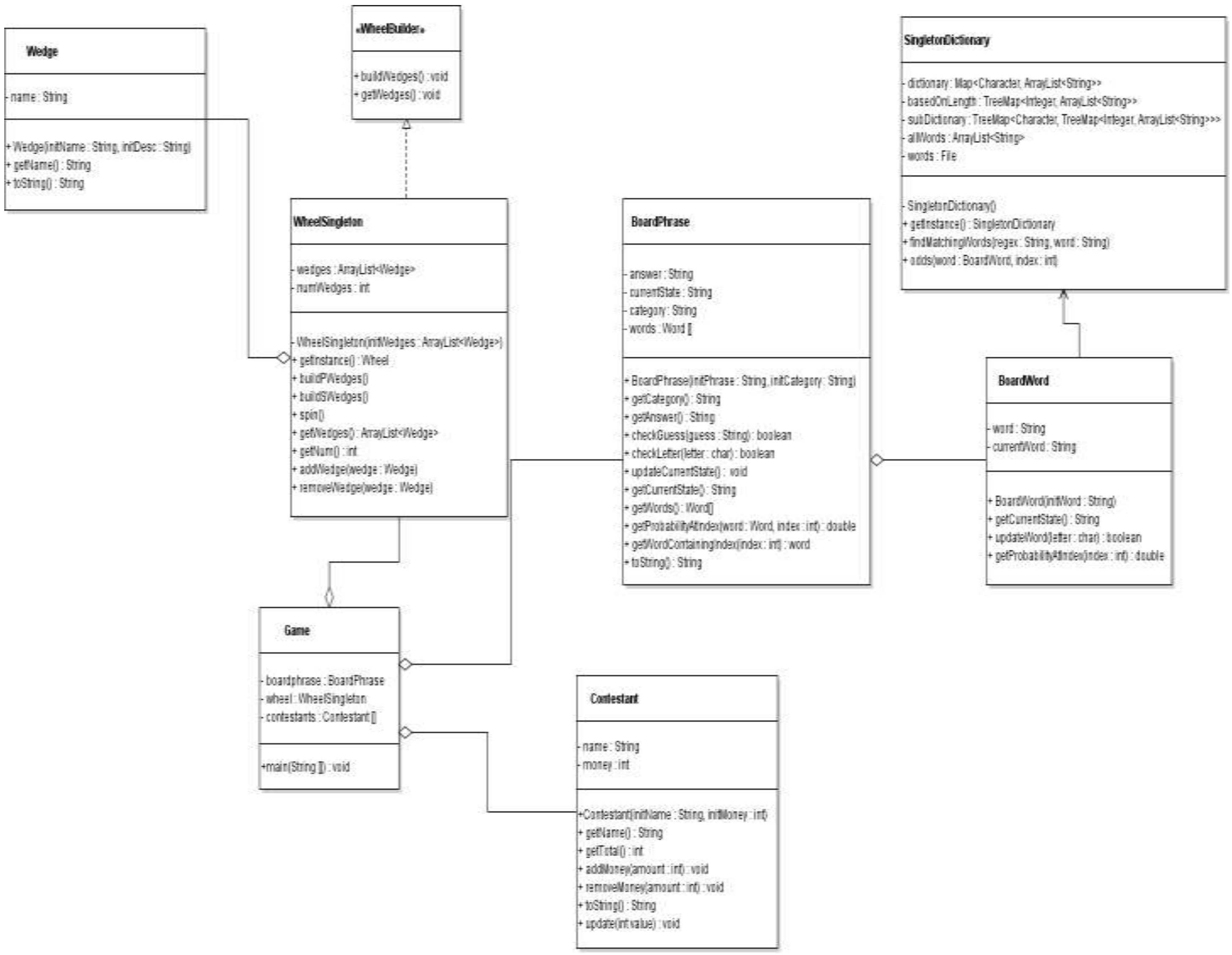
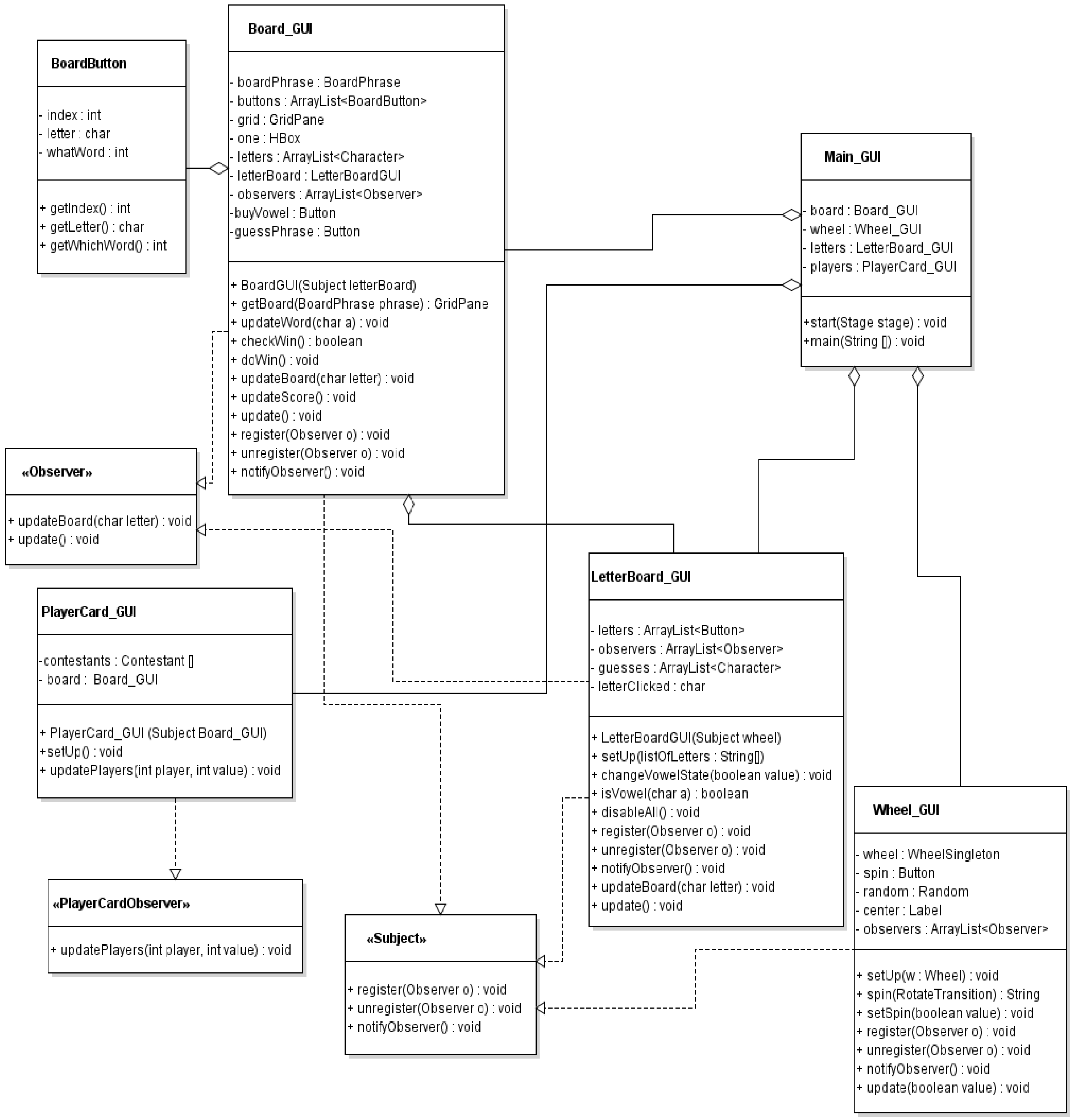


Figure 3.6: Detailed Game UML Class Diagram (focal point/main)

Final Logic Big Picture:


Wedge
- name : String
+ Wedge(initName : String, initDesc : String)
+ getName() : String
+ toString() : String
«WheelBuilder»
+ buildWedges() : void
+ getWedges() : void
WheelSingleton
- wedges : ArrayList<Wedge>
- numWedges : int
- WheelSingleton(initWedges : ArrayList<Wedge>)
+ getInstance() : Wheel
+ buildPWedges()
+ buildSWedges()
+ spin()
+ getWedges() : ArrayList<Wedge>
+ getNum() : int
+ addWedge(wedge : Wedge)
+ removeWedge(wedge : Wedge)
BoardPhrase
- answer : String
- currentState : String
- category : String
- words : Word []
+ BoardPhrase(initPhrase : String, initCategory : String)
+ getCategory() : String
+ getAnswer() : String
+ checkGuess(guess : String) : boolean
+ checkLetter(letter : char) : boolean
+ updateCurrentState() : void
+ getCurrentState() : String
+ getWords() : Word[]
+ getProbabilityAtIndex(word : Word, index : int) : double
+ getWordContainingIndex(index : int) : word
+ toString() : String
SingletonDictionary
- dictionary : Map<Character, ArrayList<String>>
- basedOnLength : TreeMap<Integer, ArrayList<String>>
- subDictionary : TreeMap<Character, TreeMap<Integer, ArrayList<String>>>
- allWords : ArrayList<String>
- words : File
- SingletonDictionary()
+ getInstance() : SingletonDictionary
+ findMatchingWords(regex : String, word : String)
+ odds(word : BoardWord, index : int)
BoardWord
- word : String
- currentWord : String
+ BoardWord(initWord : String)
+ getCurrentState() : String
+ updateWord(letter : char) : boolean
+ getProbabilityAtIndex(index : int) : double
Game
- boardphrase : BoardPhrase
- wheel : WheelSingleton
- contestants : Contestant []
+main(String []) : void
Contestant
- name : String
- money : int
+Contestant(initName : String, initMoney : int)
+ getName() : String
+ getTotal() : int
+ addMoney(amount : int) : void
+ removeMoney(amount : int) : void
+ toString() : String
+ update(int value) : void

GUI Breakdown



Board_GUI
- boardPhrase : BoardPhrase
- buttons : ArrayList<BoardButton>
- grid : GridPane
- one : HBox
- letters : ArrayList<Character>
- letterBoard : LetterBoardGUI
- observers : ArrayList<Observer>
-buyVowel : Button
-guessPhrase : Button
+ BoardGUI(Subject letterBoard)
+ getBoard(BoardPhrase phrase) : GridPane
+ updateWord(char a) : void
+ checkWin() : boolean
+ doWin() : void
+ updateBoard(char letter) : void
+ updateScore() : void
+ update() : void
+ register(Observer o) : void
+ unregister(Observer o) : void
+ notifyObserver() : void
BoardButton
- index : int
- letter : char
- whatWord : int
+ getIndex() : int
+ getLetter() : char
+ getWhichWord() : int
Main_GUI
- board : Board_GUI
- wheel : Wheel_GUI
- letters : LetterBoard_GUI
- players : PlayerCard_GUI
+start(Stage stage) : void
+main(String []) : void
«Observer»
+ updateBoard(char letter) : void
+ update() : void
LetterBoard_GUI
- letters : ArrayList<Button>
- observers : ArrayList<Observer>
- guesses : ArrayList<Character>
- letterClicked : char
+ LetterBoardGUI(Subject wheel)
+ setUp(listOfLetters : String[])
+ changeVowelState(boolean value) : void
+ isVowel(char a) : boolean
+ disableAll() : void
+ register(Observer o) : void
+ unregister(Observer o) : void
+ notifyObserver() : void
+ updateBoard(char letter) : void
+ update() : void
PlayerCard_GUI
-contestants : Contestant []
- board : Board_GUI
+ PlayerCard_GUI (Subject Board_GUI)
+setUp() : void
+ updatePlayers(int player, int value) : void
Wheel_GUI
- wheel : WheelSingleton
- spin : Button
- random : Random
- center : Label
- observers : ArrayList<Observer>
+ setUp(w : Wheel) : void
+ spin(RotateTransition) : String
+ setSpin(boolean value) : void
+ register(Observer o) : void
+ unregister(Observer o) : void
+ notifyObserver() : void
+ update(boolean value) : void
«PlayerCardObserver»
+ updatePlayers(int player, int value) : void
«Subject»
+ register(Observer o) : void
+ unregister(Observer o) : void
+ notifyObserver() : void

Figure 3.8: Detailed GUI related UML Class Diagrams

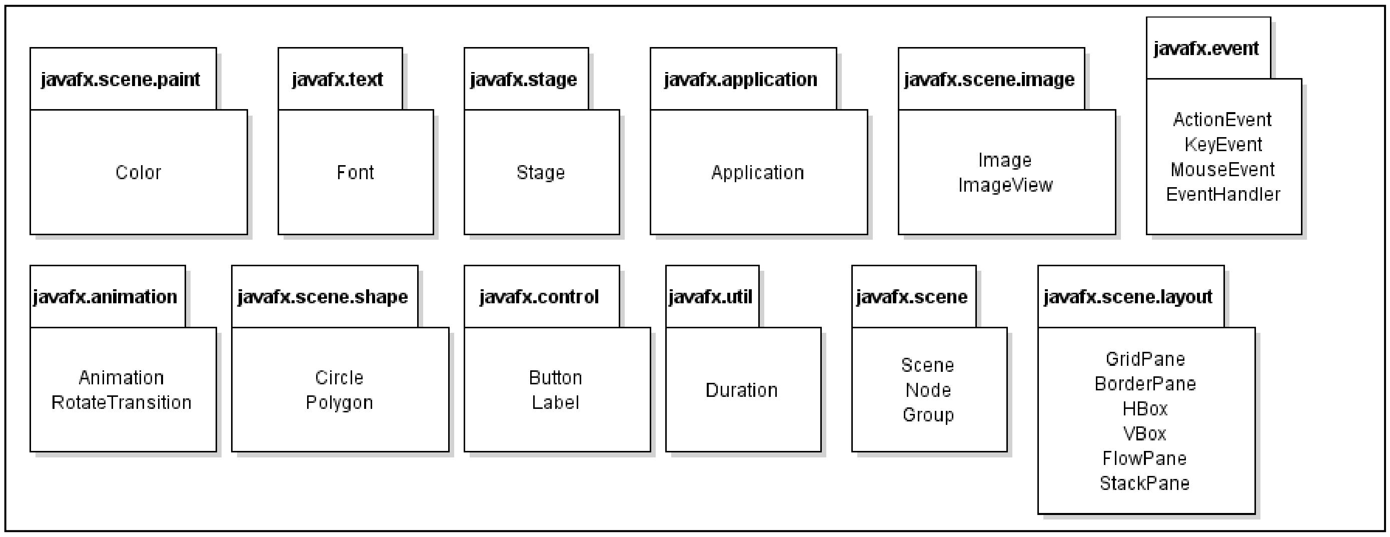


Figure 3.8: Packages to be used by GUI classes. More elaborate explanations provided in section 2.3.

3 Method-Level Design Viewpoint

Now that the general architecture of the classes has been determined, it is time to specify how data will flow through the system. The following explanations and UML Sequence Diagrams describe the methods called within the code to be developed in order to provide the appropriate event responses.

A major design pattern used throughout the program is the Observer design pattern. As such some classes serve as subjects whereas others serve as observers and are notified of changes in the subject and updated accordingly. It is important to understand the overlapping subject-observer relationships in this application.

1. **LetterBoardGUI – BoardGUI –** in this case the board GUI will be updated once a user clicks on a letter on the letter board; the LetterBoard GUI serves as the subject and the Board GUI is the observer.
2. **WheelGUI – LetterBoardGUI –** in this case the letter board GUI will be updated once the user clicks the spin button (the letter buttons will be enabled; one can only select a letter after spinning assuming he/she does not land on bankrupt or lose a turn).
3. **BoardGUI – PlayerCardGUI –** in this case the player card will be updated when the board is updated. If a letter is correctly guess the player card will be updated to reflect the updated score of a player. If an incorrect letter is guess the player card will be updated to reflect the change in turns.

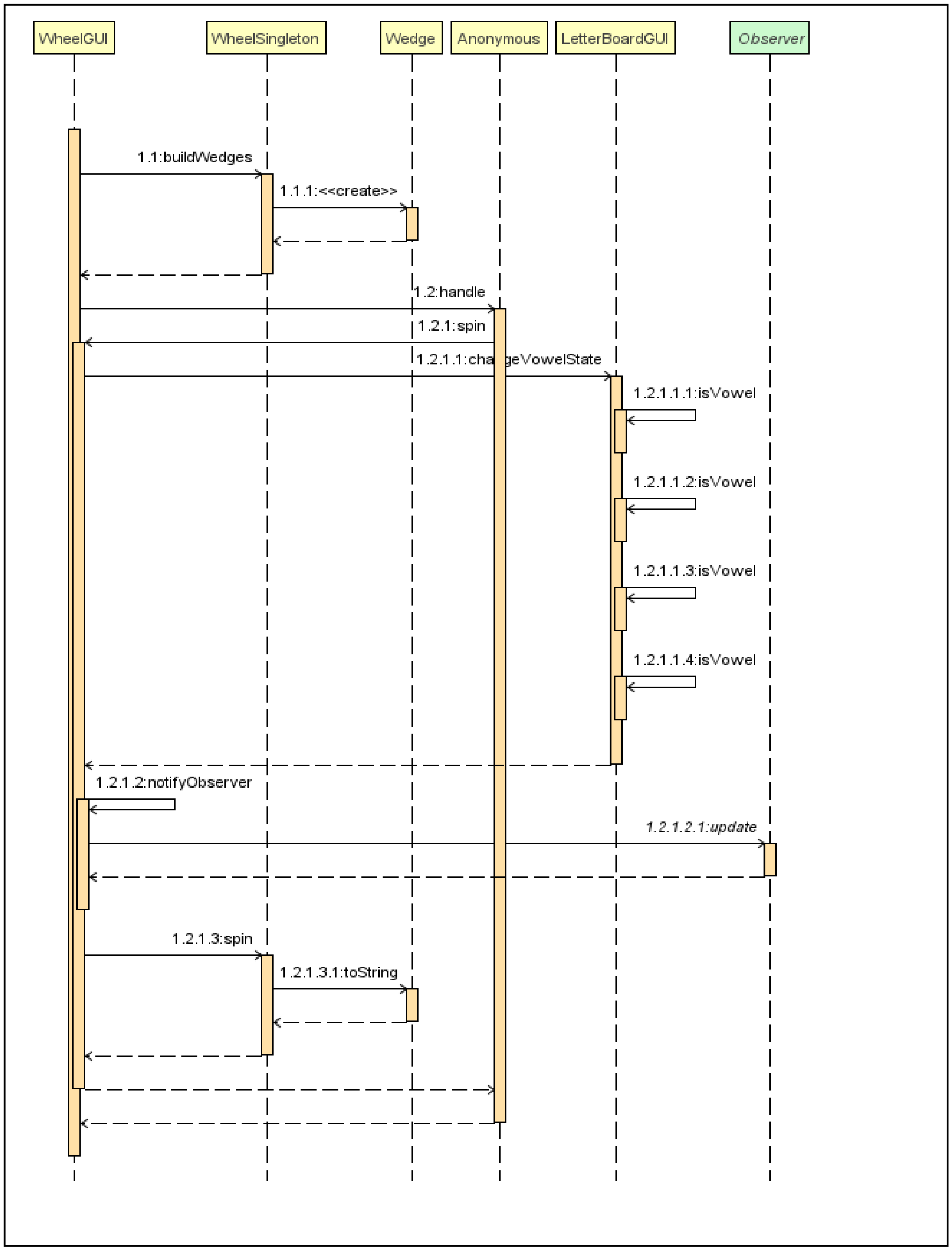


Figure 4.1: UML Sequence Diagram for the set up method in WheelGUI. The set up method initializes the GUI for the wheel. The first handle method call between Wedge and Anonymous inner class object represents the result of clicking spin (wheel spins). This results in changeVowelState() being called which activates all consonants on the letter board and disables all vowels. The wheel then notifies its other observer, the main BoardGUI so that the buy vowel and guess phrase buttons can be enabled (these buttons must be disabled until a contestant has spun the wheel).

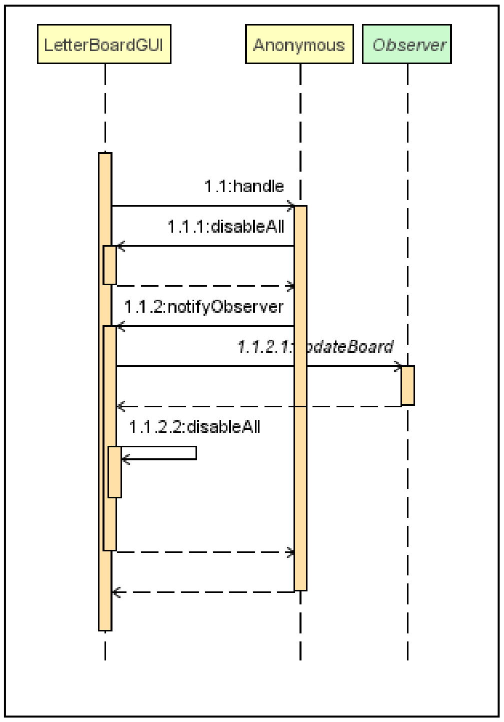


Figure 4.2: UML Sequence Diagram for the set up method in LetterBoardGUI. The set up method initializes the GUI for the letter board. The handle method call between LetterBoardGUI and the Anonymous inner class object represents the result of clicking a letter on the letter board. This results in disableAll() being called which disables all letters (this is necessary to ensure that a contestant cannot repeatedly guess letters without first spinning). The letter board then notifies its other observer, the main BoardGUI so that the buy vowel and guess phrase buttons can be enabled (these buttons must be disabled until a contestant has spun the wheel and guessed at least one letter).

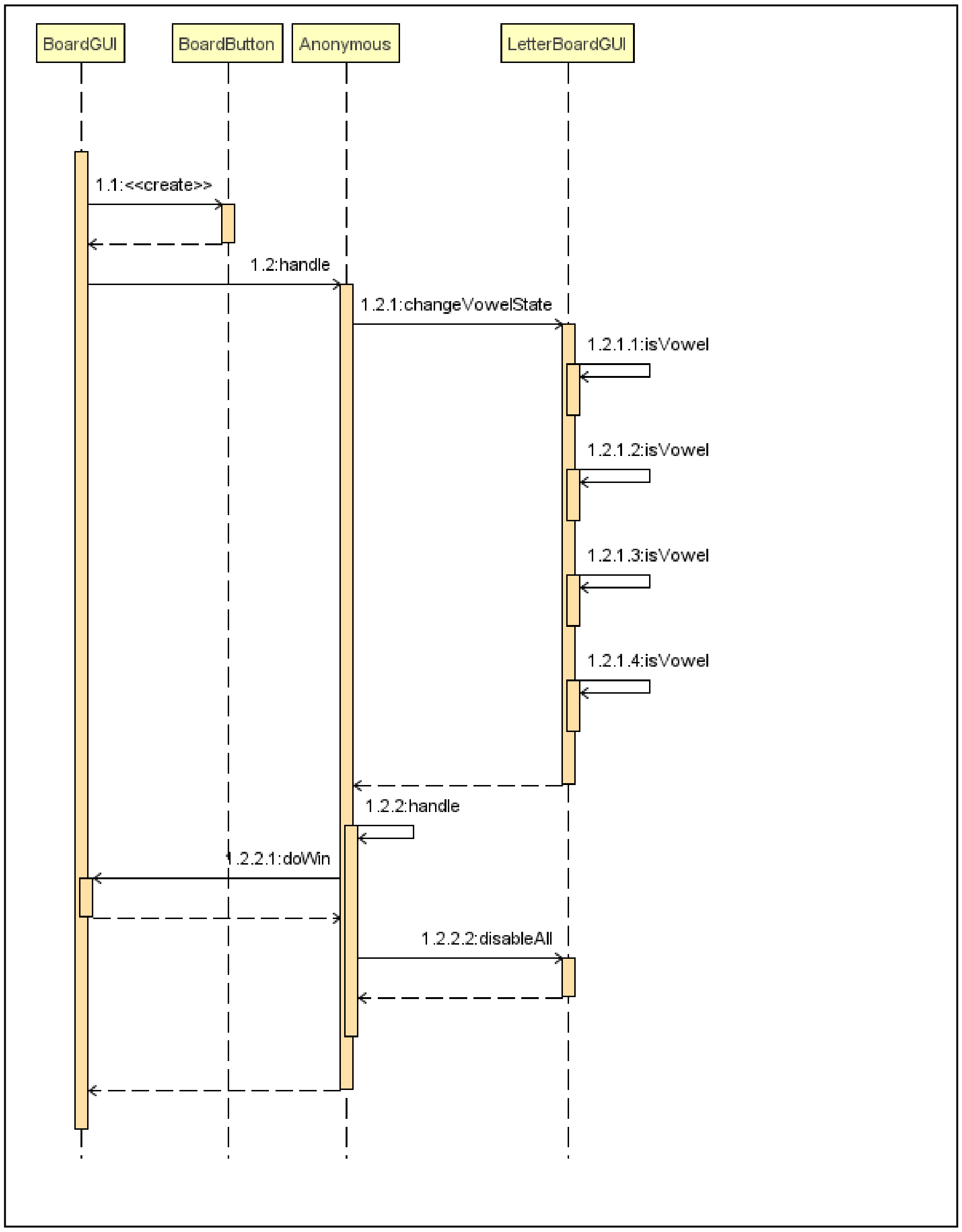


Figure 4.3: UML Sequence Diagram for the getBoard() method in BoardGUI. The getBoard() method initializes and returns the GUI for the main board. The handle method call between BoardGUI and the Anonymous inner class object represents the result of clicking the "Buy a Vowel" button. This results in changeVowelState() being called which disables all consonants and enables all remaining vowels so that a contestant can choose amongst the vowels. The second handle method in the Anonymous inner class object is the handle method for the "Guess Phrase" button. Clicking on the button prompts the user to enter his/her guess and then this guess is checked to see if it is correct. If it is, doWin() is performed. If not, disableAll() is called to signal the end of this player's turn and disable the letter board until the next player spins the wheel.

5. File Structure and Formats

The Wheel of Fortune application and will contain a single, executable JAR file titled WheelOfFortune.jar. Note that all necessary text and art files must accompany this program. Figure 5.1 specifies the necessary file structure the launched application should use.

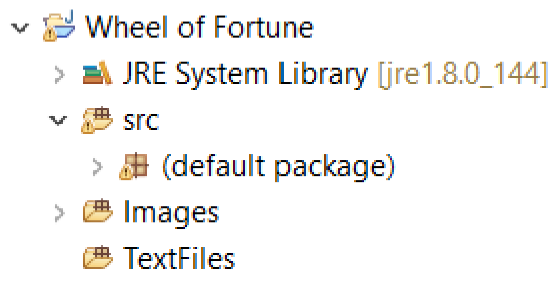


Figure 5.1: Wheel of Fortune File Structure

Wheel of Fortune folder with 5 text files each text file representing a different category. There will also be a text file that will act as a dictionary that contains a majority of words in the English language. This file will be accessed in order to accurately calculate the letter probabilities at given indices. These files will go under the TextFiles source folder in the Wheel of Fortune Java Project. The Images source folder will contain the image of the wheel to be used. All java files will be stored in the default package in the src folder.

6. Supporting Information

Note that this document should serve as a reference for those implementing the code, so we'll provide a table of contents to help quickly find important sections.

6.1 Table of contents

6.2 Appendixes

N/A