CS2011 Final Project Report

Introduction:

This project is a Lexical Decision Task (LDT). The purpose of this task is to create a game where the user must decide if a set of 4 words is belonging to the English language or not. The game must have 6 rounds and the user is given random word lists from the word dictionary each round. There were 2 main classes used to create this task, “WordFileReader” and “TrialParticipant”. The classes were then imported into a main file called “main.py”.

WordFileReader Class:

The WordFileReader is a child class of the parent class FileReader. This is an example of inheritance.

```
class WordFileReader(FileReader):

    def display_filename(self):
        print(f"the file name is : {self.get_filename}")
```

Inheritance is a process of object oriented programming where one class (the child class) inherits or gains access to all the public methods and attributes of another (parent) class. Here the FileReader class is being inherited by WordFileReader, which allows the FileReader method “get_filename” to be called and used in the WordFileReader method. The

The “all_words” method in WordFileReader uses the FileReader parent class to call the “read_all” method from FileReader to read a text file into the method and store it as a variable. The purpose of this method is to transform the text file into the correct nested dictionary format required for the LDT. Encapsulation in this class enhanced the usability and modularity of the code. Encapsulation is a key principle of object oriented design and involves containing data (attributes) and the methods that use the data in a single unit, a class. Examples of encapsulation in the WordFileReader class include the “get_round_range”, as there is controlled data access.

```
class WordFileReader(FileReader):
    def get_round_range(self, ran_rounds):
            if round_range_start >= 0 or round_range_end >= 0: #ensures list para
                if len(ran_rounds) == 2: # ensures list total is 2 so theres only
                    if ran_rounds[0] < ran_rounds[1]: # ensures start value is le
                        for key in self.full_dict:
                            if int(key) in round_range:  #if the key is in the ra
                                count += 1
                                round_range_dict[count] = self.full_dict[key] #ac
                        #print(round_range_dict)
                        return(round_range_dict)
```

The attribute “self.full_dict” is accessed and operated on within the “get_round_range” method, with its usage restricted to the method only.

Method overriding is when a method inside a child class has the same name as a method in the parent class. The parent method is then overridden by the newer child method. Method

overriding can increase allow for custom behaviour of methods, where child classes can modify methods from the parent class without changing the parent class code. Method overriding was not implemented in this class as it was not necessary to call any parent method functions in the child class. This is because the WordFileReader class did not need to use the parent methods for a different function to what they were defined as in the parent class (reading the file).

TrialParticipant Class:

TrialParticipant is the class that coordinates and runs the LDT, using the word dictionary from the "all_words" method of the WordFileReader class. The class constructor holds the instance variables for the users selections, word lists, round positions and correct, incorrect choices etc. This class stores and return the correct and incorrect values (get.correct() and get.incorrect()) and computes the users choices and evaluates whether it's the right answer in the "response()" method.

```
def response(self, selection):
    type = self.word_type[self.selection] # english or non english chosen from the list "word_type" randomly
    #key_association = self.possible_selections[selection] # stores value of the right answer
    if selection == "y":  # if selection is yes then the key is english
        key = "english"
    else:                 # if no then key is non english
        key = "non-english"
    if type == key:  # if word type is englsih then its correct return true
        print("The participant selected: %s" % selection)
        #print("the current word type is: %s" % key)
        #print("The correct response for this was: %s" % type)
        return True
    else:
        print("The participant selected: %s" % selection)
        #print("the current word type is: %s" % key)
        #print("The correct response for this was: %s" % type)
        return False
```

The "set.words" method is an example of a method in TrialParticipant that implements encapsulation. This method restricts access to data and provides a controlled way to set the words of the LDT, instead of directly changing the hardcoded dictionary of words from outside the class. The attributes "self.position" and "self.(in)correct_choices" are managed and handled entirely in the method and protected from external modification.

The "set_words" method is shown below, with comments explaining its functionality.

```
def set_words(self, new_words):
    self.position = 1 #restarts game with new words
    self.incorrect_choices = 0 # resets correct and incorrect counters
    self.correct_choices = 0
    invalid = False  # check for debug
    if type(new_words) == dict: # verifies if word set is a dictionary
        for key, value in new_words.items(): # loops through the keys in the dictionary
            #key = int(key)
            if type(key) == int and type(value) == dict:  # ensures keys are integers and values are a dictionary to c
                for nested_key, nested_value in value.items(): # loops through the nested keys and values in the value
                    if type(nested_key) == str and type(nested_value) == list and (len(nested_value) >=2): # ensures n
                        invalid = False
                    else:
                        invalid = True
                        break
            else:
                invalid = True
    else:
        invalid = True
    if invalid == True:
        print("invalid set of words")
    elif invalid == False:
        TrialParticipant.words = new_words # sets the game words as the new words
        #print(new_words)
```

```
134   words.items(): # loops through the keys in the dictionary
135
136   t and type(value) == dict:  # ensures keys are integers and values are a dictionary to conform to nested dict format
137   , nested_value in value.items(): # loops through the nested keys and values in the value dictionary
138   sted_key) == str and type(nested_value) == list and (len(nested_value) >=2): # ensures nested key is string and nested value
139                                                 # is a list and that there is at least 2 values in the lists.
140   d = False
```

Main.py:

The main.py file is where the instance of the TrialParticipant and WordFileReader class is created and the game is run using a while loop under the condition of the number of rounds, (6 determined by the “increment position” method in TrialParticipant). The main file is a good example of object oriented programming (OOP) as it creates an instance of two modular classes in an isolated file using the import feature, and no knowledge of either class’s inner workings is needed to run the main file, this principle is called abstraction. The main file greets the user in a user-friendly format and lists user-relevant data to the user throughout the game, keeping them informed on the LDT’s progress. This includes a welcome message, the users name and their consent to participate. It also updates the player on their correct answers and the round position.

In this file WordFileReader is interacting with the TrialParticipant class to set the set of words created in WordFilereader into the LDT using the “set_words” method of the TrialParticpant class. This file allows the user to pass in a text file and transform it to a dictionary and play an instance of the LDT with their own name and the new set of words from the text file. This is an example of code reuse where the methods do not need to be coded again and can be called by the user.

Feature 1:

This feature asked the player if they wish to change the set of words. If yes, then the code creates a new instance of the WordFileReader with the new text file and uses the “all_words” and “set_words” methods to implement the new dictionary.

```
        # Additional Feature 1: Changing word set
        change_words = str(input("Would you like to change the words? (y/n): ")) # allows users to change word type, allo
        if change_words == "y":
            file_2 = WordFileReader("new_words.txt") # creates new instance with new word file so the class methods can b
            changed_dict = file_2.all_words()
            trial_participant.set_words(changed_dict)
            run_game = True
        else:
            pass
```

This feature uses encapsulation well. There is good code reuse here with the “all_words” and “set_words” methods called and used from the main classes. Inheritance was not implemented for this feature as there was no need to call any methods from the parent class in the child class for the methods to run using the new instance data. Method overriding was not necessary for this feature as each method was defined uniquely and did not need a separate class to perform these functions more efficiently. Object interactions took place in the form of having the “changed_dict” variable operated on by the “set_words” method.

Feature 2:

The second additional feature added a hard mode setting to the LDT which asks the user if they would like to make it more difficult after they achieve 3 correct answers. This hard mode involves inverting the answer keys “y” and “n” so it adds a layer of difficulty when choosing the right answer.

```
else:
    hard = False
    # Additional Feature 2: Changing response keys after 3 correct guesses
    if trial_participant.get_correct() == 3 and hard != True:
        hard_mode = (input("Would you like to make it harder? (y/n): "))
        if hard_mode == "y":
            hard = True
            test = trial_participant.updated_response(participant_selection) # runs the new response function to change the respo
            if test == True:
                trial_participant.increment_correct()  # calls methods to keep track of incorrect or correct choices
            elif test == False:
                trial_participant.increment_incorrect()

        else:
            pass
    else:
        hard = False
        test = trial_participant.response(participant_selection) # runs the response function to test the users response
        if test == True:
            trial_participant.increment_correct()  # calls methods to keep track of incorrect or correct choices
        elif test == False:
            trial_participant.increment_incorrect()

    next = trial_participant.increment_position() # increments position to continue game
```

If the player selects hard mode it calls an updated version of the “response” method in the TrialParticipant class with the answer keys swapped, shown below.

```
def updated_response(self, selection):
    type = self.word_type[self.selection] # english or non english chosen from the list "word_type" randomly
    #key_association = self.possible_selections[selection] # stores value of the right answer
    if selection == "n":  # if selection is no then the key is english
        key = "english"
    else:                 # if yes then key is non english
        key = "non-english"
    if type == key:  # if word type is englsih then its correct return true
        print("The participant selected: %s" % selection)
        return True
    else:
        print("The participant selected: %s" % selection)
        return False
```

This method makes use of encapsulation with the data and being operated on in a secure contained environment and imports instance variable “self.selection” which is only accessed within the method. This method returns a Boolean value which is then evaluated in the feature code.