Course: CSC340.03

Student: Mark Kim

Instructor: Duc Ta

Assignment Number: 01

Due Date & Time: 09-15-2019 at 11:55 PM

Assignment 01

PART A – Class Design Guidelines

Cohesion

Classes should be designed in a manner where all elements (characteristics and behaviors) should support a single coherent purpose. Essentially, a class should be focused and specialized and describe a single element. In the case of our current assignment, a poorly designed class would combine players, general manager, president, and student into a single class. Since each have different roles, characteristics, and behaviors, it is appropriate to separate each into separate classes.

High cohesion in a class allows the class to be more easily maintained and improves reusability. Maintenance is simplified because each class has a focused role and its properties and methods are aligned with that role. In an incohesive class, methods and properties of the class will be harder to understand since its role will be unclear. Reusability improves because the focused nature of the class allows it to be inserted where needed. For example, a class dedicated to displaying output can be reused throughout a program to display anything as needed. Since the role is specific and focused, when the mechanism for displaying output needs to be modified, making those changes is a straightforward affair. If its role were not clear,

identifying the changes that need to be made to a class becomes hard to decipher, making maintenance more difficult.

Consistency

As with any code, it is important to remain consistent within and throughout classes. Just like any other portion of a program's code, generally accepted naming conventions and style should be adopted and maintained within a class as it is outside of the class. Liang notes that typically, the order with which elements are placed is as follows:

1. Data declaration
2. Constructors
3. Methods

Additionally, data, methods, and classes should be descriptive. Likewise, similar functions should share names as illustrated below:

```
class Teacher {
	public displayName() {
		System.out.print(name);
	}
}

class Student {
	public displayName() {
		System.out.print(name);
	}
}
```

Consistent

```
class Teacher {
	public displayName() {
		System.out.print(name);
	}
}

class Student {
	public displayFirstName() {
		System.out.print(name);
	}
}
```

Inconsistent

To preserve consistency, providing a public no-arg constructor should be compulsory unless one is not supported, in which case a reason should be clearly stated and documented. By providing a public no-arg constructor, clarity is improved and testing can be simplified.

Encapsulation

Encapsulation is the concept of hiding information from the rest of the program which is a fundamental of object-oriented programming. By hiding the implementation of a class, its contents are protected from access by other parts of the program. Access to the data is now restricted and is only made accessible to the rest of the program with getters and setters (if access is necessary). As can be seen below, access to information and methods can be restricted and access granted by getters and setters:

```
class Car {
        private double ispeed = 0; //access only within class
        private double fspeed = 0; //access only within class
        public void setispeed(double s) { //setter with outside access
            ispeed = s;
            setfspeed(); //calls method within class
        }
        private void setfspeed() { //access only within class
            fspeed = ispeed * 2;
        }
        public double getfspeed() { // getter with outside access
            return fspeed;
        }
}
```

In the example above, access to the method setfspeed() is restricted to the class that contains it which allows the method setispeed(double s) to call it since they are both members of the same class.

There are several purposes for using encapsulation: security, flexibility, reusability, and ease of maintenance. By encapsulation, the contents of a class can be hidden from a user. This prevents unauthorized access to the data and functions within the class. Furthermore, by restricting access to data via getters and setters, we can make data read-only or write-only as our needs dictate which improves the flexibility of the program. Separating data and methods within a class improves modularity, allowing the class to be reused throughout a program; this modularity allows one to modify the class as needs require without requiring changes to the rest

of the code. Lastly, since the code is wrapped up in its own unit, the code can be easily tested and maintained without affecting the rest of the program.

Clarity

A cohesive class should, by design, inevitably result in a clear class. By ensuring that a class is cohesive, understanding it should come easily. This understanding is also a result of maintaining consistency. With no confusion arising from inconsistent naming or style, one can focus on the purpose of the class rather than trying to decipher specific elements of the class.

Along with using descriptive names, methods should produce a result that is intuitive. For example, if a method returned the contents of an array from one index to the next, a user would expect the contents to include the index numbers used. So if the user requests the contents of the array {a, b, c, d, e, f} from index [1] to [4], the class should produce {b, c, d, e} and not, for instance, {c, d, e} or {b, c, d}.

Because a class can be used in many ways, classes should be unrestrictive in nature. Its use should not be limited by requiring data or functions to be entered or executed in a specific order. These limiting factors would obscure the function of the class and may cause its use to result in unexpected behavior. Moreover, if data field can be calculated from another, it should not be declared as a separate field.

Good clarity:

```
public class Employee {
	int yearOfHire = 2000;

	public boolean isEligible() {
		return 2019 - yearOfHire >= 5;
	}
}
```

Bad clarity:

```
public class Employee {
```

```
        int yearOfHire = 2000;
        boolean eligible = true;
}
```

As can be seen above in the first method, because eligibility is not declared and is calculated, the process with which eligibility is calculated is clear and easily understood. Because the second method is simply declared, any mechanisms used to determine its value is obscured, decreasing clarity.

Instance vs. Static

Whether a variable or method is declared as an instanced or static depends on whether it is shared by all instances or not. For example, if all objects in a class have a method to say, "hello," the method should remain static. If members of the class can say something and the content of what they can say changes accordingly, the method should be instanced. This can be further complicated if a static variable is called by an instanced method. Below is an example of both a static and instance method within a class:

```
public class Employee {

        public boolean isEligible() {              //instance
                return 2019 - yearOfHire >= 5;
        }

        public static void sayHi() {               //static
                System.out.println("Hi");
        }
}
```

Since eligibility is dependent on the year the employee was hired, the method which calculates eligibility requires the instanced variable yearOfHire to complete, which is specific to the particular entity. Because the method sayHi() does not require any instanced data and is shared by the entire class, it is declared as static.

Constructors should always be instanced since they are used to actually create an instance. By their very nature, constructors produce an instance, which enter instanced data into

that particular instance. If the contrary were true, the data would be static and the need for the constructor would be unnecessary.

PART B – Java Programming, Data Structures, and Data Design

1. Program Analysis to Program Design

We were provided with sample output along with some rules with which we were required to follow. First, the rules stated that we were to store the original data in a set of enum objects and each keyword, part of speech, and definition was to be stored in separate data fields. Next, we were instructed to use existing data structures or create new ones to store our dictionary's data. The data is to be loaded from the database into our dictionary's data structure before the program asks for user input. Lastly, the user program interface has to take user input and return the results of the user's queries. The output from the user's queries are to match exactly to the complete sample output provided.

Since the complete output is provided, understanding the problem to be solved becomes relatively simple. We need to enter all the data provided and then ensure our program produces the results requested. This required us to identify all the data available from the sample output and all possible combinations of input the user could enter. This will yield us the parameters for the queries, and the entirety of the entries in the database.

To store the data in enum objects, I thought that it would be much more efficient to store each word in a single object with separate arrays storing the part of speech and definitions respectively. There were many other structures that I considered before settling on the structure. Among the structures I considered were: three dimensional arrays, two dimensional arrays for the speech and definitions, and queues. My familiarity with the single dimensional arrays won

out. Mostly, I wanted the transfer of the information from the enum to the dictionary to be a simple affair.

As for the dictionary, I decided to use Google Guava ArrayListMultimap. To be completely honest, I used this data structure to familiarize myself to it. It turned out to be pretty useful for this assignment. Some of the other data structures I employed were ArrayLists, Arrays, and a TreeSet. I used the ArrayLists for the ability to quickly and easily determine if it contained a parameter and also to create the console output. The TreeSet was used to simplify the removal of duplicate entries.

2. Program Implimentation

The program works as intended. Nevertheless, I can think of many ways to improve the program. First of all, I believe that there are redundant steps throughout my code. In addition, it seems that the testing application that the grader provided does not work with my code. I believe that given enough time, I could fix my code to work seamlessly with the testing application. Mostly, the code could be made more elegant and streamlined.

I can also think of many improvements that can be made that are beyond the scope of this project (and would prevent the output from perfectly matching the sample output). For one, we could improve the dynamism of the program so that it could take in the search parameters in different order. For example, I could make the program capable of taking in the “distinct” and part of speech parameters in either of the second and third arguments. Also, it would be interesting if the dictionary could make suggestions to misspelled words.

PART C – Better Code Formatting and Junit

I am a little unsure what the extra credit portion was asking for and unfortunately I did not have the time to respond to the discussion forum to get clarification to this part. As

discussed earlier in this report, I realized too late that the grader wanted the query completed in such a way as to interface with the DictionaryTester. Hopefully this will not prevent me from receiving extra credit!

Zip file contents:

documentation	(folder containing all documentation)
MKim-Assignment1_S19_Report.pdf	(this report)
ConsoleOutput.pdf	(console output)
src	(folder containing all java files)
Dictionary.java	(dictionary driver)
DictionaryEnum.java	(dictionary enum database)