Course: CSC 340.04
Student: Kullathon “Mos” Sitthisarnwattanachai, **SFSU ID:** 921425216
Teammate: n/a, **SFSU ID:** n/a
Assignment Number: 02
Assignment Due Date & Time: 02-15-2022 at 11:55 PM

PART A – OOP Class Design Guidelines, 15 points

Y. Daniel Liang’s 8 Class Design Guidelines:

http://csc340.ducta.net/WEEK-01/JAVAtoCPP-ClassDesignGuidelines.pdf

Please choose 5 guidelines and discuss them in-depth. For each guideline, use at least one page for your discussion. It is OK to use code to help demonstrate your points. The code portion, if any, should not take up more than 1/3 of each guideline’s discussion.

- **Cohesion** is one of the principles in Liang’s class design guidelines. The principle describes how a class should be designed around a single entity and that any operations (methods) related to the class should be placed within it. This principle allows the program to be organized around the object-oriented programming (OOP) paradigm. For example, a class representing a `Student` should contain information that would normally be associated with a student, such as name, student, ID, and classes. This makes sense as classes are supposed to serve as a “blueprint” for the object that it is describing.

```
class Student {
   private String name;
   private String studentId;
   private List<String> classes;

   // constructors and methods omitted
}
```

Additionally, operations related to the `Student` should be contained inside of the class so that: (a) it may access and modify data fields that are associated with the `Student` class and (b) the methods are organized in a *cohesive* manner.

```
class Student {
   // fields omitted

   public boolean addClass(String classCode) {
       return this.classes.add(classCode);
   }
   public boolean removeClass(String classCode) {
       return this.classes.remove(classCode);
   }
}
```

Finally, if a class outgrows its original purpose or if an entity represents a broader idea, then the entity may be represented by several classes to ensure that a class doesn't become too large, which would in turn make the code harder to maintain. Continuing with the `Student` example, if a need arises to support different types of student (say a transfer student) whereby special properties and operations may be needed for that specific type and no other types of student, it may be appropriate to refactor the `Student` class to contain fields and methods that apply to *all students*, then create a subclass of `Student` called `TransferStudent`, to implement specific details and operations that only apply to transfer students.

- **Consistency** refers to the practice of adhering to standards and conventions within a project and throughout a certain programming language. Consistency allows the programmer to better maintain a project, especially when it may be worked on by

multiple people. For example the naming convention in Java expects variable and method names to be in lowerCamelCase, constants in ALL_CAPS, and class names in UpperCamelCase. While these rules aren't enforced by the compiler, adherence ensures that programmers understand the purpose of the relevant namespaces at a glance. The naming of the variables, methods, classes, etc. are also key in maintaining consistency. In Java (and all programming languages I know of), method names should be a verb that describes the action that it accomplishes, whereas variable and class names should be a noun that is descriptive of what it represents. For example, `int i` is almost always an unacceptable declaration for an integer (outside the context of a loop counter) as it does not describe the number or what it represents. When the variable is used later on in the code, where other variables may be introduced in the process, it may be impossible to keep track of what the variable does, which may ultimately lead to unintended changes. In addition to naming, there are also broad conventions on class structures that ensure consistency across different implementations. For example, class constants and variables are expected to be at the top of the class, followed by its constructors and methods, respectively. Below is an example:

```
class Person {

   String name;
   int age;

   public Person() {
       this.name = "";
       this.age = 0;
   }

   public Person(String name, int age) {
       this.name = name;
       this.age = age;
```

```
    }

}
```

- **Encapsulation** ensures that any data related to each class are confined within themselves. In the guideline, Liang encourages the use of access modifiers in Java such as the “private” modifier to ensure that any data or method that would only concern the class itself is not exposed outside the scope of the class. The private modifier prevents access to the data/methods from outside the class they were declared in. This can be helpful if those fields or methods are only relevant within the scope of the class and not outside of it. This would prevent confusion about which variables and methods are to be accessed by other developers that are making use of the class. For example, variables that are only used to keep track of the internal state of the class and methods that are used to modify those variables (or other internal “helper methods”) can be kept hidden from the end user. Preventing access also ensures that the end user does not accidentally modify any values that aren’t meant for use by them, which makes the class easier to maintain. To expose data fields to the end user, Liang recommends defining setter and getter methods, so that they may be accessed or modified by the end user explicitly, as opposed to accessing them directly using the dot notation.

```
class Student {

    private String name;
    private int age;

    public Student() {
```

```
        this.name = "";
        this.age = 0;
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }

}
```

- **Clarity** can be achieved if the three aforementioned guidelines: *cohesion*, *consistency*, and *encapsulation* are well-executed. While those guidelines help improve the overall usability and maintainability of your classes, we can also achieve clarity by explicitly defining a “general contract” of what the class represents or what a method would achieve. An easy-to-understand, concise language helps other people (and your future self) understand the purpose of the class and what it achieves. In turn, it could also be used to explain what the class is *not* designed for (“breaking the contract”). For classes that can be extended or methods that may be implemented, this helps the user understand how they may utilize the class and implement them in other applications and ensure consistent behavior across the program. It is also important that the classes are designed in a “sensible” manner so that it can be incorporated in other contexts without confusion. Data fields that belong to the class should have a defined and documented purpose. Values that may be derived from other fields should not be defined as a separate field, but instead implemented as a method. Defining a similar field can cause confusion for the end user and decrease the overall maintainability of the class. Similarly, methods should have a clear purpose, defined by their name and

supplemented with documentation if necessary. A method should serve a single purpose and should be designed in such a way that it may be utilized by the end user without confusion. If a method accepts an argument or relies on the values of the data fields, one must ensure that their implementation can be invoked by the end user in any combination or order without hindering the class' functionality.

- To employ effective class design, we must also make use of **instance and static modifiers** appropriately. The use of instance and static elements are an integral part of object-oriented programming. A general rule of thumb is that anything that belongs to the class itself and not a particular object should be declared as static. A static variable can be accessed by all instances of the class and through the class itself. It should be noted that static variables and methods should be accessed using the dot notation via the class itself. This prevents confusion as it makes clear that such a field belongs to the class and not a particular instance. In addition, access to static fields should be defined explicitly via their respective getter and setter methods. Passing a parameter from a constructor to set a static field directly is considered bad practice as it may lead to confusion that the field belongs to a particular instance and not the class itself because a constructor is used to create an *instance* of a class. Keeping the static fields and methods separate helps ensure that they are used properly, as static fields and methods belong to the class and may be accessed by referencing the class directly without the need to create a new instance. Whereas instance fields and methods belong to a particular instance, as such they may not be invoked or accessed without instantiation and may not be accessed directly through the class or within other static contexts.

PART B – Java Programming, Data Structures, and Data Design, 85 points

We are hired to implement an interactive dictionary. Our dictionary takes input from users and uses the input as a search key to look up values associated with the key. Requirements:

- **Coding**: No hard coding, https://en.wikipedia.org/wiki/Hard_coding. *Please think about Dynamic and Scalable.*

- **Data Source**: Store the original data in a set of **enum** objects. Each keyword, each part of speech, and each definition must be stored in a separate data field. Do not combine them such as storing three parts in one String.

- **Data Structure**: Use existing data structures or creating new data structures to store our dictionary's data.

- **Data Loading**: When our program starts, it loads all the original data from the Data Source into our dictionary's data structure. Data Loading must finish before our program starts interacting with users.

- **User Interface**: A program interface allows users to input search keys. This interface then displays returned results. Our program searches the dictionary's data (not the Data Source) for values associated with the search keys.

- **Identical Output**: Our program's output must be **identical** to the complete sample run's output. The **complete sample output** is posted at

http://csc340.ducta.net/Assignments/Assignment-02-PartB-CompleteSampleOutput.pdf

1. **Program Analysis to Program Design, 15 points.** *Please think about Interviews.*

   In at least 1 full page, please explain the following **in detail**:

   - Your analysis of the provided information and the provided complete sample output. *Please think about Clients and Sales.*

   - First, I look at how the program interacts with the user. From the Complete Sample Output file, I can see that the program simply takes in an input from the user, then the program prints out the results to the screen. In general, it takes in the provided search term and any relevant keywords to look for definitions and display them on the screen.

     Then, I look for search entries that provide all of the definitions needed for the program to produce identical results. These are ones without additional search arguments.

     The first few searches demonstrate that the input is case-insensitive. Then, the keywords that are applied after shows that they modify the output of the definitions. The “reverse” keyword reverses the order at which the definitions appear and “distinct” removes duplicate entries. From observation, we see that the definitions are ordered by the lexicographical order of their respective part of speech, then the order of their definitions.

Further, the output shows that the additional search arguments must be provided in a particular order (but may be omitted) in order for it to be accepted. That is, a valid argument may not be accepted if it is placed in an incorrect order (see search 20).

Finally, I look for edge cases that the program should handle, such as blank input, special keywords, etc.

- What problem you are solving. Please explain it clearly then define it concisely. *Please think about Problem Solving and Interviews.*

- First, we define how the data should be stored and interpreted. Since we are required to use enums, there isn't much choice on where to store the data.

  Then, we need to make sure that the data can be loaded and manipulated according to the user. We need to figure out how to parse the user's request, then how we design the data structure around those requirements.

  The manner in which the search arguments are parsed are defined by the Complete Sample Output, so we mainly workaround how our data structure can accommodate the request that the user may provide. Given that we need to filter out duplicates and define the order for our output, it would be best to implement the Comparator interface in order to work with existing APIs.

- How you store data in enum objects. And why. *Please think about Data Structures and Data Design.*

- Since enum instances cannot be created during runtime and that each instance and their identifiers must be unique, it makes it suitable for each instance to represent a single term (e.g., “book,” “arrow,” etc.).

  Keeping in mind that the definitions for each term: (a) are associated with a particular part of speech — i.e., a pair; (b) may be duplicated for each term, and need not be unique for a given part of speech; (c) needs to be comparable for the purpose of sorting and removing duplicates, it makes sense to create a new data type to store the definitions.

  And so, each term (enum instance) is associated with one or more instances of the custom class Definition, which we define below.

- Which data structures you use/create for your dictionary. And why. *Please think about Data Structures and Data Design.*

- The dictionary is stored in a simple HashMap type, creating a relationship between a term (string) and an array of the custom type, Definition.

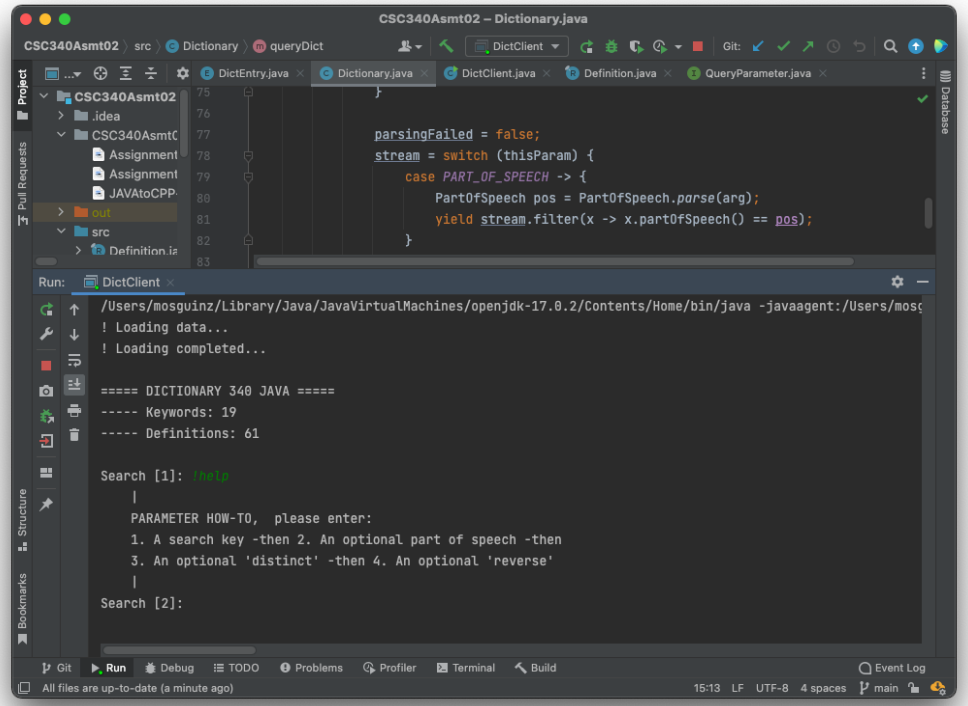
  Given the “pair-like” nature between the part of speech and a definition, it seems appropriate to model each definition after a map’s Entry type. We also implement the Comparable interface in order to be able to define how each Definition should be compared for the purpose of sorting. This allows us to make use of the Stream API, making it easy to manipulate the order of the definitions.

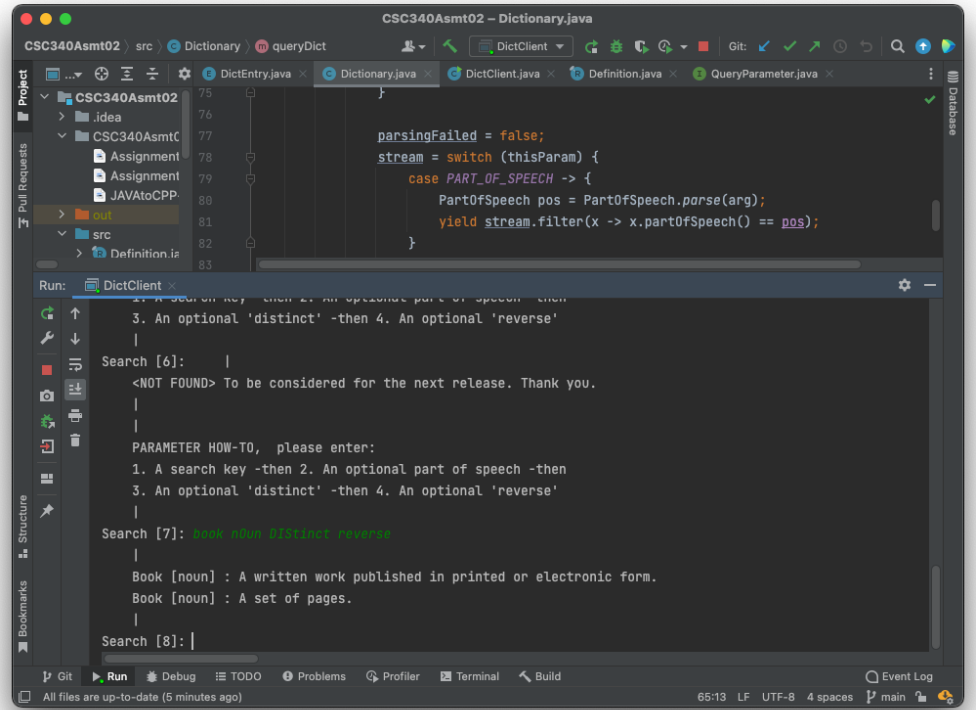
Related to the dictionary, we also use enum types for the parts of speech and the query parameters as they are a set of predefined values. The PartOfSpeech enum instances are used as keys in the Entry type, and the QueryOption type is used to parse and determine the order at which the parameters should be parsed.

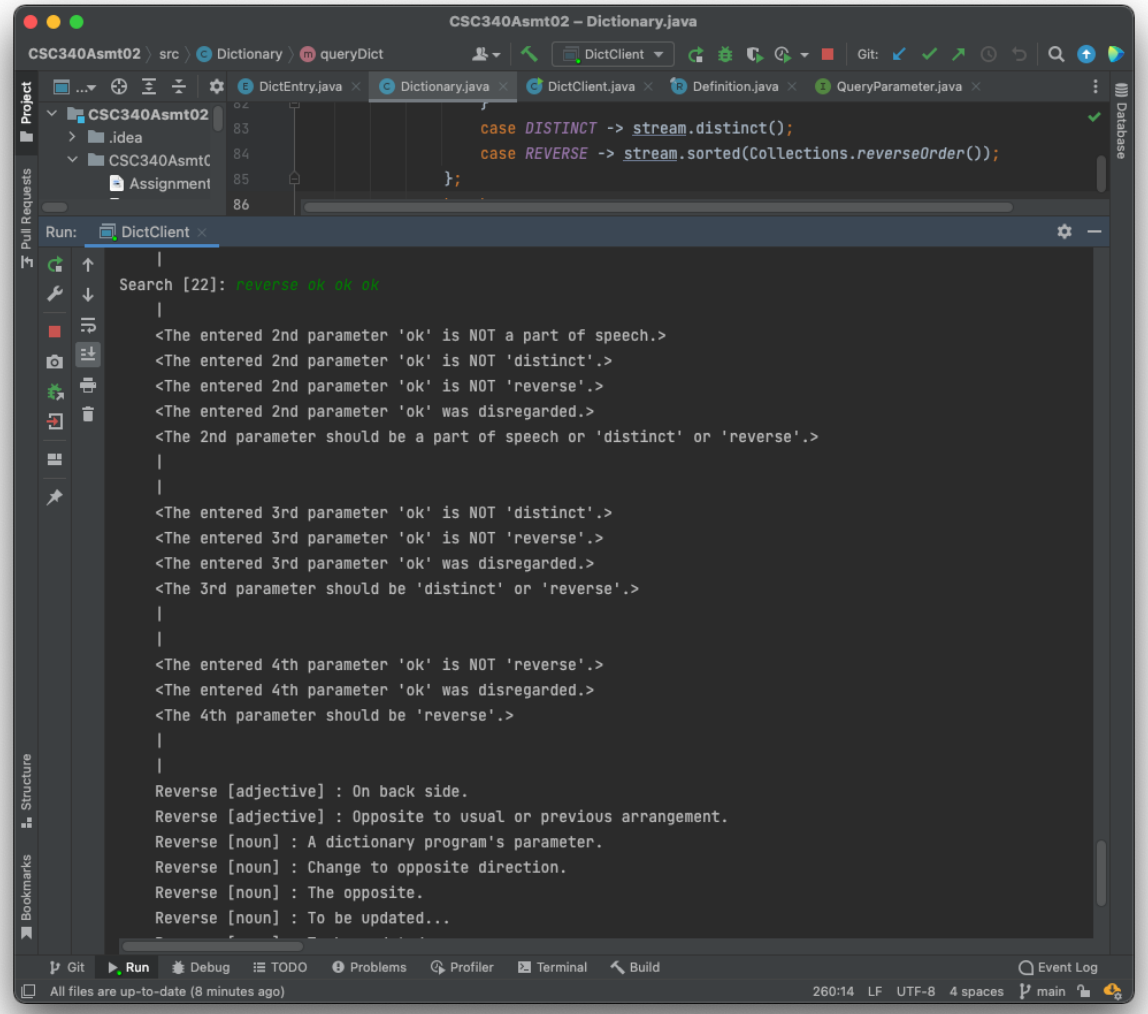
2. **Program Implementation, 70 points.** *Please think about Interviews.*

- Implement your program to meet all the requirements.
- See the source code in the src/ folder.
- In your assignment report, demonstrate your program to your grader/client.
- Below are a few of the screenshots demonstrating the program.

> **Implementation note**
>
> The program was built and tested on JDK 17. Most notably, it makes use of the new record class introduced in JDK 14. So it *should* probably work with JDK versions 14 and above.
>
> The main method is in src/DictClient.java.


CSC340Asmt02 – Dictionary.java
CSC340Asmt02 › src › Dictionary › queryDict
DictClient
Git:
DictEntry.java
Dictionary.java
DictClient.java
Definition.java
QueryParameter.java
Project
Pull Requests
Structure
Bookmarks
Database
CSC340Asmt02
.idea
CSC340AsmtC
Assignment
Assignment
JAVAtoCPP
out
src
Definition.ja
75 }
76
77 parsingFailed = false;
78 stream = switch (thisParam) {
79 case PART_OF_SPEECH -> {
80 PartOfSpeech pos = PartOfSpeech.parse(arg);
81 yield stream.filter(x -> x.partOfSpeech() == pos);
82 }
83
Run: DictClient
/Users/mosguinz/Library/Java/JavaVirtualMachines/openjdk-17.0.2/Contents/Home/bin/java -javaagent:/Users/mosg
! Loading data...
! Loading completed...
===== DICTIONARY 340 JAVA =====
----- Keywords: 19
----- Definitions: 61
Search [1]: !help
|
PARAMETER HOW-TO, please enter:
1. A search key -then 2. An optional part of speech -then
3. An optional 'distinct' -then 4. An optional 'reverse'
|
Search [2]:
Git
Run
Debug
TODO
Problems
Profiler
Terminal
Build
Event Log
All files are up-to-date (a minute ago)
15:13 LF UTF-8 4 spaces main


CSC340Asmt02 – Dictionary.java
CSC340Asmt02 › src › Dictionary › queryDict
DictClient
Git:
DictEntry.java
Dictionary.java
DictClient.java
Definition.java
QueryParameter.java
Project
Pull Requests
Structure
Bookmarks
Database
CSC340Asmt02
.idea
CSC340AsmtC
Assignment
Assignment
JAVAtoCPP
out
src
Definition.ja
75 }
76
77 parsingFailed = false;
78 stream = switch (thisParam) {
79 case PART_OF_SPEECH -> {
80 PartOfSpeech pos = PartOfSpeech.parse(arg);
81 yield stream.filter(x -> x.partOfSpeech() == pos);
82 }
83
Run: DictClient
3. An optional 'distinct' -then 4. An optional 'reverse'
|
Search [6]:
<NOT FOUND> To be considered for the next release. Thank you.
|
|
PARAMETER HOW-TO, please enter:
1. A search key -then 2. An optional part of speech -then
3. An optional 'distinct' -then 4. An optional 'reverse'
|
Search [7]: book nOun DIStinct reverse
|
Book [noun] : A written work published in printed or electronic form.
Book [noun] : A set of pages.
|
Search [8]:
Git
Run
Debug
TODO
Problems
Profiler
Terminal
Build
Event Log
All files are up-to-date (5 minutes ago)
65:13 LF UTF-8 4 spaces main



- Does your program work properly?

- Yes. “Works on my machine!”

  Tested on JDK 17 and was able to produce identical output using the exact input specified in the Complete Sample Output file.

- How will you improve your program?

- Initially, it was planned to be scalable in mind. However, during the course of implementation, I feel like what I have done may be a bit overkill (and maybe even improper) and probably resulted in the opposite of scalable.

  If I had the time, I would refactor the way the query parameters are defined and parsed. I think that using enum may have been the right start in defining those parameters. I thought that they were appropriate because: (a) they are a set of predefined values, and (b) they had other behaviors that needed to be grouped with them — i.e., parsing and their different string representation (for the error messages).

  I wasn't sure on how I should have handled the different ways they were meant to be parsed. For example, the part of speech needed to work with the existing enum that is used to internally store the different entries as well as parsing it as an argument. Whereas the other two keywords, "distinct" and "reverse" simply needed to be checked if they were those words themselves.