May 2016

Laboratory Assignment 5

Dictionary

Farcas Silviu Vlad

30425/1

Contents

1. Introduction

1.1 Task Objective

The task of this assignment is defined as follows:

Consider the implementation of one of the following:

a) A dictionary of Romanian language or a dictionary of English language or

b) A dictionary of synonyms (thesaurus) for Romanian or English language.

It is required to use Java Collection Framework Map for the implementation.

Define and implement a domain specific interface (populate / add / remove / copy / save /

search, etc.). Consider the implementation of specific utility programs for dictionary

processing. For example:

- Implement a method for checking dictionary consistency. A dictionary is consistent, if all

words that are used for defining a certain word are also defined by the dictionary.

*- Implement dictionary searching using * (any string, including null) and ? (one character).*

For example, you can search for a?t.*

Use the above examples to warm up your imagination.

Note.

The good things acquired as a result Homework 4 (i.e. contracts, invariants, assert, separating

the interface from implementation, javadoc, etc.) will be also used for this homework.

1.2. Personal Approach

The aim of this paper is to present one way of implementing the required system, based on a simple and minimal GUI that provides the user with the possibilities of applying basic commands on a dictionary. The user will be able to add new words, delete words, add or delete definitions, update words or populate the dictionary. While populating, all the above mentioned operations perform in "populate" mode, that is, we don't check for dictionary consistency, only at the end of the population process. We will discuss what dictionary consistency is later. Our implementation could be used both for a usual dictionary and a thesaurus, though we will follow the thesaurus approach in our presentation.

2. Problem Description

2.1. Problem Analysis

First of all, what is a dictionary?

Wikipedia:

> A dictionary is a collection of words in one or more specific languages, often alphabetically (or by radical and stroke for ideographic languages), with usage of information, definitions, etymologies, phonetics, pronunciations, translation, and other information;[1] or a book of words in one language with their equivalents in another, also known as a lexicon.[1] It is a lexicographical product designed for utility and function, curated with selected data, presented in a way that shows inter-relationship among the data.

And what is a thesaurus?

Wikipedia:

> In general usage, a thesaurus is a reference work that lists words grouped together according to similarity of meaning (containing synonyms and sometimes antonyms), in contrast to a dictionary, which provides definitions for words, and generally lists them in alphabetical order. The main purpose of such reference works is to help the user "to find the word, or words, by which [an] idea may be most fitly and aptly expressed" – to quote Peter Mark Roget, architect of the best known thesaurus in the English language.

A dictionary is of most use for linguists and not only. We will build our dictionary based on HashMap of DictionaryEntries and their definitions.

2.2 Modeling

Our dictionary will have as its attribute a HashMap of DictionaryEntries as key and and ArrayList of DictionaryEntries as values. The pair <key, value> will thus be <DictionaryEntry, ArrayList<DictionaryEntry>>. This is because we need an entry may have multiple definition, which can be formed of words that must also be defined in the dictionary. This is because of the dictionary consistency specification.

2.3. Use cases

We can remove a word (and all appearances of that word in the definitions in the dictionary), we can remove a definition, we can add a new word (and in the interface logic we will also be redirected to the define function, presented in the following), we can define a word by adding a new definition, we can search using patterns (the “*” means any string including null, the “?” means just one character) and we can enter populate mode, where all these operations can be performed at the expense of not checking the dictionary consistancy, that is not asserting

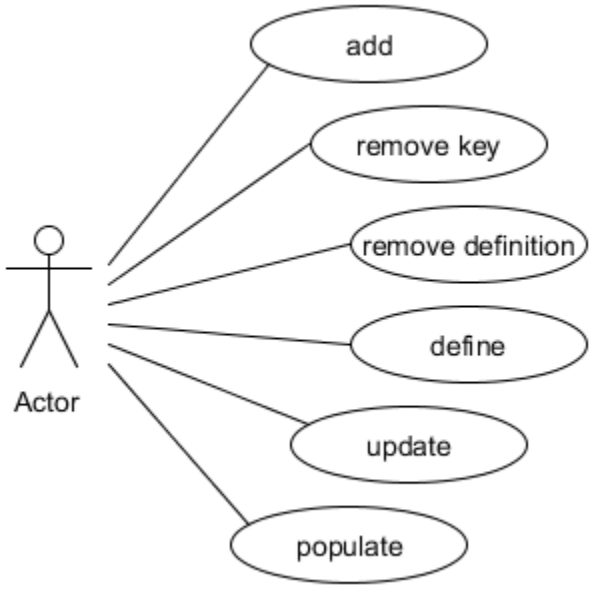
WellFormed(). However, we don't exit population mode if we do not have a consistent dictionary.

3. Design

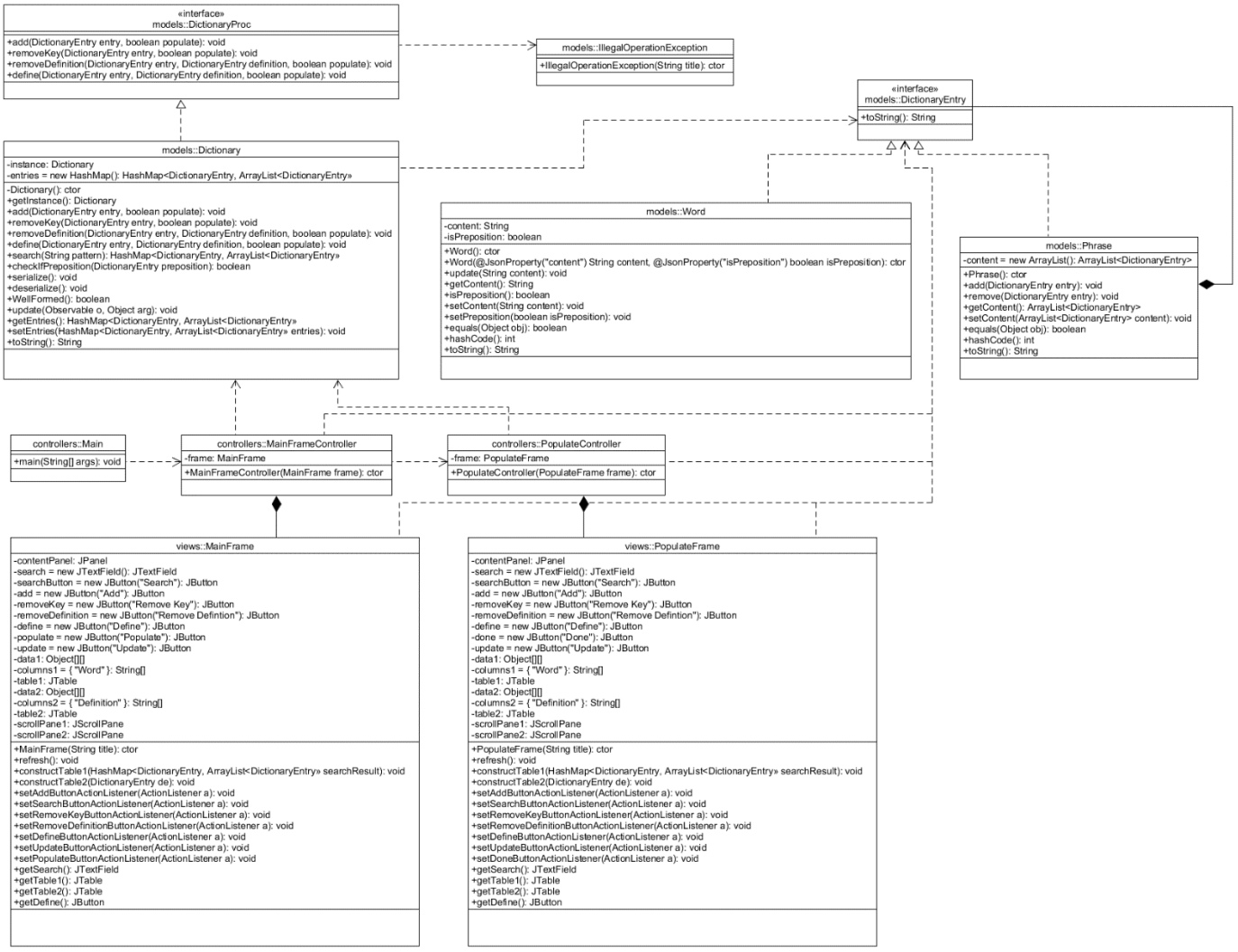
3.3. UML Diagrams

In the following we will present Use Case diagram, Class Diagram and Sequence Diagram.

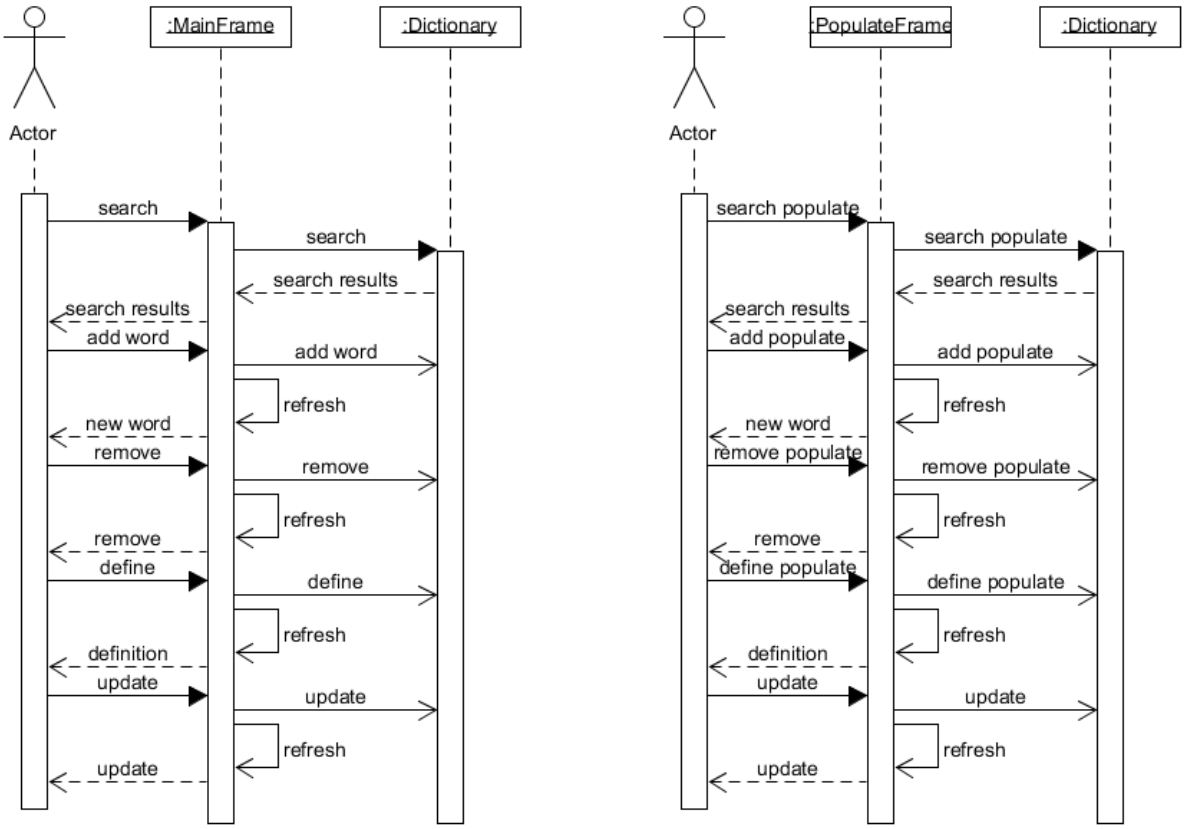
3.3.1. Use Case Diagram



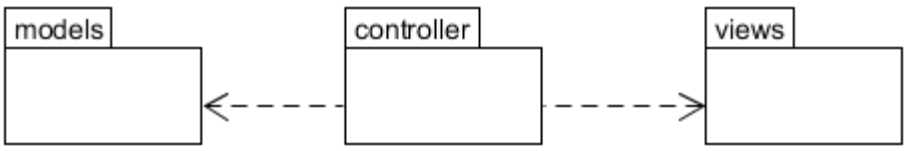
1.1.2. Class Diagram (next page)



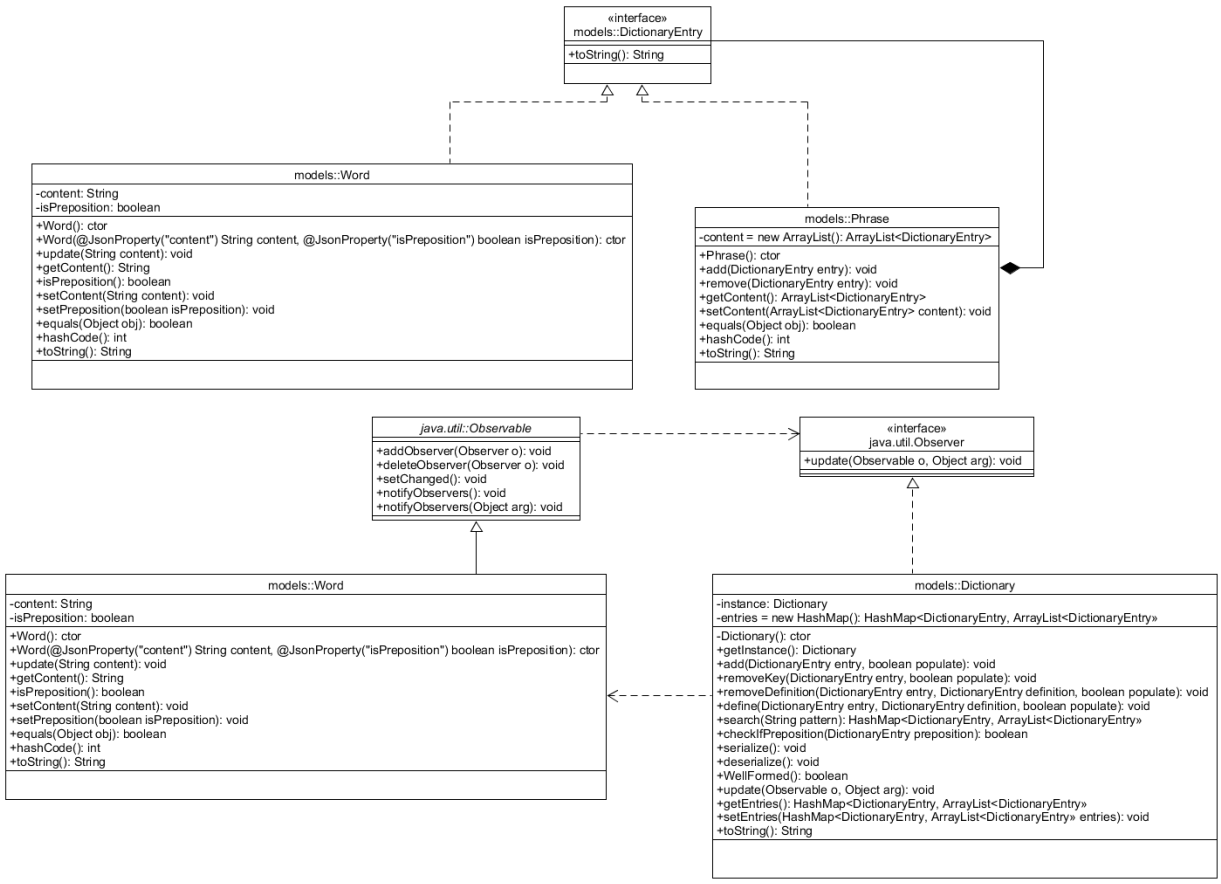
1.1.3. Sequence Diagram (next page)



1.1.4. Package Diagram



1.1.5. Design Patterns



1.2. Data Structures

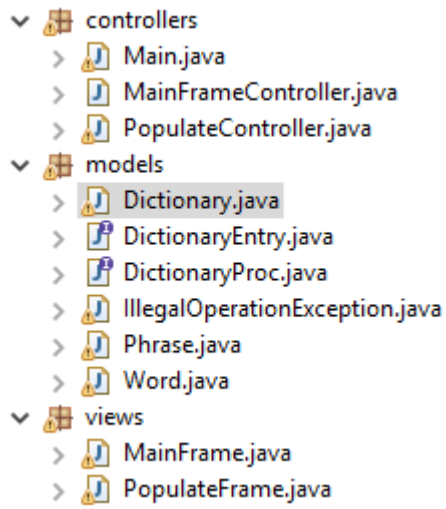
The main use in our project is of HashMap and ArrayList. HashMap hashes every key in a hashTable and associates each key to a value. We will use HashMap<DictionaryEntry, ArrayList<DictionaryEntry>> as a structure in Dictionary class. What is a DictionaryEntry? It can be both a Word and a Phrase, which can be composed of more Words or even more phrases, however our implementation assures only one level of recursion.

1.3. Class Design

The problem domain was decomposed into sub-problems, and for each sub-problem a solution was build. By assembling the solutions we obtain the general solution to our problem. The sub-problems were:

- how to correctly model a dictionary and correctly model functions on dictionaries and get the desired result?
- how will be implemented the user interface?
- how can we assure user interaction with our models?

For each of the above mentioned, we came up with a package that would support the features of dictionary processing, the user interface and user interaction with our model. These packages follow the Model-View-Controller software architecture and they are as follows:



We first started designing our **model** package.

The Dictionary class has at its core a HashMap<DictionaryEntry, ArrayList<DictionaryEntry>>. It follows the singleton pattern, with a single instance of the class being created. It contains the basic operations methods: add, removeKey, removeDefinition, define, search. Each of those methods implements the rpeconditions and postconditions commented in the interface DictionaryProc, which will be described later. Special interest needs to be given to the serialize() and deserialize(), which are completely reproduced below:

```
public void serialize() {
            ObjectMapper mapper = new ObjectMapper();
            mapper.enableDefaultTyping();
```

```
            class DictionaryEntrySerializer extends
JsonSerializer<DictionaryEntry> {
                  @Override
                  public void serialize(DictionaryEntry arg0, JsonGenerator
arg1, SerializerProvider arg2)
                              throws IOException, JsonProcessingException {
                        arg1.writeFieldName(String.valueOf(arg0.toString()));
                  }
            }
            SimpleModule module = new SimpleModule();
            module.addKeySerializer(DictionaryEntry.class, new
DictionaryEntrySerializer());
            mapper.registerModule(module);
            try {
                  String json = mapper.writeValueAsString(this);
                  FileWriter wr = new FileWriter("Dictionary.json");
                  wr.write(json);
                  wr.close();
            } catch (JsonProcessingException e1) {
                  e1.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }

      public static void deserialize() {
            ObjectMapper mapper = new ObjectMapper();
            mapper.enableDefaultTyping();
            class ExampleClassKeyDeserializer extends KeyDeserializer {
                  @Override
                  public Object deserializeKey(final String key, final
DeserializationContext ctxt)
                              throws IOException, JsonProcessingException {
                        return new Word(key, false);
                  }
            }

            class ExampleJacksonModule extends SimpleModule {
                  public ExampleJacksonModule() {
                        addKeyDeserializer(DictionaryEntry.class, new
ExampleClassKeyDeserializer());
                  }
            }
            mapper.registerModule(new ExampleJacksonModule());
            try {
                  BufferedReader br = new BufferedReader(new
FileReader("Dictionary.json"));
                  instance = mapper.readValue(br, Dictionary.class);
                  br.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
```

Serialization and deserialization is done using Jackson. A JSON file is used, following the JSON format. Because we need to serialize a

HashMap which has objects as key, during serialization thos objects are converted to Strings using the toString method defined or not (In our case, DictionaryEntry interface defines a method toString(), which is implemented by Phrase and Word using streams. But when deserializing, there is no way of knowing how to reconstruct the object serialized based on that string. For that, we have implemented a custom key deserializer that builds a Word out of the keys written in file.

We shall also take a look at the function WellFormed() defined in the Dictionary class. It basically verifies if each word used in a definition of a dictionary entry is also defined in the dictionary. The content of the method is fully reproduced below:

```
public boolean WellFormed() {
            for (DictionaryEntry de : entries.keySet())
                  for (DictionaryEntry de2 : entries.get(de))
                        if (de2 instanceof Phrase) {
                              for (DictionaryEntry de3 : ((Phrase)
de2).getContent())
                                    if (!(((Word) de3).isPreposition()))
                                          if (!(entries.containsKey(de3)))
                                                return false;
                        } else if (!(((Word) de2).isPreposition()))
                              if (!(entries.containsKey(de2)))
                                    return false;
            return true;
      }
```

The WellFormed() method is called by each of the operation methods when the dictionary is not in populate mode. When in populate mode, we only check the dictionary consistency at the end of the population, when the user presses “Done”.

The other method present in the Dictionary class is the update method. This will be described in the Design Pattern section.

The method checkIfPreposition() checks whether a given DictionaryEntry is a preposition inside the dictionary. The method is used when interpreting data in JTable in the user interface, when we need to distinguish whether a word is or not a preposition.

Then we have the **IllegalOperationException** class, which basically is the exception that is thrown whenever one of the add, define, removeKey or RemoveDefinition is not executed properly (note that this is not the case when the assertions fail).

The **Word** class implements the DictionaryEntry interface. It has a content and a boolean attribute isPreposition which says whether the Word is a Preposition. To ease the use of the HashMap, we overriden the equals() and hashCode() methods, in order we could easily access the keys in the maps using object references. The content of these methods is fully reproduced below:

```
public boolean equals(Object obj){
            if (!(obj instanceof Word)) return false;
            if(this.content.equals(((Word)obj).getContent()) &&
(this.isPreposition == ((Word)obj).isPreposition())) return true;
            return false;
      }

      public int hashCode(){
            return 37*(37 * 23 + content.hashCode()) + (isPreposition ? 0 :
1);
      }
```

The **Observer** pattern is implemented having the Word as an Observable and the Dictionary as Observer. The Word object has a method called update() which updates the content of the Word object, and calls the corresponding update() method in the Dictionary class.

The **Phrase** class implements the DictionaryEntry interface and contains an ArrayList<DictionaryEntry> as its content. As we mentioned before, there is only one level of recursion, that is, **Phrase** only contains Words, not other Phrases. This is the basis of the **Composite** pattern, which diagram has already been presented.

The **views** package has two main frames: the MainFrame, which is the frame showed when opening the application, and PopulateFrame, which builds the populate mode. The MainFrame has the following fields:

```
private JPanel contentPanel;
      private JTextField search = new JTextField();
      private JButton searchButton = new JButton("Search");
```

```
        private JButton add = new JButton("Add");
        private JButton removeKey = new JButton("Remove Key");
        private JButton removeDefinition = new JButton("Remove Defintion");
        private JButton define = new JButton("Define");
        private JButton populate = new JButton("Populate");
        private JButton update = new JButton("Update");
        private Object[][] data1;
        private String[] columns1 = { "Word" };
        private JTable table1;
        private Object[][] data2;
        private String[] columns2 = { "Definition" };
        private JTable table2;
        private JScrollPane scrollPane1;
        private JScrollPane scrollPane2;
```

It also has the method constructTable1(HashMap<DictionaryEntry, ArrayList<DictionaryEntry>> searchResult) which builds the dictionary entries table out of the search results. The method constructTable2(DictionaryEntry de) constructs the table with defintions gien a word selected.

The controller package has the controllers Main, MainFrameController and PopulateController. The last two have as attribute a corresponding view and they manage to construct the interaction between the views and the model.

1.4. Interfaces

We have two interfaces: DictionaryProc and DictionaryEntry. DictionaryProc defines the functionality of the Dictionary class which implements DictionaryProc, and defines as comments the preconditions and postconditions for each method. Here we will fully reproduce the pre and post conditions in Dictionary class:

```
/**
        * @pre entry != null
        * @pre !(entries.containsKey(entry))
        * @post entries.containsKey(entry)
        * @post size() == size()@pre + 1
        */
       public void add(DictionaryEntry entry, boolean populate) throws
IllegalOperationException;

       /**
```

```
        * @pre entry != null
        * @pre entries.containsKey(entry)
        * @post !(entries.containsKey(entry))
        * @post size() == size()@pre - 1;
        */
       public void removeKey(DictionaryEntry entry, boolean populate) throws
IllegalOperationException;

       /**
        * @pre entry != null
        * @pre definition != null
        * @pre entries.containsKey(entry)
        * @pre entries.get(entry).contains(definition)
        * @post entries.containsKey(entry)
        * @post !(entries.get(entry).contains(definition))
        * @post entries.get(entry).size() + 1 == entries.get(entry).size()@pre
        */
       public void removeDefinition(DictionaryEntry entry, DictionaryEntry
definition, boolean populate) throws IllegalOperationException;
       /**
        * @pre entry != null
        * @pre definition != null
        * @pre entries.containsKey(entry)
        * @pre entries.get(entry) != null
        * @post !(entries.get(entry).isEmpty())
        * @post entries.get(entry).size() == entries.get(entry).size()@pre + 1
        */
       public void define(DictionaryEntry entry, DictionaryEntry definition,
boolean populate) throws IllegalOperationException;
```

All these pre and post conditions will be implemented with assertions in the Dictionary class.

The DIctionaryEntry interface is the base interface used in the Composite pattern. This will be therefore examined in the Design Patterns section.

1.5. Relations

In Object Oriented Programming, it is a good habit to design classes as loosely coupled as possible, in order to avoid error propagation. I tried to stick with this principle and, with few exceptions, I think I managed to fulfill this requirement. In this design, one can found the following relations: inheritance, aggregation, dependency and innerness.

1.6. Packages

The design follows the Model-View-Controller (MVC) architecture. It has three packages: models, views, controllers.

1.7. Algorithms

Let's analyze the toString() methods in the Phrase and Dictionary classes. These were implemented using JAVA 8 streams.

Phrase:

```
public String toString() {
            return
content.stream().map(Object::toString).collect(Collectors.joining(" "));
      }
```

Dictionary:

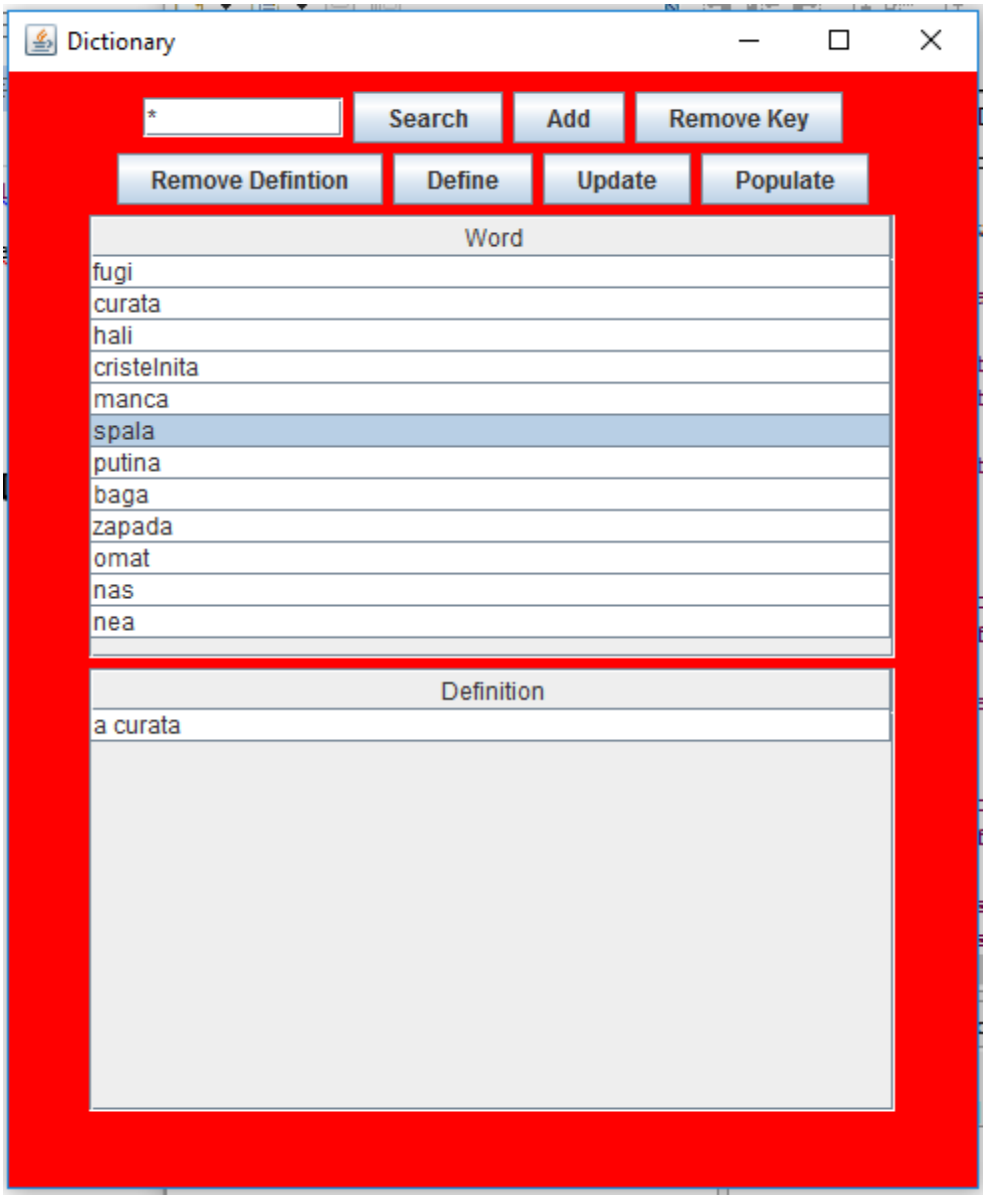
```
public String toString() {
            return entries.entrySet().stream().map(entry -> entry.getKey() +
" = " + entry.getValue())
                        .collect(Collectors.joining("; ", "[", "]"));
      }
```

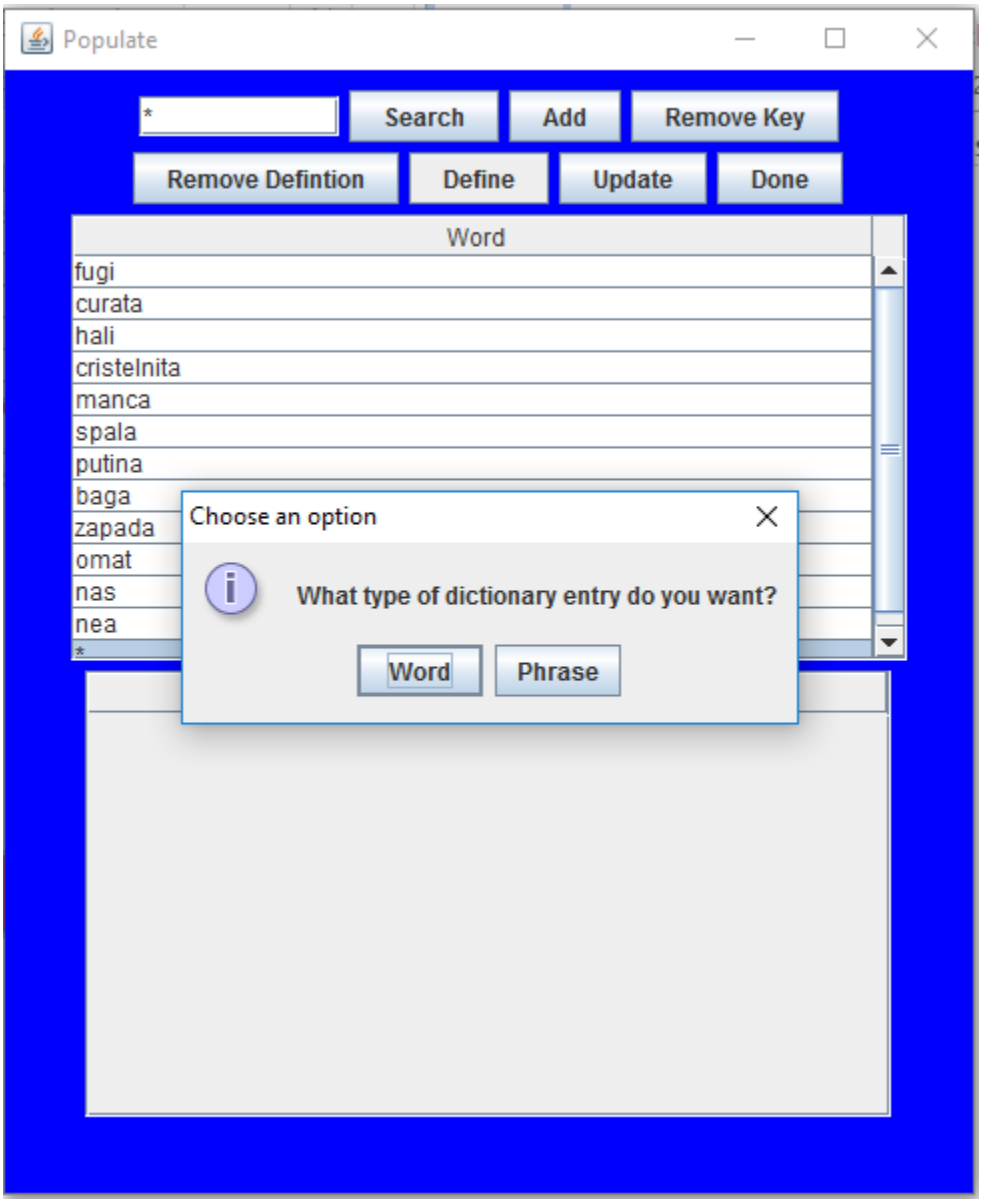
1.8. Design Patterns

In the requirements, we were expressly told to use two Design Patterns (DP). We have chosen Observer (which was implemented in the last assignment) and Composite. The class diagram of these patterns was presented in the diagrams sections. The Observer pattern has as Observable the Word class (since every DictionaryEntry key in the dictionary is a Word), which has a method update(String update), which virtually does nothing…It passes the content to be updated to the update(Observable o, Object obj) method in the Dictionary class (which is the Observer) which then makes the necessary updates in the Dictionary, updating all occurrences of the observable word in the dictionary.

The Composite pattern is implemented by the classes Word, Phrase and DictionaryEntry. Every Word or Phrase is a DictionaryEntry, and Phrase can be composed of more DictionaryEntries, which can be either Word or Phrases, which can be composed of ... However, in our implementation, we keep an order of recursion of just one, that is, a Phrase can only be composed of other Words.

1.9. User Interface





2. Implementation and Testing

The project was developed in Eclipse IDE using Java 8, on a Windows 10 platform. It should maintain its portability on any platform that has installed the SDK. It was heavily tested, but new bugs could be

discovered in the near future. One of the main inconveniences are that the program is not protected against all kinds of illegal input data, only to some. This could be the subject of future development.

3. Results

The application is a user-friendly, helpful app that can perform basic dictionary operations and save the results in a JSON file. It can be extended into a more powerful tool that can handle both Words and Phrases as keys. But for now, the app can be considered a good starting point.

4. Conclusions

4.1. What I've Learned

TIME IS PRECIOUS! I had to solve complicated problems of time management which taught me good organization skills. I learned that a good model is always a key to a successful project and that a bad model could ruin your project in the end. I learned never to get stuck on one little bug, and if I do, ask for help, either from the TA or from colleagues or on different forums. I am looking forward to apply what I have learned in future projects.

4.2. Future Developments

We could try to make the program more user – protected in that we could assume the user doesn't know to enter valid data each time.

5. Bibliography

[1] http://www.uml.org/

[2] Barry Burd, *Java For Dummies*, 2014

[3] Kathy Sierra, Bert Bates, *Head First Java*, 2005

[4] Steven Gutz, Matthew Robinson, Pavel Vorobiev, *Up to Speed with Swing*, 1999

[5] Joshua Bloch, *Effective Java*, 2008

[6] http://www.stackoverflow.com/

[7] http://docs.oracle.com/

[8] http://www.tutorialspoint.com/

[9] http://www.json.org/