**Buzea Vlad-Calin**

**30422**

**Table of Contents**

**1. Objective of the assignment 3**

**2. Problem analysis 4**

**2.1. Modelling 4**

**2.2. Use cases 5**

**2.3. Scenarios 6**

**3. Design 7**

**3.1. UML Class Diagram ..7**

**3.2. Data structures ..8**

**3.3. Algorithms 10**

**3.4. Class design 10**

**3.5. Interface 11**

**4. Implementation and testing**

**5. Results**

**6. Conclusions, what has been learned, further developments**

**7. Bibliography**

**1.** **Objective of the assignment**

TP Lab – Homework 5

1. Study the Java Collection Framework Map

http://download.oracle.com/javase/1.4.2/docs/api/java/util/Map.html

2. Consider the implementation of one of the following:

i. A dictionary of Romanian language or a dictionary of English language or

ii. 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.

**2. Problem analysis**

**2.1. Modeling**

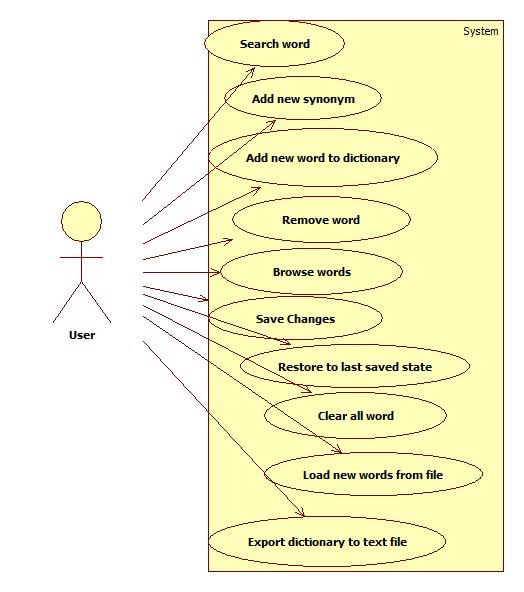
I have chosen to implement a thesaurus for the Romanian language.

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.

Although including synonyms, a thesaurus should not be taken as a complete list of all the synonyms for a particular word. The entries are also designed for drawing distinctions between similar words and assisting in choosing exactly the right word. Unlike a dictionary, a thesaurus entry does not give the definition of words.[1]

So, the aim will be to create a easy to use program, that will allow the user to find synonyms for the desired words. Furthermore the user may browse words in case he does not know exactly what he is looking for, or search for a specific format of a word.

**2.2. Use Cases**

****

**2.3. Scenarios**

Scenario I

a)Identification Summary

Title: Addition of a new word

Summary: This case allows the user to add a new word, with its first synonym, to the dictionary.

Actor: user

b)Flow of Events

Preconditions: The user interface did not malfunction and the user input data was correct.

Main success scenario:

1. The user starts the Dictionary application.
2. The user chooses the “Browse” tab from the frame.
3. The user clicks the “Add new word” button.
4. An input panel is made visible on the window.
5. The user completes both the word and synonym fields.
6. The user clicks “Add” button.
7. The introduced words are checked for validity (only letters, both fields having at least 1 character)
8. The panel containing the fields for adding a new word is made invisible.
9. If the input was valid, the new words appear in the browsing list.
10. If the input was invalid, an ERROR message is shown (“Please verify input”).

Scenario II

a)Identification Summary

Title: Search for a word in the dictionary

Summary: In this scenario, the user wishes to search for a word, but he does not know it’s exact form, only some parts of the word.

Actor: user

b)Flow of Events

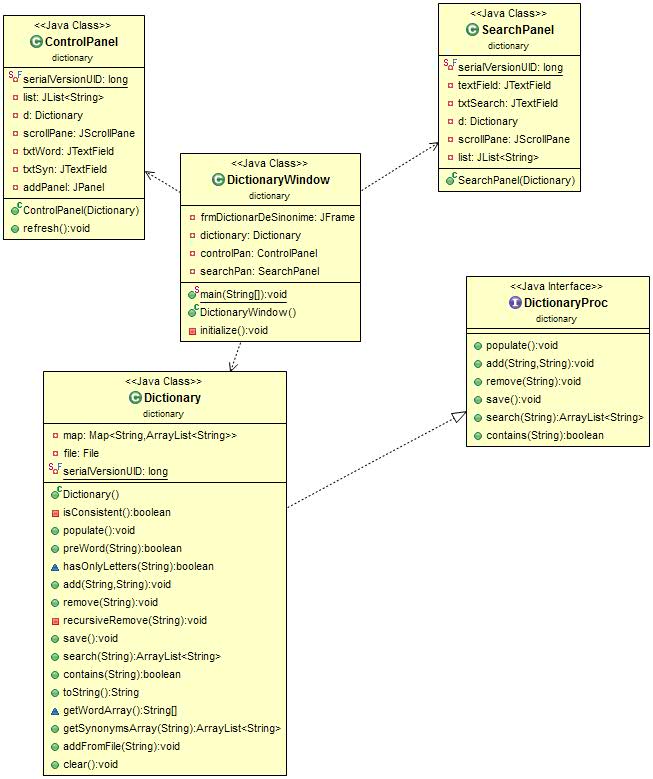
Preconditions: The user interface did not malfunction.

Main success scenario:

1. The user starts the Dictionary application.
2. The user selects the “Index” tab.
3. The user introduces the word in the text field, substituting a specific number of unknown letters with ‘?’ and undetermined number of unknown letters with ‘\*’;
4. The user clicks the “Search” button.
5. All compatible results will be displayed in the list bellow the text field.
6. If the user wants to see the synonyms of a word from the list, he selects the word and clicks “View synonyms”
7. The old list will be discarded and a new list with the synonyms corresponding to the selected word appears.

**3. Design**

**3.1. Class Diagram**

****

**3.2. Data Structures**

The objective clearly specifies the usage of the Map interface, that is described bellow:

public interface **Map**<K,V>

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract class rather than an interface.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw UnsupportedOperationException if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an UnsupportedOperationException if the invocation would have no effect on the map. For example, invoking the putAll(Map) method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically NullPointerException or ClassCastException. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

This interface is a member of the Java Collections Framework.

Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the containsKey(Object key) method says: "returns true if and only if this map contains a mapping for a key k such that (key==null ? k==null : key.equals(k))." This specification should not be construed to imply that invoking Map.containsKey with a non-null argument key will cause key.equals(k) to be invoked for any key k. Implementations are free to implement optimizations whereby the equals invocation is avoided, for example, by first comparing the hash codes of the two keys. (The Object.hashCode() specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying Object methods wherever the implementor deems it appropriate.

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

From the above mentioned list of implementing classes, I have chosen the HashMap class, that is described bellow:

public class **HashMap**<K,V>

extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.

If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

Note that this implementation is not synchronized. If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the Collections.synchronizedMap method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

Map m = Collections.synchronizedMap(new HashMap(...));

The iterators returned by all of this class's "collection view methods" are fail-fast: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

This class is a member of the Java Collections Framework.

**3.3 Algorithms**

The Algorithms used in this project are those corresponding to hashtables:

* Hashtable insert : compute value of hash function(corresponding to transmitted key) and insert value at the end of the linked list corresponding to that table entry.
* Hashtable retrieve: compute value of hash function(corresponding to transmitted key) and search the value in linked list corresponding to that table entry
* Hashtable delete: compute value of hash function(corresponding to transmitted key) and search the value in linked list corresponding to that table entry; if found, delete it from the linked list.

**3.4 Class Design**

**CRC CARDS**

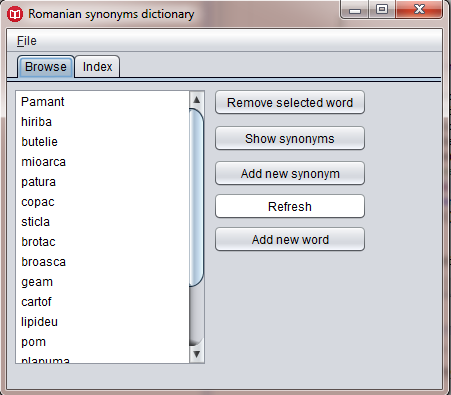
|  |  |
| --- | --- |
| **Dictionary** | |
| Attributes, Functions: | Collaborator classes |
| Contains words and a corresponding array of synonyms to each word.  Can search for a word  Can add new synonyms to a word  Can add a new word to the dictionary  Can remove a word from the dictionary  Can read new words from a file  Can be saved to a file | String, ArrayList<String> |

|  |  |
| --- | --- |
| **DictionaryWindow** | |
| Attributes, Functions: | Collaborator classes |
| Contains the two tabs, menu bar and the menu actions  Can save the dictionary  Can restore the dictionary  Can add new words from a specified file  Can export dictionary to specified file | ControlPanel  SearchPanel |

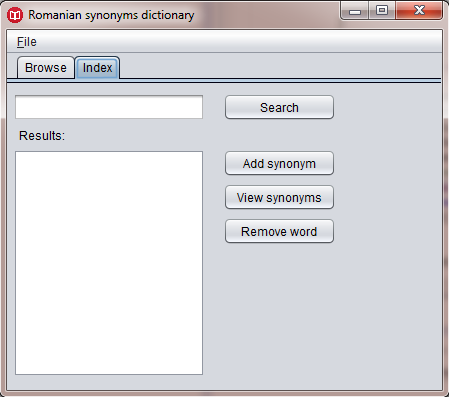
|  |  |
| --- | --- |
| **ControlPanel** | |
| Attributes, Functions: | Collaborator classes |
| Can browse words  Can add new synonyms to a word  Can view synonyms of a word  Can remove a word  Can add new word |  |
| **SearchPanel** | |
| Attributes, Functions: | Collaborator classes |
| Can search for words  Can view synonyms  Can display results  Can add new synonyms to a word  Can remove word |  |

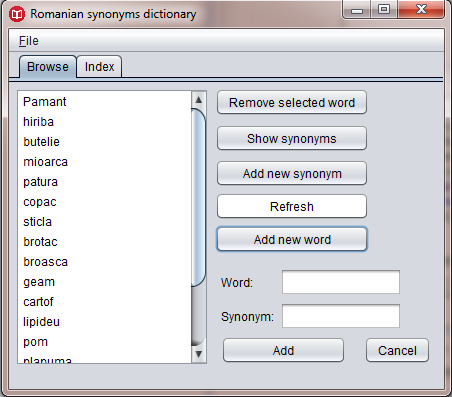
**3.5. Interface**

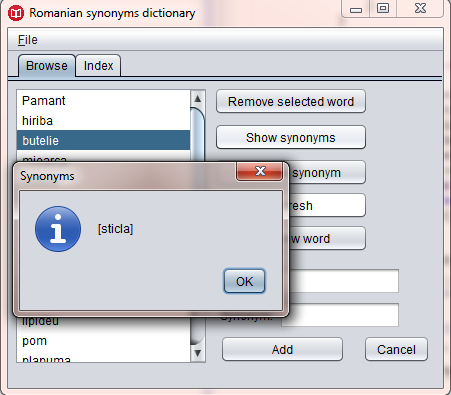
The User Interface was inspired by the java j2se7 and has the following aspect:

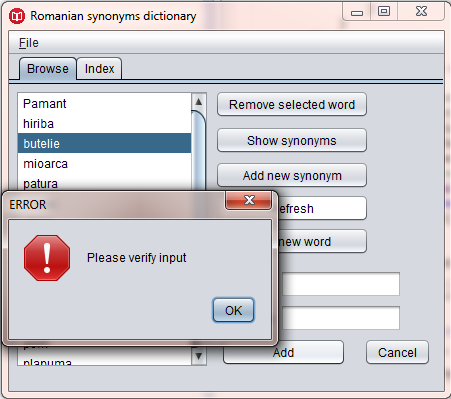
****

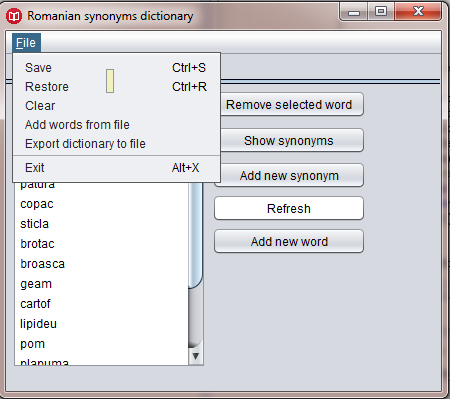
We can clearly see the tabs for browse and search(index). Also in the above picture we can see the operations available when browsing a word.

****

****

****

****

****

**4. Implementation and testing**

1. **Results**
2. **Conclusions, what has been learned, further developments**
3. **Bibliography**

[1]<http://en.wikipedia.org/wiki/Thesaurus>

<http://www.dictionardesinonime.ro/>

<http://www.vogella.com/tutorials/JavaRegularExpressions/article.html>