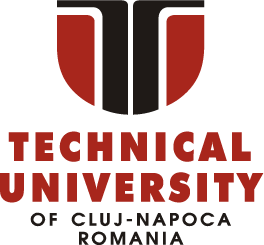
**TECHNICAL UNIVERSITY OF CLUJ-NAPOCA**

2nd year of study, Computer Science



Laboratory Work – Assignment 5

Dictionary of synonyms (Thesaurus)

Lecturer: Prof. Dr. Eng. Ioan Salomie

Teaching Assistant: Delia Balaj

Student: Diana-Gabriela Popescu

Group: 30425

**Table of contents**

**1. Introduction**

1.1 Task objectives . . . . . . . . . . . . . . . . . . . . . . . . . .page 3

1.2 Personal approach . . . . . . . . . . . . . . . . . . . . . . . .page 3

**2. Problem description**

2.1 Problem analysis . . . . . . . . . . . . . . . . . . . . . . . . .page 4

2.2 Modeling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 5

2.3 Scenarios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 5

2.4 Use cases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .page 5

**3. Projection**

3.1 UML diagrams . . . . . . . . . . . . . . . . . . . . . . . . . . page 8

3.2 Data structures . . . . . . . . . . . . . . . . . . . . . . . . . .page11

3.3 Class projections . . . . . . . . . . . . . . . . . . . . . . . .page 12

3.4 Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 17

3.5 Relationships . . . . . . . . . . . . . . . . . . . . . . . . . . . page17

3.6 Packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 18

3.7 Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 18

3.8 Graphical User Interface . . . . . . . . . . . . . . . . . . page18

**4. Implementation and testing** . . . . . . . . . . . . . . . . . . . . . . . .page 19

**5. Results** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 19

**6. Conclusions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 19

**7. Bibliography** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . page 19

**1. Introduction**

1.1 Task objectives

The task of the assignment is defined as it follows: “Design *a Dictionary of Synonyms* for Romanian or English language. It is required to use Java Collection Framework Map for the implementation.

**Problem specification**

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.

1.2 Personal approach

This documentation paper aims to present a way of solving the problem of Dictionary Management. As for user interface, there will be developed algorithms for user to interact with the Graphical User Interface (GUI) and, for some methods, the output will also be displayed in the console. The solution is obtained by means of implementing several specific operations. This type of operations were chosen due to the fact that they are the most used and the most important operations regarding Dictionary Management.

**2. Problem description**

2.1 Problem analysis

The analysis of a problem starts from examining the real model of Order Management or the model we confront with in the real world and passing the problem through a laborious process of abstractization. Hence we identify our problem domain and we try to decompose it in modules easy to implement. Always, having a good model will ease the way the operations are performed and will make more complex programs be clear to read and easy to maintain. The application should implement all the requirements given in the task description. The given task has been interpreted in the form of a **Dictionary of Synonyms**.

This Dictionary of Synonyms should provide users with the possibility of searching for words, adding new words to the thesaurus, removing unused words and listing all its content. All the information about the Dictionary’s state is saved in a file after the user closes the application.

A **thesaurus** (plural "thesauri") 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.

2.2 Modeling

Based on the information presented above, I thought on the actual implementation of the solution. I started to think about what classes are required and how classes should be organized in packages. One major advantage of having a dictionary of synonyms on the computer is the possibility of searching and finding quickly the word which is searched for. Moreover, a user will have the possibility to add his/her most often used words to the dictionary so that he/she can find them when they are needed. A thesaurus collection of words could be published also on the web, so that remote users can access it. At this point, the problem of protecting data arises, which in this small application has not been taken into account.

2.3 Scenarios

The scenarios are as it follows: the user can choose to add a synonym, to remove a synonym, to search for an entry in the dictionary. The user can choose from the available operations to be performed on the input words.

2.4 Use cases

The use cases are strongly related to the user running the application. A **use case diagram** at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different [use cases](https://en.wikipedia.org/wiki/Use_case) in which the user is involved. This application can be used from a single perspectives: regular view.

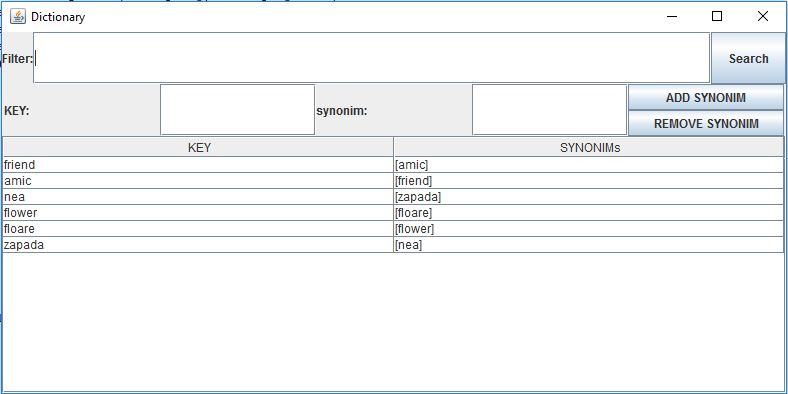


Illustration 1 : Graphical User Interface ( regular view )

**3. Projection**

3.1 UML diagrams

a) Use-case diagram

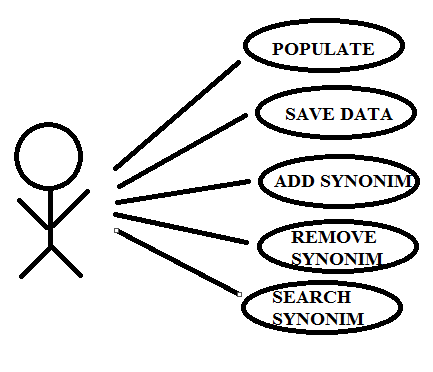


Illustration 2 : Use case diagram

The use case diagram presents the actor, which is the user that interacts with the application. He can perform several actions such as adding a new word / synonym, removing a word / synonym and searching for a word / synonym.

b) Class Diagram

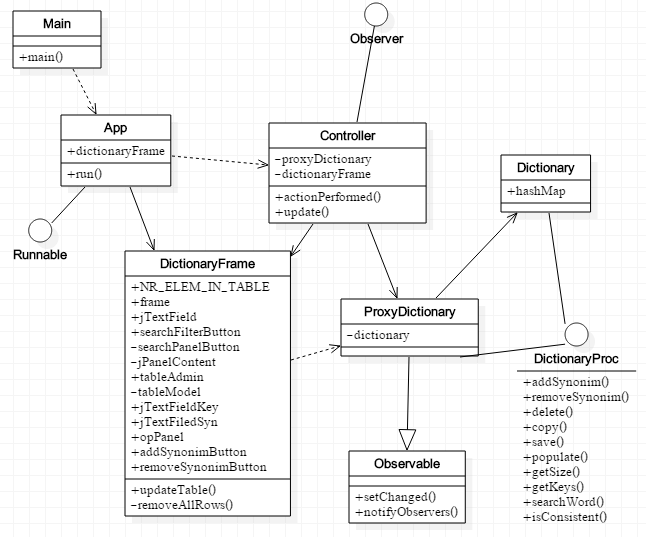


Illustration 3 : Class diagram

As presented in the class diagram, there are various relationships between the classes of the system. For example, there is one class “DictionaryFrame”, which is used by two classes: “App”, “Controller”.

c) Sequence Diagram

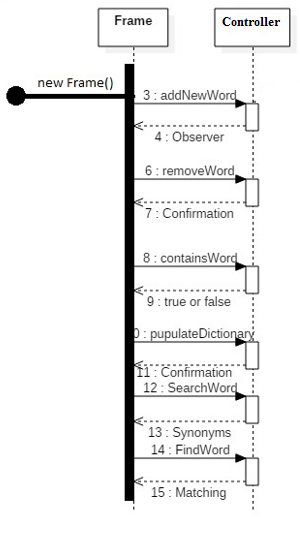


Illustration 4 : Sequence diagram

d) Package Diagram

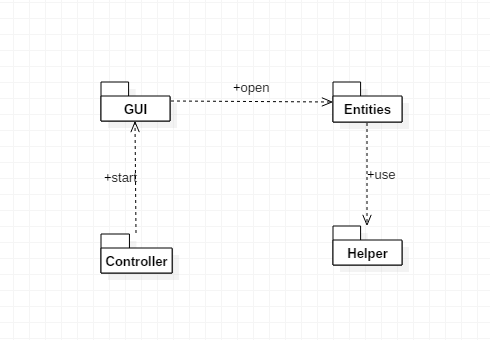


Illustration 5 : Package diagram

e) Design Pattern diagrams

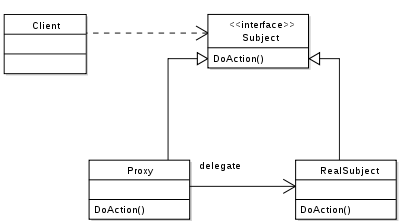


Illustration 6 : PROXY design pattern

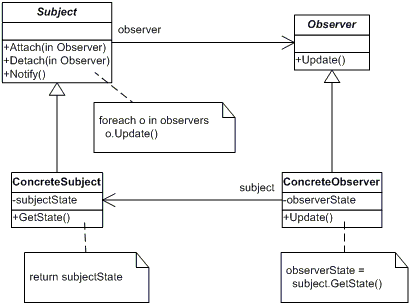


Illustration 7 : OBSERVER design pattern

3.2 Data structures

The data structures used in the application are either primitive data types ( int or double ) or new types that are defined by the designed classes. Furthermore, there were used arrays of the types presented above. Not to be forgotten is the use of GUI-like types, such as: JFrame, JButton, JLabel, JPanel.

In this project implementation I have used several data structures, but the most significant one is HashMap. Hash Map is actually 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.

Advantage: 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). It's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Disadvantage: **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.)

Chosen data structure: Hash Map

Since this application will not use multithreading, there is no problem with using Hash Map. Moreover, we can come across the situation of permitting null entries. Another advantage of using Hash Map is that this collection is collision safe, because the entries of the table act like a linked list. When you put a new entry into the same bucket, it just adds to the linked list. If the hash of the key in the map collides with an existing key, the Map will re-arrange or keep the keys in a list under that hash. No keys will get overwritten by other keys that happen so be sorted in the same bucket.

3.3 Class projections

Class projection refers mainly to how the model was thought, how the problem was divided in sub-problems, each sub-problem representing more or less the introduction of a new class. For the beginning, it has to be mentioned that, in my design, I used four different packages for organizing the classes and the interfaces. These are: “view”, “models”, “controller” and “test”. The mentioned packages are to be described next, along with the classes belonging to them.

1. Package “view” – contains the design related to the user interface (GUI and console).

* class **DictionaryFrame** – class to be used. There are plenty of other instance variables, as described below:
* **public** JFrame frame; -the specific frame.
* **public** **static** **int** *NR\_ELEM\_IN\_TABLE*;
* **private** JPanel opPanel;
* **public** JTextField jTextField;
* **public** JButton searchFilterButton;
* **private** JPanel searchPanelButton;
* **private** JScrollPane jPanelContent;
* **public** JTable tableAdmin;
* **private** DefaultTableModel tableModel;
* **public** JTextField jTextFieldKey;
* **public** JTextField jTextFieldSyn;
* **public** JButton addSynonimButton;
* **public** JButton removeSynonimButton;

The constructor of the class instantiates and initializes the instance variables to the default values.

The implemented methods are:

* **public** **void** updateTable(ProxyDictionary dictionary)
* **public** **void** updateTable(ProxyDictionary dictionary, String searchForID)
* **private** **void** removeAllRows()

1. Package “models”

* interface **DictionaryProc** - describes the capabilities of the dictionary. The methods are:
* **public** **void** addSynonim(String key, String synonim);
* **public** **void** deleteSynonim(String key);
* **public** **void** copy(String key, String synonim);
* **public** **void** delete(String key, String synonim);
* **public** ArrayList<String> searchWord(String key);//regex
* **public** **void** populate();//deserialization
* **public** **void** save();//serialization
* **public** **boolean** isConsistent();
* **public** Set<String> getKeys();
* **public** ArrayList<String> getSyn(String key);
* **public** **int** getSize();
* class **Dictionary**-implementing **DictionaryProc** interface. There is one instance variable, as described below:
* **public** Map<String, ArrayList< String > hashMap;

The constructor of the class instantiates and initializes the instance variables to the default values.

The class implements the methods described by the “DictionaryProc” interface :

* **public** **void** addSynonim(String key, String synonim);
* **public** **void** deleteSynonim(String key);
* **public** **void** copy(String key, String synonim);
* **public** **void** delete(String key, String synonim);
* **public** ArrayList<String> searchWord(String key);//regex
* **public** **void** populate();//deserialization
* **public** **void** save();//serialization
* **public** **boolean** isConsistent();
* **public** Set<String> getKeys();
* **public** ArrayList<String> getSyn(String key);
* **public** **int** getSize();
* class **ProxyDictionary** – extending Observable and implementing **DictionaryProc** interface. There is one instance variable, as described below:
* **private** Dictionary dictionary = **new** Dictionary();

The constructor of the class instantiates and initializes the instance variables to the default values.

The class implements the methods described by the “DictionaryProc” interface :

* **public** **void** addSynonim(String key, String synonim);
* **public** **void** deleteSynonim(String key);
* **public** **void** copy(String key, String synonim);
* **public** **void** delete(String key, String synonim);
* **public** ArrayList<String> searchWord(String key);//regex
* **public** **void** populate();//deserialization
* **public** **void** save();//serialization
* **public** **boolean** isConsistent();
* **public** Set<String> getKeys();
* **public** ArrayList<String> getSyn(String key);
* **public** **int** getSize();

1. package “controller”

* class **Controller** -implementing Observer and ActionListener describes the relationships between the GUI interface and the logical structures of the application.

The instance variables are:

* **private** DictionaryFrame dictionaryFrame;
* **private** ProxyDictionary dictionary;

The constructor of the class instantiates and initializes the instance variables to the default values.

Implemented methods:

* + - public void actionPerformed(ActionEvent event) - for dealing with the events that occur when interacting with the GUI.
    - public void update( Observable arg0, Object arg1)
* class **App** - implementing Runnable describes the way the application runs. It has a “DictionaryFrame instance” variable:
* private DictionaryFrame dictionaryFrame;

The constructor of the class instantiates and initializes the instance variable to the given value.

Then, it instantiates a Controller object by calling its constructor with the instantiated “DictionaryFrame” as parameter, all this in one method:

* public void run()
* class **Main** - contains the main method. It runs the App.

**public** **static** **void** main(String[] args) {

**new** App().run();

}

1. package “test”

* JUnit **Test** – is where the testing of the available methods.

The instance variables are:

* **private** ProxyDictionary proxyDictionary;

The tested methods:

* **public** **void** addSynonim(String key, String synonim);
* **public** **void** deleteSynonim(String key);
* **public** **void** copy(String key, String synonim);
* **public** **void** delete(String key, String synonim);
* **public** ArrayList<String> searchWord(String key);//regex

3.4 Interface

This section being already developed in detail until now, I will remember briefly some important facts. The user interface is mainly realized by means of “java.swing” package. I used instances of JFrame, JPanel, JButon, JLabel. Also, I used FlowLayout and GridLayout for organizing components inside the frame and panels. The buttons and the selection of a cell in the table are the main way of the user interacting with the application.

3.5 Relationships

As presented in the class diagram, there are various relationships between the classes and interfaces of the system. For example, there is one interface “DictionaryProc”, which is implemented by “Dictionary” and “ProxyDictionary”.

There are also some other dependencies and associations due to the fact that some methods use as parameter types of a different class.

3.6 Packages

The program is divided in four packages, as mentioned before. The package “view” contains class: “DictionaryFrame”. The package “models” contains classes: “Dictionary”, “ProxyDictionary” and the interface “DictionaryProc”. The package “test” contains JUnit “Test”. The package “controller” contains classes: “Controller”, “App” and “Main”.

3.7 Algorithms

The main algorithms regarding dictionary are the following: adding a new word / synonym, removing a word / synonym, search for a word / synonym. They are designed as methods handling holders and accounts, as it follows:

* **public** **void** addSynonim(String key, String synonim);
* **public** **void** deleteSynonim(String key);
* **public** **void** copy(String key, String synonim);
* **public** **void** delete(String key, String synonim);
* **public** ArrayList<String> searchWord(String key);//regex

3.8 Graphical User Interface

An important fact to be mentioned is the use of “java.swing” package. I used instances of JFrame, JPanel, JButoon, JSlider, BasicArrowButton, JLabel. Also, I used FlowLayout and GridLayout for organizing components inside the frame and panels. The buttons and the sliders are the main way of the user interacting with the application.

**4. Implementation and testing**

The implementation was done in Eclipse and it was also tested in this environment. However the program should maintain its portability. Concerning the code implementation I did not make use of laborious algorithms, but I have rather stayed faithful to the classical algorithm. The personal touch in the implementation is also felt in the way the Graphical User Interface is thought.

Testing implies checking for any errors and warnings in the program or limitations of this program. A testing class was implemented. In this case, for each method, an exception is thrown, which can also indicate a possible error when running the app. So, from this point of view, there are lots of errors that will be avoided this way. Other possible scenarios will be tried as future development.

**5. Results**

The application is user friendly and useful in performing basic order/customer operations such as: deposit, withdraw, adding a new account, removing an account, adding a new holder. As the application is developed on a Java platform, it is highly portable and allows it to run on several operating systems. The application is to be used by anyone who is interested in performing these types of operations and has a basic knowledge about bank, holders and accounts.

**6. Conclusions**

The application can be further developed, by adding some other operations on dictionary, words and synonyms. For me personally, the design of this system helped me think about the way classes should be organized in packages.

**7. Bibliography**

* <http://staruml.io/>

-for downloading a tool for creating class diagrams.

* <http://bellekens.com/2010/12/20/uml-composition-vs-aggregation-vs-association/>

-for better understanding of the relationships between classes.