*Technical University of Cluj-Napoca  
Faculty of Automation and Computers  
Department of Computer  
2nd Semester 2016*

*Programming Techniques*

*Homework 5*

*Student: Ghiurutan Alexandru   
 Group: 30425*

***Contents:***

1. *Problem Specification ................................................................ 3*
2. *Example of working .............................................................. 4*
3. *Design*

*3.1 Use case Diagram …............................................... 5*

*3.2 UML Diagrams …………………………………… 6*

*3.3 Sequence Diagram………………………………….. 7*

*3.4 Package Diagram…………………………………... 8*

*3.5 Classes Design ............................................................ 9*

*3.5.1 Gui Class*

*3.5.2 InOut Class*

*3.5.3 SearchView Class*

*3.5.4 Dictionary Class*

*3.5.5 Word Class*

*3.6 Packages and Interfaces............................................. 15*

*3.7**Runnable Interface …............................................... 16*

*3.8 User Interface........................................................... 17*

1. *Using and testing the application …………………………….. 19*
2. *Conclusions …………………..……………………………..... 19*

***1. Problem Specification***

TP Lab–Homework 5

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.

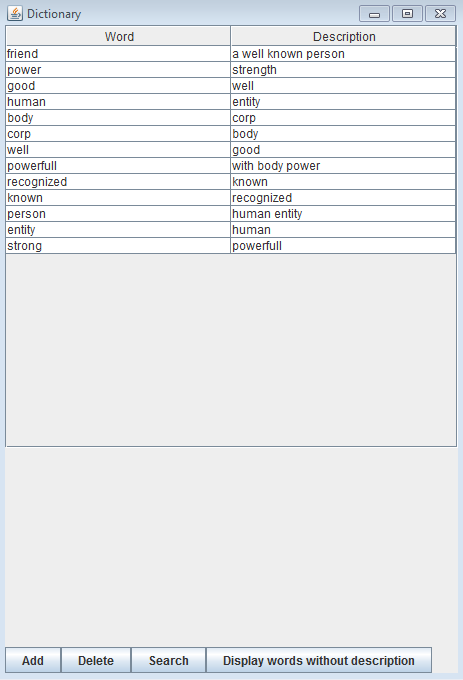
Note.

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

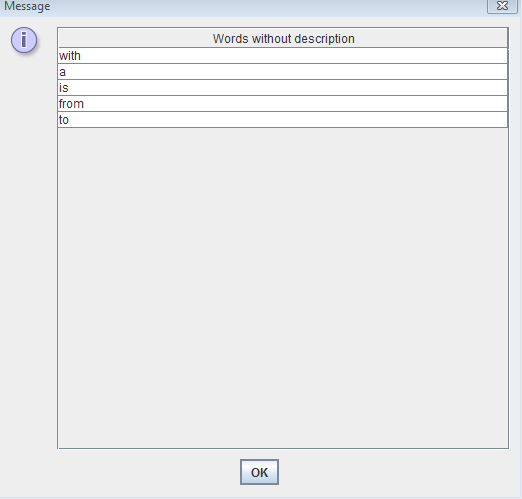
the interface from implementation, javadoc, etc.) will be also used for this homework.

***2. Example of working***

In the picture below ,we can see the logging window from the start , were both the administrator and the customers from the bank can enter .



If we log in as admin we can see the window below ,with all the buttons as options like : displaySpendingAccounts , addPerson , removePerson etc .

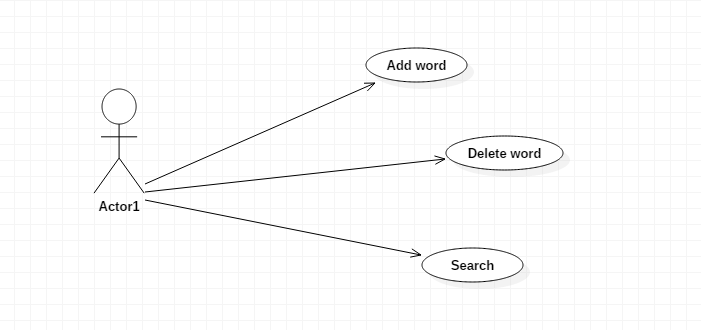
******

***3. Design***

***3.1 Use case Diagram***

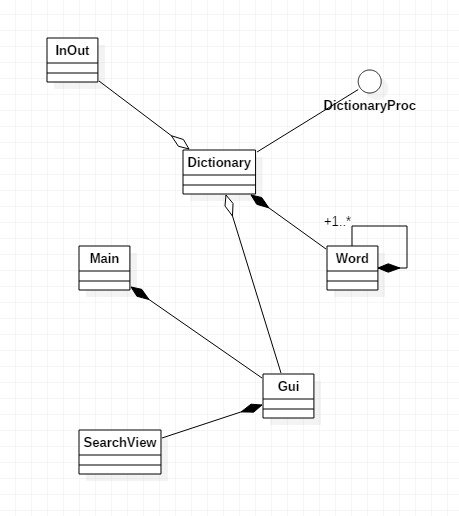
For resolving the problem specifications, I’ve chosen to use four different classes so that the design of the application would be as good as possible. The names of the classes and relations between them are represented in the figure below.

About their use and the reasons why I have chosen to organize the objects as it is written are explained in section 3.2: “Classes Design”



To get a better view related to the attributes of each class, there are below the UML diagrams for each class. Thus, we can see every class with objects and their methods.

***3.2 UML Diagram***



***3.3 Sequence diagram***

Word

Gui

Dictionary

Create Word

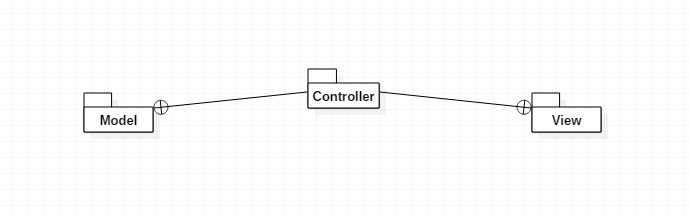
Retrives Word

Add Word to dictionary

Returns the HashMap of Words

Prints table

***3.4 Package Diagram***

******

***3.5 Classes Design***

1. Gui Class : public class Gui extends JFrame implements ActionListener

This class is designed to create a graphical user interface so the application would be easier to use. We consider the Gui class as a subclass of the predefined class "JFrame" so we can use objects of type "button", "frame" or "panel".

In this class there is declared the function public static void main (String [] args) which makes the whole application running. As a result of this run, on the screen it will open a window through which the user can enter data and perform operations that are possible with this application.

Observe the six TextFields for reading the necessary information at the top of the window. The user can insert any integer, each of them representing:

Number of existing queues in the shop

Service time interval: for each client, the program randomly generates a value in the specific range (contains the minimum and the maximum); this has the significance of time which takes for the customer to be served

Range of arrival: like the “service interval” this is randomly generated for each customer when he/she arrives at a queue.

Simulation time

There are two buttons: one to start the program after input data were introduced (Start) and the other for closing the application (Quit).

Below them there is a space for placing the graphical representation of the queues and the clients for this shop simulation.

The Attributes of the Gui Class:

To achieve the desired GUI we need several types of attributes:

The Constructor of the Gui Class: public Gui ()

It initializes all the attributes declared above; the window with the following elements will be constructed: frame, buttons, text fields. Here there are the frame settings, such as the size, the visibility, title and some predefined operation such as:

public Gui ( ) {

this.setTitle("Dictionary");

this.setLocationRelativeTo(null);

dictionary = Dictionary.getInstance();

dictionary.start();

inOut = new InOut();

jT = new JTable();

initializeNorthSection();

operations = new JPanel();

operations.setLayout(new BoxLayout(operations, BoxLayout.X\_AXIS));

add = new JButton("Add");

add.addActionListener(this);

delete = new JButton("Delete");

delete.addActionListener(this);

search = new JButton("Search");

search.addActionListener(this);

displayOthers = new JButton("Display words without description");

displayOthers.addActionListener(this);

operations.add(add);

operations.add(delete);

operations.add(search);

operations.add(displayOthers);

this.add(operations, BorderLayout.SOUTH);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.pack();

this.setVisible(true);

}

What is more, for every button (operation) there will be added an “ActionListener” which contains the instructions that are needed to be executed in the moment a certain operation button is clicked. For each button there is a different class declared inside the Gui class which implements the “ActionListener” interface.

More information about the “ActionListener” implementation are in the chapter about the user interface where there are presented all the swing components used in the Gui constructor.

The Methods of the Gui Class:

Gui contains methods for achieving operations when you press one of the two existing buttons in the interface. Moreover, it contains the main function which is run every time the application is opened.

initializeFields ( ) method:

This method aims to achieve the interface, this means that within this function we declare a new object's constructor called Gui to achieve early window:

Methods for the button:

It declares a class which implements the predefined class called ActionListener. To get the information from the text field we use a function :

String userText = userArea.getText ( ) ;

1. InOut Class: public class InOut

This class uses XML to serialize and deserialize the hashmap of the dictionary and the set with words without description.

Methods of the InOut Class:

public void writeDictionary() {

dictionary = Dictionary.getInstance();

XMLEncoder encoder = null;

try {

encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("words.xml")));

} catch (FileNotFoundException fileNotFound) {

System.out.println("ERROR: While Creating or Opening the File words.xml");

}

encoder.writeObject(dictionary.getDictionary());

encoder.close();

}

@SuppressWarnings("unchecked")

public HashMap<Word, String> readDictionary() {

HashMap<Word, String> result = null;

try {

XMLDecoder decoder = new XMLDecoder(new BufferedInputStream(new FileInputStream("words.xml")));

// @SuppressWarnings("unchecked")

result = (HashMap<Word, String>) decoder.readObject();

decoder.close();

} catch (FileNotFoundException e) {

System.out.println("ERROR: While Creating or Opening the File words.xml");

}

return result;

}

public void writeOtherWords() {

dictionary = Dictionary.getInstance();

XMLEncoder encoder = null;

try {

encoder = new XMLEncoder(new BufferedOutputStream(new FileOutputStream("others.xml")));

} catch (FileNotFoundException fileNotFound) {

System.out.println("ERROR: While Creating or Opening the File others.xml");

}

encoder.writeObject(dictionary.getOtherWords());

encoder.close();

}

@SuppressWarnings("unchecked")

public HashSet<String> readOtherWords() {

HashSet<String> result = null;

try {

XMLDecoder decoder = new XMLDecoder(new BufferedInputStream(new FileInputStream("others.xml")));

result = (HashSet<String>) decoder.readObject();

decoder.close();

} catch (FileNotFoundException e) {

System.out.println("ERROR: While Creating or Opening the File others.xml");

}

return result;

}

SearchView Class: public class AdminView extends JFrame

SearchView contains methods for displaying the words that you seached.

The Attributes of the SearchView Class :

private Set<Entry<Word, String>> words;

private DefaultTableModel tableModel;

private JScrollPane scrollPane;

private JTable table;

private Object[][] rows;

private Object[] columns = { "Word", "Description" };

The Constructor of the AdminView Class:

public SearchView(Set<Entry<Word, String>> words) {

this.setTitle("Search");

this.setLocationRelativeTo(null);

this.words = words;

initializeTable();

this.pack();

this.setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

this.setVisible(true);

}

The methods of the SearchView Class:

private void initializeTable() {

tableModel = new DefaultTableModel(rows, columns);

table = new JTable();

table.setModel(tableModel);

table.setEnabled(false);

fillTable();

scrollPane = new JScrollPane(table);

this.add(scrollPane, BorderLayout.NORTH);

}

private void fillTable() {

if (words.size() != 0) {

rows = new Object[words.size()][2];

Iterator<Entry<Word, String>> it = words.iterator();

while (it.hasNext()) {

Entry<Word, String> newWord = it.next();

Object[] row = { newWord.getKey().getWord(), newWord.getValue() };

tableModel.addRow(row);

}

}

}

public void updateTable() {

if (words.size() != 0) {

tableModel = new DefaultTableModel(rows, columns);

tableModel.setRowCount(0);

table.setModel(tableModel);

rows = new Object[words.size()][2];

Iterator<Entry<Word, String>> it = words.iterator();

while (it.hasNext()) {

Entry<Word, String> newWord = it.next();

Object[] row = { newWord.getKey().getWord(), newWord.getValue() };

tableModel.addRow(row);

}

}

}

1. Dictionary Class: public class Dictionary implements Observer, DictionaryProc

Here are made the main operations of the the application.

The Attributes of the CustomerView Class:

private static Dictionary instance;

private HashMap<Word, String> dictionary;

private Set<String> otherWords;

private Iterator<Word> it;

private Iterator<Entry<Word, String>> iterator;

private InOut inOut;

The Constructor of the Dictionary Class:

private Dictionary() {

}

The methods of the CustomerView Class:

public void start()

public void addWord(Word word, String explanation)

public void addOtherWord(String otherWord)

public boolean containsOtherWord(String otherWord)

public void removeWord(Word word)

public boolean containsWord(String newWord)

public Word getWord(String word)

public void printContent()

public int getTotalNrOfWords()

Public Set<Entry<Word,String>> getAllWordsInDictionary()

public HashMap<Word, String> getDictionary()

public HashSet<String> getOtherWords()

public Set<Entry<Word, String>> getSearchResults(String searchWord)

public boolean isWellFormed()

public void actionPerformed(ActionEvent e)

**3.6 *Packages and Interfaces***

A Java package is a mechanism for organizing Java [classes](http://en.wikipedia.org/wiki/Class_%28computer_science%29) into [namespaces](http://en.wikipedia.org/wiki/Namespace_%28computer_science%29). Java packages can be stored in compressed files called [JAR files](http://en.wikipedia.org/wiki/JAR_file), allowing classes to download faster as a group rather than one at a time. Programmers also typically use packages to organize classes belonging to the same category or providing similar functionality. A package provides a unique namespace for the types it contains. Classes in the same package can access each other's package-access members.

A package allows a developer to group classes (and interfaces) together. These classes will all be related in some way – they might all have to do with a specific application or perform a specific set of tasks.

For this application the following packages are imported, each of them having a certain role for the proper working of the application. We import them in the Gui Class (most of them relate to the user interface properties):

* import java.awt: Contains all of the classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components.
  + java.awt.BorderLayout: A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center.
  + java.awt.Color: The Color class is used encapsulate colors in the default RGB color space or colors in arbitrary color spaces identified by a [ColorSpace](http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/color/ColorSpace.html).
  + java.awt.Dimension: This encapsulates the width and height of a component (in integer precision) in a single object.
  + java.awt.GridLayout: The GridLayout class is a layout manager that lays out a container's components in a rectangular grid for a better view of all the buttons and textfields which are added to the main panel.
* import java.awt.event
  + java.awt.event.ActionEvent;
  + java.awt.event.ActionListener;
* import javax.swing: Typical Swing applications do processing in response to an event generated from a user gesture. For example, clicking on a JButton notifies all ActionListeners added to the JButton. That’s why we use this package for creating the user interface Gui.
  + javax.swing.JButton;
  + javax.swing.JFrame; javax.swing.JLabel; javax.swing.JPanel;

The Oberver design pattern and Observer interface:

Observer pattern is one of the behavioral design pattern. Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called Observer and the object that is being watched is called Subject. According to GoF, observer pattern intent is;

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Subject contains a list of observers to notify of any change in it’s state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change and either it can send the update while notifying the observer or it can provide another method to get the update.

Observer should have a method to set the object to watch and another method that will be used by Subject to notify them of any updates.

Java provides inbuilt platform for implementing Observer pattern through java.util.Observable class and java.util.Observer interface. However it’s not widely used because the implementation is really simple and most of the times we don’t want to end up extending a class just for implementing Observer pattern as java doesn’t provide multiple inheritance in classes.

Java Message Service (JMS) uses Observer pattern along with Mediator pattern to allow applications to subscribe and publish data to other applications.

Model-View-Controller (MVC) frameworks also use Observer pattern where Model is the Subject and Views are observers that can register to get notified of any change to the model.

The Singleton design pattern

The Singleton pattern is deceptively simple, even and especially for Java developers. In this classic JavaWorld article, David Geary demonstrates how Java developers implement singletons, with code examples for multithreading , class loaders , and serialization using the Singleton pattern. He concludes with a look at implementing singleton registries in order to specify singletons at runtime.

The Composite design pattern

Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

We are demonstrating use of composite pattern via following example in which we will show employees hierarchy of an organization.

Model View Controller design pattern

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

Model - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.

View - View represents the visualization of the data that model contains.

Controller - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

***3.7 Runnable Interface***

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being active simply means that a thread has been started and has not yet been stopped.

In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behaviour of the class.

In this program for each queues from the array of queues declared in the Shop class, we declare a thread. This means that we will have a run method in the class Queue which will determin whether or not to execute an action depending on the time. In this case the queue will remain unchanged while a client is being served and only after this time has passed the client is removed from the queue. This is done with the following instructions:

try {

Thread.sleep(time);

}

catch (InterruptedException e) {}

clienti.remove();

***3.8 User Interface***

When running the application, the window will open and it will provide to the user the possibility of giving inputs and choosing what operation he likes to be executed. This window is constructed in the Gui class using some predefined classes and instructions.

The user interface is based on the properties of the above mentioned packages. All the objects we need are declared as attributes of the Gui class and they are initialized in the constructor of this class. For executing the operation commanded by the user we use the predefined functions from the ”ActionListener” interface.

The ActionListener functions

The listener interface is for receiving action events. The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method. When the action event occurs, that object's actionPerformed method is invoked. In this case the only events that occur are when the user clicks on one of the operation buttons from the graphical interface.

We take as an example the instructions that need to be executed when clicking on the “P(val)” button, which leads to the determination and displaying the value of the polynomial written in ”First Polynomial” field in point val. For this we need another class which implements the ActionListener:

private class StartListener implements ActionListener

This class will contain the method which executes all the instruction needed in order to fulfill the selected operation.

public void actionPerformed (ActionEvent e)

***4. Using and testing the application***

The application was tested with the JUnit classes:

import org.junit.Test;

import static org.junit.Assert.assertEquals;

In order to use the application open Homework4.JAR. This will open a window which generates the Gui class. Thus the user can enter the desired values and selecting the operations by pressing one of the buttons.

***5.Conclusions***

Achieving such a program may be hard both in terms of algorithms, graphical structure.

For a better performance there should be implemented all cases where exceptions can occur and the application stops working due to an error made ​​by the user. Also, the division method should be done so it can calculate the reminder as well. Another thing that could be improved is the display so that it would be more elegant.

***6.References***

[*http://users.utcluj.ro/~jim/OOPE/*](http://users.utcluj.ro/~jim/OOPE/)

[*http://docs.oracle.com/javase/7/docs/api/overview-summary.html*](http://docs.oracle.com/javase/7/docs/api/overview-summary.html)

[*http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html*](http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html)

[*http://stackoverflow.com/*](http://stackoverflow.com/)