Dictionary Application

Ghiurău Alexandra Cristina

Technical University of Cluj Napoca

Group: 30424

**Abstract.** Lack of simple, easy to use applications in the field of *Dictionaries* led to an effective and very intuitive approach on creating one. Using only Java and Android SDK for both the user interface and background programming, one was able to develop a standalone *Dictionary* application which includes a multitude of functions and options.

**Introduction.** The problem stands in the lack of a more general application for handling Dictionary related activities. This field, the one of *Dictionaries* is of great importance both in Literature and Computer science with a wide range of applicability. It is no simple matter dealing with Dictionaries, thus the need for a more comprehensive application grew greater as time went by.

The application supports the following operations: Adding a word alongside synonyms, removing a word and its definitions, searching for words and their definitions, loading words and definitions from a file and saving the newly formed dictionary to the file.

**Objective of project.** To design, program and test a *Dictionary Application* capable of simulating a variable number of requests in different situations.

**Problem analysis**

1. **Modeling:**

We have a single entity which is modeled here, that is, the Dictionary.

*Dictionary class:*

* Further details about implementation will be discussed in the latter sections of the documentation.

Note that, there are more classes than the ones specified above. The other classes will be thoroughly presented in the later section of the documentation entitled *class development*.

1. **Scenarios.**

We start with a formal definition of a *scenario.*

*A scenario is a particular sequence of sequences which develops from the beginning to the end of a “use case”.*

1. ***Identification summary:***

**Title:** *Inserting a word and its definitions.*

**Summary:** this use case scenario allows the user to add a word and its definitions.

**Actors:** User which inputs all the above mentioned data.

**Creation date:** 24/05/16

**Date of update:** 24/05/16

**Version:** 1.0

**Person in charge:** Ghiurău Alexandra.

1. ***Flow of events:***

*Preconditions:* - the user interface did not malfunction in any way and the system on which the application runs on has the minimum required resources for the application to run “smoothly”.

*Main success scenario:*

1. The user inserts the word.
2. The user inserts the synonyms.
3. The user presses *add*.
4. Program adds the word and definitions in the hash map.
5. The user is free to insert other data and / or press any button for any operation which will be performed.

\* Provided that there are so few steps, there is no apparent need for splitting the *Flow of events* into two separate columns, one containing the Actor actions, and the other containing the system actions.

*Alternative sequences:*

A1) the auser leaves a field without inserting any value in it: **step 1 kicks in**

1. Program informs the user that “something went wrong”.

**Scenario returns to step 1.**

*Error sequences:*

**E1) The operation somehow freezes the application thus rendering it unable to accept anymore user input: the user will know when this happens and will have to restart the application (FAIL!)**

*Post-conditions:* none.

1. ***UI Requirements:***

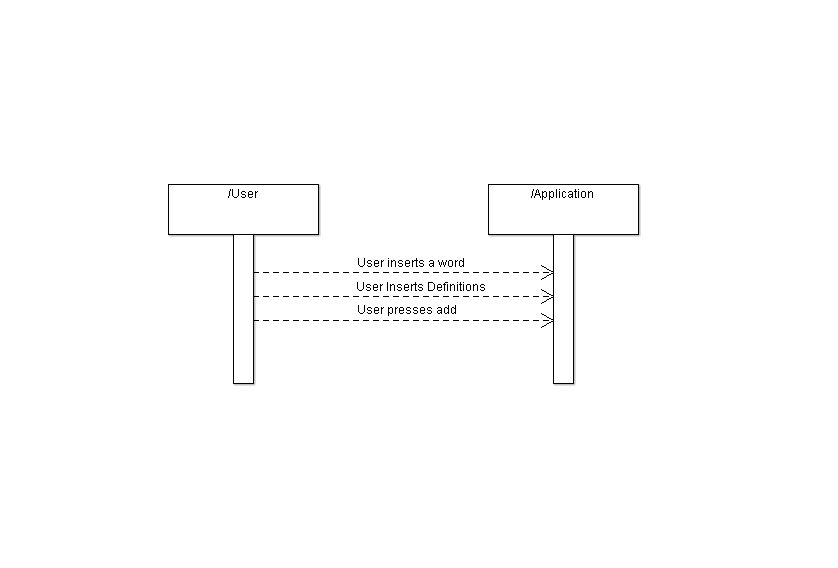
The I/O mechanism for the *Dictionary Application* must contain:

* One Java compatible device which can compile and run Java standalone applications
* A keyboard from which the user can input a number of seconds (optional).
* A mouse is not needed but strongly recommended if the user cannot operate with the designated mnemonics.
* A screen onto which he can see the input / output for the application.

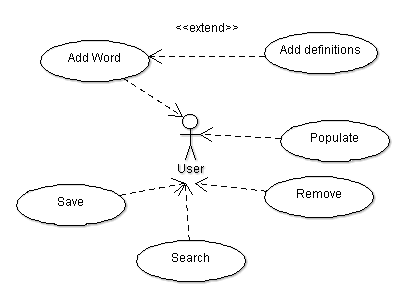
1. ***Non-functional constraints***

* *Response time:*
* No constraints on this part. One can use the Application indefinitely.
* *Availability*
* Application is available 24/7

Based on the “*Main Success Scenario”* we can build the following *Sequence Diagrams:*



Next, we will examine the *Use Case Diagram* which revolves around the main actor, the user. It includes all the relations between the possible cases the user may find himself in.



**Design**

1. **UML Diagrams**

(\*A ClassDiagram.png will be present in the attached .zip file for a clearer view\*)

**2. Data Structures**

*The following classes will represent the Dictionary.*

***Dictionary*** class:

The role of this class is to simulate a dictionary.

*Attributes:*

* *dictionary*: a HashMap object which represents the whole dictionary

*Methods:*

* *addSynonym:* adds a synonym
* *removeSynonym:* remove a synonym
* *getSynonyms:* returns synonyms
* *loadDefinitions:* loads from file
* *save:* saves to file
* *populate:* populates / parses from file
* *searchDefinition:* searches a definition
* *isConsistent:* invariant method
* *removeDefinition:* removes a definition
* *addDefinition:* adds a definition

***Other*** *data structures used:* none worth mentioning.

1. **Algorithms**

*The most notable* algorithm is the *isConsistent* invariant:

**public** **boolean** isConsistent(){

**for** (String word : dictionary.keySet()) {

**for** (String syn : dictionary.get(word)) {

**try** {

**if** (!dictionary.get(syn).contains(word))

**return** **false**;

} **catch** (Exception e) {

**return** **false**;

}

}

}

**return** **true**;

}

Another algorithm worth mentioning is the insertion in dictionary:

**public** **void** addSynonym(String word, String synonym){

ArrayList<String> values = dictionary.get(word);

**if** (values == **null**) {

values = **new** ArrayList<String>();

}

**if** (!values.contains(synonym)) {

values.add(synonym);

dictionary.put(word, values);

}

**assert**(isConsistent());

}

Of course another algorithm worth mentioning is the search in dictionary(I have used pattern syntax to find matches from words searched found in java.util.regex.Pattern):

**public** ArrayList<String> searchDefinition(String word){

**assert**(dictionary!=**null**);

ArrayList<String> results = **new** ArrayList<String>();

**int**[] v=**new** **int**[100];

**for**(**int** i=0;i<v.length;i++){

v[i]=0;

}

**for**(**int** i=0;i<word.length();i++){

**if**(((word.charAt(i)=='?') || (word.charAt(i)=='\*')) && i==0){

word.replace(word.charAt(i), '+');

word='.'+word;

}

**else** **if**(word.charAt(i)=='?'){

word=word.substring(0,i)+"."+word.substring(i+1,word.length());

}

**else** **if**(word.charAt(i)=='\*'){

word=word.substring(0,i)+"."+word.substring(i+1,word.length());

v[i]++;

}

System.*out*.println("transf word is: "+word);

}

**try**{

**for**(**int** i=0;i<v.length;i++){

**if**(v[i]!=0){

word=word.substring(0,i+1)+"\*"+word.substring(i+2,word.length());

}

}

} **catch**(Exception e){

}

**try** {

Pattern p = Pattern.*compile*(word);

**for** (String candidate : dictionary.keySet()) {

Matcher m = p.matcher(candidate);

**if** (m.find()) {

results.add(candidate);

}

}

} **catch** (PatternSyntaxException e) {

System.*out*.println("Incorrect pattern syntax: " + word);

}

**return** results;

}

1. **Class Design**

A simple yet effective approach is found to the problem of *class design* for this specific project. Not much to say here though…

We have the *Dictionary* class, the *DictionaryInterface* interface and the Main class.

1. **Interfaces**

*DictionaryInterface* is the dictionary interface which contains all the methods which the *Dictionary* class will include.

1. **Relations**

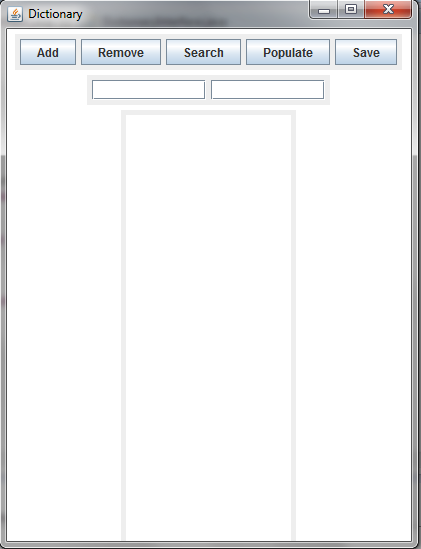
*Associasons* are present between the following classes:

* *SynonymMain* and *Dictionary* classes with the *dictionary* object.

1. **Packages**

In the current implementation, all classes are saved in one package. Further development may bring a repartition in different packages.

1. **User Interface**



A a a a aa a

**Implementation and testing**

**Performance and Code Efficiency**

Code efficiency can be found in the whole application. A very limited number of code lines usually do a lot of work and this can be seen in all the classes. Most of the classes have a comment to code ratio of about 50%.

Using native Android components made everything *that more* efficient. Even though it is not clearly seen, the whole program is rather constructed in the MVC (Model, View, Controller) paradigm. The whole *View* parts are covered only in the *Activity* classes while the *Model* is depicted from the *Helper* classes and the *Controller* in the other classes.

The application did not require very complex data structures to be used; rather everything had to be implemented as light as possible to enable reusage of code. Avoiding complications made it easier for an efficient implementation to emerge from coding and design.

**Graphical User Interface**

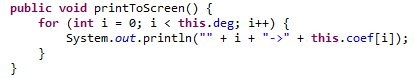
The whole GUI can is covered in the previous chapters.

**Testing**

Several testing methods were applied in the different stages of the development process:

*Using toString() methods:*

Each class was provided with *toString()* methods to facilitate the output of a certain variable / data to the console line. Using System.out.prinln(…) in each of these methods enabled a quick feedback of important data which was of great importance in the early stages of the developing process. Here is an example of how such a method looks like:



Pros: quick and easy;

Cons: messy and in some cases impractical;

*Using the Debugger:*

Eclipse provides quite an intuitive debugger which enables tracing in the deepest areas of the code. Both in the middle part of the developing process and when applying the finishing touches, the *debugger* tool was indispensable. Data structures like *arrays* and *queues* prove very difficult to trace without a debugger and as the application mainly revolves around such data structures, one cannot stress out how useful the *Eclipse Debugger* was.

Pros: enables in depth tracing and step by step execution tracing;

Cons: time consuming and may prove difficult at first;

*Trial and error:*

When first encountering a problem, you don’t usually jump straight away to the *debugger* tool. First of all, you try again to see if it “busts” the second time. If it does, you rewrite a little code which, in the first few seconds, you think causes the problem. It is worth mentioning that, in *many* cases, even if this solution may not completely solve the problem, it solves around 60% of it. And that is a big deal, for a few seconds of rewriting some code! Trial and error comes in handy especially when there are no major algorithms which need be considered or no complex data structures involved. There is always the risk that your quick solution may be in the detriment of the final solution, but it is a risk well worth taking, considering how fast one could undo the last changes. (ctrl+z)

Pros: way fast!

Cons: doesn’t usually solve the problem, and if it does, it mostly only contributes to the final solution.

*Logical rethinking:*

This isn’t really a testing method rather than it being a developing method, however. Some modules may be written in the “urge” of the moment without first developing the class diagrams, and so on. Therefore, when something goes wrong, the first thing one should do is grab a piece of paper and start solving the problem *on that piece of paper*, one logical step at a time. It is amazing how many logical inconsistencies can slip through a programmers mind when he’s in a *mad state* of coding. This “debugging” method can easily repair those inconsistencies and save a lot of later repair work.

Pros: if done in time, saves o lot of future work.

Cons: requires time, a piece of paper and a pencil.

**Results**

Through hard work and intensive testing, one achieved a pretty robust standalone application for *Polynomial Processing.* The following table will show the functionality progress of the implemented methods:

|  |  |
| --- | --- |
|  |  |
| Function | **Final result** |
| *Insertion of a word* | Fully functional |
| *Inserting definitions* | Fully functional |
| *Load data* | Fully functional |
| *Save data* | Fully functional |
| *Remove word*  *Search Word* | Fully functional  Fully functional |

**Conclusions**

***What did this project teach you?***

Besides renewing my knowledge about Dictionaries, the project thought me how easy it is to implement quite complex functions in a very limited number of code lines.

The importance of starting with the class design and so on before coding was also pointed out in this first project. Even though it was not such a complex project and it did not require a lot of class design, it is an important thing to learn about approaching a problem in an *Object Oriented Programming* way.

The last important thing this project thought me was that a little hard work in the beginning can save you a lot on performance later on in the developing process. Choosing to dynamically implement the queues with lists proved to be great choice. One did not have to deal with size-related matters !

**Further developments**

Oh jolly, I couldn’t wait to get to this part.

1. Implement graphics!
2. Make an applet out of it!
3. Make an option to be able to save the result of an operation in a file
4. Improve GUI aspect

**Bibliography**

1. [www.google.com](http://www.google.com)
2. www.oracle.com

22)