

## Universitatea Tehnică Cluj-Napoca

## Facultatea de Automatică şi Calculatoare

## Secţia: Calculatoare, engleză

## Programming Techniques

## ~ Dictionary ~

Student: Ban Bogdan Marian

Group id: 30425, second year

Professor: Salomie Ioan

Table of Contents

1. Task Objectives. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 03

2. Problem analysis, modelling, scenarios, use cases. . . . . . . . . . . . . . . . . .03

3. Projection (UML diagrams, data structures, class projections, . . . . . . . .08

Interface, relations, packages . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .09

Sequence ). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .09

4. Implementation and testing. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

5. Results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

6. Conclusions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

7. Bibliography. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .11

1. Task Objectives

1. Study the Java Collection Framework Map

https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html

2. 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.Problem analysis, modelling, scenarios, use cases

The problem involves the construction of a dictionary that can be populated with terms and their definitions, we need to have the opportunity to look for words in the dictionary. Other operations allowed as addition and removal of terms but dictionaries finish populating read from external files. We need a data structure to retain terms and their definitions.

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.

We need assertions contracts and to ensure the proper functioning of the application.

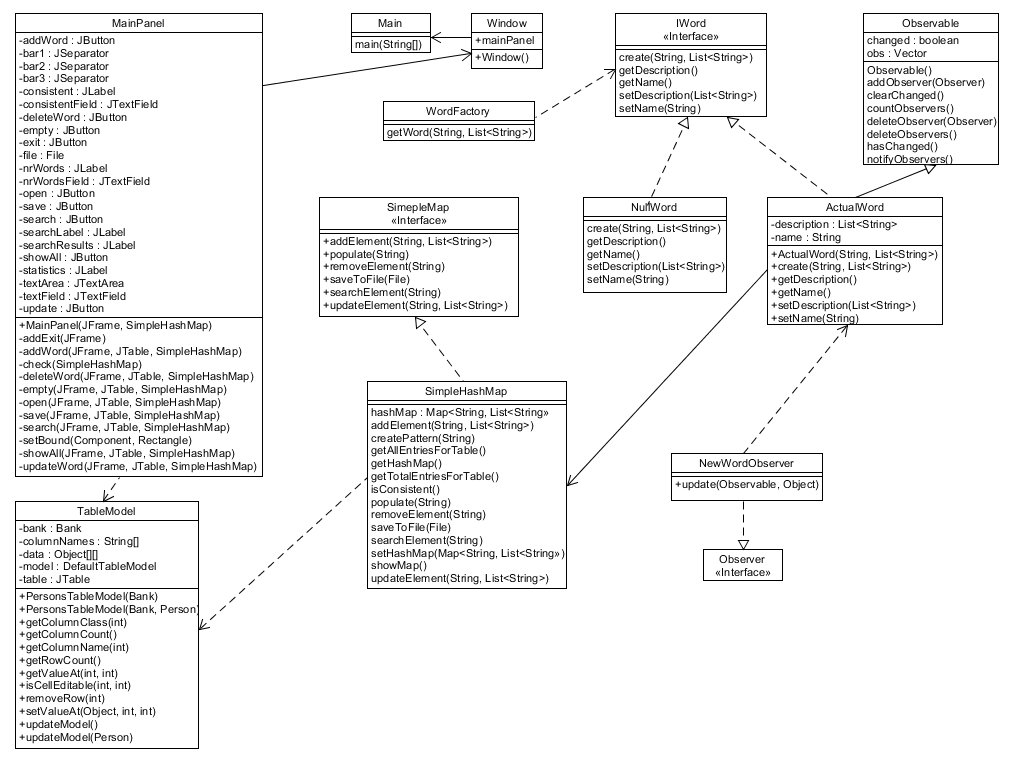
We must use an interface for data structure chosen, it must be separated from implementation.

Comments using standard java javadoc we can generate a good and useful for those who want to better understand the functioning of the application.

To implement the Dictionary we needed a structure that retain terms and definitions have chosen a defined class HashMap SimpleHashMap. This implements the interface SimpleMap. Each entry in HashMap consists of two fields one key term in our case and one that we value di definition for that term.  
We need an easy to use graphical interface that allows the user to perform the desired operation on the words but also to easily find everything he needs. Window class do this.  
  
  
Scenarios  
  
The user can use the HashMap's property Serializable and is available right from the entrance application a number of terms.  
When we want to improve our variety of words we can add new ones, either manually one by one are reading from external files composed in such a manner consistent with the methods to be implemented in application read.  
The user can choose between .txt or .xml files depends on the situation and the ease with which these files can build.  
Another scenario is saving the data dictionary in a .txt file to be read by another application to run or to make a backup simply. All the property Serializable we save all data when we leave the application.  
  
Use cases  
  
This dictionary can be used easily by any user, he just need to open the app and look for words that interests him.  
More advanced users may build external files with the terms of certain areas and have access to only those terms or even maintain all terms whether or not they need them.

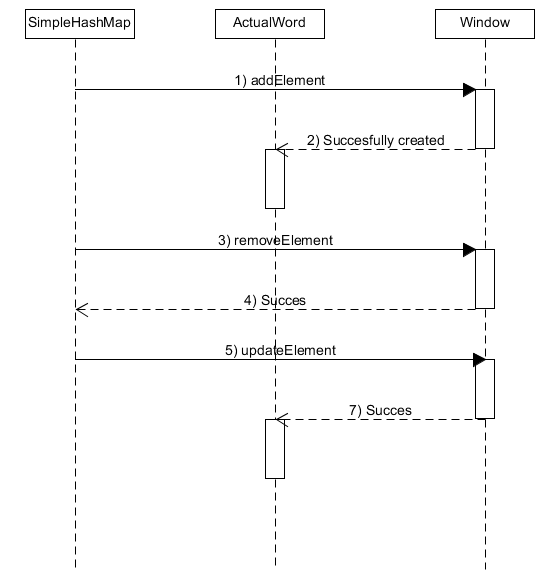
3.Projection

UML Class Diagram



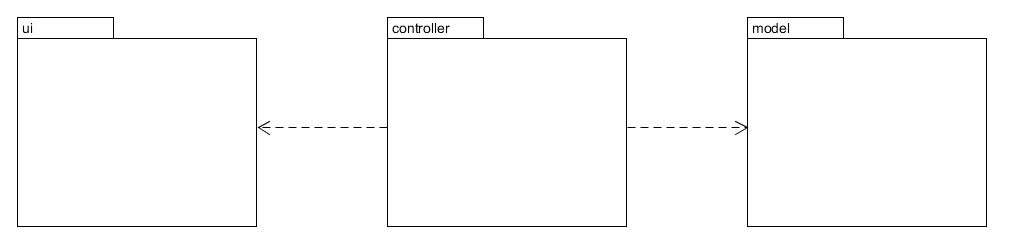
Use case Diagram

Sequence diagram



A sequence diagram of the objects SimpleHashMap,ActualWord and Window.

Package diagram



Packages

The packages are controller, ui and model.

The classes chosen for this project are specific and each one of them handles an important part of the program.

4.Implementation and testing

This dictionary is very practical and can be integrated in any field in order to serve as an electronic dictionary.

Due to the possibility of adding new terms from external files inside the application but provides the user quickly and easily terms of handling.

Design patterns

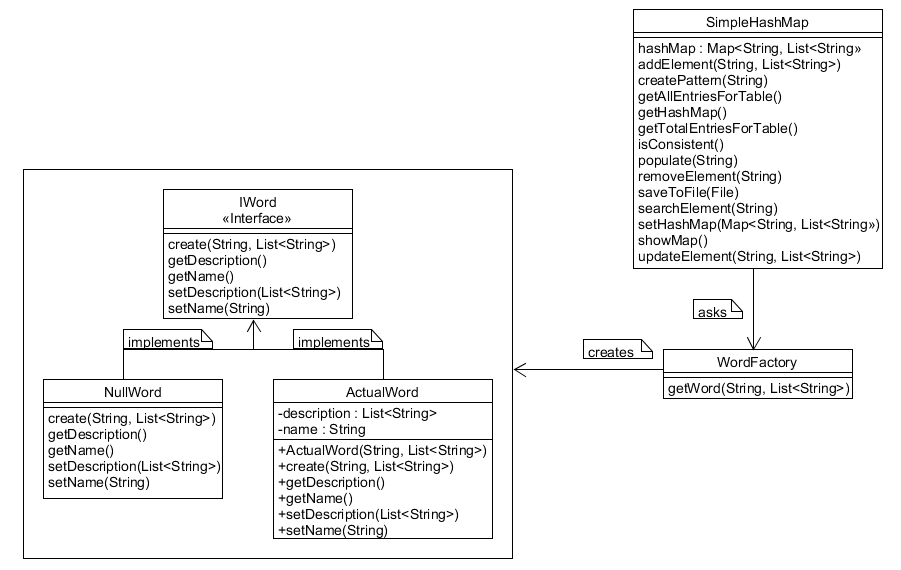
Factory pattern

Factory pattern is one of most used design pattern in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

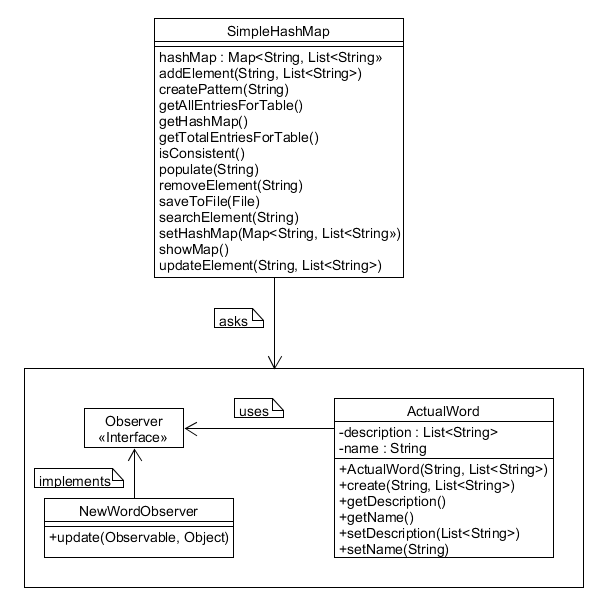
Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).



Observer pattern

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.



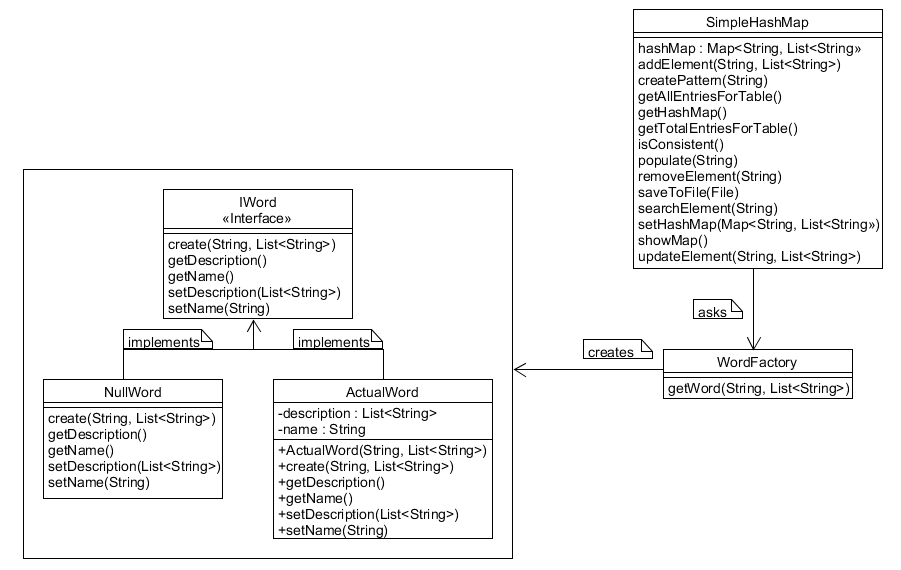
Observer -- any object that wishes to be notified when the state of another object changes

Observable -- any object whose state may be of interest, and in whom another object may register an interest

Null Object pattern

In Null Object pattern, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available.

In Null Object pattern, we create an abstract class specifying various operations to be done, concrete classes extending this class and a null object class providing do nothing implemention of this class and will be used seemlessly where we need to check null value.



5. Results

The results can be seen from the application. As for this we obtained an easy application that creates a dictionary application to open any json dictionary type.

6. Conclusions

Personally I worked the hardest for this project because it is the last one and the using of the hash map to store keys as strings and a List as values was interesting to implement. Also the fact that I can open many dictionaries , it does not stick just to one.

The project is very stable and it is very easy to use.

7. Bibliography

https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html