Intro:

Language used is java.

1- data structures:
   a. beyond the class structure, we use vectors to represent the input numbers and operations over them
   b. the dictionary used to load, store, retrieve words

Class Summary
Dictionary
Game
LettersAndNumbers
LettersMatch
Match
NumbersMatch
Player
TimeOutThread
UI

Field Summary	
java.util.HashSet	set
Constructor Summary	
Dictionary()	
Method Summary	
java.lang.String	add(java.lang.String str)
boolean	contains(java.lang.String str)
void	loadFromFile(java.lang.String str)

We have many choices regarding the dictionary, one is to build one using linked lists for example, and a good choice is to sort the elements using the length of the words to ease retrieval.

But here we use already built structure and encapsulate it with a class Dictionary to suit out needs.
The heart is a HashSet, for fast storage and retrieval.

We load the dictionary from a file: any text file is suitable, all words in the file will become available to the dictionary:

Read a file
While(has more lines read one){
    Tokenize the line and add to the dictionary
}

2- distribution of letters over players:
we got the game done in this way:
    a game has two players and a current match, which could be a Letters match or a Numbers match, each match has it's own rules.

So we created two classes: LettersMatch and NumbersMatch descendants of Match class.

The LettersMatch has this:
    Buffer of vowel letters and another buffer for non buffere letters, plus a maximum of distributions and current one.

Class LettersMatch

```
java.lang.Object
  └ lettersandnumbers.Match
      └ lettersandnumbers.LettersMatch
```

Field Summary	
(package private) static int maxCount	
(package private) static int	nonVowelLength
(package private) static java.lang.StringBuffer	nonVowelSet
(package private) static int	vowelLength
(package private) static java.lang.StringBuffer	vowelSet
Constructor Summary	

LettersMatch()	
Method Summary	
boolean	canGetMore() checks if we can get any more letters from this match, which returns false when both players took their shares
char	getNonVowel() if picks haven't exceeded the maximum level we return a random non vowel letter from the suitable buffer
char	getVowel() as the non vowel but returns a vowel

Distribution of letters between players is controlled from the UI, by allowing each one to pick a letter in a roll.

3- InDict: mentioned earlier
4- GetBestWord:

```
/**
 * out of 9 letters form the longest word that exists in the dictionary
 * so this is how we do it:
 * instead of forming all possibilities out of 9 letters which could reach
 * infinity we iterate
 * through all words in dictionary and see if the word could be formed out
 *of the 9 letters
 * then we return the longest match
 * [alternative approach using a sorted list against lengths of strings in the
 * dictionary or through another interface to it]
 * @param mix StringBuffer 9 letters buffer
 * @return String
 */
public String getBestWord(StringBuffer mix) {
   int maxLen = 0;
   String longest = null;
   Iterator itr = dict.set.iterator();
   while (itr.hasNext()) {
      String temp = (String) itr.next();
      if (canMutate(mix, new StringBuffer(temp))) {
         if (temp.length() > maxLen) {
            maxLen = temp.length();
            longest = temp;
         }
      }
   }
   return longest;
}
```

This algorithm has a complexity of M mutations where M is the rank of the dictionary set

```
/**
 * if we can constitute target string from a mix of letters then we can do the
 * mutation and return true
 */
public boolean canMutate(StringBuffer mix, StringBuffer target) {
    for (int i = 0; i < target.length(); i++) {
        if (mix.indexOf("" + target.charAt(i)) < 0)
            return false;
    }
    return true;
}
```

The mutate method has complexity of N where N is the length of the word. So this makes the getBestResult method of O(N*M) which could be considered O(M) where N is a constant average of the whole words which can't exceed a number like 6 in English.

5- GetBestResult:
We've got a list of integers and a result, we need to find the operations between these numbers to get the result.

This algorithm could be very annoying if we considered this:
Association and operation precedence:

a+b+c+d+e+f

a+b+c*d*f
a+b+c*d*f
a+b+(c*d)*f
a+(b+c)*d*f
(a+b+c)*d*f

So possibilities could be too far to reach.

So what we've done is the simplest way:
No precedence, no association, no mutation of position
And the problem becomes like this:
Find all mutations of operations between n numbers of constant-relative position.

a +b+ c+ d+ e+ f
a +b+ c+ d+ e- f
a+ b+ c+ d+ e* f

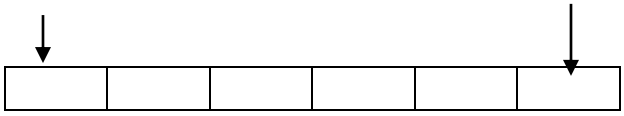
a+ b+ c+ d+ e/f

.

.

The algorithm becomes like this

```
char[] getBestResult(int[] list,int res){
      char op[]=new char[list.length-1];
         for each operation of the 4 do
                  if(mix(list,1,basicOp[i],list[0],op,res)){
                        return op;
                  }
   }
```



```
boolean mix(int[] list, int index,char op,int res,char[] ops,int finalRes){
        if(final state reached){
                for each OP in OPSET do
                        temp=list[index-1] OP res
                        if(temp==finalRes)
                                record operation
                                return true
        }
        Else{
                For each OP in OPSET do
                        Temp=list[index-1] OP res
                        If(mix(list,index+1,OP,temp,ops,finalRes))
                                record OP
                                return true
        }
}
```

So from the first index tell the last, the current operation mix is passed down in the res variable and the operation recording is done in the way back.

The complexity of the algorithm is simply (the mutation of 4 different operations over an n-1 locations)

C(getBestResult)=m! * (n-1)!

Where m is the rank of operations OPSET and n is the rand of the numbers vector

This method tries to find the exact operation mix over the passed numbers, if such one exists

Next we view some of the classes and their associated attributes.

Class Game

Field Summary	
(package private) Dictionary	dict
(package private) Match	match
(package private) int	matchCount
(package private) static int	maxMatches
(package private) Player	player1
(package private) Player	player2
(package private) int	turn

Constructor Summary	
Game(java.lang.String name1, java.lang.String name2, java.lang.String path)	

Method Summary	
boolean	canMutate(java.lang.StringBuffer mix, java.lang.StringBuffer target) if we can constitute target string from a mix of letters then we can do the mutation and return true
char[]	getBestResult(int[] list, int res) passing a list of integers, find the sequence of operations + - * / to get the result passed try all possibilties tell one is found or none no association or precedence is considered
java.lang.String	getBestWord(java.lang.StringBuffer mix) out of 9 letters form the longest word that exists in the dictionary so this is how we do it: instead of forming all possibilities out of 9 letters which could reach infinity we iterate through all words in dictionary and see if the word could be formed out of the 9 letters then we return the longest match [alternative approach using a sorted list against lengths of strings in the dictionary or through another interface to it]
boolean	mix(int[] list, int index, char op, int res, char[] ops, int finalRes)
boolean	playNextMatch() the match sequence is L L N L L N L L N...

Class NumbersMatch

Field Summary	
char[]	mix
int[]	numbers
int	res

Constructor Summary
NumbersMatch()

Method Summary	
static int	calcRes(int[] list, char[] op) passing list of integers and operations we do the math and return the result
void	generateNumbers() randomly generate 6 numbers
void	solve(int[] nums, int res) getBestReult
void	suggestMixOp() suggests a random operation distribution over mix variable

Class Player:

Constructor Summary
Player(java.lang.String name)

Method Summary	
void	add2score(int val)
int	getScore()

class **TimeOutThread:**

Field Summary	
(package private) boolean	stop
(package private) long	timeout
(package private) UI	ui

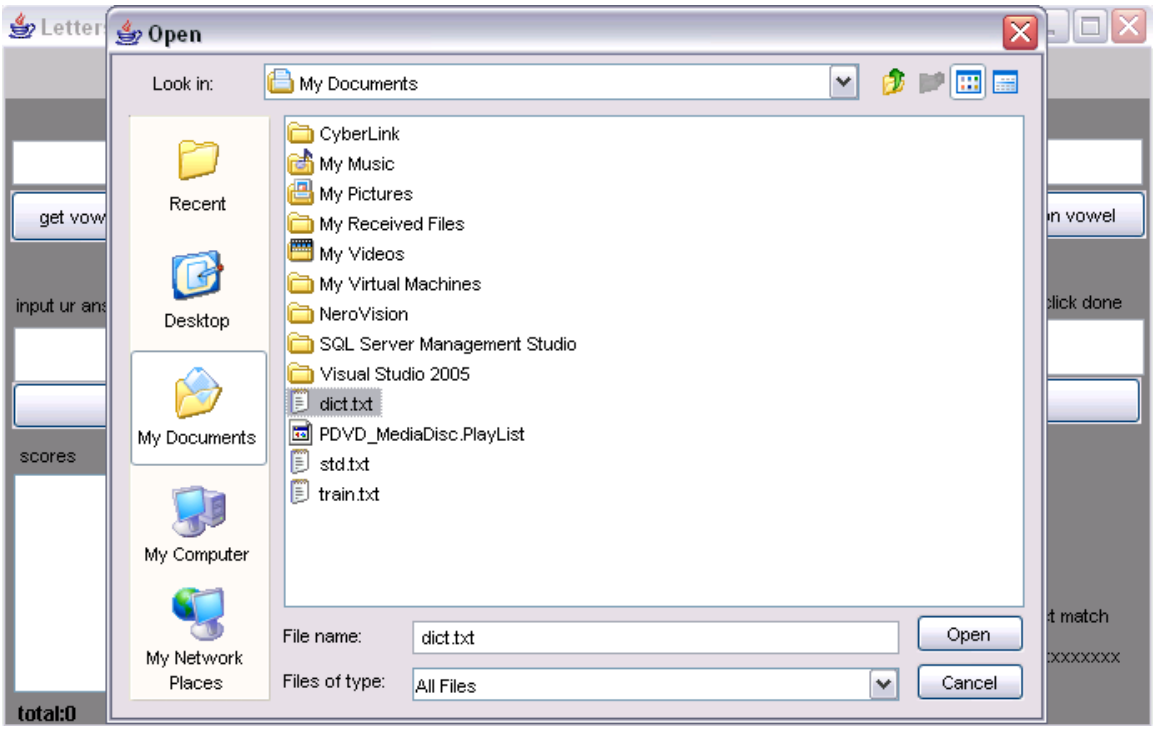
Constructor Summary
TimeOutThread(UI ui)

Method Summary

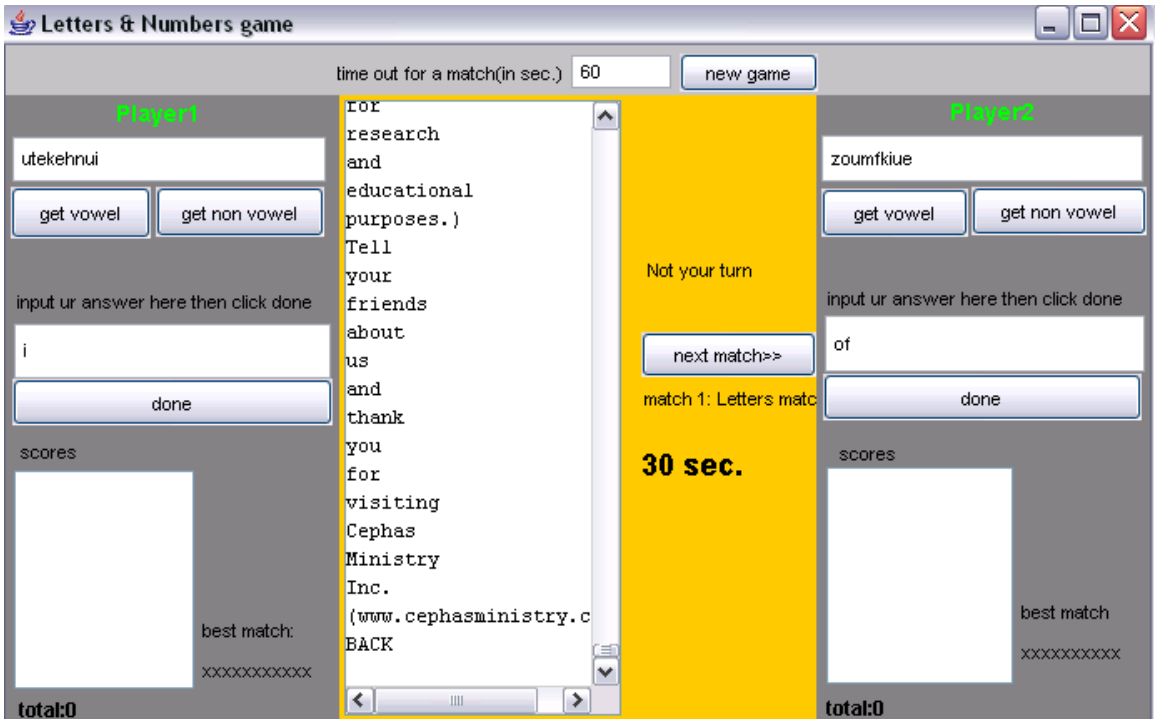
void	run()

If current match is time out we do the math and stop the match

UI :



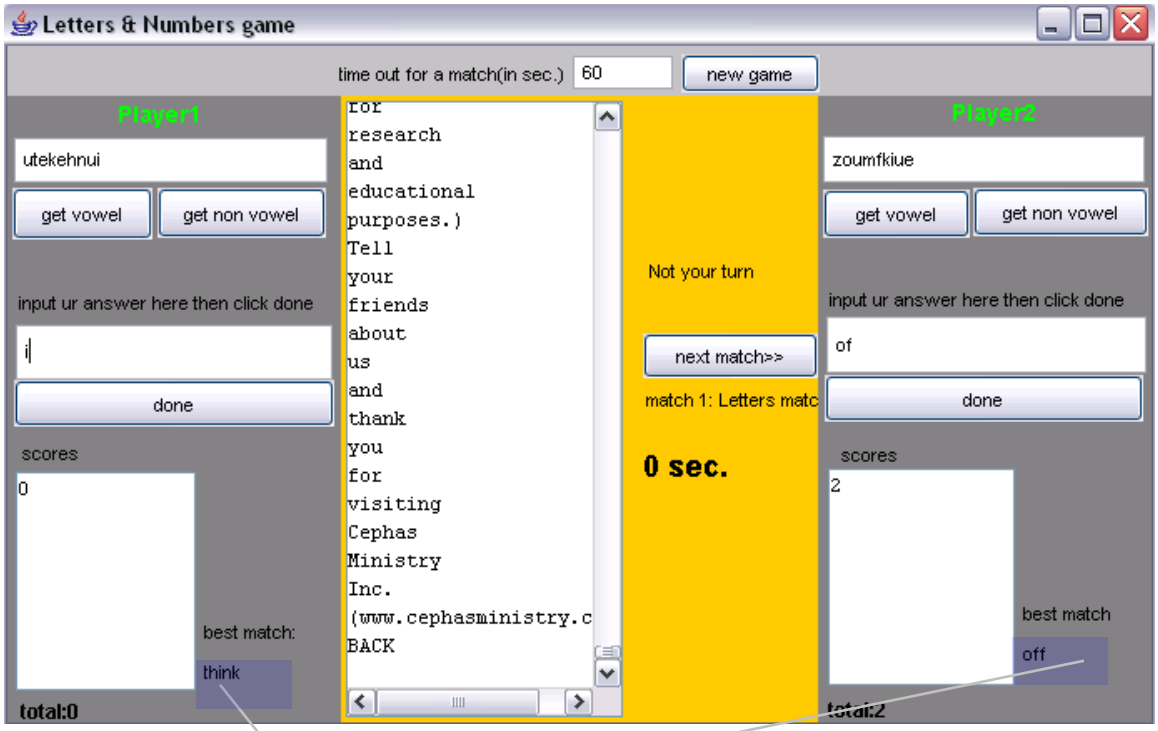
You choose new game and need to select a dictionary file (any text file with words works just fine)
You define the time out value in the field bellow then we begin by a letter match.
players can pick their letters, each from his own panel, one at a time, when they both end their share (9 letters here) the timer fires immediately and they have to put the answer in the white text field bellow, the dictionary is vied inside the text area in the middle.



above 30 sec. have passed.

If each kept the answer as it is we get the match result like bellow: player1 gets 0 for no match to his choice and player2 gets 2 for one match of two letters

The left players could choose THINK as best available mix and right player could choose OFF as the best mix

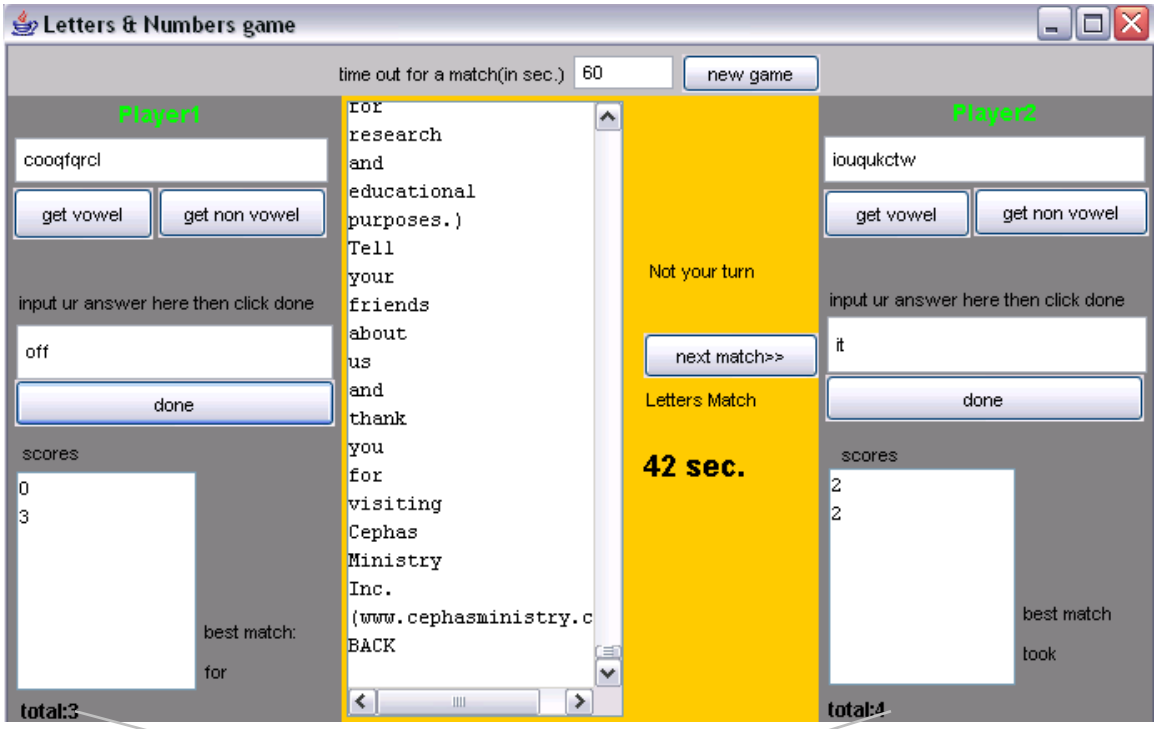


Best matches

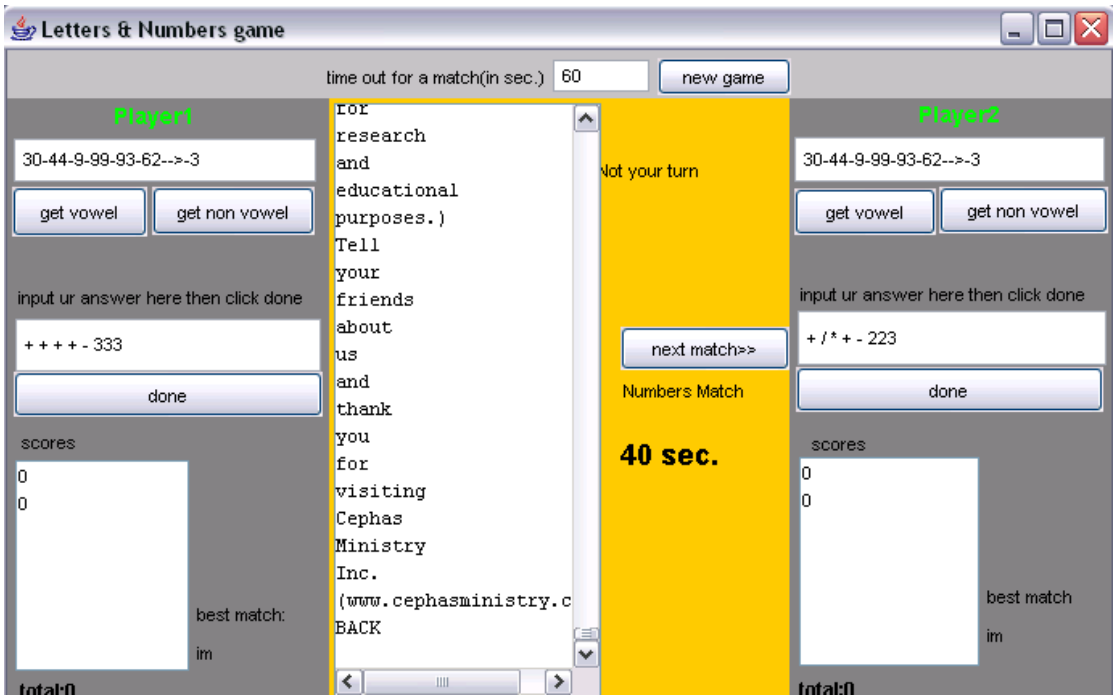
A player can also click done to announce that he's done, so if the other player clicked done too, the match ends and result calculation begins immediately without waiting for the time out.

Press next match after done of the current match (clicking any button out of order wont affect anything, they are all tied up with each other so no button miss the game, for example: pressing NEXT MATCH wont do a thing unless current match is over)

The second match is a letters match too, both ended by pressing done and first player gets 3 points and player2 get 2 and total result becomes 3-4



The same in the numbers match case but:



Later when done you get the mix of operations