Project 5 Report

Atharva Pendse & Zhaosong Zhu | CS 314H

Assignment Goals

In the Boggle Game project, the objectives are to implement a *Boggle* game, consisting of *GameDictionary, GameManager, and Boggle(GUI),* design a user interface with which multiple users can play the game with a randomly created board, and implement an algorithm that searches all available words based on both the board and dictionary given. Through our research algorithm and GUI design, we are to develop a relatively comprehensive understanding of how to implement a data structure and an iterator that helps find any word on the board, and how to design a simple UI using Swing. Our personal goals include writing out an efficient data structure that searches word quickly on the Boggle board, learning about basics of UI design, and finishing bonus Karma activities to explore the relationship between the configuration of a Boggle board and scoring.

Solution Design

The design of the Boggle project involves three aspects: *GameDictionary*, which implements a *BoggleDictionary* interface, *GameManager*, which implements *BoggleGame* interface, and *Boggle*, which is a GUI that operates based on information from GameDictionary and GameManager object. The *GameDictionary* class aims to load dictionary using a data structure called *Trie* and thus allow us to find if a word or prefix exists in the dictionary with an Iterator. The *BoggleGame* receives different inputs from the user and keeps track on all the conditions when playing the game, including a score of each user and whether a word is used. The *Boggle* is the GUI we design to allow users to play the game with a decent user experience visually. To see if all these classes are actually working, we also design a test harness for each of these classes.

A. High level details on implemented classes and methods

1. ***Graphical UI in Boggle class***

   To provide users a visual gaming experience, we implement a graphical user interface in *Boggle* class, as shown in *figure 1*. The main functionalities include submitting a typed word, switching players, and receiving prompts about scoring. Thus, we add an *ActionListener* to each button on the GUI. The *ActionListener* is triggered whenever a button is pressed, and we perform a set of operations.

   For the *submit answer* butter, the ActionListener will do the following things: obtain user's input with formatting, use *addWord* method in *GameManager* class to check if this word exists in dictionary and is not used beforehand, give a prompt about points, highlight the correct word's position on the board, update current user scoring, and finally set the input text box to empty for next input.

For the *next player* button, the ActionListener will first give a prompt about the user's final score and set the board words to black. Then, it will check whether the current player # +1 is still valid, if yes, then act. Otherwise, the two buttons on the GUI will be disabled, meaning the game is over.

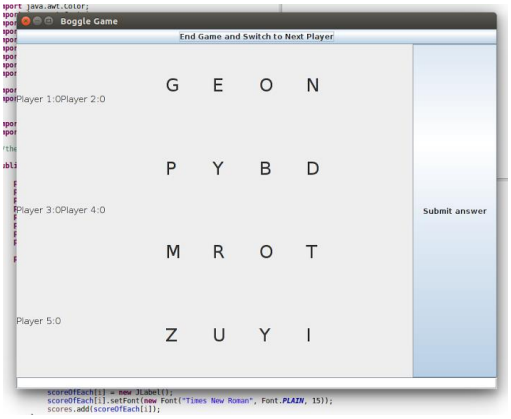


Figure 1

2. ***Board Search and Dictionary Search***

As specified in the assignment instruction, we implemented two types of search methods - board search and dictionary search in the GameManager class. Each method is composed of two parts, including the search method itself and a recursive function.

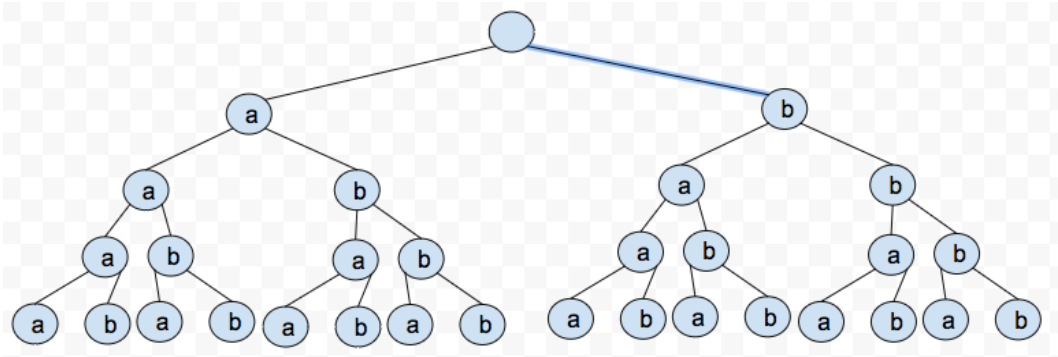
In the board search, a for loop will iterate through each element on the board. At the same time, a recursive function will be called, where a String will be constructed. In the beginning, a *String cur* will be passed in as the current character that the method is looking at. Then, we will perform two if checks - if the *String cur* is a legitimate 4 or more letters word in the dictionary(which is impossible in the first three turns), and if *cur* is a legitimate prefix. If yes, then we know a word might be found by this thread. Then, we will call on *boardSearchRecursive(visited, row, col, cur)* recursively, until a word and its corresponding points(from the indexes we iterate through) is formed and added to the list of all words on the current board.

In the dictionary, the similar mindset is applied. In the beginning, an iterator is used to store all the words into an *ArrayList*. Then, as we iterate through the items in the *ArrayList*, the recursive method is called to check each word character by character, thus determining whether the word is on the board. Our base case is that when a word is only left with one character after recursion, and that character is matched up with the character at the indexes on the board, then we add this word to our list. In doing so, we can reduce a number of search efforts.

3. ***Trie Data Structure***

A 'trie,' also called a radix tree, is a useful data structure for storing dictionaries. A trie takes at most O(k) time for accessing any node, where k is the maximum length of a word in the given dictionary. While a Trie is O(k), k is constant for the given dictionary and is generally not a very large number.

Each node except the root holds a character. As we traverse down a particular path of nodes, all nodes on that path are added to a string. The resulting string we get is part of the dictionary. If we travel all the way down to a leaf, then the word formed by the resulting string is part of the dictionary. If not, then the resulting string is the prefix of a word in the dictionary. For example, the Trie given below generates all strings of length up to four which contain only the characters a and b.



4. ***Iterator within GameDictionary class***

   The constructor for our iterator iterates through all words in the Trie and stores them in an ArrayList. The iterator then just iterates through all the words stored in the ArrayList. The benefit of storing the words in an ArrayList is that although we are sacrificing O(n) memory, each call of the iterator.next() now takes constant time.

B. Abstractions

1. ***Recursion***

   In both our Board Search and Dictionary search methods, we have used recursion to get all words efficiently. Our Board search method calls the *BoardSeearchRecursive()* method on each character on the board. The *BoardSearchRecursive()* method then calls itself repeatedly until all words on the board starting from that character have been found. The Dictionary search method uses the Trie iterator to iterate through all words in the dictionary. For each word, it checks which characters on the board match with the first character of that word, and calls *DictionarySearchRecursive()* on those characters.

2. ***Application of LinkedHashMap for isPrefix() and contains() methods***

   In the *isPrefix()* and *contains()* methods, *LinkedHashMap* is used to help determine the truth value. Using the idea of checking character by character, we use a for loop to go through each character of the word. During this process, we will obtain the children of the current node, starting from the root node. Then, we will also see if the current character as a key is in the LinkedHashMap - if no, then this word is not a prefix or in the dictionary; if yes, then we will change the current node to the node relating to the current key, *currentNode = hashMap.Get(key);* then going on with the for a loop.

3. ***Elements used in GUI***

   To design our graphical UI, we used an object-oriented mindset to classify the entire UI as a frame with the container, on which we can add *JButton, JTextField, JLabel, and JOptionPane* with a GridLayout. This mindset is especially helpful when we are trying to implement the Boggle Board on the graphical UI. By thinking about the board as an array

of JLabels instead of chars, we can just copy the original array to the JLabels array with each word at correct position!

Project Discussions

A. *GameDictionary and GameManager* Designs, Assumptions, Strength, and Weakness

As we write up codes for *GameDictionary* and *GameManager* class, one important consideration, as mentioned in the instructions, is to perform the search in a dictionary or a given board in *O(log n)* time.

- *GameManager*

1. *Designs for setGame() method*

   As described in the assignment, the *setGame()* method is designed to allow the debugger to provide a square board and perform testing with a new game. To satisfy this goal, we will "reset" the following variables for a given *GameManager* variable. A test to see if the new board is square is performed beforehand.

Char [][] currentBoard	currentBoard = newBoard;
int boardSize	boardSize = newBoard.length;
ArrayList<HashSet<String>> userWordSet	userWordSet = new ArrayList<HashSet<String>>(playerNum);
int [] score	score = new int[playerNum];
String password	last word = null;

   Since except the Board and its properties, each user only needs a set to store all words he has used, individual score and the last successfully added word. We believe we have reset necessary variables.

2. ***BoardSearch, DictionarySearch and efficiency***

   In the board search, a for loop will iterate through each element on the board. At the same time, a recursive function will be called, where a String will be constructed. In the beginning, a *String cur* will be passed in as the current character that the method is looking at. Then, we will perform two if checks - if the *String cur* is a legitimate 4 or more letters word in the dictionary(which is definitely impossible in the first three turns), and if *cur* is a legitimate prefix. If yes, then we know a word might be found by this thread. Then, we will call on *boardSearchRecursive(visited, row, col, cur)* recursively,

until a word and its corresponding points(from the indexes we iterate through) is formed and added to the list of all words on the current board.

The efficiency of board search varies depending on the particular configuration of the board. For a given board with I valid words in it, we found the efficiency to be O(s), where s is the total number of prefixes of all the words found (prefixes obviously include the words themselves.)

Average case for Board: We found that on average, a 4 x 4 board produces 35 valid words, with an average length of 4. This means that the average runtime is O(35 x 4).

In the dictionary search, the similar mindset is applied. In the beginning, an iterator is used to store all the words into an *ArrayList*. Then, as we iterate through the items in the *ArrayList*, the recursive method is called to check each word character by character, thus determining whether the word is on the board. Our base case is that when a word is only left with one character after recursion, and that character is matched up with the character at the indexes on the board, then we add this word to our list. In doing so, we can reduce a number of search efforts.

The overall efficiency of Dictionary search is proportional to n, the number of words in the dictionary (since we iterate through all of them once). This is because according to the assignment description, we will anyway have to iterate through the entire dictionary. It is also proportional to the number of characters from the start each word has on the given board.

So, which search strategy is better? Based on a 4x4 board and the dictionary we are given, BoardSearch is apparently a better choice. However, to answer this question, we have to consider more factors:

1. The Number of valid words in the board
   As we have pointed out before, the time efficiency of Board search depends on the number of valid words on that particular board. Therefore, if the board has a high number of possible words, it will take longer to run. If we have a large board size (say, 100) and the board is generated randomly (not a special case like all Xs), then that board will obviously contain a lot more words than a simple 4 X 4. The time taken to perform a Board Search on such a board would be quite high.

   Another case will be board with considerable size. For example, a board with size of 1 million. In this case, board search is slow, assuming there's a good mix of letters on the board, which doesn't allow us to limit a lot of options by using *isPrefix().*

2. The length of dictionary

Based on our implementation, dictionary search is slow with a large number of words. The best case for a dictionary with *n* words will be iterate through the *n* words. For each one of them, the best case will be the first letter is always not inside the HashSet of the node. Thus, even in the impossible best case, we will still need to perform a search for each item anyway. With a large input *n*, the dictionary search is going to be slow, since there's nearly no way to prevent this iteration.

3. The average length of each word in dictionary
   Assume a dictionary, in which each word has a length >50. Then, we will provide a board large enough to find all of those words with length >50. In this case, whichever search performed will be slow since merely the process of trunking/adding each character from the dictionary word/ board character will need to be executed 50 times.

   Thus, with an extremely large board and word length, the search strategies won't matter as much since they are all slow. However, with really small board size and word length, both methods should perform as well regarding speed.

3. When to update *lastAddedWord*
   getLastAddedWord() method is expected to return a List of Points, representing the location of each letter on the Boggle game. Since the points array is set up during word search in our implementation, in the normal case we will *return allwords.get(lastword)*. So it becomes interesting when we should update *lastword* to ensure a correct return result. By referring to the description of *getLastAddedWord()* method, "**Returns** A list of Points (concerning the board array) showing the previous successfully added word. If there is no previous word, return null.", It implies that the *lastword* must be the last successfully added word. Thus, in the *addWord* method, the *lastword* variable will only be updated if the user didn't score 0.

- *GameDictionary*

1. The advantage of Trie data structure
   A 'trie,' also called a radix tree, is a useful data structure for storing dictionaries. A trie takes at most O(k) time for accessing any node, where k is the maximum length of a word in the given dictionary. While a Trie is O(k), k is constant for the given dictionary and is generally not a very large number.

   Although a HashSet uses the constant time for accessing words, there are two problems with it:
   a. It has a high constant factor for accessing elements, and
   b. There is no way to use the *isPrefix()* method by just storing all the words in a HashSet.

   Inserting a word into a trie:

To insert a word from the given dictionary in the Trie, we iterate down the Trie character by character of the word, until we hit a node where we cannot progress further. When this happens, we add new nodes for each character left in the word after that and set the boolean value word of the leaf node to true.

The *GameDictionary.contains(word)* method now simply iterates down the Trie character by character of *word.* If the node where it stops has boolean value isword = false, then the method returns false. If you are unable to go further down but still have some characters of *word* left, then again the method will return false. If the node where it stops has isword = true, then it returns true.

The *GameDictionary.isPrefix(word)* method works the same as contains(), except it doesn't need to check the boolean value isword.

2. Search efficiency for *isPrefix()* and *contains()* method
   Similar to the discussions about board search and dictionary search, we believe that they are in line with the efficiency requirements. Using the idea of checking character by character, we use a for loop to go through each character of the word. During this process, we can actually determine very quickly whether a word is in the dictionary.

   As we go through a String character by character, we will obtain the children of the current node at the end of each iteration, starting from the root node. Since the child of each node is a LinkedHashMap, whose keys are the possible letters that come after a certain letter. Thus, we can check if this HashMap contains this key(the character) in constant time - if no, then this word is not a prefix or in the dictionary; if yes, then we will change the current node to the node relating to the current key, *currentNode = hashMap.get(key);* then going on with the for a loop. In doing so, we only need to iterate through the word itself and perform a constant-time check for each character.

   The number of times we need to move down the Trie is equal to the number of characters in the word in the worst case (which is when the argument to isPrefix() is actually a valid word in the dictionary.) Therefore, the worst-case running time of isPrefix() is o(k) where k is the length of the argument.

- *Weakness and possible improvements*
  We find out that we have implemented both board search and dictionary search with reliance on recursion. For example, in the dictionary search, for each word from the dictionary, we will call on the *dictionarySearchRecursive(visited, currentword, j, k)* , which is a recursive method as we obtain a substring of the current word each time until getting to the last letter of the word to see if it's a letter on the given location of the board. As we know, recursion is hard to trace and find errors, so we need to have faith in its correctness. Besides, in the dictionary we are given, there's almost no word with a length longer than 20. Suppose in an extreme case where the dictionary contains all

words with length of 1000, the efficiency of the search will be greatly compromised, and memory may even run out!

B. Edge cases and Potential Solutions

1. Given an erroneous player number input
   From the perspective of white box testing, there are in total two places that a user can give a wrong player number input - at the initialization of GUI and *newGame* method of GameManager. We choose to use different approaches to deal with this:

   For the GUI input, we used a while loop in the *Boggle* class such that only when the user's input is correct will we change the condition to false. In doing so, the GUI will not allow just exit due to an exception, which is more user-friendly.

   For the initialization of GameManager, we will throw an Exception if the number of players is either not an integer or smaller than 1. The reason is twofold. For the one thing, we can't have a graphical interaction with the initialization in here, so it is better just to throw an exception. Besides, it might also not be a good way to automatically set the # of players to one, since this is not the intention, as said on Piazza.
2. Null/ empty/captalized string as a parameter
   This situation is most likely to happen for these methods: addWord(), isPrefix(), setSearchTactic().

   In the case where the input is null, we will perform a check at the very beginning of each method to return *false* for null. This is reasonable because a null equals to nothing. However, in the setSearchTactic(), we will accept null as the default search tactic, to allow a search to be performed properly.

   If the input is empty, the isPrefix() method may return true, since it checks whether "" is a value for the HashSet. However, "" is similar to null so we should still return false all the time.
   If the input is capitalized or has spaces in between, we will change all the inputs to lower case and trim all the spaces exists in the input. However, this relies on the assumption that no word in the dictionary has space in between, which is true for the dictionary given.

3. Load a new dictionary on a given dictionary object
   We used a flag to signal whether the *loadDictionary()* method is executed. If the method is already executed, which means there's already a dictionary set up with Trie structure, we will throw an error message and just exit the *loadDictionary()* method.

C. GUI Design considerations

Since I possess very limited knowledge about Java GUI design, I started off the design by referring to the GUI offered in the *Tetris* project. As we have done more research on what elements we can use in the GUI, we started our design by using a frame with the container, on which we can add *JButton, JTextField, JLabel, and JOptionPane* with a GridLayout.

For the entire GUI's panel, we used a Border Layout, which supports a layout like *figure 2.*

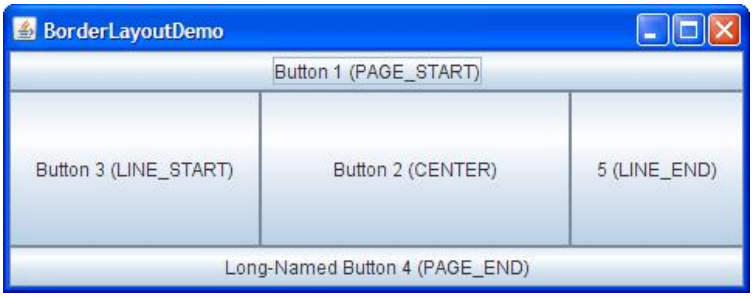


Figure 2 https://docs.oracle.com/javase/tutorial/uiswing/layout/border.html

However, this layout itself is not sufficient for us to implement a Boggle board and give each person's points. Thus, we used two GridLayout at LINE_START and CENTER to accommodate all the JLabels we used for scoring and Boggle board.

Since each of the JLabels are labels only, we wrote two methods *updateScore(JLabel[] scoreOfEach, int[] score, JPanel scores)* and *boldWord(JLabel[][] gameboard, List<Point> words)*. They will iterate through the array of Board and player scores, and set the text for each JLabel properly.

Besides the JLabels, the two buttons are the most direct way for users to interact with the UI. They are put with an ActionListener to monitor the click. For the *submit answer* butter, the ActionListener will do the following things: obtain user's input with formatting, use *addWord* method in *GameManager* class to check if this word exists in dictionary and is not used beforehand, give a prompt about points, highlight the correct word's position on the board, update current user scoring, and finally set the input text box to empty for next input.

For the *next player* button, the ActionListener will first give a prompt about the user's final score and set the board words to black. Then, it will check whether the current player # +1 is still valid, if yes, then act. Otherwise, the two buttons on the GUI will be disabled, meaning the game is over.

D. Karma & Personal Reflections

1. Karma discussion

   The karma asks us to think about the board configurations that would produce the highest scores. If we were to restrict the arrangements to the ones allowed by the given cubes, it would be a simple brute-force algorithm that found the total score for each possible board configuration. (There would be 16! X 6^16 total possible combinations, not excluding repetitions) . The stricter case here, though, is if we are not restricted to using the cubes from the given cube file. If we are allowed to use any character in any of the 16 available positions, we will get 26^16 total board configurations. Obviously, brute force would not be applicable in this case.

In this scenario, we can use a method similar to the one used in assignment 2 (Random writer). This method may not give the absolute best solution, but it will certainly give a configuration that is better than most random configurations. First, we analyze all the words in the dictionary and check the frequency with which each character appears after every other character in all words of the dictionary combined. This is the same as what we did in the random writer, with the value of k set to 1. Once we have all this data, we can simply choose our first character as the one that occurs most frequently in our dictionary, place it randomly on the board, find the character that occurs most frequently after it, place it horizontally, vertically or diagonally next to the first character, and so on.

2. What we learned from this project and future improvements
   For the one thing, this project allows us to apply the concepts of Trie(aka Radix Tree). By implementing the Trie class in the *GameDictionary* class, we can speed up the word searching speed with the hierarchy of the tree. For example, when finding the word "zoo," we will start from the tree node to see if 'z' is a key. If so, then we obtain the child of the current node, which is another HashSet and we repeat the same process. If any letter cannot be found, then that means the word or prefix is not in the dictionary. In doing so, we only need to iterate through the word itself and perform a constant-time check for each character. This thinking process makes us feel enjoyable too.

   Besides, this is the third project that we work with someone else. Not only we have a better working schedule, but we have also set up Git through Github to sync our work. More details are under pair programming experience.

   The peer testing process has also helped us greatly. For example, when thinking about testing cases for others, we were able to fix words with a combination of lower and upper case. Many thanks to the feedback from other groups, we are also able to fix some other corner cases, including null or "" as parameter and load dictionary on an already loaded dictionary object. We wish to do more peer testing in the future.

3. Pair programming experience
   A. Date and working time logs

Date	# of minutes	Items Accomplished
10/21/2017	Together: Zhaosong - 30 Atharva -30	Read through assignment descriptions together; Implement the basics of *GameManager* class
10/23/2017	Together: Zhaosong - 25 Atharva - 35	Brainstorm for the Trie structure; Research on GUI design
10/26/2017	Together:	Implement Tris structure for

	Zhaosong - 90 Atharva - 85	dictionary search; Implement part of Board search
10/27/2017	Together: Zhaosong - 55 Atharva - 45 Individual: Zhaosong - 40	Continue to work on *GameManager* class; Finish up GUI;
10/28/2017	Together: Zhaosong - 60 Atharva - 70	GUI testing; Getting ready for project submission
10/31/2017	Together: Zhaosong - 180 Atharva - 180 Individual: Zhaosong - 70 Atharva - 40	Write up parts of the report; Submit peer testing feedbacks; Junit testing running and debugging;
11/2/2017	Together: Zhaosong - 180 Atharva - 180 Individual: Zhaosong - 70 Atharva - 40	Improve solution based on peer testing results; Finish up report;

B. Experiences

Having done pair programming for three projects, we are getting more comfortable to cooperate with one another.

First, we find that it is easier for both of us to come up solutions or insights. For example, during the peer testing, each one of us was only able to come up around 5-7 items for code testing. Through discussion, however, we were able to think of some more intriguing corner cases for testing. A couple of examples will be whether we are allowed to change search tactics in the UI, whether a dictionary will read words with - in between.

Besides, pair programming gives each of us more incentives to work on the project. If working alone, we are likely to postpone the programming homework because of personal schedule. However, when working together, since we will periodically send message to one another to ask if the one wants to work collaboratively. With the experience of project 4, we have had a good idea about each one's schedule, based on which we try to work fluidly both in pair and individually.

Regarding improvements, we were able to set up git on each person's computer and create a repository on GitHub to take account of versioning. Git greatly helped us to synchronize with each one's record. We no longer need to copy the entire project folder to Google Drive and share it. We will explore more functionalities of Git in the future.

Testing

A. Testing Methodology

The JUnit testing for this project includes two sections: *Dictionary*, *Game*. Detailed explanations are shown below:

Dictionary: This test intends to test whether the dictionary can load or throw an error appropriately under various parameters passed in.

As specified below in the table, we have performed tests in eight different aspects. To start off, we adopted the tests from the original sanity test, which uses a small dictionary so that it's easy to keep track on inputs and outputs.

Starting from line 95, the test's focus is on the full functionalities of the dictionary object, as explained below:

- Load another dictionary on the current *GameDictionary* object;
    - EXPECTED: able to load it and any operation should return correct results
- Test if the dictionary contains "" or *null*; test if "" or *null* is a prefix;
    - EXPECTED: all of them should be false
- Check inputs with a combination of upper/lower case letter and spaces can be read
    - EXPECTED: if the word with correct format is in dictionary, then *contains()* and *isPrefix()* should all be true
- Check if the iterator can iterate through all words in the given dictionary
    - EXPECTED: Yes
- Loading an empty dictionary
    - EXPECTED: *contains* and *isPrefix* return *FALSE*
- Loading no dictionary
    - EXPECTED: *contains* and *isPrefix* return *FALSE*
- Loading a non-existing file
    - EXPECTED: an exception with the message that "Error when loading dictionary!" is expected

In referral to the interface *BoggleDictionary*, which contains *loadDictionary(), isPrefix, and contains(),* we believe that our test is relatively comprehensive - covers both the provided methods and corner cases that may happen at each parameter.

Testing item	Aspect covered

Sanity test given with a small dictionary	*loadDictionary(), isPrefix, and contains()*
Edge cases for dictionary parameters	*isPrefix, contains()*
Contains with inputs uppercase/lowercase	
isPrefix with inputs uppercase/lowercase	
Compare the result of all words from iterator and manual iteration of the dictionary file	Iterator
Loading empty/ doesn't exist dictionary	*loadDictionary()*
Not loading dictionary and call *contains, isPrefix*	
Loading a dictionary for two times on the same object	

Implementation:

We used JUnit 4 for running all our unit tests. For the most part, we used the assertEquals and assertArrayEquals methods to check if the actual values matched with the expected values. Following is a list of the things we tested and their line numbers:

Testing item	Code line in *testFullDictionary()*
Sanity test given with a reloaded small dictionary	125-157
Edge cases for dictionary parameters	100-104
Contains with inputs uppercase/lowercase	107-109
isPrefix with inputs uppercase/lowercase	113-121
Compare the result of all words from iterator and manual iteration of the dictionary file	160-184
Loading empty/ doesn't exist dictionary	187-198
Not loading dictionary and call *contains, isPrefix*	201-203

Negative tests:

We used a couple of asserts(false, a==b) statements to make sure that the test harness detected that we were passing in the wrong inputs and that it wasn't just passing all the cases.

Testing item	Code line in *testDictionary() and testFullDictionary()*
Loading a dictionary for two times on the same object	125-157
Loading empty/ doesn't exist dictionary	187-198
Not loading dictionary and call *contains, isPrefix*	201-203

Game: This section tests the methods in the *GameManager* class. We tried to come up with cases that could potentially cause errors, as described below.

As specified below in the table, we have performed tests in eight different aspects. To start off, we adopted the tests from the original sanity test, which uses a small dictionary so that it's easy to keep track on inputs and outputs.

Starting from line 210, the test's focus is explained below:

- Call *getLastAddedWord()* and *getScores()*
    - EXPECTED: *getLastAddedWord()* should return null while *getScores()* should be an array with 0s.
- Check if the board is what it looks like
    - EXPECTED: should be the same for 4x4 or non-4x4 board inputs
- Check if different search strategies on the same board will return the same words
    - EXPECTED: the same amount of words in correspondence should be returned under *SEARCH_DICT, SEARCH_BOARD, null* search methods.
- Try to play a valid word and an invalid word
    - EXPECTED: an exception with the message that "Invalid player number" is expected
- Load a blank cube file
    - EXPECTED: an exception with the message that "Invalid Board size for the given Cube file." is expected
- Load a cube file of not a perfect square
    - EXPECTED: none of the char on the board should be null or ‘’
- Loading an invalid player number
    - EXPECTED: an exception with the message that "Error when loading dictionary!" is expected
- Use GameManager before game is loaded
    - EXPECTED: an exception with the message that "Initialize game first" is expected

In referral to the interface *BoggleGame*, which contains *newGame(), addWord(), setGame(), setSearchTactic,* excluding all the getter methods. We believe that our test is relatively comprehensive - covers both the provided methods and corner cases that may happen at each parameter.

Testing item	Aspect covered

Call *getLastAddedWord()* and *getScores()*	*getLastAddedWord()* and *getScores()*
Check if the board is what it looks like	*getBoard(), setGame()*
Check if different search strategies on the same board will return the same words	getAllWords(), setSearchTactic()
Try to play a valid word and an invalid word	addWord()
Loading an invalid player number	
Load a blank cube file	*newGame()*
Load a cube file of not a perfect square	
Use GameManager before game is loaded	

Implementation:

We used JUnit 4 for running all our unit tests. For the most part, we used the assertEquals and assertArrayEquals methods to check if the actual values matched with the expected values. Following is a list of the things we tested and their line numbers:

Testing item	Code line in *testGame()*
Game initialization and variables	211-231
setGame with non-4x4 board size	234-238
Compare results of different search tactics	241-263
addWord() with valid/invalid word/# of player	266-276
Point array testing	282-285
Blank/ non-square cube files	288-301
Invalid player number	304-310

Negative tests:

We used a couple of asserts(false, a==b) statements to make sure that the test harness detected that we were passing in the wrong inputs and that it wasn't just passing all the cases.

Testing item	Code line in *testGame()*

addWord() with valid/invalid word/# of player	266-276
Blank/ non-square cube files	288-301
Invalid player number	304-310
Use GameManager before game is loaded	312-316

Black Box Testing:

While we were coding the assignment, we played Boggle over and over to make sure that the particular feature we had just completed worked. Black-box testing was also very helpful in reminding us what was still left to complete. In this particular game, playing the game also allows us to see if there's any problem in UI design. This is a list of things we tested while playing with the Boggle UI:

Check invalid number of players
Check if all words we can use match with our program
Input with Uppercase and lowercase/ Combination of letters and words
With blank in between/ end/ beginning
See if you can change the search tactic
See if the subsequent players could use the words used by player 0 to gain points

In addition to actually playing the game, we used System.Out.Print statements in several places for debugging the code.

White box testing:

As explained above, we created two JUnit test classes BoardTest and PieceTest specifically for testing our code.

Testing Exceptions:

The exception may happen in both the *GameDictionary* and *GameManager* class. In our testing harness, we have tested the cases below:

1. Load multiple dictionaries
2. Load a nonexistent dictionary
3. Use a blank cube file
4. Invalid player number

In each of these cases, we will catch the exception. Since the exception is with a specific message of the error, we will see if the error message with the exception is matched up with our expectations, thus testing exceptions.

Strengths and Weaknesses in our Testing Methodology

Strengths:
Testing the code increased our confidence in its correctness. Since we used JUnit, we were able to automate the testing process. Instead of having the program print diagnostic statements and examining those statements to make inferences about the correctness of our program, we were able to have the JUnit test method check for accuracy automatically. Having an automatic checking mechanism shortened the time required for testing individual cases considerably.
Weaknesses:
We came up with a significant amount of test cases and tried to test as many cases as we could think of, but couldn't code all of them due to time constraints. However, the peer testing process helps open our mind in creating a more comprehensive testing harness.