Homework Turnin

Name:	Ameya Singh
Account:	ameyas (ameyas@uw.edu)
Student ID:	1868457
Section:	AQ
Course:	CSE 143 18au
Assignment:	a4
Receipt ID:	a58664616d3e257f2f3438dde238cfe4

Warning: Your turnin is 1 day late. Assignment a4 was due Thursday, October 25, 2018, 11:30 PM.

Turnin script completed with output:

Turnin Successful!

The following file(s) were received:

HangmanManager.java (8419 bytes)

```java
/*
 * Author: Ameya Singh
 * CSE 143 AQ
 * TA: Soham P.
 * Homework 4: HangmanManager
 */

import java.util.*;

/**
 * HangmanManager manages a game of hangman where the computer chooses the
 * solution word at the last possible instance. Handles internal logic of the
 * game and exposes methods that allow guesses to be easily recorded.
 *
 * @author Ameya Singh
 */
public class HangmanManager {
    /**
     * Holds the current patterns based on the current guesses.
     */
    private Map<String, Set<String>> patternMap;
    /**
     * Holds the dictionary of currently possible words.
     */
    private Set<String> dictionary;
    /**
     * Holds a records of all of the user guesses.
     */
    private Set<Character> guesses;
    /**
     * Holds the current pattern in play.
     */
    private String pattern;
    /**
     * Holds the length of words in the game.
     */
    private int length;
    /**
```

```java
     * Holds the number of incorrect guesses the user can still make.
     */
    private int chancesLeft;

    /**
     * Constructs a new HangmanManager using the passed dictionary, word length,
     * and max number of incorrect guesses. Game will use all words of the
     * length passed in less any duplicates as options for the word to be
     * guessed. Word length must be greater than or equal to 1 and max number
     * of wrong guesses must be greater than or equal to 0.
     *
     * @param dictionary Any Collection of Strings from which to pick words to
     *                   be used in the game.
     * @param length     Length of words which will be picked from the dictionary
     *                   and used in playing the game.
     * @param max        Represents the maximum number of incorrect guesses the player
     *                   can make.
     * @throws IllegalArgumentException Thrown if passed length is less than 1.
     * @throws IllegalArgumentException Thrown if passed max is less than 0.
     */
    public HangmanManager(Collection<String> dictionary, int length, int max) {
        if (length < 0) {
            throw new IllegalArgumentException();
        }
        if (max < 0) {
            throw new IllegalArgumentException();
        }

        patternMap = new HashMap<String, Set<String>>();
        this.dictionary = new TreeSet<String>();
        guesses = new HashSet<Character>();
        pattern = "";
        this.length = length;
        chancesLeft = max;

        initDictionary(dictionary);
        initPattern();

        patternMap.put(pattern, this.dictionary);
    }

    /**
     * Private helper that initializes the dictionary of words the program will
     * use based on the desired length of words.
     *
     * @param dictionary Dictionary to filter words of only of passed length.
     */
    private void initDictionary(Collection<String> dictionary) {
        for (String word : dictionary) {
            if (word.length() == length) {
                this.dictionary.add(word);
            }
        }
    }

    /**
     * Initializes the pattern to dashes in all spaces.
     */
    private void initPattern() {
        for (int i = 0; i < length; i++) {
            pattern += "-";
        }
    }

    /**
     * Returns the current set of words being considered by the HangmanManager.
     *
     * @return Set of words currently being considered.
     */
    public Set<String> words() {
        return Collections.unmodifiableSet(patternMap.get(pattern));
    }

    /**
     * Returns the current number of incorrect guesses the player can sill make.
     *
     * @return Returns the number of incorrect guessed left.
     */
    public int guessesLeft() {
        return chancesLeft;
    }

    /**
     * Returns the current set of letters the player has guessed.
     *
     * @return Returns set of letter guesses the player has made.
     */
    public Set<Character> guesses() {
        return Collections.unmodifiableSet(guesses);
    }

    /**
     * Returns the current pattern for the hangman game accounting for the
     * guesses that have been made. Formatted such that guessed made are shown
     * and letters that have not been guessed are shown as dashes.
     *
     * @return Returns the pattern of the current game.
     * @throws IllegalStateException Thrown if the set of words corresponding to
     *                               the pattern is empty.
     */
    public String pattern() {
        if (patternMap.get(pattern).isEmpty()) {
            throw new IllegalStateException();
        }

        return this.pattern;
    }

    /**
     * Records the next guess made by the user. Returns the number of
     * occurrences of the guessed letter in the pattern and updates all other
     * fields as appropriate.
     *
     * @param guess Letter guessed by the user.
     * @return Returns the number of occurrences of the guess in the pattern.
     * @throws IllegalStateException    Thrown if the player has no guesses left
     *                                  (Guesses left are less than 1).
     * @throws IllegalStateException    Thrown if there are no words that
     *                                  correspond to the current pattern.
     * @throws IllegalArgumentException Thrown if the set of words that match
     *                                  the current pattern is not empty but
     *                                  the character has been guessed before.
     */
    public int record(char guess) {
        if (chancesLeft < 0) {
            throw new IllegalStateException();
        }
        if (patternMap.get(pattern).isEmpty()) {
            throw new IllegalStateException();
        }
        if (!guesses.add(guess)) {
            throw new IllegalArgumentException();
        }
        updatePatternMap(guess);
        setPattern();

        int count = countPattern(guess);
        if (count == 0) {
            chancesLeft -= 1;
        }
        return count;
    }

    /**
     * Private helper that updates the pattern map using the passed guess.
     *
     * @param guess Guess to update pattern map to represent.
     */
    private void updatePatternMap(char guess) {
        dictionary = patternMap.get(pattern);
        patternMap.clear();
        for (String word : dictionary) {
            String pattern = getPattern(word, guess);
            if (patternMap.containsKey(pattern)) {
                patternMap.get(pattern).add(word);
            } else {
                Set<String> wordSet = new HashSet<String>();
                wordSet.add(word);
                patternMap.put(pattern, wordSet);
            }
        }
    }

    /**
     * Private helper that gets the pattern for a word based on the passed guess
     * and the previous guesses.
     *
     * @param word  Word whose pattern is to be returned.
     * @param guess Current guessed letter.
     * @return Returns the pattern for the word.
     */
    private String getPattern(String word, char guess) {
        String out = "";
        for (int i = 0; i < length; i++) {
            if (word.charAt(i) == pattern.charAt(i)) {
                out += pattern.charAt(i);
            } else if (word.charAt(i) == guess) {
                out += guess;
            } else {
                out += "-";
            }
        }
        return out;
    }

    /**
     * Private helper that sets the the current pattern to the pattern that
     * contains all guesses and allows for the largest number of possible words
     * to be chosen by the game.
     */
    private void setPattern() {
        Object[] keyArr = patternMap.keySet().toArray();

        String maxPattern = (String) keyArr[0];
        for (Object key : keyArr) {
            String keyString = (String) key;
            if (patternMap.get(keyString).size() >
                    patternMap.get(maxPattern).size()) {
                maxPattern = keyString;
            }
        }
        this.pattern = maxPattern;
    }

    /**
     * Private helper that counts the occurrences of the passed letter in the
     * current pattern.
     *
     * @param guess Letter to count in the pattern.
     * @return Returns number of occurrences of letter in pattern.
     */
    private int countPattern(char guess) {
        int count = 0;
        for (char c : pattern.toCharArray()) {
            if (c == guess) {
                count++;
            }
        }
        return count;
    }
}

```