**Faculty of Computing**

**SE-314: Software Construction**

**Class: BESE 13AB**

# Lab 09: Abstract Data Type - II

**CLO-03:** Design and develop solutions based on Software Construction principles.  
**CLO-04:** Use modern tools such as Eclipse, NetBeans etc. for software construction.

**Date: 18th Nov 2024**

**Time: 10:00 AM** **- 12:50 PM   
 02:00 PM – 04:50 PM**

|  |  |
| --- | --- |
| **NAME** | MUHAMMAD SAAD UMER |
| **CLASS** | BESE 13A |
| **CMS** | 408485 |

**Instructor: Dr. Mehvish Rashid  
Lab Engineer: Mr. Aftab Farooq**

**Introduction:**

# Lab 09: ADT- II

Students will have hands-on experience on designing, testing, and implementing abstract data types. Given a set of specifications, you will write unit tests that check for compliance with the specifications, and then implement code that meets the specifications. Use GitHub to collaborate with your group members and mention the work distribution.

**Material:**

<https://ocw.mit.edu/ans7870/6/6.005/s16/psets/ps2/>

Lectures on LMS regarding designing Abstract Data Types   
**Lab Tasks** (TEST RESULTS)



**GitHub Link**

**https://github.com/MSaadUmer**

**Task1: Test Graph Poet**

|  |
| --- |
| **IMPLEMENTATION AND DOCUMENTATION OF DEVISED CASES** |
| IMPLEMENTATION  package poet;  import static org.junit.Assert.\*;  import org.junit.Test;  import java.io.File;  import java.io.IOException;  import java.util.List;  public class GraphPoetTest {    // Sample corpus files for testing  private static final File SEVEN\_WORDS\_FILE = new File("test/poet/seven-words.txt");  private static final File EMPTY\_FILE = new File("test/poet/empty.txt");  private static final File MALFORMED\_FILE = new File("test/poet/malformed.txt");    // Constructor Tests  @Test  public void testGraphPoetWithValidFile() {  try {  GraphPoet poet = new GraphPoet(SEVEN\_WORDS\_FILE);  assertNotNull(poet); // Ensure the GraphPoet is initialized  } catch (IOException e) {  fail("IOException should not occur with valid file.");  }  }  @Test(expected = IOException.class)  public void testGraphPoetWithInvalidFile() throws IOException {  new GraphPoet(new File("nonexistent-file.txt"));  }  // Test for empty file (edge case)  @Test  public void testGraphPoetWithEmptyFile() {  try {  GraphPoet poet = new GraphPoet(EMPTY\_FILE);  assertNotNull(poet); // Ensure GraphPoet is still created  // Ensure it doesn't generate poems (empty graph)  List<String> poem = poet.generatePoem(5); // Assuming 5 words  assertTrue("Poem should be empty for empty file", poem.isEmpty());  } catch (IOException e) {  fail("IOException should not occur with empty file.");  }  }  // Malformed file (e.g., file with invalid structure)  @Test  public void testGraphPoetWithMalformedFile() {  try {  GraphPoet poet = new GraphPoet(MALFORMED\_FILE);  assertNotNull(poet);  List<String> poem = poet.generatePoem(5);  assertNotNull("Poem should not be null", poem);  } catch (IOException e) {  fail("IOException should not occur with malformed file.");  }  }  // Poem Generation Tests  @Test  public void testGeneratePoemWithValidFile() {  try {  GraphPoet poet = new GraphPoet(SEVEN\_WORDS\_FILE);  List<String> poem = poet.generatePoem(3);  assertEquals("Poem should contain 3 words", 3, poem.size());  // Add more assertions here depending on expected behavior of the poem  assertTrue("Poem should not be empty", poem.size() > 0);  } catch (IOException e) {  fail("IOException should not occur with valid file.");  }  }  // Edge Case: Large File  @Test  public void testGeneratePoemWithLargeFile() {  try {  GraphPoet poet = new GraphPoet(new File("test/poet/large-file.txt"));  List<String> poem = poet.generatePoem(100);  assertTrue("Poem should be generated even for large files", poem.size() > 0);  } catch (IOException e) {  fail("IOException should not occur with large file.");  }  }  // Edge Case: File with repetitive content  @Test  public void testGeneratePoemWithRepetitiveFile() {  try {  GraphPoet poet = new GraphPoet(new File("test/poet/repetitive.txt"));  List<String> poem = poet.generatePoem(3);  // Assert that the generated poem has meaningful content, not just repeated words  assertNotEquals("Poem should not be just a repetition of the same word", poem.get(0), poem.get(1));  } catch (IOException e) {  fail("IOException should not occur with repetitive file.");  }  }  // Test for assertion error with invalid corpus (if applicable)  @Test(expected = AssertionError.class)  public void testAssertionsEnabled() {  assert false; // Ensure assertions are enabled  }  }  DOCUMENTATION  **Test Constructor Behavior:** We test the constructor GraphPoet(File) to ensure that it behaves correctly with valid files, invalid files, empty files, and malformed files. Each of these tests should ensure that no unexpected exceptions are thrown unless explicitly expected.  **Test Poem Generation:** The test testGeneratePoemWithValidFile() verifies that the poem is generated properly for a valid file, and testGeneratePoemWithEmptyFile() ensures that an empty file results in no poem.  **Edge Case Testing:**   * Testing with large files (testGeneratePoemWithLargeFile()). * Testing files with repetitive content (testGeneratePoemWithRepetitiveFile()). |

**Task2: Implement GraphPoet in GraphPoet.java .**

|  |
| --- |
| **CODE** |
| package poet;  import graph.Graph;  import java.io.File;  import java.io.IOException;  import java.nio.file.Files;  import java.util.\*;  public class GraphPoet {  private final Graph<String> graph = Graph.empty(); // Empty graph to store words and their adjacencies  /\*\*  \* Create a new poet with the graph from corpus (as described above).  \*  \* @param corpus text file from which to derive the poet's affinity graph  \* @throws IOException if the corpus file cannot be found or read  \*/  public GraphPoet(File corpus) throws IOException {  // Read the corpus from the file  List<String> lines = Files.readAllLines(corpus.toPath());  // Create a map to count adjacencies between words  Map<String, Map<String, Integer>> wordGraph = new HashMap<>();  // Process each line in the corpus  for (String line : lines) {  // Tokenize the line into words, removing punctuation and converting to lowercase  String[] words = line.split("[\\s]+"); // Split by any whitespace  for (int i = 0; i < words.length - 1; i++) {  String word1 = normalize(words[i]);  String word2 = normalize(words[i + 1]);  // Update the adjacency map (word1 -> word2)  wordGraph.putIfAbsent(word1, new HashMap<>());  wordGraph.putIfAbsent(word2, new HashMap<>());  wordGraph.get(word1).put(word2, wordGraph.get(word1).getOrDefault(word2, 0) + 1);  }  }  // Now, populate the graph with the adjacencies and their weights  for (Map.Entry<String, Map<String, Integer>> entry : wordGraph.entrySet()) {  String word = entry.getKey();  for (Map.Entry<String, Integer> adjacency : entry.getValue().entrySet()) {  String adjacentWord = adjacency.getKey();  int weight = adjacency.getValue();  // Add the edge to the graph  graph.add(word, adjacentWord, weight);  }  }  }  /\*\*  \* Normalize the word: convert to lowercase and remove any non-alphanumeric characters  \*/  private String normalize(String word) {  return word.toLowerCase().replaceAll("[^a-zA-Z0-9]", "");  }  /\*\*  \* Generate a poem.  \*  \* @param input string from which to create the poem  \* @return poem (as described above)  \*/  public String poem(String input) {  String[] words = input.split("\\s+");  List<String> poem = new ArrayList<>();  // For each consecutive pair of words, find the best bridge word  for (int i = 0; i < words.length - 1; i++) {  String word1 = normalize(words[i]);  String word2 = normalize(words[i + 1]);  // Add the first word in the pair to the poem (preserving the case)  poem.add(words[i]);  // Find the best bridge word (if any)  String bridgeWord = findBestBridgeWord(word1, word2);  if (bridgeWord != null) {  poem.add(bridgeWord); // Add the bridge word in lowercase  }  }  // Add the last word in the input  poem.add(words[words.length - 1]);  return String.join(" ", poem);  }  /\*\*  \* Find the best bridge word between two words.  \* A bridge word is the word that connects two words via a two-edge path with the maximum weight.  \*  \* @param word1 first word  \* @param word2 second word  \* @return the bridge word, or null if no bridge exists  \*/  private String findBestBridgeWord(String word1, String word2) {  int maxWeight = 0;  String bestBridgeWord = null;  // Iterate through all neighbors of word1 and find the maximum weight path  for (String neighbor : graph.neighbors(word1)) {  int weight = graph.getEdgeWeight(word1, neighbor) + graph.getEdgeWeight(neighbor, word2);  if (weight > maxWeight) {  maxWeight = weight;  bestBridgeWord = neighbor;  }  }  return bestBridgeWord;  }  // Additional methods for testing and debugging  /\*\*  \* String representation of the graph for debugging  \*/  @Override  public String toString() {  return graph.toString();  }  // Optional checkRep method to ensure the graph's representation is valid  private void checkRep() {  for (String word : graph.vertices()) {  for (String neighbor : graph.neighbors(word)) {  assert word != null && neighbor != null : "Null words detected in graph";  assert !word.isEmpty() && !neighbor.isEmpty() : "Empty words detected in graph";  }  }  }  } |

## You must use Graph in the rep of GraphPoet , but the implementation is otherwise entirely up to you

**Deliverables:**

Compile a single word document by filling in the solution part and submit this Word file on LMS. In case of any problems with submissions on LMS, submit your Lab assignments by emailing it to [aftab.farooq@seecs.edu.pk.](mailto:aftab.farooq@seecs.edu.pk.)