Assignment 1 Aayush Sapkota PRT 452

## Question no 1:

GitHub-repo link : https://github.com/aayushsapkota/HIT452-assignment-1-aayush-sapkota

## Question no 2:

**Class : Connectedgraph.java**

import java.util.InputMismatchException;

import java.util.Scanner;

public class Connectedgraph {

public static void main(String[] args) {

int numberOnNode, origin;

Scanner scan = null;

try{

System.out.println("Please, Enter the number of nodes in the graph");

scan = new Scanner(System.in);

numberOnNode = scan.nextInt();

int adj\_matrix[][] = new int[numberOnNode + 1][numberOnNode + 1];

System.out.println("Please, Enter the adjacency matrix");

for (int i = 1; i <= numberOnNode; i++) {

for (int j = 1; j <= numberOnNode; j++) {

adj\_matrix[i][j] = scan.nextInt();

}

}

for (int i = 1; i <= numberOnNode; i++) {

for (int j = 1; j <= numberOnNode; j++) {

if (adj\_matrix[i][j] == 1 && adj\_matrix[j][i] == 0) {

adj\_matrix[j][i] = 1;

}

}

}

System.out.println("Please, Enter the origin for the graph");

origin = scan.nextInt();

BFS checkGraphConnection = new BFS();

checkGraphConnection.breadthFristSearch(adj\_matrix, origin);

}catch(InputMismatchException IM)

{

System.out.println("Please Enter in 1 0 1 or 1 0 format");

}

scan.close();

} }

**Class: BFS.java**

import java.util.LinkedList;

import java.util.Queue;

public class BFS {

private Queue<Integer> queue;

private int i, uniqueObject;

public BFS() {

queue = new LinkedList<Integer>();

}

public String breadthFristSearch(int adj\_matrix[][], int origin) {

int numberOfNode = adj\_matrix[origin].length - 1;

int[] visited = new int[numberOfNode + 1];

visited[origin] = 1;

queue.add(origin);

while (!queue.isEmpty()) {

uniqueObject = queue.remove();

i = uniqueObject;

while (i <= numberOfNode) {

if (adj\_matrix[uniqueObject][i] == 1 && visited[i] == 0) {

queue.add(i);

visited[i] = 1;

}

i++;

}

}

boolean connected = false;

for (int vertex = 1; vertex <= numberOfNode; vertex++) {

if(visited[vertex] == 1) {

connected = true;

} else {

connected = false;

}

}

if (connected) {

System.out.println("Graph is connected");

return "Graph is connected";

}

else {

System.out.println("Graph is not connected");

return "Graph is not connected";

}

}

}

## The methods were created and then commented to make sure it fails. Then, uncommented and tested to if it passes.

Individual test method for each method is created to check it works. The methods ware checked against the actual result of method to expected result and both of them passed. Testing was done using Junit which is a framework that is part of Xunit collective framework. And some refactoring of code was done. This included keeping BFS in a separate class as it has its own unique function and methods were shorten for effieciency.

public void testBreadthFristSearch() {

System.out.println("breadthFristSearch");

int adj\_matrix[][] = {{0, 1, 1}, {1, 0, 0}, {1, 0, 0}};

int origin = 0;

BFS instance = new BFS();

String result = instance.breadthFristSearch(adj\_matrix, origin);

String expectedResult = "Graph is connected";

String anotherExpectedResult = "Graph is not connected";

if (expectedResult.equals(result)) {

System.out.println("Pass 1");

} else if (anotherExpectedResult.equals(result)) {

System.out.println("Pass 2");

} else {

fail("Didnot get expected result!");

}

}

public void testMain() {

System.out.println("main");

int origin = 0;

int adj\_matrix[][] = {{0, 1, 1}, {1, 0, 0}, {1, 0, 0}};

BFS checkGraphConnection = new BFS();

String result = checkGraphConnection.breadthFristSearch(adj\_matrix, origin);

String expectedResult = "Graph is connected";

String anotherExpectedResult = "Graph is not connected";

if (expectedResult.equals(result)) {

System.out.println("Pass");

} else if (anotherExpectedResult.equals(result)) {

System.out.println("Pass");

} else {

fail("Didnot get expected result!");

}

}

## Question no 3:

Answer:

### Objective:

* + To test whether the code Bad Smells are directly associated with faults.

### Targeted Code Bad Smells:

* + Data Clumps, Message Chains, Middle Man, Speculative Generality, and Switch Statements

### Use the following data sets:

* + Apache Common Packages (Common IO, Common Logging, Common Codec, Common DbUtils, Common DBCP, and Common Net)

### Choose some bad smells.

* + Manually investigate 5 source code samples from the Apache commons
  + Zimmerman et al.’s (2007) fault identification approach:
  + Locate “bug”, “fix(ed)” and “update(d)” token in CVS comment messages.
  + If a version entry in CVS contains one or more above tokens and those tokens are followed by numbers, this version entry is seen as a bug fixing update.

### Bugs:

#### DBCP – 80 Bugzilla ID – 24082

max = Math.max(max, Integer.parseInt((String)obj));

This code is an example of data clumps as it has identical variables and tries to make sense of these variables. This code results into an error in code because the code assumes that the key it gets is an int, this is unnecessary delegation. This problem shows that code smells may result in bugs.

#### IO-448

The code waits too long to detect an exception so, it doesnot detect an interruption. The code has a bad smell of message chain:

Thread.currentThread.interrupt();

The smell doesn’t affect the execution of the code so, not confirming any relation between code smells and faults.

#### NET-551

public static final long copyStream(InputStream source, OutputStream dest, int bufferSize, long streamSize, CopyStreamListener listener)

throws CopyStreamException

{

return copyStream(source, dest, bufferSize, streamSize, listener, true);

}

The code re-uses the chars field for both the number of characters and the single characters. This is a type of data clumps and make code hard to read.

The existence of this bad smell might have led to the bug showing incorrect value for bytesTransferred.

#### Logging-147

Switch(type) {

|  |
| --- |
| case SimpleLog.LOG\_LEVEL\_TRACE: buf.append("[TRACE] "); break; |
| case SimpleLog.LOG\_LEVEL\_DEBUG: buf.append("[DEBUG] "); break; |
| case SimpleLog.LOG\_LEVEL\_INFO: buf.append("[INFO] "); break; |
| case SimpleLog.LOG\_LEVEL\_WARN: buf.append("[WARN] "); break; |
| case case SimpleLog.LOG\_LEVEL\_ERROR: buf.append("[ERROR] "); break; |
| SimpleLog.LOG\_LEVEL\_FATAL: buf.append("[FATAL] "); break; |

}

Simplelog.log uses switch statements and has data clumps. This update is not safe and uses fields written twice, log is called by two threads. These might be the results of having multiple bad smells in the code.

#### IO-389

FileUtils.java : Between the call to File.listFiles and the call to FileUtils.sizeOf, a file may be deleted. This can cause FileUtils.sizeOf to throw an IllegalArgumentException, indicating that the file does not exist.

This bug is caused a result of data clumps as it has too many identical variables but tries to do too many functions at once. so, no focus has been given to edge cases like this.

### Conclusion:

From the analysis of these 5 different bugs, we can see that most of the time if a program has a bad smell then, it is prone to faults.

### References for bugs:

Apache Commons Issues. (2017). Available at: https://issues.apache.org/jira/browse/ZOOKEEPER-2883?jql=ORDER%20BY%20key%20DESC [Accessed 28 Aug. 2017].