Question No 1:

My GitHub account where I make a repository on the file name Assignment1-PRT452 is here:

https://github.com/loveinspk/Assignment1-PRT452

Question no 2:

A) Small java program to input a graph weather graph is connected or not. This was tested with Junit which is supported by xunit.

```
import java.util.*;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;
public class graphConnected
{
   private Queue<Integer> queue;
   public graphConnected()
   {
      queue = new LinkedList<Integer>();
   }
   public void bfs(int adjacency_matrix[][], int source)
   {
      int numberofnodes = adjacency_matrix[source].length - 1;
      int[] visited = new int[numberofnodes + 1];
      int i, element;
      visited[source] = 1;
      queue.add(source);
      while (!queue.isEmpty())
      {
```

```
    element = queue.remove();
    i = element;
    while (i <= numberofnodes)
    {
        if (adjacency_matrix[element][i] == 1 && visited[i] == 0)
        {
            queue.add(i);
            visited[i] = 1;
        }
        i++;
    }
}
boolean connected = false;

for (int vertex = 1; vertex <= numberofnodes; vertex++)
{
    if (visited[vertex] == 1)
    {
        connected = true;
    } else
    {
        connected = false;
        break;
    }
}
if (connected)
{
    System.out.println("The graph given is connected");
```

```
    } else
    {
        System.out.println("The graph given is disconnected");
    }
}

public static void main(String... arg)
{
    int number_no_nodes, source;
    Scanner scanner = null;

    try
    {
        System.out.println("Enter the number of nodes in the graph");
        scanner = new Scanner(System.in);
        number_no_nodes = scanner.nextInt();

        int adjacency_matrix[][] = new int[number_no_nodes + 1][number_no_nodes + 1];
        System.out.println("Enter the adjacency matrix");
        for (int i = 1; i <= number_no_nodes; i++)
            for (int j = 1; j <= number_no_nodes; j++)
                adjacency_matrix[i][j] = scanner.nextInt();

        for (int i = 1; i <= number_no_nodes; i++)
        {
            for (int j = 1; j <= number_no_nodes; j++)
            {
                if (adjacency_matrix[i][j] == 1 && adjacency_matrix[j][i] == 0)
```

```
            {
               adjacency_matrix[j][i] = 1;
            }
          }
        }

        System.out.println("Enter the source for the graph");
        source = scanner.nextInt();

        graphConnected undirectedConnectivity= new graphConnected();
        undirectedConnectivity.bfs(adjacency_matrix, source);

      } catch (InputMismatchException inputMismatch)
      {
        System.out.println("Wrong Input Format");
      }
      scanner.close();
   }
}
```

First output of this graph code

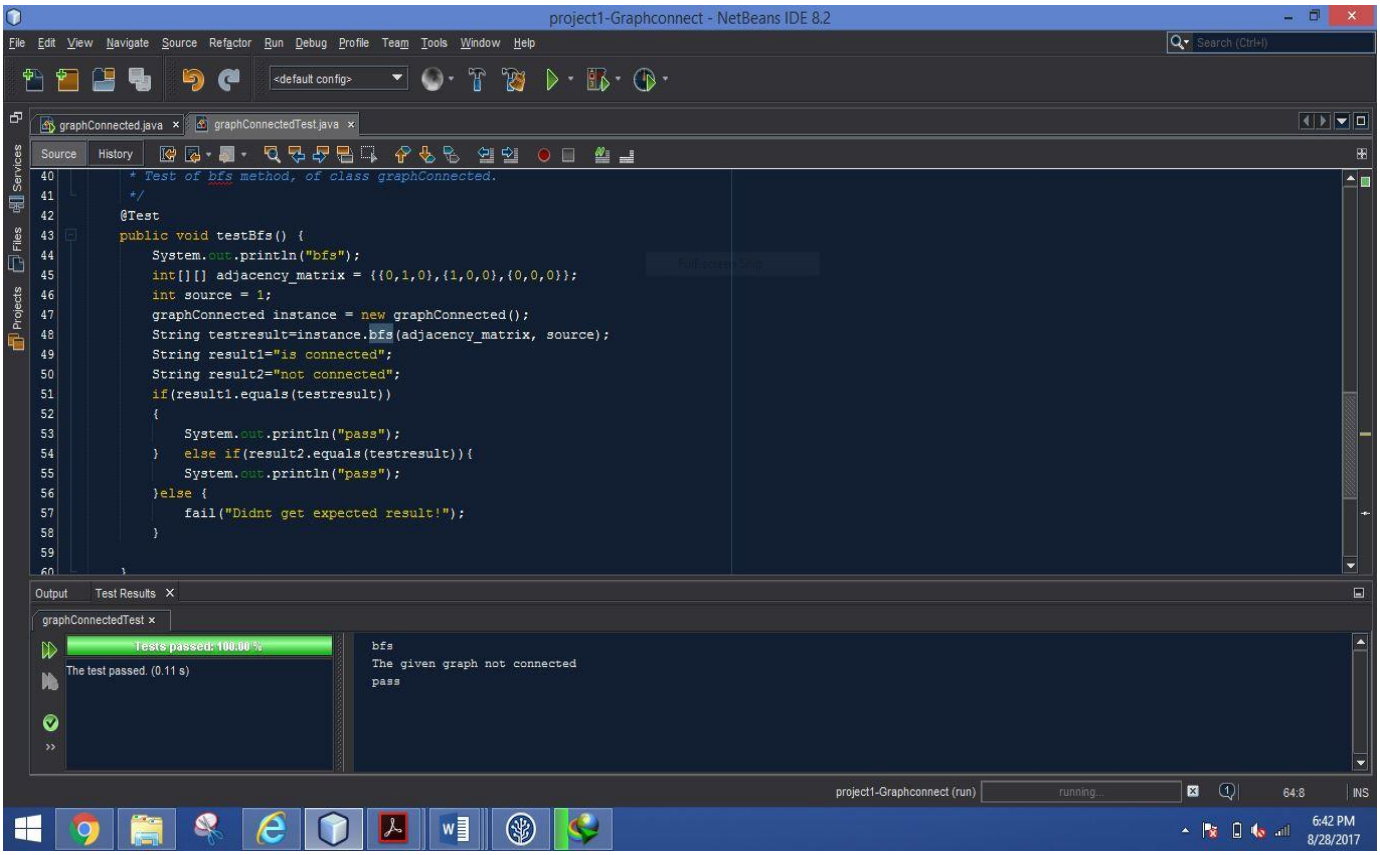
Enter the number of nodes in the graph
3
Enter the adjacency matrix
0 1 1
1 1 0
0 0 1
Enter the source for the graph
1
The given graph is connected
BUILD SUCCESSFUL (total time: 26 seconds)
b)

B) Test of connected graph

1. Test pass but graph is not connected

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class graphConnectedTest {

    public graphConnectedTest() {
    }

    public static void setUpClass() {
    }

    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    public void tearDown() {
    }
```

```
public void testBfs() {
    System.out.println("bfs");
    int[][] adjacency_matrix = {{0,1,0},{1,0,0},{0,0,0}};
    int source = 1;
    graphConnected instance = new graphConnected();
    String testresult=instance.bfs(adjacency_matrix, source);
    String result1="is connected";
    String result2="not connected";
    if(result1.equals(testresult))
    {
       System.out.println("pass");
    }   else if(result2.equals(testresult)){
       System.out.println("pass");
    }else {
       fail("Didnt get expected result!");
    }

}


    }
```

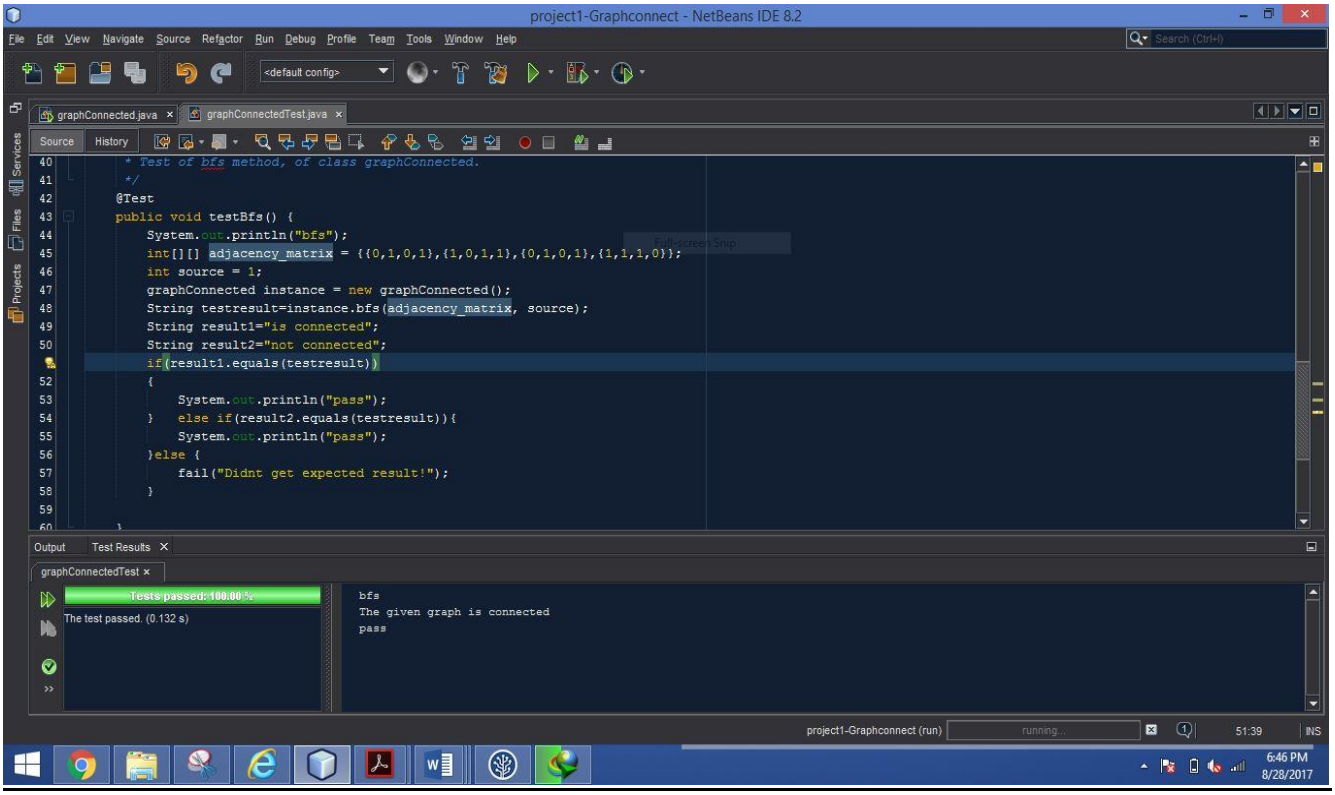


Test pass with connect graph

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
public class graphConnectedTest {

    public graphConnectedTest() {
    }
```

```
public static void setUpClass() {
}
      public static void tearDownClass() {
}
  public void setUp() {
}
   public void tearDown() {
}
    public void testBfs() {
    System.out.println("bfs");
    int[][] adjacency_matrix = {{0,1,0,1},{1,0,1,1},{0,1,0,1},{1,1,1,0}};
    int source = 1;
    graphConnected instance = new graphConnected();
    String testresult=instance.bfs(adjacency_matrix, source);
    String result1="is connected";
    String result2="not connected";
    if(result1.equals(testresult))
    {
       System.out.println("pass");
    }   else if(result2.equals(testresult)){
       System.out.println("pass");
    }else {
       fail("Didnt get expected result!");
    }

}
```

```
}
```



Question No 3.

Objective

Normally, code smell refers as a bad smell. In computer programming that indicate some symptom in the code of the program the probably indicates a deeper problem. According to the Martin Fowler, “this is a surface indication which correlated with the deeper problem of the system. Actually, code smell is not a bug and not any technical incorrect and don't right now keep the program from working of the code. Code smells are indicating the violation of the fundamentals design principle and effects on the quality of design. Code smell indicates the weakness in the quality of design which might be slowing down the programming development or increases the failure and risk of bugs in future. Fowler et al. (1999) suggests refactoring is the way to solve the such kind of trouble which is indicated by the code smell whereas faults refers to the errors into the programming code. With the faults in programs cannot give effective results. Code smell and faults are correlated to each other but code smell is not failure of the program which only indicates that the weakness of the quality in programming code.

Targeted code bad smells

Data clumps:

In Object Oriented Programming, Data Clump is the term that provides to the other groups of variables which can be passed around together throughout the program. It is like as other code smells which indicate the various problems in the program design and implementation. In data clumps is considered as specific kind of level of bad code smell that might be the indication of the weak programming code written. In general, those types of code smell should be refactored. The process of reducing data clumps generating a various type of code smell (In a data class that can only stores the data where doesn't stores any methods for operating on the data; however, new programmer will encourage to see the actual functionality which might be included in the program.

Message chains:

This type of code smell arises when the number of classes are highly coupled to the other multiple classes in like a chain form. To demonstrates this, for example we have class A who want to take data from class E. For retrieving data, firstly object A needs to retrieve object E from object D Object C object B. The chain seems like that

a.get B(). get C(). get D(). get E();

here, when A want to access the information from Class E, A unnecessarily access data from B C D along in the way, when it wants data only to get form E. Message chains code smell could be refactored.

Middle man:

Middle man also be a code smell. It is a class which delegates most of its function to other classes. This smell can be the result of the essential task of the class being gradually moved to other classes.

Speculative Generality:

This is also a one type to bad smell of programming code when programmer write a code, code is created "just in case" to support and predicated to prevent problem which will arise in future features that never get implemented. So, code becomes confusing and harder to understand for client. The major solutions for this problem is to remove unused abstract classes, try to collapse each other. Secondly, using inner class to eliminate the unnecessary delegation of functionality to other class.

Switch statements:

The switch statement also be a code smell. In this smell, programmer relatively used to switch and case major operators which is the major marks of object oriented program. In Switch statement code, the switch operators could be distributed in various place in the program. When the programmer wants to put new condition or requirements, have to find from all switch code and modify it. The switch statement code smell should be refactored. As a rule of thumb, we can solve this smell through polymorphism.

From the different data sets, I choose five bugs from Apache common package.

Five different Bugs

Bug 1:

From common IO: 2.5 ExceptionInInitializerError
Status: Resolved
Priority: Major
Created: 09/May/2017
Updated: 17/May/2017
Resolved: 17/May/2017

Description:

In its static block, org.apache.commons.io.Java7Support executes:

ClassLoader cl = Thread.currentThread().getContextClassLoader();

This can be null.

In that case, I believe the class should fall back to using:

Java7Support.class.getClassLoader();

But someone with a better understanding of the security implications should weigh in on that change.

Thus, here the Message box code smell arise. So, the company is resolved this problem.

Bug 2:

Form common logging: **Wrong sample code in org/apache/commons/logging/package.html (using static and this.class)**
Status: closed
Priority: Minor
Created: 10/April/2003
Updated: 29/December/2009
Resolverd: 29 December/2009

Description:

In the org/apache/commons/logging/package.html file, there is sample code:

public class Foo
{ static Log log = LogFactory.getLog(this.class); ... }

This code cannot be compiled because of using "static" and "this.class" in the same statement. Should be change to Foo.class.

Thus, here data clumps code smell arises.

Bug 3:

From Common Net: SubnetUtils.SubnetInfo.isInRange("0.0.0.0") returns true for CIDR/31, 32
Status: Resolved
Priority: Minor
Created: 09/July/2017
Updated: 04/August/2017
Resolved: 04/August/2017

Description:

Code:

```
import org.apache.commons.net.util.SubnetUtils;

public class A {
public static void main(String[] args)
{
System.out.println(new SubnetUtils("192.168.1.0/30").getInfo().isInRange("0.0.0.0"));
System.out.println(new SubnetUtils("192.168.1.0/31").getInfo().isInRange("0.0.0.0"));
System.out.println(new SubnetUtils("192.168.1.0/32").getInfo().isInRange("0.0.0.0")); }

}
```

Result:
false
true
true

Expected:
false
false
false

Thus, the Duplicate code smell arises here which can be resolved.

Bug 4:

From common logging 147: SimpleLog.log - unsafe update of shortLogName

Status: Closed
Priority: Major
Created: 18/july/2012
Updated: 20/March/2012
Resolved: 18/July/2012

Problem:

```
 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 SimpleLog.LOG_LEVEL_ERROR: buf.append("[ERROR] ");
break;
 case SimpleLog.LOG_LEVEL_FATAL: buf.append("[FATAL] ");
break;
 }
```

Here, the data clumps code smell arises.

Bug 5:

From Common Logging: RandomStringUtils' random method infinite loop 1 of 162

Status: Closed

Resolution: Invalid

Created: 11/Nov/2016

Updated: 11/Nov/2016

Resolved: 11/Nov/2016

Code error:

```
RandomStringUtils.random(1, 0, 0, false, true, new char[]

{ 'a' }

)
```

The above code has infinite loops and lots of parameters. So, this code is considered as Message Chain smell.

References

1) FOWLER, M., BECK, K., BRANT, J., OPDYKE, W. & ROBERTS, D. (1999) *Refactoring: Improving the Design of Existing Code*, Addison Wesley.

2) Fowler, M. (2006). *bliki: CodeSmell*. [online] martinfowler.com. Available at: https://martinfowler.com/bliki/CodeSmell.html.

3) Refactoring.guru. (2014). *Data Clumps*. [online] Available at: https://refactoring.guru/smells/data-clumps.

4) Refactoring.guru. (2014). *Message Chains*. [online] Available at: https://refactoring.guru/smells/message-chains

5) Refactoring.guru. (2014). *Remove Middle Man*. [online] Available at: https://refactoring.guru/remove-middle-man.

6) Sourcemaking.com. (2007). *Design Patterns and Refactoring*. [online] Available at: https://sourcemaking.com/refactoring/smells/switch-statements [Accessed 28 Aug. 2017].

7) Sourcemaking.com. (2007). *Design Patterns and Refactoring*. [online] Available at: https://sourcemaking.com/refactoring/smells/speculative-generality.

8)) SEAMAN, C. B., SHULL, F., REGARDIE, M., ELBERT, D., FELDMANN, R. L., GUO, Y. & GODFREY, S. (2008) Defect categorization: making use of a decade of widely varying historical data. *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement.* Kaiserslautern, Germany, ACM.

9) Team, A. (2017). *Apache Commons – Apache Commons*. [online] Commons.apache.org. Available at: https://commons.apache.org/.