```java
//*********************************************************************************
//
// File:    UndirectedGraph.java
// Package: ---
// Unit:    Class UndirectedGraph
//
//*********************************************************************************

import java.util.ArrayList;
import java.util.LinkedList;
import edu.rit.pj2.vbl.DoubleVbl;
import edu.rit.util.Random;

/**
 * Class UndirectedGraph represents an undirected graph meaning that if
 * there exists an edge connecting some vertex A to some vertex B, then
 * that same edge connects vertex B to vertex A.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class UndirectedGraph {

    // private data members
    private ArrayList<UndirectedEdge> edges;
    private ArrayList<Vertex> vertices;
    private int v;

    // Prevent construction
    private UndirectedGraph() {

    }

    /**
     * Private constructor used internally by the static random graph
     * method
     * @param v the number of vertices in the graph
     */
    private UndirectedGraph(int v) {
        this.v = v;
        vertices = new ArrayList<Vertex>(v);
        edges = new ArrayList<UndirectedEdge>();
        for(int i = 0; i < v; i++) {
            vertices.add(new Vertex(i));
        }
    }

    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the id of the start vertex
     * @param goal the id of the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(int start, int goal) {
        return BFS(vertices.get(start), vertices.get(goal));
    }

```

```java
    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the reference to the start vertex
     * @param goal the reference to the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(Vertex start, Vertex goal) {
        int distance = 0, verticesToProcess = 1, uniqueNeighbors = 0;
        LinkedList<Vertex> queue = new LinkedList<Vertex>();
        boolean[] visited = new boolean[v];
        visited[start.n] = true;
        Vertex current, t2;
        queue.add(start);
        while(!queue.isEmpty()) {
            current = queue.removeFirst();
            if(current.equals(goal)) {
                return distance;
            }
            for(int i = 0; i < current.edgeCount(); i++) {
                t2 = current.getEdges().get(i).other(current);
                if(!visited[t2.n]) {
                    visited[t2.n] = true;
                    queue.add(t2);
                    uniqueNeighbors++;
                }
            }
            verticesToProcess--;
            if(verticesToProcess <= 0) {
                verticesToProcess = uniqueNeighbors;
                uniqueNeighbors = 0;
                distance++;
            }

        }
        return 0;
    }

    /**
     * Accumulate the distances of each pair of vertices into
     * a "running total" to be averaged
     *
     * @param thrLocal the reference to the "running total"
     * Prof. Alan Kaminsky's library handles averaging this
     * accumulated value.
     */
    public void accumulateDistances(DoubleVbl.Mean thrLocal) {
        for(int i = 0; i < v; i++) {
            for(int j = i + 1; j < v; j++) {
                int distance = BFS(i, j);
                // only accumulate the distance if the two vertices
                // are actually connected
                if(distance > 0) {
                    thrLocal.accumulate(distance);
                }
            }
        }
    }
```

```java

    /**
     * Generate a random graph with a PRNG, a specified vertex count and
     * an edge probability
     *
     * @param prng Prof. Alan Kaminsky's Perfect Random Number Generator
     * @param v number of vertices to use
     * @param p edge probability between vertices
     * @return the randomly generated graph
     */
    public static UndirectedGraph randomGraph(Random prng, int v, double p) {
        UndirectedGraph g = new UndirectedGraph(v);
        UndirectedEdge edge;
        Vertex a, b;
        int edgeCount = 0;
        for (int i = 0; i < v; i++) {
            for (int j = i + 1; j < v; j++) {
                // connect edges
                // always order it `i` then `j`
                if(prng.nextDouble() <= p) {
                    a = g.vertices.get(i);
                    b = g.vertices.get(j);
                    edge = new UndirectedEdge(edgeCount++, a, b);
                    g.edges.add(edge);
                }
            }
        }
        return g;
    }
}

```