96/100 nicely done, well documented, efficiently implemented


```
package ps4;

import java.io.*;
import java.util.*;
import java.lang.*;

/**
 * This class implements a testing driver for BipartiteGraph.  The
 * driver manages BipartiteGraphs whose nodes and edges are strings.
 **/

public class BipartiteGraphTestDriver {

        // Fields
        // Insert your code here.
        // (Consider the code below as a starting point.)

        HashMap graphs;

        /**
         * @effects Constructs a new test driver.
         */
        public BipartiteGraphTestDriver () {
                graphs = new HashMap ();
        }

        /**
         * @requires graphName != null
         * @modifies this
         * @effects Creates a new graph that shall be named
         * graphName.  The graph is initially empty.
         */
        public void createGraph(String graphName) {
                graphs.put(graphName, new BipartiteGraph());
        }

        /**
         * @requires createGraph(graphName) has been called on this
         *                      && nodeName != null
         *                      && neither addBlackNode(graphName,nodeName)
         *                                 nor addWhiteNode(graphName,nodeName)
         *                                      has already been called on this
         *
         * @modifies graph named graphName
         * @effects Adds a black node represented by the string nodeName
         * to the graph named graphName.
         */
        public void addBlackNode(String graphName, String nodeName) {
                BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);

    g.addNode(nodeName, NodeColor.BLACK);
        }

        /**
         * @requires createGraph(graphName) has been called on this
         *                      && nodeName != null
         *                      && neither addBlackNode(graphName,nodeName)
         *                                 nor addWhiteNode(graphName,nodeName)
         *                                      has already been called on this
         *
         * @modifies graph named graphName
         * @effects Adds a white node represented by the string nodeName
         * to the graph named graphName.
         */
        public void addWhiteNode(String graphName, String nodeName) {
                BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);

    g.addNode(nodeName, NodeColor.WHITE);
        }

        /**
         * @requires createGraph(graphName)
         *                      && ((addBlackNode(parentName) && addWhiteNode(childName))
         *                                  || (addWhiteNode(parentName) && addBlackNode(childName)))
         *                      && edgeLabel != null
         *                      && node named parentName has no other outgoing edge labeled edgeLabel
         *                      && node named childName has no other incoming edge labeled edgeLabel
         * @modifies graph named graphName
         * @effects Adds an edge from the node parentName to the node
         * childName in the graph graphName.  The new edge's label is the
         * string edgeLabel.
         */
        public void addEdge(String graphName,
                                                String parentName,
                                                String childName,
                                                String edgeLabel) {
                BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);

    g.addEdge(parentName, childName, edgeLabel);
        }

        /**
```

15/15

didn't you fix your .emacs file?


```
         * @requires createGraph(graphName)
         * @return a space-separated list of the names of all the black
         * nodes in the graph graphName, in alphabetical order.
         */
        public String listBlackNodes(String graphName) {
                BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);
    StringBuffer str = new StringBuffer();
    List l = new LinkedList();

    for (Iterator i = g.blackNodeIterator(); i.hasNext(); )
    {
            l.add(i.next());
    }

    Collections.sort(l);

    for (Iterator i = l.iterator(); i.hasNext(); )
    {
            str.append((String) i.next());
            if (i.hasNext())
        str.append(" ");
    }

    return str.toString();
        }

        /**
         * @requires createGraph(graphName)
         * @return a space-separated list of the names of all the white
         * nodes in the graph graphName, in alphabetical order.
         */
        public String listWhiteNodes(String graphName) {
                BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);
    StringBuffer str = new StringBuffer();
    List l = new LinkedList();

    for (Iterator i = g.whiteNodeIterator(); i.hasNext(); )
    {
            l.add(i.next());
    }

    Collections.sort(l);

    for (Iterator i = l.iterator(); i.hasNext(); )
    {
            str.append((String) i.next());
            if (i.hasNext())
        str.append(" ");
    }

    return str.toString();
        }

        /**
         * @requires createGraph(graphName) && createNode (parentName)
         * @return a space-separated list of the names of the children of parentName
         * in the graph graphName, in alphabetical order.
         */
        public String listChildren(String graphName, String parentName) {
                BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);
    StringBuffer str = new StringBuffer();
    List l = new LinkedList();

    for (Iterator i = g.listChildren(parentName); i.hasNext(); )
    {
            l.add(i.next());
    }

    Collections.sort(l);

    for (Iterator i = l.iterator(); i.hasNext(); )
    {
            str.append((String) i.next());
            if (i.hasNext())
        str.append(" ");
    }

    return str.toString();
        }

        /**
         * @requires createGraph(graphName) && createNode (childName)
         * @return a space-separated list of the names of the parents of
         * childName in the graph graphName, in alphabetical order
         */
        public String listParents(String graphName, String childName) {
                BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);
    StringBuffer str = new StringBuffer();
    List l = new LinkedList();
```

pull this out into a private method so you can eliminate duplicate code

can take an Iterator and return a String

```
        for (Iterator i = g.listParents(childName); i.hasNext(); )
        {
                l.add(i.next());
        }

        Collections.sort(l);

        for (Iterator i = l.iterator(); i.hasNext(); )
        {
                str.append((String) i.next());
                if (i.hasNext())
            str.append(" ");
        }

        return str.toString();

            }

            /**
             * @requires addEdge(graphName, parentName, c, edgeLabel) for some string c
             * @return the name of the child of parentName that is connected by the
             * edge labeled edgeLabel, in the graph graphName.
             */
            public String getChildByEdgeLabel (String graphName, String parentName, String edgeLabel) {
        BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);

        return (String) g.getChildByEdgeLabel(parentName, edgeLabel);
            }

            /**
             * @requires addEdge(graphName, p, childName, edgeLabel) for some string p
             * @return the name of the parent of childName that is connected by the
             * edge labeled edgeLabel, in the graph graphName.
             */
            public String getParentByEdgeLabel (String graphName, String childName, String edgeLabel) {
        BipartiteGraph g = (BipartiteGraph) graphs.get(graphName);

        return (String) g.getParentByEdgeLabel(childName, edgeLabel);
            }
    }
```

excellent

AF, RI


```
    // $Id: BipartiteGraph.java,v 1.11 2004/03/10 00:23:21 dan Exp $

    // Passed BipartiteGraphTest today -- drkp 2004/03/09

    package ps4;

    import java.util.Collections;
    import java.util.Iterator;
    import java.util.List;
    import java.util.LinkedList;
    import java.util.Map;
    import java.util.HashMap;
    import java.util.Set;
    import java.util.HashSet;
    import java.util.AbstractSet;


    /**
     * BipartiteGraph represents a directed bipartite graph with labeled
     * edges. This is a set of black and white nodes connected by directed
     * edges such that no edge connects two nodes with different
     * colors. Each node and edge is labeled with an Object associated
     * with it; any object can be used so long as it implements behavioral
     * equality. No two nodes can have the same label. And no two edges
     * can have the same label if they share a source or destination node.
     *
     * <p>
     *
     * We define specification types
     *          Node := (color : NodeColor , label : Object)
     *          Color := ( black | white )
     *  and Edge := (source : Node, destination : Node, label : Object)
     *
     * @specfield Nodes := List(Node)
     * @specfield Edges := List(Edge)
     */

    public class BipartiteGraph
    {
            // Internal class for tracking nodes.
            private class Node
            {
                    public Object label;
                    public NodeColor color;

                    // Maps between edge label and corresponding
                    // parent/child node
                    public Map<Object, Node> parents;
                    public Map<Object, Node> children;

                    public Node(Object label, NodeColor color)
                    {
                            parents = new HashMap();
                            children = new HashMap();

                            this.label = label;
                            this.color = color;
                    }
            }

            private Map<Object, Node> whiteNodes, blackNodes;

            // Abstraction function:
            //   Nodes = union(blackNodes, whiteNodes
            //   Edges = an edge (source, dest, label) is in Edges if there is
            //           an edge going from [illegible]
            //           source.children.get(label) == dest

            // Representation invariant:
            //   whiteNodes != null && blackNodes != null
            //   && whiteNodes and blackNodes do not contain nulls
            //           and
            //   [illegible]
            //   && for all nodes n in whiteNodes, n.color = White
            //   && for all nodes n in blackNodes, n.color = Black
            //   && for all nodes n in whiteNodes and blackNodes,
            //           n.parents != null && n.children != null
            //           && for each pair p in n.parents and n.children
            //                   p.value.color != n.color
            //                   && a corresponding link is in the corresponding
            //                           inverse, p.value.children or p.value.parents

            private void checkRep()
            {
                    for (int i = 0; i < 2; i++)
                    {
                            Map nodeMap, otherNodeMap;
                            NodeColor nodeColor;
```

nice, clean

```
                // check either white or black nodes on each pass
                if (i == 0)
                {
                        nodeMap = whiteNodes;
                        otherNodeMap = blackNodes;
                        nodeColor = NodeColor.WHITE;
                }
                else
                {
                        nodeMap = blackNodes;
                        otherNodeMap = whiteNodes;
                        nodeColor = NodeColor.BLACK;
                }

                for (Iterator it = nodeMap.keySet().iterator(); it.hasNext(); )
                {
                        Object key = it.next();
                        Node n = (Node) nodeMap.get(key);

                        Assert.assertNotNull(n);
                        Assert.assertSame(key, n.label);
                        Assert.assertNull(otherNodeMap.get(key));
                        Assert.assertEquals(nodeColor, n.color);
                        Assert.assertNotNull(n.parents);
                        Assert.assertNotNull(n.children);

                        for (Iterator it2 = n.parents.keySet().iterator(); it2.hasNext(); )
                        {
                                Object edgeLabel = it2.next();
                                Node n2 = (Node) n.parents.get(edgeLabel);

                                Assert.assertNotNull(n2);
                                Assert.assertNotSame(n.color, n2.color);
                                Assert.assertEquals(n, n2.children.get(edgeLabel));
                        }

                        for (Iterator it2 = n.children.keySet().iterator(); it2.hasNext(); )
                        {
                                Object edgeLabel = it2.next();
                                Node n2 = (Node) n.children.get(edgeLabel);

                                Assert.assertNotNull(n2);
                                Assert.assertNotSame(n.color, n2.color);
                                Assert.assertEquals(n, n2.parents.get(edgeLabel));
                        }
                }
        }
}

/* Private helper function for finding the Node record associated
 * with a given label, regardless of which Map it's in
 */
private Node findNode(Object label)
{
        Node n;

        n = (Node) whiteNodes.get(label);
        if (n == null)
                n = (Node) blackNodes.get(label);

        return n;
}

/**
 * @effects constructs a new BipartiteGraph containing no
 * nodes or edges.
 */
public BipartiteGraph()
{
        whiteNodes = new HashMap();
        blackNodes = new HashMap();

        checkRep();
}

/**
 * Adds a new node to the graph with the specified label and
 * color.
 *
 * @requires none
 * @modifies this
 * @effects Nodes = Nodes + (color, label)
 * @throws NullPointerException if label = null or color = null
 * @throws IllegalArgumentException if a node with the same label
```

requires label, color ≠ null or omit @requires

drop node not already in graph

@requires none is somewhat misleading


```
 * already exists
 */
public void addNode(Object label, NodeColor color)
{
        checkRep();

        if (color == null)
                throw new NullPointerException();

        if (findNode(label) != null)
                throw new IllegalArgumentException("node already exists");

        if (color.equals(NodeColor.WHITE))
        {
                whiteNodes.put(label, new Node(label, color));
        }
        else if (color.equals(NodeColor.BLACK))
        {
                blackNodes.put(label, new Node(label, color));
        }
        else
        {
                throw new IllegalArgumentException("unexpected color");
        }

        checkRep();
}

/**
 * Adds a labeled edge between two nodes.
 *
 * @requires none
 * @modifies this
 * @effects Edges = Edges + (sourceNode, destNode), where
 * sourceNode and destNode are nodes from Nodes with names
 * source and dest respectively.
 * @throws NullPointerException if parent = null, child =
 * null, or label = null
 * @throws IllegalArgumentException if no node labeled parent or
 * child exists in the graph, if another edge with the same source
 * or destination node and same label exists in the graph, if the
 * edge already exists, or the source and destination nodes have
 * the same color.
 */
public void addEdge(Object parent, Object child, Object label)
{
        checkRep();

        if (parent == null || child == null || label == null)
                throw new NullPointerException();

        Node parentNode = findNode(parent);
        Node childNode = findNode(child);

        if (parentNode == null || childNode == null)
                throw new IllegalArgumentException();

        if (parentNode.color.equals(childNode.color))
                throw new IllegalArgumentException("both nodes have same color");

        // check whether an edge with the same label and either the
        // same source or same destination exists

        if (parentNode.children.containsKey(label)
                || childNode.parents.containsKey(label))
                throw new IllegalArgumentException("edge label conflict");

        // check whether the edge already exists
        for (Iterator i = listChildren(parent); i.hasNext(); )
        {
                if (i.next().equals(child))
                        throw new IllegalArgumentException("edge already exists");
        }

        // finally, add the edge
        parentNode.children.put(label, childNode);
        childNode.parents.put(label, parentNode);

        checkRep();
}

/**
 * Counts the number of nodes.
 *
 * @requires none
 * @returns the total number of nodes in the graph
```

again, requires should reflect throws

I prefer IllegalArgumentException

NPE could be anything; IAE more instructive

```
     */
    public int countNodes()
    {
        checkRep();

        return whiteNodes.size() + blackNodes.size();
    }

    /**
     * Counts the nodes with a given color.
     *
     * @requires none
     * @returns the number of nodes n such that n.color = color
     */
    public int countNodes(NodeColor color)
    {
        int count = 0;

        checkRep();

        if (color.equals(NodeColor.BLACK))
            return blackNodes.size();
        else if (color.equals(NodeColor.WHITE))
            return whiteNodes.size();
        else    // TODO: add support for pink and turquoise nodes
            throw new IllegalArgumentException("unexpected color");
    }

    /**
     * Provides an iterator over the set of labels of
     * black nodes. The iterator does not support the remove()
     * operation.
     *
     * @requires none
     * @returns an iterator over the set of labels of black nodes
     */
    public Iterator blackNodeIterator()
    {
        checkRep();

        return Collections.unmodifiableCollection(blackNodes.keySet())
                .iterator();
    }

    /**
     * Provides an iterator over the set of labels of
     * white nodes. The iterator does not support the remove()
     * operation.
     *
     * @requires none
     * @returns an iterator over the set of labels of white nodes
     */
    public Iterator whiteNodeIterator()
    {
        checkRep();

        return Collections.unmodifiableCollection(whiteNodes.keySet())
                .iterator();
    }

    /**
     * Checks whether the graph contains a node by label.
     *
     * @requires none
     * @returns true if a node n exists in Nodes with n.label =
     * label
     * @throws NullPointerException if label == null
     */
    public boolean containsNode(Object label)
    {
        checkRep();

        if (label == null)
            throw new NullPointerException();

        return (findNode(label) != null);
    }

    /**
     * Returns the color of a node, by label.
```



```
     *
     * @requires none
     * @returns n.color, where n is the node in Nodes such that
     * n.label = label
     * @throws NullPointerException if label = null
     * @throws IllegalArgumentException if no such node n exists
     */
    public NodeColor getNodeColor(Object label)
    {
        checkRep();

        if (label == null)
            throw new NullPointerException();

        Node n = findNode(label);

        if (n == null)
            throw new IllegalArgumentException("node not found");

        return n.color;
    }

    /**
     * Returns an iterator over the list of labels of children of
     * a given node. The iterator does not support the remove()
     * operation.
     *
     * @requires none
     * @returns an iterator to a list of the labels of children of
     * the specified node: m.label for all nodes m such that there
     * exists an edge e with e.source = n and e.dest = m, where n
     * is the node with n.label = label
     * @throws NullPointerException if label == null,
     * @throws IllegalArgumentException if no node n with n.label =
     * label exists in Nodes
     */
    public Iterator listChildren(Object label)
    {
        checkRep();

        if (label == null)
            throw new

        if
            throw new

            new

        for

        checkRep();

        return Collections.unmodifiable

    /**
     * Returns an iterator over the list of labels of parents of
     * a given node. The iterator does not support the remove()
     * operation.
     *
     * @requires none
     * @returns an iterator to a list of the labels of parents of
     * the specified node: m.label for all nodes m such that there
     * exists an edge e with e.source = m and e.dest = n, where n
     * is the node with n.label = label
     * @throws NullPointerException if label == null,
     * @throws IllegalArgumentException if no node n with n.label =
     * label exists in Nodes
     */
    public Iterator listParents(Object label)
    {
        checkRep();

        if (label == null)
            throw new NullPointerException();

        Node n = findNode(label);
        if (n == null)
            throw new IllegalArgumentException
```

I avoid l as a variable name since it looks like a one, maybe it's personal preference


```
                List l = new LinkedList();

                for (Iterator i = n.parents.values().iterator(); i.hasNext(); )
                {
                        Node n2 = (Node) i.next();
                        l.add(n2.label);
                }

                checkRep();

                return Collections.unmodifiableList(l).iterator();
        }


        /**
         * Finds the label of a child node, given the labels of the
         * parent and the edge.
         *
         * @requires none
         * @returns e.dest.label, where n is the node such that
         * n.label = parentLabel and e is the edge such that e.source = n
         * and e.label = edgeLabel
         * @throws NullPointerException if parentLabel = null or
         * edgeLabel = null
         * @throws IllegalArgumentException if no such node n or edge e
         * exists
         */
        public Object getChildByEdgeLabel(Object parentLabel,
                Object edgeLabel)
        {
                if (parentLabel == null || edgeLabel == null)
                        throw new NullPointerException();

                Node n = findNode(parentLabel);
                if (n == null)
                        throw new IllegalArgumentException("no such node");

                Node n2 = (Node) n.children.get(edgeLabel);
                if (n2 == null)
                        throw new IllegalArgumentException("no such edge");

                return n2.label;
        }


        /**
         * Finds the label of a parent node, given the labels of the
         * child and the edge.
         *
         * @requires none
         * @returns e.source.label, where n is the node such that
         * n.label = childLabel and e is the edge such that e.dest = n
         * and e.label = edgeLabel
         * @throws NullPointerException if childLabel = null or
         * edgeLabel = null
         * @throws IllegalArgumentException if no such node n or edge e
         * exists
         */
        public Object getParentByEdgeLabel(Object childLabel,
                Object edgeLabel)
        {
                if (childLabel == null || edgeLabel == null)
                        throw new NullPointerException();

                Node n = findNode(childLabel);
                if (n == null)
                        throw new IllegalArgumentException("no such node");

                Node n2 = (Node) n.parents.get(edgeLabel);
                if (n2 == null)
                        throw new IllegalArgumentException("no such edge");

                return n2.label;
        }


        /**
         * Behavioral equality test
         *
         * @returns true iff this and o are behaviorally
         * (i.e. referentially) equivalent.
         */
        public boolean equals(Object o)
        {
                return (this == o);
        }
```



```
        /**
         * Remove a node and all associated edges.
         *
         * @requires none
         * @effects Nodes = Nodes - n, where n.label = label;
         * Edges = Edges - E, where E is the set of all e with e.source =
         * n or e.dest = n
         * @throws NullPointerException if label == null
         * @throws IllegalArgumentException if no node labelled label
         * exists
         */
        public void removeNode(Object label)
        {
                checkRep();

                if (label == null)
                        throw new NullPointerException();

                Node n = findNode(label);
                if (n == null)
                        throw new IllegalArgumentException("node not found");

                [illegible]
                Set keys = new [illegible]
                for [illegible]
                {
                        [illegible]
                }

                [illegible]
                keys = new [illegible]
                for [illegible]
                {
                        [illegible]
                }

                [illegible]

                checkRep();
        }


        /**
         * Remove an edge, given the source endpoint label and the edge
         * label.
         *
         * @requires none
         * @effects Edges = Edges - e, where e.source.label = nodeLabel
         * and e.label
         * @throws NullPointerException if label == null or
         * nodeLabel == null
         * @throws IllegalArgumentException if no node labeled nodeLabel
         * exists, or if no edge labeled edgeLabel exists from nodeLabel
         */
        public void removeEdge(Object nodeLabel, [illegible]
        {
                checkRep();

                if (nodeLabel == null || edgeLabel == null)
                        throw new NullPointerException();

                Node n = findNode(nodeLabel);
                if (n == null)
                        throw new IllegalArgumentException("source node not found");

                [illegible]

                if [illegible]
                {
                        [illegible]
                }

                else

                        throw new [illegible]

                [illegible]
        }
}
```

nice support for removal

17/20

Dan, your overview, AF, and RI are thorough and excellent. The implementation is clean and the specs are good, except:

(1) I wouldn't write "requires: none" on things that throw exceptions. "requires: none" suggests that you can pass anything and the method will execute okay which isn't quite true since it may throw an exception. Bloch mentions that it may be obsessive to write that the params should not be null all over the place, so I would just omit the requires clause (which implies "requires: true") and then hopefully your clients will read more of your spec instead of stopping after reading "requires: none"

(2) Throwing IllegalArgumentExceptions is more helpful than throwing NPEs. NPEs should happen "by accident" whereas an IllegalArgumentException is more helpful for the client in identifying his error.

14/15

```
// $Id: Simulator.java,v 1.6 2004/03/10 00:33:37 dan Exp $

// Passed SimulatorTest today -- drkp 2004/03/09

package ps4;

import java.util.Iterator;
import java.util.Map;
import java.util.HashMap;
import junit.framework.Assert;


/**
 * The Simulator class represents a pipe-and-filter simulator.
 *
 * @specfield pipes : list<pipe>
 * @specfield filters : list<filter>
 * @specfield edges : list<edge>
 * @derivedfield nodes : union(pipes, filters)
 */

public class Simulator
{
        BipartiteGraph graph;
        Map/*<Object, Pipe>*/ pipes;
        Map/*<Object, Filter>*/ filters;

        /* Abstraction function:
           pipes = pipes
           filters = filters
           edges = graph edges
         Representation invariant:
           graph is ...
           ...
           ...
           ... */

        void checkRep()
        {
    Assert.assertNotNull(graph);
    Assert.assertNotNull(pipes);
    Assert.assertNotNull(filters);

    for (Iterator it = graph.blackNodeIterator(); it.hasNext() ;)
    {
            Assert.assertNotNull(pipes.get(it.next()));
    }

    for (Iterator it = graph.whiteNodeIterator(); it.hasNext() ;)
    {
            Assert.assertNotNull(filters.get(it.next()));
    }

    for (Iterator it = pipes.entrySet().iterator(); it.hasNext() ;)
    {
            Map.Entry entry = (Map.Entry) it.next();
            Assert.assertTrue(entry.getValue() instanceof Pipe);
            Assert.assertTrue(graph.containsNode(entry.getKey()));
            Assert.assertEquals(NodeColor.BLACK,
        graph.getNodeColor(entry.getKey()));
    }

    for (Iterator it = filters.entrySet().iterator(); it.hasNext() ;)
    {
            Map.Entry entry = (Map.Entry) it.next();
            Assert.assertTrue(entry.getValue() instanceof Filter);
            Assert.assertTrue(graph.containsNode(entry.getKey()));
            Assert.assertEquals(NodeColor.WHITE,
          graph.getNodeColor(entry.getKey()));
    }
        }


        /**
         * @effects constructs a new Simulator object with empty pipes
         * and filters lists
         */
        public Simulator()
        {
    graph = new BipartiteGraph();
    pipes = new HashMap();
    filters = new HashMap();

    checkRep();
        }
```

these pipes and filters HashMaps contain redundant information: you should just extract this info from the graph

AF should say if Pipe is white or black node

```
        /**
         * Add a new pipe to the simulator.
         *
         * @requires label != null, pipe != null, nodes does not contain
         * a pipe or filter with the same label
         * @effects pipes = pipes + p;
         */

        public void addPipe(String label, Pipe p)
        {
checkRep();

graph.addNode(label, NodeColor.BLACK);
pipes.put(label, p);

                checkRep();
        }


        /**
         * Add a new filter to the simulator.
         *
         * @requires label != null, filter != null, nodes does not contain
         * a pipe or filter with the same label
         * @effects filters = filters + f
         */

        public void addFilter(String label, Filter f)
        {
                checkRep();

                graph.addNode(label, NodeColor.WHITE);
                filters.put(label, f);

                checkRep();
        }


        /**
         * Add a new edge connecting a pipe and filter.
         *
         * @requires parentLabel and childLabel are labels of two nodes in
         * nodes; exactly one of the two is a pipe and exactly one is a
         * filter; (parentLabel, childLabel) is not already in edges
         * @effects edges = edges + (parentLabel, childLabel)
         */

        public void addEdge(String parentLabel, String childLabel,
                String edgeName)
        {
                graph.addEdge(parentLabel, childLabel, edgeName);
        }


        /**
         * Directly inject an input object into a pipe.
         *
         * @requires pipeLabel is the label of a node in pipes, o != null
         * @effects o is injected into pipe p, where p.label = pipeLabel
         */

        public void injectIntoPipe(String pipeLabel, Object o)
        {
                Pipe p = (Pipe) pipes.get(pipeLabel);

                p.inject(o);
        }


        /**
         * Directly extract an object from a pipe's output.
         *
         * @requires pipeLabel is the label of a node in pipes
         * @effects returns the first object extracted from the output
         * queue of the pipe p with p.label = pipeLabel, or null if the
         * pipe's output queue is empty.
         */

        public Object extractFromPipe(String pipeLabel)
        {
                Pipe p = (Pipe) pipes.get(pipeLabel);

                if (p.outputAvailable())
                        return p.extract();
                else
                        return null;
        }


        /**
```



```
         * Simulate all pipes and filters in the simulator
         *
         * @requires none
         * @effects Simulates each pipe in pipes for one time slice, then
         * simulates each filter in filters for one time slice.
         */

        public void simulate()
        {
                for (Iterator it = pipes.values().iterator(); it.hasNext(); )
                {
                        Pipe p = (Pipe) it.next();
                        p.simulate(graph, 1);
                }

                for (Iterator it = filters.values().iterator(); it.hasNext(); )
                {
                        Filter f = (Filter) it.next();
                        f.simulate(graph, 1);
                }
        }


        /**
         * Obtain an iterator over the contents of a pipe
         *
         * @requires pipeLabel is the label of a node in pipes
         * @effects returns an unmodifiable iterator over the list of
         * elements in the specified pipe
         */

        public Iterator pipeContentsIterator(String pipeLabel)
        {
                Pipe p = (Pipe) pipes.get(pipeLabel);

                return p.contentsIterator();
        }

        /**
         * Get a Pipe object by label
         *
         * @returns the Pipe whose label is label, or null if no such
         * pipe exists
         */

        Pipe getPipeByLabel(Object label)
        {
return (Pipe) pipes.get(label);
        }


        /**
         * Get a Filter object by label
         *
         * @returns the Filter whose label is label, or null if no such
         * filter exists
         */

        Filter getFilterByLabel(Object label)
        {
return (Filter) filters.get(label);
```

Mar 09, 04 19:33 **IntPipe.java** Page 1/1

```
// $Id: IntPipe.java,v 1.6 2004/03/10 00:33:36 dan Exp $

// Passed SimulatorTest today -- drkp 2004/03/09

package ps4;

/**
 * An IntPipe stores a flow of integers, with a delay of one time
 * slice and infinite capacity.
 * @specfield label : Object
 * @specfield simulator : Simulator (the simulator that contains this
 * Pipe)
 */

public class IntPipe extends Pipe
{
        // Abstraction function and rep invariant:
        //   Trivial -- same as Pipe superclass

        /**
         * @effects constructs a new IntPipe with the given label and
         * simulator
         * @requires label != null, sim != null
         */

        public IntPipe(Object label, Simulator sim)
        {
    super(label, sim);

    checkRep();
        }


        /**
         * @requires timeSlice > 0
         * @modifies this
         * @effects Simulates this IntPipe for the length of time
         * given by timeSlice.
         */

        public void simulate(BipartiteGraph g, double timeSlice)
        {
    while (!inQueue.isEmpty())
    {
            Object o = inQueue.remove(0);

            if (o instanceof Integer)
            {
        // Only add to the output queue if the input is an
        // integer; otherwise, discard it
        outQueue.add(o);
            }
    }

    checkRep();
        }
}
```

nice design

20/20

owners

Mar 09, 04 19:33 **Pipe.java** Page 1/2

```
package ps4;

import
import
import
import
import

/**
 * The Pipe abstract class is implemented by each of the pipes in the
 * pipe-and-filter simulation.
 *
 * @specfield inputQueue : List<Object>
 * @specfield outputQueue : List<Object>
 * @specfield label : Object
 * @specfield simulator : Simulator (the simulator that contains this
 * Pipe)
 */

public abstract class Pipe implements Simulatable
{
        protected List inQueue, outQueue;
        protected Object label;
        protected Simulator sim;

        // Abstraction function:
        //  inputQueue = inQueue, outputQueue = outQueue
        // Rep invariant
        //  label != null, sim != null, inQueue != null, outQueue != null

        public void checkRep()
        {
    Assert.assertNotNull(label);
    Assert.assertNotNull(sim);
    Assert.assertNotNull(inQueue);
    Assert.assertNotNull(outQueue);
        }


        /**
         * @effects constructs a new Pipe with the given label and
         * simulator
         * @requires label != null, sim != null
         */

        public Pipe(Object label, Simulator sim)
        {
    if (label == null || sim == null)
            throw new NullPointerException();

    this.label = label;
    this.sim = sim;
    inQueue = new
    outQueue = new

    checkRep();
        }


        /**
         * Inject an object into the pipe.
         *
         * @requires o != null
         * @modifies this
         * @effects Adds o to the end of the input queue of the pipe.
         */

        public void inject(Object o)
        {
    checkRep();

    if (o == null)
            throw new NullPointerException();

    inQueue.add(o);

            checkRep();
        }


        /**
         * Extract an object from the pipe.
         *
         * @requires At least one element is in the output queue.
         * @modifies this
         * @returns The first element from the output queue of the pipe.
         * @effects Removes the first element from the output queue.
         */
```

Consider renaming AbstractPipe

what if you wanted to have a Pipe that accepts nulls? subclass inherits spec

```
        public Object extract()
        {
    checkRep();

    if (outQueue.isEmpty())
            throw new RuntimeException("no element available");

    return outQueue.remove(0);
        }


        /**
         * Check if any output is available from the pipe
         *
         * @requires none
         * @returns true output is available; i.e. if the output queue
         * contains at least one element.
         */

        public boolean outputAvailable()
        {
    checkRep();

    return !(outQueue.isEmpty());
        }


        /**
         * Obtain an iterator over the contents of the pipe
         *
         * @requires none
         * @effects returns an unmodifiable iterator over the list of
         * elements in this pipe, in both the output and input queues,
         * starting with the one that will be output next, and ending
         * with the one most recently input.
         */

        public Iterator contentsIterator()
        {
    checkRep();

    List l = new ArrayList();

    l.addAll(outQueue);
    l.addAll(inQueue);

    checkRep();

    return Collections.unmodifiableList(l).iterator();
        }

}
```


Consider renaming Abstract Filter

```
package [illegible]

import [illegible]

/**
 * The Filter abstract class represents a filter in the
 * pipe-and-filter simulator.
 *
 * @specfield label : Object
 * @specfield simulator : Simulator (the simulator that contains this Filter)
 */

public abstract class Filter implements Simulatable
{

        protected Object label;
        protected Simulator sim;

        // Abstraction function:
        //   label = label; simulator = sim
        // Representation invariant:
        //   label != null, simulator != null

        public void checkRep()
        {
    Assert.assertNotNull(label);
    Assert.assertNotNull(sim);
        }


        public Filter(Object label, Simulator sim)
        {
    if ([illegible])
            throw new [illegible]

    this.label = label;
    this.sim = sim;

    checkRep();
        }


        /**
         * Emit an object into a connected pipe.
         *
         * @requires o != null, g is a BipartiteGraph containing node
         * this.label
         * @effects Places o into the pipe found by traveling along the
         * outgoing edge labeled edgeLabel from the node labeled
         * this.label in the BipartiteGraph g. If edge edgeLabel does not
         * exist, nothing happens (the output is discarded).
         */

        protected void emit(BipartiteGraph g, Object edgeLabel, Object o)
        {
    checkRep();

    if (o == null || g == null || edgeLabel == null)
            throw new NullPointerException();

    try
    {
            Object pipeLabel = g.getChildByEdgeLabel(label, edgeLabel);
            Pipe p = sim.getPipeByLabel(pipeLabel);
            p.inject(o);
    }
    catch (IllegalArgumentException e)
    {
            // ignore. If no edge is found, this just means that the
            // output connection is disconnected; we discard the
            // output.
    }

    checkRep();
        }


        /**
         * Check whether input is available from a connected pipe.
         *
         * @requires g is a BipartiteGraph containing node this.label with
         * incoming edge edgeLabel
         * @returns true if an object is available for extraction in the
         * pipe found by travelling along the edge labeled edgeLabel from
         * the node labeled this.label in the BipartiteGraph g
         */

```

```
    protected boolean inputAvailable(BipartiteGraph g, Object edgeLabel)
    {
checkRep();

if (g == null || edgeLabel == null)
        throw new NullPointerException();

Object pipeLabel = g.getChildByEdgeLabel(label, edgeLabel);
Pipe p = sim.getPipeByLabel(pipeLabel);

checkRep();

return p.outputAvailable();
    }

    /**
     * Receive an object from a connected pipe.
     *
     * @requires g is a BipartiteGraph containing node this.label with
     * incoming edge edgeLabel
     * @returns The first object received from the pipe found by
     * travelling along the edge labeled edgeLabel from the node
     * labeled this.label in the BipartiteGraph g, or null if the
     * pipe is empty.
     */
    protected Object receive(BipartiteGraph g, Object edgeLabel)
    {
checkRep();

if (g == null || edgeLabel == null)
        throw new NullPointerException();

Object pipeLabel = g.getChildByEdgeLabel(label, edgeLabel);
Pipe p = sim.getPipeByLabel(pipeLabel);

checkRep();

if (p.outputAvailable())
        return p.extract();
else
        return null;
    }

    /**
     * Emit an object to a specific pipe, by pipe label.
     *
     * @requires o != null, g is a BipartiteGraph containing node
     * this.label
     * @effects Places o into the pipe whose node label is nodeLabel
     * in the BipartiteGraph g. If the node does not exist, nothing
     * happens (the output is discarded).
     */
    protected void emitByPipeLabel(BipartiteGraph g, Object pipeLabel,
Object o)
    {
checkRep();

if (o == null || g == null || pipeLabel == null)
        throw new NullPointerException();

try
{
        Pipe p = sim.getPipeByLabel(pipeLabel);
        if (p != null)
    p.inject(o);
}
catch (IllegalArgumentException e)
{
        // ignore. If no edge is found, this just means that the
        // output connection is disconnected; we discard the
        // output.
}

checkRep();
    }

    /**
     * Check whether input is available from a connected pipe.
     *
     * @requires g is a BipartiteGraph containing node this.label and
     * node pipeLabel
     *
     * @returns true if an object is available for extraction in the
     * pipe with label pipeLabel
     */
```



```
    protected boolean inputAvailableByPipeLabel(BipartiteGraph g, Object pipeLabel)
    {
checkRep();

if (g == null || pipeLabel == null)
        throw new NullPointerException();

Pipe p = sim.getPipeByLabel(pipeLabel);

if (p != null)
        return p.outputAvailable();
else
        return false;
    }

    /**
     * Receive an object from a connected pipe.
     *
     * @requires g is a BipartiteGraph containing node this.label and
     * node nodeLabel
     * @returns The first object received from the pipe with label
     * nodeLabel, or null if the pipe is empty.
     */
    protected Object receiveByPipeLabel(BipartiteGraph g, Object pipeLabel)
    {
checkRep();

if (g == null || pipeLabel == null)
        throw new NullPointerException();

Pipe p = sim.getPipeByLabel(pipeLabel);

if (p.outputAvailable())
        return p.extract();
else
        return null;
    }
}
```

```
// $Id: PlusFilter.java,v 1.6 2004/03/10 00:33:36 dan Exp $

// Passed SimulatorTest today -- drkp 2004/03/09

package ps4;
import java.util.Iterator;

/**
 * A PlusFilter sums the inputs presented to it on each input pipe
 * and places the output on the output pipe each timeslice. It may be
 * connected to any number of input and output pipes, including zero;
 * each output pipe behaves identically. If input that is not an
 * Integer is received on an input pipe, it is ignored.
 *
 * @specfield label : Object
 * @specfield simulator : Simulator (the simulator that contains this Filter)
 */

public class PlusFilter extends Filter
{
        // Abstraction function and rep invariant:
        //  same as superclass

        /**
         * @effects constructs a new PlusFilter with the given label
         * @requires label != null
         */

        public PlusFilter(Object label, Simulator sim)
        {
    super(label, sim);

    checkRep();
        }


        /**
         * @requires timeSlice > 0
         * @effects Simulates this PlusFilter for the length of time
         * given by timeSlice.
         */

        public void simulate(BipartiteGraph g, double timeSlice)
        {
    int result = 0;

    // sum the values on each input pipe (if they are available)

    for (Iterator it = g.listParents(label); it.hasNext(); )
    {
            Object pipeLabel = it.next();
            if (inputAvailableByPipeLabel(g, pipeLabel))
            {
        Object o = receiveByPipeLabel(g, pipeLabel);
        if (o instanceof Integer)
                result += ((Integer) o).intValue();
            }
    }

    for (Iterator it = g.listChildren(label); it.hasNext(); )
    {
            Object pipeLabel = it.next();
            emitByPipeLabel(g, pipeLabel, new Integer(result));
    }

    checkRep();
        }
}
```



```
package ps4;

import java.io.*;
import java.util.*;

/**
 * This class implements a testing driver for Simulator.  The
 * driver manages Simulators for Integer arithmetic.
 **/

public class SimulatorTestDriver {

        // Fields
        HashMap simulators, simulatorByNode;

        /**
         * @effects Constructs a new test driver.
         */
        public SimulatorTestDriver() {
                simulators = new HashMap();
                simulatorByNode = new HashMap();
        }


        /**
         * @requires simName != null
         * @modifies this
         * @effects Creates a new Simulator that shall be named
         * simName.  The simulator's graph is initially empty.
         */
        public void createSimulator(String simName) {
                simulators.put(simName, new Simulator());
        }


        /**
         * @requires createSimulator(simName) &&
         *                   nodeName != null
         *                   && nodeName has not been used in a previous
         *                           addPipe, addPlusFilter, or addGCDFilter
         *                           call on this object
         * @modifies this, simulator named simName
         * @effects Creates a new IntPipe named by the string nodeName
         * The node can be referred to by name after it has been created
         * with this method.
         * Adds a pipe represented by the string nodeName
         * to the simulator named simName.
         */
        public void addPipe(String simName, String nodeName) {
    Simulator sim = (Simulator) simulators.get(simName);
    sim.addPipe(nodeName, new IntPipe(nodeName, sim));
    simulatorsByNode.put(nodeName, sim);
        }

        /**
         * @requires createSimulator(simName) &&
         *                   nodeName != null
         *                   && nodeName has not been used in a previous
         *                           addPipe, addPlusFilter, or addGCDFilter
         *                           call on this object
         * @modifies this, simulator named simName
         * @effects Creates a new PlusFilter named by the string nodeName.
         * The node can be referred to by name after it has been created
         * with this method.
         * Adds a filter represented by the string nodeName
         * to the simulator named simName.
         */
        public void addPlusFilter(String simName, String nodeName) {
                Simulator sim = (Simulator) simulators.get(simName);
                sim.addFilter(nodeName, new PlusFilter(nodeName, sim));
                simulatorsByNode.put(nodeName, sim);
        }

        /**
         * @requires createSimulator(simName) &&
         * @requires nodeName != null
         *                   && nodeName has not been used in a previous
         *                           addPipe, addPlusFilter, or addGCDFilter
         *                           call on this object
         * @modifies this, simulator named simName
         * @effects Creates a new GCDFilter named by the string nodeName.
         * The node can be referred to by name after it has been created
         * with this method.
         * Adds a filter represented by the string nodeName
         * to the simulator named simName.
         */
        public void addGCDFilter(String simName, String nodeName) {
    Simulator sim = (Simulator) simulators.get(simName);
    sim.addFilter(nodeName, new GCDFilter(nodeName, sim));
    simulatorsByNode.put(nodeName, sim);
        }

}
```

Mar 09, 04 19:33 **SimulatorTestDriver.java** Page 2/2

```
        /**
         * @requires createSimulator(simName)
         *                  && (          (addPipe(parentName) && addFilter(childName))
         *                              || (addFilter(parentName) && addPipe(childName)) )
         *
         *                  && edgeLabel != null
         *                  && node named parentName has no other outgoing edge labeled edgeLabel
         *                  && node named childName has no other incoming edge labeled edgeLabel
         *
         * @modifies simulator named simName
         * @effects Adds an edge from the node parentName to the node
         * childName in the simulator simName.  The new edge's label is the
         * string edgeLabel.
         */
        public void addEdge(String simName,
                            String parentName,
                            String childName,
                            String edgeLabel) {
    Simulator sim = (Simulator) simulators.get(simName);
    sim.addEdge(parentName, childName, edgeLabel);
        }

        /**
         * @requires addPipe(pipeName)
         * @modifies simulator containing pipe named pipeName
         * @effects pushes the integer value into the pipe named pipeName
         */
        public void injectInput (String pipeName,
                                 int value) {
    Simulator sim = (Simulator) simulatorsByNode.get(pipeName);
    sim.injectIntoPipe(pipeName, new Integer(value));
        }

        /**
         * @requires addPipe(pipeName)
         * @return a space-separated list of the integer values currently in
         * the pipe named pipeName
         */
        public String listContents (String pipeName) {
    Simulator sim = (Simulator) simulatorsByNode.get(pipeName);
    StringBuffer s = new StringBuffer();

    Object o;

    for (Iterator it = sim.pipeContentsIterator(pipeName); it.hasNext(); )
    {
            s.append(" ");
            s.append(((Integer) it.next()).toString());
    }

    String str = s.toString();

    if (str.equals(""))
    {
        return str;
    } else {
        return str.substring(1); // chop off the leading space
    }

        }

        /**
         * @requires createSimulator(simName)
         * @modifies simulator named simName
         * @effects runs simulator named simName for a single 1-second time slice
         */
        public void simulate (String simName) {
    Simulator sim = (Simulator) simulators.get(simName);

    sim.simulate();
        }
}
```

Mar 10, 04 22:25 **test.out** Page 1/1

```
SUMMARY FOR ps4.stafftests.SimulatorTest
Pass : 3 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 3
SUMMARY FOR ps4.stafftests.BipartiteGraphTest
Pass : 9 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 9
SUMMARY FOR ALL TESTS
Pass : 12 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 12
GRADES
ps4.stafftests.SimulatorTest: 15/15
ps4.stafftests.BipartiteGraphTest: 15/15
GRADES TOTAL: 30/30
```

15/15

I checked out your ps4 in eclipse to get your problem4.txt. Your write-up was clear and thorough without overstating — great job!

15/15