1. CLASSES WITH SPECIFICATION

The subsections 1.1 and 1.2 contain the `PrinterServer` class and the `SimpleListNode` class with specification respectively.

1.1 The PrinterServer class

Listing 1: The `PrinterServer` class with specification

```
package server;

import java.util.concurrent.locks.
    ReentrantLock;

/**
 * This class models a printer server.
 * @author Martijn Roo
 * @version Revision: 1.0
 */
public class PrinterServer {

  /*@
   * public resource state(int i) =
   *   Perm(queue[i],100) **
   * ((queue[i] != null) -*
   *   queue[i].nodeState());
  @*/


  /**
   * The queue of this printer server.
   */
  private SimpleListNode[] queue;
  /**
   * The list of locks where each lock
   * protects an element of the queue.
   */
  private ReentrantLock[] locks;

  /**
   * Constructs this printer server.
   * @requires true;
   * @ensures Perm(queue, 100) **
   *   Perm(length, 100) ** Perm(locks, 100);
   */
  private PrinterServer() {}

  /**
   * Constructs a PrinterServer with the
   * specified number of users in the queue.
   * Makes queue, length and locks read-only.
   * @param numberOfUsers The number of users
   *   the printer server supports.
   * @requires Perm(numberOfUsers, p);
   * @ensures Perm(queue, 100-p) **
   *   Perm(length, 100-p) **
   *   Perm(locks, 100-p) **
   *   (\forall int j; 0 <= j &&
   *   j < length; Perm(locks[j], 100-p));
   */
  public PrinterServer(int numberOfUsers) {
    this();
    init(numberOfUsers);
  }

  /**
   * Initializes this printer server. Makes
   * queue, length and locks read-only.
   * @param numberOfUsers The number of users
   *   the printer server supports.
   * @requires Perm(queue, 100) **
   *   Perm(locks, 100) **
   *   Perm(numberOfUsers, p);
   * @ensures Perm(queue, 100-p) **
   *   Perm(locks, 100-p) **
   *   (\forall int j; 0 <= j && j < length;
   *     Perm(locks[j], 100-p) **
   *     locks[j] != null **
   *     queue[j] != null);
   */
  private void init(int numberOfUsers) {
    queue = new SimpleListNode[numberOfUsers
    ];
    locks = new ReentrantLock[numberOfUsers];

    for(int i = 0; i < queue.length; i++) {
      queue[i] = new SimpleListNode(null);
      /*@ pred inv = state(i); @*/
      locks[i] = new ReentrantLock/*@<inv>@*/
    ();
      //@ locks[i].commit();
    }
  }

  /**
   * Adds a PrintJob to the queue of the
   * specified user.
   * @param printJob The added print job.
   * @param id The ID of the user.
   * @requires printJob != null ** id >= 0 **
   *   id < queue.length ** queue[id] != null
   *   ** Perm(id, p) ** Perm(locks[id], p)
   *   ** Perm(printJob, p);
   * @ensures queue[id] != null **
   *   Perm(locks[id], p);
   */
  public void addToQueue(int id, String
    printJob) {
    locks[id].lock();
    queue[id].add(printJob);
    locks[id].unlock();
  }

  /**
   * Removes specific print job from queue.
   * @param printJob The removed print job.
   * @param id The id of the user.
   * @return The print job that was removed
   *   or <code>null</code> if not found.
   * @requires printJob != null ** id >= 0 **
   *   id < queue.length ** Perm(id, p) **
   *   Perm(locks[id], p) **
   *   Perm(printJob, p) **
   *   queue[id] != null;
   * @ensures Perm(locks[id], p) **
   *   queue[id] != null **
   *   Perm(removedElement, 100);
   */
  public String removeFromQueue(int id,
    String printJob) {
    String removedElement = null;
    locks[id].lock();
    removedElement = queue[id].remove(
    printJob);
    locks[id].unlock();
    return removedElement;
  }

  /**
   * Removes the first print job from queue.
   * @param id The id of the user.
   * @return The removed print job or <code>
   *   null</code> if the list was empty.
   * @requires id >= 0 **
   *   id < queue.length **
   *   Perm(locks[id], p) ** Perm(id, p) **
   *   queue[id] != null;
   * @ensures Perm(locks[id], p) **
   *   queue[id] != null **
   *   Perm(removedElement, 100);
   */
  public String removeFromQueue(int id) {
    String removedElement = null;
    locks[id].lock();
    removedElement = queue[id].remove();
    locks[id].unlock();
    return removedElement;
  }
```

```
  /**
   * Removes a user's list of print jobs from
   * the queue.
   * @param id The id of the user.
   * @requires id >= 0 **
   *   id < queue.length **
   *   Perm(locks[id], p);
   * @ensures Perm(locks[id], p);
   */
  public void removeUser(int id) {
    locks[id].lock();
    queue[id] = new SimpleListNode(null);
    locks[id].unlock();
  }

  /**
   * Retrieves a user's print jobs.
   * @param id The id of the user.
   * @return The print jobs of the user.
   * @requires Perm(id, p) **
   *   Perm(locks[id], p);
   * @ensures Perm(locks[id], p) **
   *   Perm(retrievedJobs, 100);
   */
  public String[] retrievePrintJobs(int id) {
    locks[id].lock();
    String[] retrievedJobs = (String[]) queue
    [id].retrieveData().toArray();
    locks[id].unlock();
    return retrievedJobs;
  }
}
```

1.2 The SimpleListNode class

Listing 2: The `SimpleListNode` class with specification

```
package server;

import java.util.ArrayList;

/**
 * A simple linked list.
 * @author Martijn Roo
 */
public class SimpleListNode {

  /*@
   * public resource nodeState() = Perm(data,
     100)**
   * Perm(next, 100)**((next != null) -* next
    .nodeState());
  @*/

  /*@
   * model int size() {
   *  int size = 1;
   *  SimpleListNode node = this;
   *  while (node.next != null) {
   *    size++;
   *  }
   *  return size;
   * }
  @*/

  /*@
   * model boolean containsLast(String
    elementData) {
   *  SimpleListNode last = this;
   *  while (last.next != null) {
   *    last = last.next;
   *  }
   *  return elementData.equals(last.data);
   * }
  @*/

  /*@
   * model int count(String elementData) {
   *  int count = 0;
   *  SimpleListNode node = this;
   *  while (node.next != null) {
   *    if(elementData.equals(node.data)) {
   *      count ++;
   *    }
   *    node = node.next;
   *  }
   *  return count;
   * }
  @*/

  private SimpleListNode next;
  private String data;

  /**
   * The constructor for this class.
   * @requires true;
   * @ensures Perm(next, 100) **
   *   Perm(data, 100);
   */
  private SimpleListNode() {}

  /**
   * Constructs a list node with the
   * specified data.
   * @param data The data.
   * @requires Perm(data, p);
   * @ensures Perm(next, 100) **
   *   Perm(data, 100) ** next == null;
   */
  public SimpleListNode(String data) {
    this();
    init(data);
  }

  /**
   * Initializes this node.
   * @param text The String in this node.
   * @requires Perm(next, 100) **
   *   Perm(data, 100) ** Perm(text, p);
   * @ensures Perm(next, 100) **
   *   Perm(data, 100) ** next == null;
   */
  public void init(String text) {
    next = null;
    data = text;
  }

  /**
   * Adds the specified element to the end of
   * this linked list.
   * @param elementData The added data.
   * @requires Perm(elementData, p) **
   *   elementData != null ** nodeState();
   * @ensures nodeState() **
   *   containsLast(elementData) **
   *   size() == \old(size()) + 1 **
   *   count(elementData) ==
   *   \old(count(elementData)+1);
   */
  public void add(String elementData) {
    if(next == null) {
      next = new SimpleListNode(elementData);
    } else {
      next.add(elementData);
    }
  }

  /**
   * Removes the first element of this list.
   * @return The removed element or <code>
   *   null</code> if the list was empty.
   * @requires nodeState();
   * @ensures nodeState() **
   *   ((\old(next)!=null) -*
   *   (next == \old(next.next))) **
   *   Perm(removedData, 100) **
```

```
   *    count(elementData) ==
   *    \old(count(elementData))-1;
   */
  public String remove() {
    String removedData = null;
    if (next != null) {
      removedData = next.data;
      next = next.next;
    }
    return removedData;
  }

  /**
   * Removes an element from this list.
   * @param elementData The removed data.
   * @return The removed data or <code>null
   *    </code> if not found.
   * @requires Perm(elementData, p) **
   *    elementData != null ** nodeState();
   * @ensures nodeState() **
   *    Perm(removedData, 100) **
   *    Perm(removedData, 100) **
   *    count(elementData) ==
   *    \old(count(elementData))-1;
   */
  public String remove(String elementData) {
    String removedData = null;
    if(next != null) {
      if(next.data.equals(elementData)) { //
    if next node is the one, remove it
        removedData = next.data;
        next = next.next;
      } else { // otherwise, call remove on
    the next node
        removedData = next.remove(elementData
    );
      }
    }
    return removedData;
  }

  /**
   * Retrieves the data of this node and all
   * nodes following this node.
   * @return The retrieved data.
   * @requires nodeState();
   * @ensures nodeState() **
   *    Perm(userJobs, 100) **
   *    this.equals(\old(this));
   */
  public ArrayList<String> retrieveData() {
    ArrayList<String> userJobs = new
    ArrayList<String>();
    if(next != null) {
      userJobs.add(data);
      userJobs.addAll(next.retrieveData());
      return userJobs;
    } else {
      userJobs.add(data);
      return userJobs;
    }
  }
}
```