272 – Homework Assignment 1

Fall 2008

Due: Wednesday, October 15th, 5:00PM

Do not discuss the problems with anyone other than the instructor or the TA.

1. Consider a simple class called `BufferControl` that keeps track of the number of items in a buffer, the number of items that have been inserted to the buffer, and the number of items that have been removed from the buffer. To simplify things we do not keep track of the items in the buffer in this class. `BufferControl` class has three integer fields: `numItems`, `numInserted`, and `numRemoved`. The constructor `BufferControl()` sets all three fields are equal to zero. `BufferControl` class has three methods: `void insert()`, `void remove()`, and `int getNumItems()`. `getNumItems()` returns the value of `numItems` and does not change the values of any fields. When `insert()` method is called `numItems` and `numInserted` are incremented by one and `numRemoved` remains the same. When `delete()` method can only be can only be called when `numItems` is greater than or equal to 1. When `delete()` method is called `numItems` is decremented by one and `numRemoved` is incremented by one and `numInserted` remains the same.

(a) Write the contract for the `BufferControl` class in **JML** by writing the pre and post-conditions for each method. Also write the (strongest) class invariant.

(b) Assume that there is another simple class called `BoundedBufferControl`. `BoundedBufferControl` class is very much like the `BufferControl` except that it has another integer field called `size`. The constructor `BoundedBufferControl(int s)` requires the input value `s` to be greater than or equal to 1, sets `size` to the input value `s` and initializes the rest of the variables to zero. The constructor `BoundedBufferControl()` sets `size` to a constant value (say 10) and initializes the rest of the variables to zero. The value of the variable `size` is not modified by any methods after construction. The behaviors of the methods `remove()` and `getNumItems()` for `BoundedBufferControl` are identical to those of corresponding methods in `BufferControl`. The method `insert()` can only be called when `numItems` is strictly less than `size`, otherwise it behaves exactly like the `insert()` method of the `BufferControl`.

Write the contract for the `BoundedBufferControl` class in **JML** by writing the the pre and post-conditions for each method. Also write the (strongest) class invariant.

(c) Consider the following two class structures: 1) `BufferControl` is superclass of `BoundedBufferControl`, 2) `BoundedBufferControl` is superclass of `BufferControl`. Based on the inheritance rules in design by contract explain whether these options would or would not work and why.

2. Write a FIFO circular queue in Java which stores a set of Objects in an array. Here are the *enqueue* and *dequeue* algorithms:

```
enqueue(Object x) {
 data[tail] = x;
 if (tail == data.length - 1)
   tail = 0;
 else
   tail = tail + 1;
}

Object dequeue() {
  Object result = data[head];
  if (head == data.length - 1)
    head = 0;
  else
    head = head + 1;
  return result;
}
```

You should implement the following methods: `enqueeu(Object)`, `Object dequeue()`, `int numItems()`, `int size()`, `boolean full()`, `boolean empty()`.

Write the contract for this class (including preconditions, postconditions and class invariant) in **JML**. Write a driver for this class (that generates some test inputs) and use **jmlc** and **jmlrac** to check for contract violations. Insert errors into to the code and generate following contract violations 1) precondition violation, 2) postcondition violation, and 3) class invariant violation. Turn in the source code with the contract specifications and the output messages from the contract violations.

3. Write a binary search tree that stores integers(>0 only). Here are the *insert* and *find* algorithms:

```
insert(int x) {
  temp=root;
  while(temp <> null) {
    if(temp.value > n) {
      if(temp->left == null) {
        temp->left=new Node;
        temp->left.value=x;
      } else {
        temp=temp -> left;
      }
    }
    if(temp.value < n) {
      if(temp->right == null) {
        temp->right=new Node;
        temp->right.value=x;
```

```
      } else {
        temp=temp->right;
      }
    }
  }
}

find(int x) {
  Node temp = root;
  while(temp != null) {
    if(temp.value == x) {
      return temp;
    } else if(temp.val < n) {
      temp=temp->right;
    } else {
      temp=temp->left;
    }
  }
  return None;
}
```

Implement a binary search tree class with the following methods: `insert(int)`, `Node find()`, `int size()` in Java (to simplify things we are not asking for the implementation of the remove method). In addition note the following conditions:

- The class constructor should initialize the root Node.
- The class should only hold integers greater than zero.
- No duplicate items should exist in the class (Hint: Can you specify this in the contract for one of the methods?).

Write the contract for this class (including preconditions, postconditions and class invariant) based on the **jContractor** conventions. Write a driver for this class (that generates some test inputs) and use **jContractor** to check for contract violations. Insert errors into to the code and generate following contract violations 1) precondition violation, 2) postcondition violation, and 3) class invariant violation. Turn in the Java source code and the contract and the output messages from the contract violations.

Submission Instructions: Solution to Problem 1 can be submitted electronically or on paper. Solutions to Problems 2 and 3 should be submitted electronically via email to puneet@cs.ucsb.edu with [CS272-HW1-YourLastName] as the subject line. For problem 2 your submission should include the Queue class, any helper classes and the driver class and the output messages should be attached as a separate text file. For problem 3 your submission should include the BinarySearchTree class, any helper classes and the driver class and the output messages should be attached as a separate text file.

Running JContractor and JML Common Tools (jmlc-unix and jmlrac-unix) on CSIL Machines:

Make sure you are using the bash shell as your shell. Edit the file .bashrc in your home directory (/cs/student/<userid>) and add the following lines:

- export CLASSPATH=$CLASSPATH:/cs/class/cs272/cs272/jcontractor-0.1/jcontractor.jar
- export PATH=$PATH:/cs/class/cs272/cs272/JML/bin

Log back in (or restart the terminal) once you have done the above steps. You should now have jContractor available in your classpath and you can run jContractor in either mode (using jContractor or jInstrument) as described in the tutorial.

Also you should now have jmlc-unix and jmlrac-unix available on your path, thus allowing you to run these tools from anywhere for JML runtime assertion checking.