Prof. Dr. A. Poetzsch-Heffter
Dipl.-Inform. Kathrin Geilmann

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2010)

Practice Sheet 8 (Hints and Comments)

Exercise 1 Introduction to JML

a)
- Advantages
  - Formal Specifications are precise, therefor they avoid the ambiguities of natural languages.
  - Formal Specifications can be executable. This allows a (semi-) automatical verification (→ theoreme provers, tools like boogie, etc.).
  - Enforces "Think before you code". The developer has to think about the precise meaning of the interfaces he uses, this allows to detect errors and problem quite early in the development process.
- Disadvantage
  - Formal Specifications are more difficult to write than code, because they need more knowledge about mathematics, abstraction, etc.
  - Another language to learn.
  - Formal specifications can be longer than descriptions in natural languages or than the specified source.
  - Every method is written twice, once in den specification language and once in code.
  - Problem: How to decide, that a specification is complete/precise enough?

b) -

c) -

d)

```
class Behaelter {

  /*@ non_null @*/                          // part of the class invariant, otherwise extractMin fails.
  int[]  a;
  int n;

  //@invariant  0 <= n && n <= a.length;     // n is a valid index in a[]

  /*@ requires        input != null;
   @  assignable      n, a;
   @  ensures         n == input.length &&
   @                  (\forall int i;  0 <= i && i < input.length;  a[i] == input[i]) &&
   @                  a.length == input.length;
   @
    */
  Behaelter( int[] input ){
      n = input.length;
      a = new int[n];
      System.arraycopy(input, 0, a, 0, n);
  }

  /*@ requires        0 < n;
    @ assignable      n, a;
    @ ensures         \result == (\min int i; 0 <= i && i < (n); \old(a[i]));
    */
  int extractMin() {
      int m = Integer.MAX_VALUE;
      int mindex = 0;
      /*@ maintaining m >= (\min int j; 0 <= j && j < a.length; a[j]);
      */
       for (int i = 0; i < n; i++) {
          if (a[i] < m) {
              mindex = i;
              m = a[i];
          }
      }
      n--;
      a[mindex] = a[n];
     return m;
  }

}
```

e) Add `n < old(n)` to the postcondition of `extractMin`.

f) Translate the documentation of the class ByteArrayInputStream directly into jml.

```
public class ByteArrayInputStream extends InputStream {
  protected /*@ spec_public non_null @*/ byte[] buf;
  protected /*@ spec_public @*/          int    count;
  protected /*@ spec_public @*/          int    mark;
  protected /*@ spec_public @*/          int    pos;

  /*@ public invariant (count>=0 && count<=buf.length) &&
    @                  (mark>=0  && mark<=count) &&
    @                  (pos>=0   && pos<=count);
    @*/

  /*@ requires   b!=null;
    @ assignable buf, pos, count, mark;
    @ ensures    buf==b &&
    @            pos==0 &&
    @            count==b.length &&
    @            mark==0;
    @*/
  public ByteArrayInputStream(byte[] b);

  /*@ requires   b!=null &&
    @            offset>=0 &&
    @            length>=0 &&
    @            (offset+length)<b.length;
    @ assignable buf, pos, count, mark;
    @ ensures    buf==b &&
    @            pos==offset &&
    @            count==(offset+length<buf.length) ? offset+length:buf.length &&
    @            mark==offset;
    @*/
  public ByteArrayInputStream(byte[] b,
                              int offset,
                              int length);

  /*@ requires true;
    @ ensures  \result==(count-pos);
    @*/
  public /*@ pure @*/ int available();

  /*@ requires true;
    @ ensures  true;
    @*/
  public void /*@ pure @*/ close() throws IOException;

  /*@ requires   true;
    @ assignable mark;
    @ ensures    mark==pos;
    @*/
  public void mark(int readAheadLimit);

  /*@ requires   true;
    @ ensures    \result==true;
    @*/
  public boolean /*@ pure @*/ markSupported();

  /*@ requires   true;
    @ assignable pos;
    @ ensures    available()==0 => (\result==-1 && pos==\old(pos)) &&
    @            available()>0  => ((\result==buf[pos] && pos==\old(pos)+1) &&
    @                               \result>=0 && \result<=255);
    @*/
  public int read();

  /*@ requires   b!=null &&
    @            off>=0 &&
    @            len>=0 &&
    @            (off+len)<b.length;
    @ assignable pos;
    @ ensures    available()==0 => (\result==-1 && pos==\old(pos)) &&
    @            available()>0  => ((\result==(len<available()) ? len:available() &&
    @                               pos==\old(pos)+\result) &&
    @                               (\forall int k; 0<k && k<\result; buf[off+k]==b[off+k]));
    @*/
  public int read(byte[] b, int off, int len);

  /*@ requires   true;
    @ assignable pos;
    @ ensures    pos==mark;
    @*/
  public void reset();

  /*@ requires   n>=0;
    @ assignable pos;
    @ ensures    \result==(available()<n) ? available():n &&
    @            pos==\old(pos)+\result;
    @*/
  public long skip(long n);
}
```

Exercise 2 Abstraction

a) Model fields form a model state, that can be different from the implemented state. The model state may be better suited for specifying the behavior of an object, because it can be more abstract than the real state. For interfaces this is the only possibility to express something about the state of the objects that implement the interfaces. The model fields (model state) has to be related by the specification to the real state.

b) Note: the following definition does not make any guarantees about FIFO - order of the queue. To guarantee FIFO, choose another type for the model field, e.g. a sequence.

```
//@ model import org.jmlspecs.models.*;

public interface Queue {
  //@ public model JMLObjectBag elements;

  /*@
    @ public normal_behavior
    @   requires !isEmpty();
    @   ensures elements.has(\result);
    @
    @ also
    @
    @ public exceptional_behavior
    @   signals(EmptyQueueException) isEmpty();
    @*/
  /*@ pure @*/ Object peek() throws EmptyQueueException;

  /*@
    @ public normal_behavior
    @   requires !isEmpty();
    @   assignable elements;
    @   ensures elements.equals(((JMLObjectBag)\old(elements)).remove(\result)) &&
    @           \result.equals(\old(peek())) &&
    @           size()==\old(size())-1;
    @
    @ also
    @
    @ public exceptional_behavior
    @   assignable \nothing;
    @   signals(EmptyQueueException) isEmpty();
    @*/
  Object dequeue() throws EmptyQueueException;

  /*@   requires item != null;
    @   assignable elements;
    @   ensures elements.equals(((JMLObjectBag)\old(elements)).add(item)) &&
    @           size()==\old(size())+1;
    @*/
  void enqueue(Object item);

  /*@
    @   ensures \result==elements.isEmpty();
    @*/
  /*@ pure @*/ boolean isEmpty();

  /*@
    @   ensures \result==elements.size();
    @*/
  /*@ pure @*/ int size();
}

class EmptyQueueException extends Exception {}
```

c) The method implementation are straightforward, mainly delegate the calls to e.

```
public class Queue {
        //@ public model JMLObjectBag elements;

        private LinkedList<Object> e = new LinkedList<Object>();

        //@ private depends state <- e;
        //@ private represents elements <- JMLObjectBag.convertFrom(e);
        ...
}
```