Prof. Dr. A. Poetzsch-Heffter
Mathias Weber, M.Sc.

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2015)

Practice Sheet 8 (Hints and Comments)

Exercise 1 Abstraction

a) Model fields form a model state, which 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)
```
//@ model import org.jmlspecs.models.*;

public interface Queue {
  //@ public instance model boolean canRead;
  //@ public instance model boolean canWrite;

  //@ public model instance JMLObjectSequence elements;
  //@ initially elements != null && elements.isEmpty();

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

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

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

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

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

class EmptyQueueException extends Exception {}
```

c) The method implementation are straightforward, mainly delegate the calls to `e`. To relate specification and implementation use depends- and represents-clauses.

```
public abstract class Queue {

  private LinkedList<Object> e = new LinkedList<Object>();
  //@ private depends elements <- e
  //@ private represents elements <- JMLObjectSequence.convertFrom(e)

  //@ public model JMLObjectSequence elements;
  ...
}
```

d) Interfaces can not have fields in Java. This is the reason why model fields can not directly be modeled as fields in all cases. Each implementation (class) would have to implement the model field and its representation as a field. The representation gives the expression that has to be executed whenever a component of the representation is changed.

Exercise 2 Behavioral Subtyping I

a) JML specifications are inherited by subclasses and classes implementing interfaces. A class inherits the visible invariants of its superclasses (-interfaces). See JML Reference Manual 8.2.4

b) *A method-specification of a method in a class or interface must start with the keyword also if (and only if) this method is already declared in the parent type that the current type extends, in one of the interfaces the class implements, or in a previous file of the refinement sequence for this type. Starting a method-specification with the keyword also is intended to tell the reader that this specification is in addition to some specifications of the method that are given in the superclass of the class, one of the interfaces it implements, or in another file in the refinement sequence.* JML Reference Manual 9.2 and in the paper "Design by Contract with JML" by Gary T. Leavens and Yoonsik Cheon.

c) Use the rules to construct the pre- and postconditions for subclasses.

```
public class Child extends Parent {
    //@ requires    i >= 0 || i <= 0
    //@ ensures     (\old(i>=0) => \result >= i)
                 && (\old(i<=0) => \result <= i);
    int m(int i){ ... }
}
```

A call to Child.m with `i = 0`, means that both parts of the precondition are fulfilled and therefor both parts of the postcondition have to be fulfilled too. As i is not assignable, pre- and post-values of i are the same and we get as only possible result 0.

d)
- Class A + Class B: ok
- Class C + Class D: No behavioural subtyping. The complete precondition of D.set() is `a > 0 || a > 10`, and the complete postcondition of D.get() is `(true => result > 10) && (true => result > 0)`.

  ```
  D d = new D();
  d. set (5); // ok
  ... d.get();    // not ok, because of the conjunction.
  ```

  The invariant changes nothing.
- Class E + Class F: No behavioural subtyping, due to a possible overflow in F.increment(). The overflow breaks the part of the postcondition, that is inherited from E.increment().

Exercise 3 Behavioral Subtyping II

```
//@ model import org.jmlspecs.models.*;
//@ model import java.util.Arrays;

public class ArrayQueue implements Queue{
  private int used = 0; //@ in canRead; in canWrite;
  private int max = 10; //@ in canRead; in canWrite;
  //@ private invariant 0 <= used && used <= max;
```

```
  private Object[] a = new Object[max]; //@ in elements;

  ...

  public boolean isEmpty() {
    return used == 0;
  }

  public int size() {
    return used;
  }


  //@ private represents canRead <- true;
  //@ private represents canWrite <- used < max;
  //@ private represents elements <-
  //@          JMLObjectSequence.convertFrom(Arrays.copyOfRange(a, 0, used));
}
```