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 2013)

Practice Sheet 9 (Hints and Comments)

Exercise 1 Behavioral Subtyping

a) At the begin of method `getChar()` the state is valid (since this is the precondition of this method). This means that `0 <= lo && lo <= cur && cur <= hi`. Since we are going to increment cur, we have to make sure that `cur = hi`!. If `cur == hi` we call `refill()` which may change `lo` and `hi` since it may change `state` and `state` depends on `lo` and `hi`. Method `refill()` may not change `valid`, so we know that `cur` is between `lo` and `hi` after returning from `refill`. If still `cur == hi` we can not proceed and return `-1`. Before incrementing `cur` we therefore know that `0 <= lo && lo <= cur && cur < hi`. After incrementing `cur` we have `0 <= lo && lo < cur && cur <= hi`. We access `buff` at index `cur-lo-1` which is `>= 0`. And since `hi-lo <= buff.length` we know that `cur-lo-1 < buff.length`.

b) File `StringReader.java`:

```
public interface StringReader extends Reader {

    /*@ public normal_behavior
      @   requires    s != null;
      @   assignable  valid, state;
      @   ensures     valid && \result == this;
      @*/
    public StringReader init(String s);
}
```

File `StringReaderImpl.java`:

```
public class StringReaderImpl extends BuffReader implements StringReader {
    private String str;
    //@ private represents svalid <- hi <= str.length();

    public StringReader init(String s) {
        str = s;
        buff = str.toCharArray();
        lo = 0;
        cur = 0;
        hi = buff.length;
        return this;
    }

    public void refill() {
        lo = cur;
        hi = Math.min(lo+buff.length, str.length());
    }

    public void close(){ }

    //@ private depends state  <- str;
    //@ private depends svalid <- str;
}
```

c) In file `BuffReader.java`:

```
/*@ public normal_behavior
  @   requires    valid;
  @   requires    0 <= c && c <= 65535;
  @   requires    lo < cur;
  @   assignable  state;
  @   ensures     getChar() == c;
  @*/
public void unread(int c){
    buff[cur] = (char)c;
    cur--;
}
```

For the unread operation to work, at least one character has to be read from the reader. The information about the inner workings of the buffered reader is not visible in `Reader`. Moving the method to the `Reader` interface and abstracting from the state would make the implementation much more complex.

Exercise 2 Concurrent Access to Shared State

The read and write access to the field `stopRequested` is atomic, which means the field can only have the value **`true`** or **`false`**. This does not guarantee that there exists only a single instance of the field for all threads of the program. In the Java memory model, every thread has its own instance of this field. Synchronization of the value of such a field is only forced by entering a synchronized block or by writing to a volatile field. Since the example program neither uses synchronized blocks nor volatile fields, the value the main threads sets is never synchronized with the background thread.

The example can simply be fixed by declaring the field `stopRequested` as volatile.