Prof. Dr. Peter Thiemann
Stephan Arlt
Martin Wehrle
Sommersemester 2011

Softwaretechnik

http://proglang.informatik.uni-freiburg.de/teaching/swt/2011/

Exercise Sheet 5

Exercise 1 (6 points)

The following Java class shows an implementation of queues in Java.

```
public class Queue {

  protected int in,out;
  protected Object[] buf;

  public Queue (int anzahl) {
    buf = new Object[anzahl];
  }

  public boolean empty() {
    return in - out == 0;
  }

  public boolean full() {
    return in - out == buf.length;
  }

  public void enqueue(Object o) {
    buf[in % buf.length] = o;
    in++;
  }

  public Object dequeue() {
    Object o = buf[out % buf.length];
    out++;
    return o;
  }
}
```

(i) Give reasonable pre- and postconditions for all methods and the constructor of the Queue class. In particular, keep in mind that integers may overflow.

(ii) A *weak class invariant* is defined as a condition that holds between calls to methods of the class, but not during the execution of such methods. Are there any weak class invariants for the Queue class?

Exercise 2 (6 points)

Consider the following Java class *IntegerInterval* that represents an interval of integer values.

```
class IntegerInterval {
  int getLowerBound() { ... }
  int getUpperBound() { ... }
  void doSomething (int i) { ... }
}
```

The methods of the class *IntegerInterval* have the following specifications:

- getLowerBound(): **requires:** *true*; **ensures:** $0 <= getLowerBound() < getUpperBound()$
- getUpperBound(): **requires:** *true*; **ensures:** $0 <= getLowerBound() < getUpperBound()$
- doSomething (int i): **requires:** $getLowerBound() <= i < getUpperBound()$; **ensures:** *true*;

Consider the class *Run* that uses the *IntegerInterval* class as follows. The main method of *Run* has the specification **requires:** *true*; **ensures:** *true*;

```
class Run {

  public static void main (String[] a) {

    int i = ...
    IntegerIntervall c = ...

    if (i >= 0 && i <= 10) {
     c.doSomething(c.getLowerBound()+(c.getUpperBound()-c.getLowerBound())*i/10);
    }

  }
}
```

Analyze the code and identify contract violations that may occur during run-time.

Exercise 3 (6 points)

In the previous exercises, we have examined the specification of programs using pre- and postconditions. In this exercise, we will use Pex, a tool from Microsoft Research that creates a set of intelligent test cases. We will see that it is usually harder to understand the semantics of a program if a set of test cases is given instead of a specification.

Familiarize yourself with Pex4Fun at `http://www.pexforfun.com/`. Provide code that matches a secret implementation. Test your solution by asking Pex. Pex either returns true if your solution is correct, or provides a counter-example for parameters for which your solution fails.

1. Provide code that matches the implementation of *Puzzle* at `http://goo.gl/t5SPC`. What does *Puzzle* compute? *Hint:* Consider the triangle example discussed in the lecture.

2. Provide code that matches the implementation of *Puzzle* at `http://goo.gl/SZVZS`. What does *Puzzle* compute?