Assignment 5: SCOOP principles

ETH Zurich

1 Interpreting a SCOOP program

1.1 Background

The code in listing 1 shows the participants of a crazy office. Note that the \textit{BOSS} class is the root of this system.

Listing 1: crazy office classes

``` SCOOP
class BOSS
create
make

feature
  evil_supervisor: separate EVIL_SUPERVISOR
  nice_supervisor: separate NICE_SUPERVISOR
  worker: separate WORKER

make
  -- Create supervisors and a worker and use the supervisors to drive the worker.
  do
    create evil_supervisor
    create nice_supervisor
    create worker
    print ("boss: I am about to ask the supervisors to do their job.")
    run (evil_supervisor, nice_supervisor)
    print ("boss: I am done.")
  end

run (a_evil_supervisor: separate EVIL_SUPERVISOR; a_nice_supervisor: separate NICE_SUPERVISOR)
  -- Use the supervisors to drive the worker.
  do
    a_evil_supervisor . convince (worker)
    a_nice_supervisor . convince (worker)
    a_evil_supervisor . convince (worker)
    a_nice_supervisor . convince (worker)

    if (a_evil_supervisor . done and a_nice_supervisor . done) then
      print ("boss: The supervisors are done.")
    end
  end
end
```
class EVIL_SUPERVISOR

feature
done: BOOLEAN
   -- Did I convince a worker?

convince (a_worker: separate WORKER)
   -- Convince 'a_worker' that he is not done as soon as he thinks that he is done.
   require
      a_worker.done
   do
      a_worker.be_not_done
      done := true
      print ("evil supervisor: I am done.")
   end

end

class NICE_SUPERVISOR

feature
done: BOOLEAN
   -- Did I convince a worker?

convince (a_worker: separate WORKER)
   -- Convince 'a_worker' that he is done as soon as he thinks that he is not done.
   require
      not a_worker.done
   do
      a_worker.be_done
      done := true
      print ("nice supervisor: I am done.")
   end

end

class WORKER

create
make

feature
make
   -- Create the worker and make him not done.
   do
      done := false
   ensure
      not done: not done
   end

done: BOOLEAN
   -- Do I think that I am done with my task?
1.2 Task

Write down one possible output of the program. Does this system terminate (i.e. all processors finish their tasks)?

2 Breakfast Running Time

2.1 Background

Reasoning about the execution times of a concurrent SCOOP program, in the context of breakfast.

2.2 Task

Consider the following SCOOP program being executed on a processor z:

`be_not_done
   -- Make me realize that I am not done.
   do
      print("worker: I am not done.")
      done := false
   end

be_done
   -- Make me realize that I am done.
   do
      print("worker: I am done.")
      done := true
   end
end`

The object-processor associations are given as follows: `pan` is handled by processor p, `bread` and `toaster` by processor q, and `meal` by processor r. The call `bread.cut` takes 20 time units until it returns, `toaster.toast` 30 time units, `pan.fry` 20 time units, `meal.compose` 40 time units, `meal.eat` 20 time units. Assume the queries are instantaneous. What is the minimum time for execution of this program? Justify your answer.

3 Baboon Crossing

3.1 Background

This task is adapted from Downey [1] and Tanenbaum [2]. There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope
is only strong enough to hold \( n \) baboons. If there are more baboons on the rope at the same time, it will break.

### 3.2 Task

Design and implement a SCOOP synchronization scheme with the following properties:

- Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
- There are never more than \( n \) baboons on the rope.
- A continuing stream of baboons crossing in one direction should not bar baboons going the other way indefinitely (no starvation).

### References
