

# Classroom 3

ETH Zurich

Date: 31 January 2006

The classroom exercise intends to help you self-evaluate your knowledge and skills and lets us gain knowledge about the performance of our students. The setup resembles the situation you will encounter during the fall exam. The assistants will be happy to clarify any problems with the formulation of the tasks, but will not solve the tasks for you. This exercise will be corrected and graded by your assistant; the grade will not have any influence on the fall exam or the testate.

In this paper, the number of empty lines reserved for your answers is not a hint on the number of lines that you should fill in.

**Please solve this exercise alone.**

## 1 Inheritance (12 points)

### Goal

In this task you will have to derive descendants using inheritance. Have a look at the classes WAGON and CARRIAGE (see listings 1 and 2) representing train wagons (Eisenbahnwagen) and train carriages (Personenwagen), respectively.

Listing 1: Class WAGON

```
deferred class
2  WAGON

4 feature -- All features

6  id: INTEGER
    -- Wagon identification number

8
   capacity: INTEGER is
10  -- Maximal number of passengers
    deferred
12  end

14  passenger_count: INTEGER
    -- Actual number of passengers

16
   set_passenger_count (n: INTEGER) is
18  -- Set 'passenger_count' to 'n'.
    require
```

```
20   valid_passenger_count: n >= 0 and n <= capacity
    do
22     passenger_count := n
    ensure
24     passenger_count_set: passenger_count = n
    end
26
invariant
28   positive_passenger_count: passenger_count >= 0
    not_more_passengers_than_capacity: passenger_count <= capacity
30 end
```

Listing 2: Class *CARRIAGE*

```
class
2  CARRIAGE

4  inherit
   WAGON

6
create
8  make

10 feature -- All features

12  make (n: INTEGER) is
    -- Initialize carriage with 'n' seats.
14    require
        positive_number: n > 0
16    do
        seat_count := n
18    ensure
        seat_count_set: seat_count = n
20    end

22  seat_count: INTEGER
    -- Number of seats

24  capacity: INTEGER is
26    -- Maximal number of passengers
    do
28    Result := 2 * seat_count
    ensure then
30    definition: Result = 2 * seat_count
    end
32
end
```

## 1.1 Sleepers

Derive a descendant SLEEPER representing train sleepers (Schlafwagen). Make sure that...

- each sleeper stores the number of beds it contains
- the capacity of a sleeper is equal to the number of its beds

Write your derived class SLEEPER below. Do not forget to define appropriate preconditions, postconditions, and invariants.

```
class
2  SLEEPER
4 .....
6 .....
8 .....
10 .....
12 .....
14 .....
16 .....
18 .....
20 .....
22 .....
24 .....
26 .....
28 .....
30 .....
32 .....
34 .....
36 .....
38 .....
40 .....
42 .....
44 .....
46 .....
48 .....
50 .....
52 .....
54 .....
```

```
56 .....  
58 .....  
60 .....  
62 .....  
64 .....  
66 .....  
68 .....  
70 .....  
72 .....  
end
```

## Solution

```
class  
2  SLEEPER  
  
4  inherit  
    WAGON  
6  
7  create  
8  make  
  
10 feature -- All features  
  
12  make (n: INTEGER) is  
    -- Initialize sleeper with 'n' beds.  
14    require  
        positive_number: n > 0  
16    do  
        bed_count := n  
18    ensure  
        bed_count_set: bed_count = n  
20    end  
  
22  bed_count: INTEGER  
    -- Number of beds  
24  
25  capacity: INTEGER is  
    -- Maximal number of passengers  
26    do  
27        Result := bed_count  
28    ensure then  
30        capacity_equals_to_bed_count: capacity = bed_count  
    end  
32  
33 invariant  
34  one_bed_capacity: capacity = bed_count  
end
```

## 1.2 Diners

Assume you have in addition to the classes WAGON and CARRIAGE (see listings 1 and 2) a class RESTAURANT representing restaurants. Have a look at that class now (see listing 3).

Listing 3: Class *RESTAURANT*

```
deferred class
2  RESTAURANT

4 feature -- All features

6  make (a_table_count: INTEGER; a_menu: STRING) is
   -- Initialize restaurant 'a_table_count' as tables
   -- and 'a_menu' as menu.
   require
10   positive_a_table_count : a_table_count > 0
   a_menu_not_void: a_menu /= Void
12  do
   table_count := a_table_count
14   menu := a_menu
   ensure
16   table_count_set : table_count = a_table_count
   menu_set: menu = a_menu
18  end

20  menu: STRING
   -- Daily menu

22
24  set_menu (new_menu: STRING) is
   -- Set menu to 'new_menu'.
   require
26   new_menu_not_void: new_menu /= Void
   do
28   menu := new_menu
   ensure
30   menu_set: menu = new_menu
   end

32
34  guest_count: INTEGER is
   -- Actual number of guests
   deferred
36  end

38  table_count: INTEGER
   -- Number of tables

40
42  invariant
   guest_count_positive : guest_count >= 0
44   table_count_greater_zero : table_count > 0
   menu_not_void: menu /= Void
46
end
```

Inheriting from any of the given classes WAGON, CARRIAGE and/or RESTAURANT (see listings 1, 2, and 3) now implement a new class DINER representing

train diners (Speisewagen). Make sure that...

- DINER provides a creation procedure to initialize the number of tables
- only four passengers can sit at a table
- DINER stores the number of its guests within the attribute `passenger_count`
- DINER has an appropriate contract

Write your derived class `DINER` below. You can use feature renaming and redefinition if necessary.

```
class
2  DINER
4 .....
6 .....
8 .....
10 .....
12 .....
14 .....
16 .....
18 .....
20 .....
22 .....
24 .....
26 .....
28 .....
30 .....
32 .....
34 .....
36 .....
38 .....
40 .....
42 .....
44 .....
```

```
46 .....  
48 .....  
50 .....  
52 .....  
54 .....  
56 .....  
end
```

## Solution

```
class  
2  DINER  
  
4  inherit  
   WAGON  
6  RESTAURANT  
   rename  
8   guest_count as passenger_count  
   end  
10  
12 create  
   make  
  
14 feature -- All features  
  
16 capacity: INTEGER is  
   -- Maximal number of passengers  
18 do  
   Result := 4 * table_count  
20 ensure then  
   capacity_four_times_table_count: capacity = 4 * table_count  
22 end  
  
24 invariant  
   four_times_table_capacity: capacity = 4 * table_count  
26 end
```

## 2 Loops (12 points)

Below you will find three different functions *binary\_search\_1*, *binary\_search\_2*, and *binary\_search\_3* all with the same signature. Every function is implementing the so called **binary search**: consider an array *an\_array* (first formal argument of the functions) of integers assumed to be in increasing order and indexed from **1** to **n** (see Figure 1);

**Binary search** is a way to decide whether a certain integer value *x* (second formal argument of the functions) appears in the array *an\_array*:

0	2	4	7	8	17	21
1	2	3	...			$n$

Figure 1: Sorted array

- if the array has one element (the precondition of the function guarantees that the array *an\_array* has at least one element), the answer is yes if and only if that element has value  $x$ ;
- otherwise compare  $x$  to the element at the array's middle point, and repeat on the lower or higher half depending on whether that element is greater or less than  $x$ .

Decide for each of the four functions,

1. if the algorithm of the function is correct
2. in case the algorithm is not correct find **one case** in which it will not work properly and explain in detail the problem (using the case which caused the problem). There might be more than one case which raises a problem, but it is enough to show here just **one case!**

Hint:

- If an algorithm is correct, you do not need to give an explanation.
- *an\_array @ m* denotes the element at index  $m$  in array *an\_array*. Note that the infix feature *@* has a precondition stating that  $m$  must be a valid index. In our case  $m$  is allowed to have the values **from 1 (not 0) to n (an\_array.count)!**
- The *//* operator denotes integer division, for example  $7 // 2$  and  $6 // 2$  have value 3.

## 2.1 Version 1:

```
binary_search_1 (an_array: ARRAY [INTEGER]; x: INTEGER): BOOLEAN is
2  -- Search 'x' in 'an_array' using binary search algorithm. Version 1;
  -- Elements of 'an_array' are in increasing order.
4  require
   an_array_count_positive: an_array.count > 0
6  local
   i: INTEGER
8   j: INTEGER
   m: INTEGER
10 do
   from
12   i := 1
   j := an_array.count
14  until
   i = j
16  loop
   m := (i + j) // 2
18   if an_array @ m <= x then
   i := m
20  else
   j := m
22  end
  end
24  Result := (x = an_array @ i)
end
```

### Explanation

The function *binary\_search\_1* is not correct. The problem with function *binary\_search\_1* is that the exit condition of the loop ( $i = j$ ) might never evaluate to true, resulting therefore in an infinite loop with the consequence that the function will never terminate.

Consider e.g. an array [2, 5] consisting of two integer elements 2 and 5 and we try to find the value 2: in the first iteration of the loop  $m = (i + j) // 2 = (1 + 2) // 2 = 1$ . Then in the if-statement  $i$  gets the value of  $m$  which is 1 and since  $i$  is not equal to  $j$  (exit condition of the loop) the loop is executed again and  $m$  is evaluated to  $m = (i + j) // 2 = (1 + 2) // 2 = 1$ . As the values of  $i$  and  $j$  are not changing anymore, once can see that the exit condition will never be fulfilled and hence we have an infinite loop.

## 2.2 Version 2:

```
binary_search_2 (an_array: ARRAY [INTEGER]; x: INTEGER): BOOLEAN is
2  -- Search 'x' in 'an_array' using binary search algorithm. Version 2;
  -- Elements of 'an_array' are in increasing order.
4  require
   an_array_count_positive: an_array.count > 0
6  local
   i: INTEGER
8   j: INTEGER
```

```

10   m: INTEGER
    found: BOOLEAN
11 do
12   from
    i := 1
14   j := an_array.count
    until
16   i = j and not found
    loop
18   m := (i + j) // 2
    if an_array @ m < x then
20     i := m + 1
    elseif an_array @ m = x then
22     found := true
    else
24     j := m - 1
    end
26 end
    Result := found
28 end
  
```

### Explanation

The function *binary\_search\_2* is not correct. One problem with function *binary\_search\_2* is that the exit condition of the loop (*i = j and not found*) might never evaluate to true, resulting therefore in an infinite loop with the consequence that the function will never terminate.

Consider again e.g. an array [2, 5] consisting of two integer elements 2 and 5 and we try to find the value 2: in the first iteration of the loop  $m = (i + j) // 2 = (1 + 2) // 2 = 1$ . Then the boolean entity *found* is correctly set to true in the if-statement. As the exit condition of the loop is not fulfilled the loop is executed once again and the value of *m* is calculated:  $m = (i + j) // 2 = (1 + 2) // 2 = 1$ . Since the value of *m* is from now on always the same, the exit condition will never be fulfilled and therefore we end with an infinite loop.

### 2.3 Version 3:

```

binary_search_3 (an_array: ARRAY [INTEGER]; x: INTEGER): BOOLEAN is
2   -- Search 'x' in 'an_array' using binary search algorithm. Version 3;
   -- Elements of an_array are in increasing order.
4 require
    an_array_count_positive: an_array.count > 0
6 local
    i: INTEGER
    j: INTEGER
    m: INTEGER
10 do
    from
12   i := 0
    j := an_array.count
14   until
    i = j
  
```

```

16 loop
    m := (i + j + 1) // 2
18 if an_array @ m <= x then
    i := m + 1
20 else
    j := m
22 end
end
24 if i >= 1 and i <= an_array.count then
    Result := (x = an_array @ i)
26 else
    Result := False
28 end
end
    
```

### Explanation

The function *binary\_search\_3* is not correct.

Consider e.g. an array [2] consisting of only one integer element 2 and we try to find the value 2: in the first iteration of the loop  $m = (i + j + 1) // 2 = (0 + 1 + 1) // 2 = 1$ . Then the if-statement is executed and  $i$  is updated to  $i = m + 1 = 1 + 1 = 2$ . Since the exit condition of the loop is not fulfilled the loop will be executed again and  $m$  is updated to  $m = (i + j + 1) // 2 = (2 + 1 + 1) // 2 = 2$ . Applying the feature call *an\_array @ m* with  $m$  being 2 will lead to a precondition violation, since 2 is not a valid index. Valid index would be a value between 1 and *an\_array.count* (here 1).

## 3 Recursion (10 points)

Consider the following classes *SINGLE\_LINKED\_LIST* [G] and *SINGLE\_CELL* [G] implementing a single linked list. The head of the list (first element of the list) is stored in the attribute *first* of the class *SINGLE\_LINKED\_LIST* [G]. Attribute *next* of class *SINGLE\_CELL* [G] delivers the next cell (instance of the class *SINGLE\_CELL* [G]). Calling *next* on the last cell (instance of the class *SINGLE\_CELL* [G]) will return a *Void* reference.

Implement the feature *invert* of class *SINGLE\_LINKED\_LIST* [G] **using recursion**, so that it inverts the order of the elements in the list. If we have e.g. the list [6, 2, 8, 5] (with 6 being the first element of the list and 5 the last element) inverting it should result in [5, 8, 2, 6]. You are allowed to introduce a new procedure to class *SINGLE\_LINKED\_LIST* [G] that you call from feature *invert*, but you are not allowed to create any objects of type *SINGLE\_CELL* [G] or *SINGLE\_LINKED\_LIST* [G]. Note that the use of loop constructs is disallowed in this task.

```

class
2 SINGLE_LINKED_LIST [G]
4 feature -- Access
    
```

```
6  first : SINGLE_CELL [G]
      -- Head element of the list, 'Void' if the list is empty
8
feature -- Basic operations
10
    invert is
12      -- Invert the order of the elements of the list .
      -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6].
14      local
16      .....
18      do
20      .....
22      .....
24      .....
26      .....
28      .....
30      .....
32      .....
34      end
36      .....
38      .....
40      .....
42      .....
44      .....
46      .....
48      .....
50      .....
52      .....
54      .....
56      .....
58      .....
60      .....
62      .....
```

```
64 .....
66 .....
68 .....
70 .....
72 .....
74 .....
76 .....
78 .....
80 .....
82 .....
84 .....
86 .....
88 .....
90 .....
  end
92
  class
94   SINGLE_CELL [G]

96   feature -- Access

98     next: SINGLE_CELL [G]
        -- Reference to the next generic list cell of a list
100
102   feature -- Element change

104     set_next (an_element: SINGLE_CELL [G]) is
        -- Set 'next' to 'an_element'.
106       ensure
          next_set: next = an_element
108     end
  end
```

## Solution

```
  class
2   SINGLE_LINKED_LIST [G]

4   feature -- Access

6   first : SINGLE_CELL [G]
```

```
8      -- Head element of the list, 'Void' if the list is empty
feature -- Basic operations
10
11  invert is
12      -- Invert the order of the elements of the list .
13      -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6].
14  do
15      invert_recursive (Void)
16  end
17
18  invert_recursive (previous: SINGLE_CELL[G]) is
19      local
20          next: SINGLE_CELL[G]
21          aux: SINGLE_CELL[G]
22      do
23          if first /= Void then
24              next := first.next
25              first.set_next(previous)
26              aux := first
27              first := next
28              invert_inner (aux)
29          else
30              first := previous
31          end
32      end
end
```