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

```plaintext
defurred class
2 WAGON
4 feature -- All features
6 id: INTEGER
-- Wagon identification number
8 capacity: INTEGER is
-- Maximal number of passengers
defurred
12 end
14 passenger_count: INTEGER
-- Actual number of passengers
16 set_passenger_count (n: INTEGER) is
-- Set 'passenger_count' to 'n'.
18 require
```

1
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
SLEEPER
Solution

class SLEEPER

inherit WAGON

create

make (n: INTEGER) is

  require
   positive_number: n > 0

  do
    bed_count := n
  ensure
    bed_count_set: bed_count = n
  end

bed_count: INTEGER

  -- Number of beds

capacity: INTEGER

  -- Maximal number of passengers

do
  Result := bed_count
ensure then
  capacity_equals_to_bed_count: capacity = bed_count
end

invariant

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

defered 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.
8 require
10 positive_a_table_count : a_table_count > 0
12 do
14 a_menu_not_void: a_menu / Void
16 table_count := a_table_count
18 menu := a_menu
20 ensure
22 table_count_set: table_count = a_table_count
24 menu_set: menu = a_menu
26 end

28 menu: STRING
--  Daily menu
30 set_menu (new_menu: STRING) is
32 --  Set menu to ‘new_menu’.
34 require
36 new_menu_not_void: new_menu / Void
38 do
40 menu := new_menu
42 ensure
44 menu_set: menu = new_menu
46 end

48 guest_count: INTEGER is
50 --  Actual number of guests
52 deferred
54 end

56 table_count: INTEGER
--  Number of tables
58 invariant
60 guest_count_positive : guest_count >= 0
62 table_count_greater_zero : table_count > 0
64 a_menu_not_void: a_menu / Void
66 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 DINER
Solution

class 2  DINER

4 inherit  WAGON
6 RESTAURANT
rename
8  guest_count as passenger_count
end

create
12  make

14 feature  --  All features
16  capacity:  INTEGER is
18  --  Maximal number of passengers
20  do
22  Result := 4 * table_count
24  ensure then
26  capacity, four_times_table_count: capacity = 4 * table_count
end

24 invariant
26  four_times_table_capacity: capacity = 4 * table_count

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.
• 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.

• \( \text{an\_array} \@ \text{m} \) denotes the element at index \( \text{m} \) in array \( \text{an\_array} \). Note that the infix feature \( \@ \) has a precondition stating that \( \text{m} \) must be a valid index. In our case \( \text{m} \) is allowed to have the values from 1 (not 0) to \( \text{n} \) (\( \text{an\_array.count} \))!

• 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
-- Search ‘x’ in ‘an_array’ using binary search algorithm. Version 1;
-- Elements of ‘an_array’ are in increasing order.
require
an_array_count_positive: an_array.count > 0
local
i: INTEGER
j: INTEGER
m: INTEGER
do
from
i := 1
j := an_array.count
until
i = j
loop
m := (i + j) // 2
if an_array @ m <= x then
i := m
else
j := m
end
end
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
-- Search ‘x’ in ‘an_array’ using binary search algorithm. Version 2;
-- Elements of ‘an_array’ are in increasing order.
require
an_array_count_positive: an_array.count > 0
local
i: INTEGER
j: INTEGER
ETHZ D-INFK
Prof. Dr. B. Meyer
Introduction to programming – Exercises
Winter 2005-2006

m: INTEGER
found: BOOLEAN

do
from
i := 1
j := an_array.count
until
i = j and not found
loop
m := (i + j) // 2
if an_array @ m < x then
i := m + 1
elseif an_array @ m = x then
found := true
else
j := m - 1
end
end
Result := found
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;
4 require
an_array_count_positive: an_array.count > 0
6 local
i: INTEGER
j: INTEGER
m: INTEGER
10 do
from
i := 0
j := an_array.count
until
i = j
The function `binary_search` 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 \( m = \frac{(i + j + 1)}{2} = \frac{(0 + 1 + 1)}{2} = 1 \). 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 \( m = \frac{(i + j + 1)}{2} = \frac{(2 + 1 + 1)}{2} = 2 \). 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.

```java
class SINGLE_LINKED_LIST [G]

feature -- Access
```
first : SINGLE_CELL [G]
    -- Head element of the list, 'Void' if the list is empty

feature -- Basic operations

invert is
    -- Invert the order of the elements of the list.
    -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6].

local

    do

end
Solution

class

SINGLE_LINKED_LIST [G]

feature -- Access

first : SINGLE_CELL [G]
feature -- Basic operations

invert is
  -- Invert the order of the elements of the list.
  -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6].
do
  invert_recursive (Void)
end

invert_recursive (previous: SINGLE_CELL[G]) is
  local
    next: SINGLE_CELL[G]
    aux: SINGLE_CELL[G]
do
  if first /= Void then
    next := first.next
    first.set_next(previous)
    aux := first
    first := next
    invert_recursive (aux)
  else
    first := previous
  end
end