Classroom 2

ETH Zurich

Date: 10. 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 Pairs... and Pairs of Pairs

Consider class \texttt{PAIR} \([G, H]\) given below, which represents pairs of elements.

1.1 A client for class \texttt{PAIR}

Fill in the source of class \texttt{CLIENT\_OF\_PAIR} so that:

- its feature \texttt{pair\_of\_integers} returns a pair made of the two integers that it receives as arguments
- its feature \texttt{pair\_of\_pairs\_of\_strings} returns a pair whose first element is a pair made of the first two strings that it receives as arguments, and whose second element is a pair made of the third and fourth strings that the routine receives as arguments

1.2 A client for class \texttt{CLIENT\_OF\_PAIR}

Fill in the source of class \texttt{ROOT\_CLASS} so that its \texttt{make} feature creates a pair of pairs of strings, where the first pair consists of the strings "a" and "b", and the second of the strings "c" and "d". Then \texttt{make} must print the concatenation of the members of the pair of pairs of strings. In other words, the output of \texttt{make} must be "abcd".

1
class 
PAIR [G, H]
create
make
feature -- Initialization
make (f: G; s: H) is
   -- Create a new pair with first member 'f' and second member 's'.
   do
      first := f
      second := s
   ensure
      first_set: first = f
      second_set: second = s
   end

20 feature -- Access
22 first : G
   -- First member of the pair
24 second : H
   -- Second member of the pair
28 end

class
CLIENT_OF_PAIR
4 feature -- Basic operations
6 pair_of_integers (i1, i2: INTEGER): ......................... is
      -- Pair made of the two integers 'i1' and 'i2'
8 do
10 ..............................................................
12 ..............................................................
14 ..............................................................
16 ..............................................................
end
18
18 pair_of_pairs_of_strings (s1, s2, s3, s4: STRING): ..................... is
      -- Pair consisting of two pairs of strings
      -- ('s1' and 's2' form one pair, 's3' and 's4' form another pair, and these 2 pairs
      are also grouped in a pair.)
20 local
24 ..............................................................
26 ..............................................................
d
2 Inversion of Linked List

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.
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

end

class
SINGLE_CELL [G]

feature -- Access

next: SINGLE_CELL [G]
  -- Reference to the next generic list cell of a list
Implement the feature `invert` of class `SINGLE_LINKED_LIST [G]`, 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]`. Do not create objects of type `SINGLE_CELL [G]` and also do not introduce any new feature in class `SINGLE_LINKED_LIST [G]` and `SINGLE_CELL [G]`. 
3 Polymorphism and dynamic binding

Consider the inheritance hierarchy shown in Figure 1 and the corresponding class text shown in Listing 1.

![Inheritance Hierarchy Diagram]

Figure 1: inheritance hierarchy

Listing 1: Classes VEHICLE, CAR

```ruby
class VEHICLE
create
make
feature -- Initialization
make is
  -- Initialize vehicle.
do ...
end

feature -- Output
print_vehicle is
  -- Print message.
do
  io.put_string ("This is a vehicle.")
end
end

class CAR
inherit VEHICLE
rename print_vehicle as print_car
redefine print_car
end
create
make

feature -- Output
print_car is
  -- Print message.
```

---

---
do
    io.put_string ("This is a car.")
end

Listing 2: Classes $SIMULATION$

class $SIMULATION$
create
make
  feature {NONE}    -- Initialization
  make is
    -- Initialize 'traffic'.
  do
    create traffic.make
  end
feature    -- Access
traffic : LINKED_LIST [VEHICLE]
    -- Vehicles in traffic
feature    -- Element change
  add_vehicle_to_traffic is
  local
  do
    ...  
    ensure
      one_more: traffic.count = old traffic.count + 1
  end
invariant
  traffic_not_void : traffic /= Void
end

Class $SIMULATION$, as shown in Listing 2 has a list of $VEHICLE$s called $traffic$. As you can see command $add_{vehicle}_{to}_{traffic}$ is incomplete. The goal of this exercise is to find three different ways to implement this command.
Example

Question

Complete the following code so that the message displayed at execution is: “This is a vehicle”, and explain why the code you wrote works (mention principles of object-oriented programming to explain).

```plaintext
add_vehicle_to_traffic is
local
v: VEHICLE do
v.print
ensure
one_more: traffic.count = old traffic.count + 1
end
```

Answer:

```plaintext
add_vehicle_to_traffic is
local
v: VEHICLE do
create v.make v.print_vehicle
ensure
one_more: traffic.count = old traffic.count + 1
end
```

Explanation

The only feature we have at our disposal to display “This is a vehicle.” is the feature `v.print_vehicle` defined in class `VEHICLE`. Therefore the last line should be `v.print_vehicle`. For this code to be valid `v` needs to be created as an instance of type `VEHICLE`; thus the second line should be `create v.make`. Hence the above code.
To do:

Question

Complete the following code so that the message displayed at execution is: “This is a car”, and explain why the code you wrote works (mention principles of object-oriented programming to explain).

Note: There is exactly one instruction missing on the first dotted line, and a part of a feature name missing on the dotted segment.

```
add_vehicle_to_traffic is
local c: CAR
end

traffic.extend (c)

ensure one_more: traffic.count = old traffic.count + 1
```

Explanation

...
Question

Complete the following code so that the message displayed at execution is: “This is a car”, and explain why the code you wrote works (mention principles of object-oriented programming to explain).

Note: There is exactly one instruction missing on the first dotted line, and a part of a feature name missing on the dotted segment.

```
add_vehicle_to_traffic is
local
v: VEHICLE
do
traffic.extend (v)

v.print
ensure
one_more: traffic.count = old traffic.count + 1
end
```

Explanation
Question

Complete the following code so that the message displayed at execution is: “This is a car”, and explain why the code you wrote works (mention principles of object-oriented programming to explain).

Note: There is exactly one instruction missing on the first dotted line, and a part of a feature name missing on the dotted segment.

```python
add_vehicle_to_traffic is
local
v: VEHICLE
c: CAR
do
...........................
v := c
traffic.extend (v)
v.print
ensure
one_more: traffic.count = old traffic.count + 1
end
```

Explanation