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

1.1 A client for class \textit{PAIR} 

Fill in the source of class \textit{CLIENT_OF_PAIR} so that:

- its feature \textit{pair_of_integers} returns a pair made of the two integers that it receives as arguments

- its feature \textit{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 \textit{CLIENT_OF_PAIR} 

Fill in the source of class \textit{ROOT_CLASS} so that its \textit{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 \textit{make} must print the concatenation of the members of the pair of pairs of strings. In other words, the output of \textit{make} must be "abcd".
class PAIR [G, H]
create

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

feature -- Access

first : G

-- First member of the pair

second: H

-- Second member of the pair
end

class CLIENT_OF_PAIR

feature -- Basic operations

pair_of_integers (i1, i2: INTEGER): ...................... is

-- Pair made of the two integers ‘i1’ and ‘i2’
do

end

pair_of_pairs_of_strings (s1, s2, s3, s4: STRING):

-- 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.)
local
do
Solution

class
2 CLIENT_OF_PAIR

feature -- Basic operations

6 pair_of_integers \((i1, i2: INTEGER): PAIR [INTEGER, INTEGER]\) is
-- Pair made of the two integers ‘i1’ and ‘i2’
8 do
create Result.make \((i1, i2)\)
10 end

12 pair_of_pairs_of_strings (s1, s2, s3, s4: STRING): PAIR [PAIR [STRING, STRING],
PAIR [STRING, STRING]] is
-- Pair consisting of two pairs of strings
14 -- (s1 and s2) form one pair, ‘s3’ and ‘s4’ form another pair, and these 2 pairs are
also grouped in a pair.
local
16 p1, p2: PAIR [STRING, STRING]
do
18 create p1.make (s1, s2)
create p2.make (s3, s4)
20 create Result.make (p1, p2)
end
end

class 2 ROOT_CLASS
create
make
feature
8
make is
10 -- Creates a pair of pairs of strings using the strings "a", "b", "c", "d"
-- and then prints the concatenation of the members of each element of this pair.
12 local
14 cp: CLIENT_OF_PAIR
16 p: PAIR [PAIR [STRING, STRING], PAIR [STRING, STRING]]
p1, p2: PAIR [STRING, STRING]
do
18 create cp
create p. pair_of_pairs_of_strings ("a", "b", "c", "d")
p1 := p.first
p2 := p.second
print (p1.first + p1.second + p2.first + p2.second)
22 end

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 2 SINGLE_LINKED_LIST [G]
feature -- Access
first : \textit{SINGLE}\_\textit{CELL} \[G\]
\quad -- Head element of the list, ‘Void’ if the list is empty

feature -- Basic operations

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

local
do

end

class \textit{SINGLE}\_\textit{CELL} \[G\]

feature -- Access

next : \textit{SINGLE}\_\textit{CELL} \[G\]
\quad -- Reference to the next generic list cell of a list

feature -- Element change

set\_next (an\_element : \textit{SINGLE}\_\textit{CELL} \[G\]) is
\quad -- Set ‘next’ to ‘an\_element’.
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].

Solution

```python
invert is
   -- Invert the order of the elements of the list.
   -- E.g. the list [6, 2, 8, 5] should become [5, 8, 2, 6]
local
   actual: SINGLE_CELL[G]
   next: SINGLE_CELL[G]
do
   from
   until first = Void
   loop
      actual := first
      first := first.next
      actual.set.next(next)
      next := actual
   end
   first := next
end
```
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**

```plaintext
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.
```

Listing 2: Classes \textit{SIMULATION}

Class \textit{SIMULATION}, as shown in Listing 2 has a list of \texttt{VEHICLE}s called \texttt{traffic}. As you can see command \texttt{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

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 `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.

```plaintext
add_vehicle_to_traffic  is
  local c: CAR
do
  create c.make
  traffic . extend (c)
  c . print_car
ensure
  one_more: traffic . count = old traffic . count + 1
end
```

Explanation

The local variable `c` is declared of type `CAR`. The only feature of class `CAR` at disposal to print a message is `print_car`. Hence the last line should be `c.print_car`. To be able to write `c.print_car`, `c` must not be `Void`. Thus, we need to create it. Because class `CAR` does not have any heir in the above example, we simply create `c` as a direct instance of `CAR` and the first missing line is: `create c.make`. Hence the code.
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.

```plaintext
add_vehicle_to_traffic is
local
v: VEHICLE
do
create {CAR} v.make
traffic.extend (v)
end
one_more: traffic.count = old traffic.count + 1
```

Explanation

The static type of \( v \) is \textit{VEHICLE}. Thus, the features available to \( v \) are the features of class \textit{VEHICLE}. Thus, the only feature applicable to \( v \) in the last line is \textit{print\_vehicle}. Now, we want the displayed message to be “This is a car.”. It means that we want the executed version of \textit{print\_vehicle} to be the one from class \textit{CAR} (called \textit{print\_car}) by relying on dynamic binding. But dynamic binding only works if the dynamic type of \( v \) differs from its static type. Here, what we need is to create \( v \) as a direct instance of class \textit{CAR} so that the version of \textit{print\_vehicle} that gets called at run time is the version defined in class \textit{CAR} (corresponding to the dynamic type of \( v \)). Therefore the first missing line is: \textit{create {CAR} v.make}. Hence the code.
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.

```java
add_vehicle_to_traffic is
local
v: VEHICLE
c: CAR
doa
create c.make
v := c
traffic.extend (v)
v.print_vehicle
ensure
one_more: traffic.count = old traffic.count + 1
end
```

Explanation

The static type of v is VEHICLE. Thus, the features available to v are the features of class VEHICLE. Thus, the only feature applicable to v in the last line is print_vehicle. Now, we want the displayed message to be “This is a car”. It means that we want the executed version of print_vehicle to be the one from class CAR (called print_car) by relying on dynamic binding. The assignment v := c helps us here. What we need is to create c as a direct instance of class CAR: create c.make. Then, the assignment v := c attaches v to this object of type CAR. (v and c point to the same object of type CAR, although they have different static types.) Thus, when executing v.print_vehicle, it is the most appropriate version of print_vehicle that is called, namely the version of print_vehicle in class CAR (corresponding to the dynamic type of v), which is called print_car and displays “This is a car.”. Hence the code.