Solution 3: Of objects and features

ETH Zurich

1 Classes vs. objects

There is no unique solution. Sample answers:

1.1 A class can be viewed as a software *module* (a piece of source code that contains descriptions of related operations and data), and a *type* (a set of objects that support the same operations).

An object can be viewed as a *collection of data* (whose structure is defined in the object's generating class) and a *member of a type* (an entity in a running program, to which the operations defined in the generating class are applicable).

1.2 A class can be viewed as the blueprint of a machine, while an object is the actual machine built according to the blueprint.

2 Query call chains

1. This chain returns a *TRAFFIC_LINE*:

$$\underbrace{\underbrace{\underbrace{Route2}_{}\;.first}_{TRAFFIC\_ROUTE}\;.line}_{TRAFFIC\_LEG}\ldots$$

$$\underbrace{\underbrace{\underbrace{\underbrace{Route2}_{TRAFFIC\_ROUTE}.first}_{TRAFFIC\_LEG}.line}_{TRAFFIC\_LINE}}$$

2. This chain returns a *TRAFFIC_POINT*:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{Route1}_{TRAFFIC\_ROUTE}.first}_{TRAFFIC\_LEG}.next}_{TRAFFIC\_LEG}.origin}_{TRAFFIC\_STATION}.location}_{TRAFFIC\_POINT}$$

3. This chain returns a *TRAFFIC_STATION*:

$$\underbrace{\underbrace{\underbrace{Line2}_{TRAFFIC\_LINE}.i\_th(\underbrace{\underbrace{Line2}_{TRAFFIC\_LINE}.count}_{INTEGER})}_{TRAFFIC\_STATION}.stop(\underbrace{\underbrace{\underbrace{Route3}_{TRAFFIC\_ROUTE}.first}_{TRAFFIC\_LEG}.line}_{TRAFFIC\_LINE})}_{TRAFFIC\_STOP}.station}_{TRAFFIC\_STATION}$$

3 Writing more feature calls

Listing 1: Class *PLANNER*

```
indexing
   description: "Planner class (Assignment 3)"
   date: "$Date$"
   revision: "$Revision$"

class
   PLANNER

inherit

   TOURISM

feature -- Explore Paris

   explore_on_click is
         -- Explore Paris!
      do
         Paris.display
         -- Paris.display must be the first line (loads and displays Paris map)

         Line1.remove_all_segments
         Line1.extend (Station_concorde)
         Line3.remove_all_segments
         Line3.extend (Station_concorde)
         Line7_a.remove_all_segments
         Line7_a.extend (Station_concorde)
         Line8.remove_all_segments
         Line8.extend (Station_concorde)
         Line2.remove_all_segments
         Line2.extend (Line3.terminal_1)
         Line2.extend (Line7_a.terminal_1)
         Line2.extend (Line1.terminal_1)
         Line2.extend (Line8.terminal_1)
         Line2.extend (Line2.terminal_1)
      end

end
```

4 In and out

Listing 2: Class *BUSINESS_CARD*

```
class
   BUSINESS_CARD

create
   fill_in
```

```
feature {NONE} -- Initialization
  fill_in
      -- Fill in the card and print it.
    do
      Io.put_string ("Your name: ")
      Io.read_line
      name := Io.last_string.twin

      Io.put_string ("Your job: ")
      Io.read_line
      job := Io.last_string.twin

      Io.put_string ("Your age: ")
      Io.read_integer
      age := Io.last_integer

      Io.put_string (text)
    end

feature -- Access
  name: STRING
      -- Owner's name.

  job: STRING
      -- Owner's job.

  age: INTEGER
      -- Owner's age.

feature -- Output
  age_info: STRING
      -- Text representation of age on the card.
    do
      Result := age.out + " years old"
    end

  text: STRING
      -- Text on the card.
    do
--    Result := name + "%N" + job + "%N" + age_info + "%N"
      Result := line (Width + 2) + "%N"
         + "|" + name + spaces (Width - name.count) + "|%N"
         + "|" + job + spaces (Width - job.count) + "|%N"
         + "|" + age_info + spaces (Width - age_info.count) + "|%N"
         + line (Width + 2) + "%N"
    end

  Width: INTEGER = 50
      -- Width of the card (in characters), excluding borders.

  line (n: INTEGER): STRING
      -- Horizontal line on length 'n'.
    do
      Result := "-"
      Result.multiply (n)
    end

  spaces (n: INTEGER): STRING
      -- String consisting of 'n' whitespaces.
    do
      Result := " "
      Result.multiply (n)
    end
end
```

- The main benefit of using the constant attribute is that the width of the card is stored in a single place. Thus, when you want to change it you just have to edit a single constant attribute definition as opposed to searching the whole program text for usages of number 50 and trying to remember, which ones of them actually refer to the width and which ones mean something else and just happen to be equal to 50.

  Another benefit is that using a meaningful name for the attribute improves code readability.

- The answer depends on the actual implementation; we give the answer for the master solution.

  If the name is too long you will get a precondition violation when calling feature *multiply* of class *STRING* from within the function *spaces*. It happens because we are trying to fill up the line with a negative (or zero) number of spaces.

  A solution would be to go through all the lines in the card, calculate the maximum of their lengths and change the width to be larger than this maximum. Otherwise, if there is a good reason for the width to be equal to 50, the *APPLICATION* class should check the user input to be sufficiently short, and if it isn't, either truncate it or ask the user to input a shorter string.