Solution 5: Assignments and control structures

ETH Zurich

1 Assignments

The solution lists the correct statements for each of the subtasks.

1. (c)
2. (a)
3. (b)
4. (d)
5. (c)
6. (d) (e)
7. (a)
8. (c) (e)
9. (b) (e)

2 Reading loops

Version A:

- The result of the comparison using = will always be **False** (*STRING* is a reference type).
- The second if-statement is inside the loop, so it will try to move "Central" in every loop iteration after it had been found. This behavior is not incorrect, as it has no effect expect for the first time, but it is inefficient.
- The corrected code of version A is shown in Listing **??**.

Version B:

- Infinite loop: there is no call to a command that advances the cursor position in the list.
- Possible precondition violation: *i.item.name* ∼ **"Central"** will most likely be tested before *i.after*, therefore trying to access an item when the cursor has already advanced past the end of the list. To get a guaranteed correct order of evaluation, switch the two conditions and use **or else** instead of **or**.
- The corrected code of version B is shown in Listing **??**.

Listing 1: Version A

```
explore
    -- Move "Central".
  local
    station: STATION
  do
    across
      Zurich.stations as i
    loop
      if i.item.name ~ "Central" then
        station := i.item
      end
    end
    if station /= Void then
      station.set_position ([0.0, 0.0])
    end
  end
```

Listing 2: Version B

```
explore
    -- Move "Central".
  local
    i: like Zurich.stations.new_cursor
  do
    from
      i := Zurich.stations.new_cursor
    until
      i.after or else i.item.name ~ "Central"
    loop
      i.forth
    end
    if not i.after then
      i.item.set_position ([0.0, 0.0])
    end
  end
```

3 Next station: loops

```
note
  description: "Route information displays."

class
  DISPLAY

inherit
  ZURICH_OBJECTS

feature -- Explore Zurich

  add_public_transport
      -- Add a public transportation unit per line.
    do
      across
        Zurich.lines as i
      loop
        i.item.add_transport
      end
    end

  update_transport_display (t: PUBLIC_TRANSPORT)
      -- Update route information display inside transportation unit 't'.
    require
      t_exists: t /= Void
    local
      i: INTEGER
      s: STATION
```

```
    do
      console.clear
      console.append_line (t.line.number.out + " Willkommen/Welcome")
      from
        i := 1
        s := t.arriving
      until
        i > 3 or s = Void
      loop
        console.append_line (stop_info (t, s))
        s := t.line.next_station (s, t.destination)
        i := i + 1
      end
      if s /= Void then
        if s /= t.destination then
          console.append_line ("...")
        end
        console.append_line (stop_info (t, t.destination))
      end
    end

  stop_info (t: PUBLIC_TRANSPORT; s: STATION): STRING
      -- Information about stop 's' of transportation unit 't'.
    require
      t_exists: t /= Void
      s_on_line: t.line.has_station (s)
    local
      time_min: INTEGER
      l: LINE
    do
      time_min := t.time_to_station (s) // 60
      if time_min = 0 then
        Result := "<1"
      else
        Result := time_min.out
      end
      Result := Result + " Min.%T" + s.name

      -- Optional task:
      across
        s.lines as i
      loop
        l := i.item
        if l /= t.line and
          ((l.next_station (s, l.first) /= Void and not
            t.line.has_station (l.next_station (s, l.first))) or
          (l.next_station (s, l.last) /= Void and not
            t.line.has_station (l.next_station (s, l.last)))) then
          Result := Result + " " + i.item.number.out
        end
      end
    end
```

```
end
```

4 Board game: Part 1

There are several possible solutions; we discuss the two most reasonable in our opinion.

The simpler solution only includes three classes:

- *GAME*: encapsulates the logic of the game (start state, the structure of a round, ending conditions).
- *DIE*: provides random numbers in the required range.
- *PLAYER*: stores the state of each player in the game and performs a turn.

We discarded *ROUND* and *TURN*: we consider them parts of the *GAME* and *PLAYER* behavior respectively, rather than separate abstractions. Additionally *PLAYER* and *TOKEN* represent the same abstraction for now.

In the simple solution we don't introduce classes for *SQUARE* and *BOARD*. The only information associated with squares in the current version of the game is their index, thus a square can be easily represented with an integer. Also the board in the current version doesn't have any specific structure (square arrangement); the only property of the board is the number of squares, which probably does not deserve a separate class and instead can be stored in *GAME*.

A more flexible solution additionally includes classes *SQUARE* and *BOARD*. Though *SQUARE* doesn't contain enough behavior for now, we anticipate that in the future versions of the game there might be squares with special properties and behavior (this anticipation is based on our knowledge of the problem domain, namely that interesting boardgames have squares of different types with different properties).

Introducing class *BOARD* makes the solution more flexible with respect to the arrangement of squares on the board. In the simple version the knowledge about "on which square does a token land if it moves n steps starting from square x" is located in class *PLAYER*. Once it becomes more complicated than just $x + n$, it is better to encapsulate such knowledge in class *BOARD*.

5 MOOC: Assignment, control structures

The order in which the questions and the answers appear here in the solution may vary because they are randomly shuffled at each attempt.

References, Assignment, and Object Structure

- Choose the appropriate initialization values for the variables below: nat_val: NATURAL (0); int_val: INTEGER (0); real_val: REAL (0.0); bool_val: BOOLEAN (False); char_val: CHARACTER (null char); string_val: STRING (Void)
- Suppose to have the following class *PERSON*:

```
class
    PERSON

create
    set_friend,
    default_create
```

```
feature -- Initialization

    set_friend (f: PERSON)
            -- Initialize current object.
        do
            friend := f
        end

feature -- Access

    friend: PERSON
end
```

In some other class, some objects of type PERSON are created and initialized:

```
create kima
create jimmy.set_friend (kima)
create buck.set_friend (jimmy)
create rhonda.set_friend (buck)
create kima.set_friend (rhonda)
```

We claim that there is a cycle in the four objects above. True or False? False

- Determine to whom the following calls apply: set_color ("red"): to Current; my_pic.set_color ("blue"): to the object attached to my_pic; till.friend.friend: to the object attached to till.friend.
- Determine if the following calls are qualified or unqualified: set_color ("red"): unqualified; my_pic.set_color ("blue"): qualified; arno.friend.friend: both qualified; draw: unqualified.
- Assuming you have the following definitions:

  ```
  s1: STRING = "Game"
  s2: STRING = " of Thrones"
  ```

  What can you say about the following Eiffel routine?

  ```
      join_strings (s1, s2: STRING)
              -- Append s2 to s1.
          do
              s1.append (s2)
          end
  ```

  It works as expected: s1 has value "Game of Thrones"; This routine produces a side effect on s1.
- What can you say about the following Eiffel routine?

  ```
      increment (num: INTEGER)
              -- Add 1 to num.
          do
              num := num + 1
          end
  ```

  It does not work as expected: num is not incremented; It does not compile. In Eiffel you cannot assign directly to a routine argument.

- Suppose to have the following class ITEM:

```
class
    ITEM

create {ORDER_LINE}
    set_description

feature {NONE} -- Initialition

    set_description (d: STRING)
            -- Set description for current object.
        do
            description := d
        end

feature -- Basic operations

    set_price (p: INTEGER)
            -- Set price for current object.
        do
            price := p
        end

feature -- Access

    description: STRING
        -- Item description.

    price: INTEGER
        -- Item price.
end
```

  Which of the following is true? Objects of class ITEM can be created from within objects of class ORDER_LINE; Feature set_description can be used as a creation procedure, but cannot be invoked normally (that is, not as a creation procedure) on an object of type ITEM from another class.

- Suppose to have the following class ITEM:

```
class
    ITEM

feature -- Basic operations

    set_price (p: INTEGER)
            -- Set price for current object.
        do
            price := p
        end

feature -- Access

    price: INTEGER
```

```
end
```

In some other class TEST, the following routine is declared:

```
swap_prices (item_1, item_2: ITEM)
        -- Swap prices of items.
    local
        temp: INTEGER
    do
        temp := item_1.price
        item_1.set_price (item_2.price)
        item_2.set_price (temp)
    end
```

Assume that in the same class TEST two references of type ITEM are declared:

```
item_one, item_two: ITEM
```

Then the following happens:

```
create item_one
item_one.set_price (7)
create item_two
item_two.set_price (4)
swap_prices (item_two, item_one)
print (item_one.price.out)
print (item_two.price.out)
```

What will be printed on the console? 47

Control Structures

- Complete the code of the following function maximum by choosing the correct instructions:

```
maximum (a, b: INTEGER): INTEGER
        -- The maximum between a and b.
do
    if a > b then
        Result := a
    else
        Result := b
    end
end
```

```
Complete the code of the following function print_relation by choosing the correct
    instructions:
print_relation (a, b: INTEGER)
        -- Prints if a > b, a < b or a = b.
do
    if a > b then
        print (a.out + ">" + b.out)
    else
        if a < b then
            print (a.out + "<" + b.out)
```

```
        else
            print ("The 2 numbers are equal.")
        end
    end
end
```

- Complete the code of the following function remainder by choosing the correct istructions. Assume d1 and d2 are positive.

```
remainder (d1, d2: INTEGER): INTEGER
        -- Compute the remainder of integer division between d1 and d2.
do
    from
        Result := d1
    until
        Result <= d2
    loop
        Result := Result - d2
    end
    -- nothing here
end
```

- Complete the code of the following function absolute_value by choosing the correct instructions:

```
absolute_value (a: INTEGER): INTEGER
        -- Absolute value of a.
do
    if a >= 0 then
        Result := a
    else
        Result := -a
    end
end
```

- Assuming that c is a CHARACTER, what will the following instruction print, if executed with c = '0'?

```
inspect c
    when '1'..'9' then
        print ("number")
    when 'a'..'z' then
        print ("lower case letter")
    when 'A'..'Z' then
        print ("upper case letter")
    when '#','@','%' then
        print ("special character")
    else
        print ("unexpected character")
end
```

It will print “unexpected character”.

- Complete the code of the following function euclid by choosing the correct expressions for the loop invariant and the loop variant:

```
euclid (a, b: INTEGER): INTEGER
        -- Greatest common divisor of a and b.
require
    a_positive: a > 0
    b_positive: b > 0
local
    m, n: INTEGER
do
    from
        m := a
        n:= b
    invariant
        euclid (a, b) = euclid (m, n)
    variant
        m + n
    until
        m = n
    loop
        if m > n then
            m := m − n
        else
            n := n − m
        end
    end
    Result := m
end
```