# Solution 5: Assignments and control structures

ETH Zurich

## 1 Assignments

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

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

## 2 Reading loops

Version A:

- The result of the comparison using = will always be **False** (*STRING* is a reference type).
- The if-statement is inside the loop: it will move all the stations until it finds the right one.
- The corrected code of version A is shown in Listing 1.

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.is_equal* (**"Central"**) may 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 order of evaluation, use **or else** instead of **or**.
- The corrected code of version B is shown in Listing 2.

Listing 1: Version A

```
explore
    -- Move "Central".
  local
    i: like Zurich.stations.new_cursor
    found: BOOLEAN
  do
    from
      i := Zurich.stations.new_cursor
    until
      i.after or found
    loop
      if i.item.name.is_equal ("Central") then
        found := True
      else
        i.forth
      end
    end
    if not i.after then
      i.item.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.is_equal
          ("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: "Creating new objects for Zurich."

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.name.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
    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.name.out
```

```
        end
      end
    end

end
```

# 4 Board game: Part 1

There are several possible solution; we discuss two that are most reasonable in our opinion.

A simpler solution includes only 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 behavior of *GAME* and *PLAYER* respectively, rather than separate abstractions. Additionally *PLAYER* and *TOKEN* represent the same abstraction for now.

In the simpler 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*.