Solution 5: References and assignments

ETH Zurich

1 City building

Listing 1: Class *CITY_BUILDING*

```
class CITY_BUILDING

inherit

  TOURISM

feature -- City creation

  explore is
      -- Create the city, central station and other needed objects.
    local
      t: TIME
    do
      create city.make ("New Zurich")
      main_window.canvas.set_city (city)

      create t.make_now
      create random.set_seed (t.milli_second)
      random.start

      create central_station.make_with_location ("Central", 0, 0)
      city.put_station (central_station)
      add_line
    ensure
      city_exists: city /= Void
      line_exists: line /= Void
      central_station_exists: central_station /= Void
      random_exists: random /= Void
    end

  city: TRAFFIC_CITY
      -- City under construction.

  central_station: TRAFFIC_STATION
      -- Central tram station.

  line: TRAFFIC_LINE
      -- Most recently added line.
```

```
random: RANDOM
    -- Random sequence.

add_line
    -- Add new line and store it in ‘line’.
  require
    city_exists: city /= Void
    central_station_exists: central_station /= Void
  local
    tram_type: TRAFFIC_TYPE_TRAM
    name: STRING
  do
    name := "Line " + (city.lines.count + 1).out -- relies on city_exists
    create tram_type.make
    create line.make_with_terminal (name, tram_type, central_station) -- relies on
        central_station_exists
    line.set_color (random_color)
    city.put_line (line)
    Console.show (name + " added")
  ensure
    line_exists: line /= Void
  end

add_station (x, y: INTEGER)
    -- Extend ‘line’ with a new station at coordinate (‘x’, ‘y’).
  require
    city_exists: city /= Void
    line_exists: line /= Void
  local
    s: TRAFFIC_STATION
  do
    create s.make_with_location ("Station " + city.stations.count.out, x, y) -- relies on
        city_exists
    city.put_station (s)
    line.extend (s) -- relies on line_exists
  end

random_color: TRAFFIC_COLOR
    -- Random color.
  require
    random_exists: random /= Void
  local
    r, g, b: INTEGER
  do
    random.forth -- relies on random_exists
    r := random.item \\ 256
    random.forth
    g := random.item \\ 256
    random.forth
    b := random.item \\ 256
    create Result.make_with_rgb (r, g, b)
  ensure
```

```
      Result_exists: Result /= Void
    end
end
```

6. The simplest way to avoid the problem is to add a call to *add_line* to the end of *explore*, which is executed immediately when the application starts (that is, before any double left clicks are processed).

8. **Local variables vs. attributes.** We want *add_line* and *add_station*, every time they are called, to be able to access the very same city object that was created once by *explore*. If more than one feature needs access to an object, it must be stored in an attribute (because local variables are only accessible within a single feature). Similar reasoning applies to other attributes.

   Note that it's technically possible to make *random* a local variable of *random_color* and create a new random sequence every time *random_color* is called. However, this is not how random number generators are generally used: as a rule a random sequence is initialized just once and used throughout the program execution. There are two main reasons:

   – Random number generation algorithms usually provide several useful properties, e.g. that subsequent numbers are sufficiently far away from each other, that the sequence is sufficiently uniformly distributed over the whole range, etc. If instead of relying on the random number generator you use current time every time you need another random number, you are loosing these properties. For example, if your program needs random numbers very often they might turn out all the same.

   – Sometimes you want to control the random numbers: for example, you would like to repeat exactly the same program execution to reproduce a fault. If you initialize the random sequence just once, the only thing you need to do to regenerate the same sequence in a future execution is to save a single seed. If you are using time, you have to save the whole sequence in order for the execution to be reproducible.

9. **Contracts.** Here is an informal argument on why the preconditions of *add_line*, *add_station* and *random_color* will always be satisfied at runtime.

   The feature *add_line* requires that the city and the central station exist. *add_line* is first called from the end of *explore*, where both objects have already been created. Afterwards it is only called after the (first) execution of *explore* has finished and has created both objects (this can be reflected in the postcondition of *explore*). There is no other feature in *CITY_BUILDING* that sets these attributes back to void, thus they remain non-void throughout the rest of the program execution.

   The second part of the reasoning above can also be applied to the precondition of *add_station*.

   The feature *random_color* requires that *random* exist. *random_color* is only called from *add_line*, which is always called after *random* was first created by *explore*.

   If we consider not only attributes, but expressions in general, our implementation relies on more of them being non-void. For example the call *city.stations.count* in *add_station* relies on *city.stations* /= **Void**. If you check out the class invariant of *TRAFFIC_CITY* you will see that this is true for every city, that is why *add_station* does not have to state it in the precondition.

   Look at the call *line.set_color* (*random_color*) in *add_line*. The feature *set_color* requires that its argument be non-void. We can make the reasoning about the correctness of this call more clear is we add **Result** /= **Void** to the postcondition of *random_color*.

2 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)

3 Phone contracts

1. The postcondition for *sleep* states **not** *screen_on*; this immediately contradicts *screen_on* (the precondition of *unlock*).
2. The precondition of *sleep* is empty, meaning that it can be invoked at any time.
3. The postcondition of *remove* (**"Jen"**) guarantees **not** *has* (**"Jen"**) and *is_ready*, which is exactly what is needed to satisfy the precondition of *add* (**"Jen"**) (together with **"Jen"** /= **Void**, which is trivially true).
4. The precondition should be extended with

   ```
   phone /= Void
   phone.is_ready
   phone.has ("Ulrich")
   ```

   to satisfy the precondition of the call to *remove*. Once the precondition is extended, we can successfully deduce the postcondition of *use* from "two_contacts" in the original precondition of *use* and "one_less_contact" in the postcondition of *remove*.
5. This routine does not require any additional preconditions to complete successfully. The precondition of the call to *add* is satisfied because **not** *phone.has* (**"Ulrich"**) follows from $phone.contact\_count = 0$ (see the postcondition of *has*). To show that the postcondition of *use* holds we apply the same reasoning as in the previous point.
6. The precondition should be extended with

   ```
   not phone.has ("Ulrich")
   ```

   The postcondition follows from the class invariant and the postcondition of *add*.

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*.