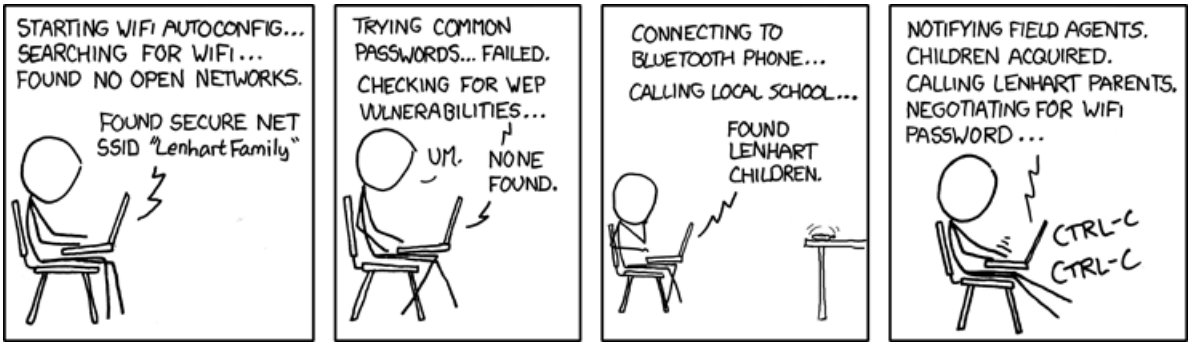
Assignment 5: References and assignments

ETH Zurich

Hand-out: 15 October 2010
Due: 26 October 2010



Zealous Autoconfig © Randall Munroe (xkcd.com)

Goals

- Test your knowledge about assignments.
- Write more contracts.
- Start to work on a more complex application.

1 City building

We have prepared a traffic project that contains a class *CITY_BUILDING*. In this class you will find four features: *explore*, *add_station*, *add_line*, and *random_color*. The application is programmed to call *add_station* when you double click with the left mouse button into the city canvas (the white area where the map is usually displayed), and feature *add_line* when you double click with the right mouse button. At the moment, double clicking will result in a message in the Console area, but no station or line is created. In this assignment, you will complete these features to do what their comments promise.

To do

1. Download
   http://se.inf.ethz.ch/teaching/2010-H/eprog-0001/assignments/05/assignment_5.zip and extract it in `traffic/example`. You should now have a new directory `traffic/example/assignment_5` with assignment_5.ecf directly in it.

2. Open and compile this new project. Open class *CITY_BUILDING* and follow the suggestions given below.

3. In feature *explore*, create a new city. To display this city on the screen, use feature *set_city* of *TRAFFIC_CITY_CANVAS*. You can get the canvas you need from the feature *main_window*.

   Add a station called "Central" at coordinate (0, 0) to the city.

   We have modified the application to automatically call *explore* at startup, but you can still call it by clicking the "Run example" button.

4. Implement feature *add_line* to add a new line to the city. The line should be of tram type and have the central station (created in step 3) as its starting terminal. The names of the newly added lines should be "Line 1", "Line 2" and so on (incrementing the number every time a new line is added). Adding a new line doesn't change the map immediately, so show a message "Line n added" in the Console area to let the user know that something happened.

5. Implement feature *add_station* to add a new station to the city and extend the most recently added line with this station. The position of the new station is given through the arguments of *add_station*. The names of the newly added stations should be "Station 1", "Station 2" and so on (incrementing the number every time a new station is added).

6. If you run the program now and try to create a station with a left double-click, the program will fail because there is no line that the new station can be added to. Modify the program in such a way that this failure doesn't occur.

7. Since all the created lines have the same default color, it is difficult to distinguish them. Implement the feature *random_color* and use it to assign a new color to each created line. To achieve this, use the class *RANDOM* that generates random numbers for you. The following code illustrates its usage:

```
local
    t: TIME
    random: RANDOM
    n, m: INTEGER
do
    create t.make_now -- Create a time object for the seed.
    create random.set_seed (t.milli_second) -- Create a random number sequence.
    random.start -- Go to the beginning of the sequence.
    n := random.item \\ 100 -- Access the first random number;
    -- take modulo 100 to get a number between 0 and 99.
    random.forth -- Go to the next number in the sequence.
    m := random.item \\ 100 -- Access the second random number.
    random.forth -- Go to the next number in the sequence.
end
```

   The *seed* in the example above is a number used to initialize the algorithm that generates a pseudo-random sequence. A fixed seed always produces the same sequence. We are using the current time (more precisely, the number of milliseconds in the current time) as a seed, so that running this program at two different times will most likely produce two different sequences.

8. In your solution you probably used both local variables and attributes. For the attributes of *CITY_BUILDING*, explain why you have chosen to make them attributes rather than locals.

9. Are there features in your solution that rely on some attributes not being **Void**? If so, express these properties as preconditions and mark (using comments) at least one line in each feature body that relies on each precondition clause.

   Will these preconditions always be satisfied when running your program? Provide an informal argument on why this is or isn't true; if possible, support your argument by adding postconditions and/or class invariants.

To hand in

Hand in the code of *CITY_BUILDING* and answers to the questions 8 and 9.

2 Assignments

In this assignment you can test your understanding of assignment instructions. Consider the following class:

```
class PERSON
create make
feature -- Initialization
  make (s: STRING)
      -- Set 'name' to 's'.
    require
      s_non_empty: s /= Void and then not s.is_empty
    do
      name := s
    ensure
      name_set: name = s
    end

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

  loved_one: PERSON
      -- Person's loved one.

feature -- Basic operations
  set_loved_one (p: PERSON)
      -- Set 'loved_one' to 'p'.
    do
      loved_one := p
    ensure
      loved_one_set: loved_one = p
    end

invariant
  has_name: name /= Void and then not name.is_empty
end
```

Below is the code of the feature *tryout*. It contains a number of declarations and creation instructions, and it is defined in a class different from *PERSON*. All features of class *PERSON* as shown above are accessible by feature *tryout*.

```
tryout
    -- Tryout assignments
  local
    i, j: INTEGER
    a, b, c: PERSON
  do
    create a.make ("Anna")
    create b.make ("Ben")
    create c.make ("Chloe")
    a.set_loved_one (b)
    b.set_loved_one (c)
    -- Here the code snippets from below are added
  end
```

To do

You will find a number of subtasks. Each contains a code snippet and statements. Assume that the code snippet is inserted at the location indicated in feature *tryout* above.

If the code snippet produces, in your opinion, a compiler error, choose option (a). If it doesn't produce a compiler error, decide for each statement whether it is correct or incorrect after the code snippet has been fully executed. This means that you can have more than one correct statement (provided the compilation went fine!). To make the answers easier to read, we call `Anna` the object whose *name* attribute is set to "Anna", and accordingly `Ben` and `Chloe` for subtasks 6 – 9.

1.	$j := 3$ $i := j$ $2 := i$	(a) The compiler reports an error. (b) i has value 2, j has value 3. (c) i and j have both value 2. (d) i and j have both value 3.
2.	$i := 7$ $j := 2$ $i := i + 3$	(a) The compiler reports an error. (b) i has value 7 and j has value 2. (c) i has value 5 and j has value 2. (d) i has value 10 and j has value 2.
3.	$i := -7$ $j := 5$ $i := j$ $j := i$	(a) The compiler reports an error. (b) i has value -7 and j has value 5. (c) i and j have both value -7. (d) i and j have both value 5.
4.	$j := 8$ $i := 19$ $j := i$	(a) The compiler reports an error. (b) i and j have both value 19. (c) j has value 19 and i holds no value any more. (d) i and j have both value 8. (e) i has value 8 and j has value 19.
5.	$i := 5$ $j := i + 7$ $i := 8$	(a) The compiler reports an error. (b) i and j have both value 8. (c) i has value 8 and j has value 12. (d) i has value 8 and j has value 15.
6.	$b := a$ $a := b$	(a) The compiler reports an error. (b) a and b are both attached to `Ben`. (c) a is a void reference and b is attached to `Anna`. (d) b is attached to `Anna` and a to `Ben`. (e) a and b are both attached to `Anna`.
7.	$b := a.loved\_one$ $b.set\_loved\_one\ (a.loved\_one)$ $a.set\_loved\_one\ (c)$	(a) The compiler reports an error. (b) The attribute *loved_one* of `Ben` references `Ben`. (c) b is attached to `Chloe`. (d) a is attached to `Anna` and b to `Ben`. (e) b is attached to `Anna` and a to `Chloe`.
8.	$b := c$ $b.loved\_one := a.loved\_one$	(a) The compiler reports an error. (b) b is attached to `Chloe` and its attribute *loved_one* references `Ben`. (c) The attribute *loved_one* of `Chloe` references `Ben`. (d) b is attached to `Ben` and c to `Chloe`.
9.	$b := b.loved\_one.loved\_one$ $a.set\_loved\_one\ (c)$	(a) The compiler reports an error. (b) b is attached to `Chloe`. (c) b is Void and the attribute *loved_one* of a is attached to `Chloe`. (d) a is attached to `Anna` and b to `Ben`. (e) The object with name `Ben` is not reachable any more.

To hand in

Hand in your answers to the questions above.

3 Phone contracts

In this task you will practice reasoning about code using contracts.

The following class models a cell phone with a touch screen. If you put the phone to sleep, the screen is off and locked. To unlock the screen you first have to turn it on (to wake the phone

up). The phone also stores a list of contacts; you can add and remove contacts once the screen in unlocked.

```
class
  PHONE

feature -- Power saving, locking
  screen_on: BOOLEAN
      -- Is the screen on?

  is_locked: BOOLEAN
      -- Is the touch screen locked?

  is_ready: BOOLEAN
      -- Is the phone is ready for use?
    do ...
    ensure
      on_and_unlocked: Result = (screen_on and not is_locked)
    end

  unlock
      -- Unlock the touch screen.
    require
      on: screen_on
    do ...
    ensure
      ready: is_ready
    end

  sleep
      -- Put the phone to sleep.
    do ...
    ensure
      off: not screen_on
      locked: is_locked
    end

  wake
      -- Wake the phone up.
    do ...
    ensure
      on: screen_on
      locked_unchanged: locked = old locked
    end

feature -- Contacts
  contact_count: INTEGER
      -- Number of contacts in the phone.

  has (name: STRING): BOOLEAN
      -- Is 'name' present in the contacts?
      -- (Uses value equality).
```

```
    require
      ready: is_ready
      name_exists: name /= Void
    do ...
    ensure
      no_contacts: (contact_count = 0) implies not Result
    end

  add (name: STRING)
      -- Add 'name' to the contacts.
    require
      ready: is_ready
      name_exists: name /= Void
      fresh_name: not has (name)
    do ...
    ensure
      entered: has (name)
      one_more_contact: contact_count = old contact_count + 1
      still_ready: is_ready
    end

  remove (name: STRING)
      -- Remove 'name' from the contacts.
    require
      ready: is_ready
      name_exists: name /= Void
      already_there: has (name)
    do ...
    ensure
      removed: not has (name)
      one_less_contact: contact_count = old contact_count - 1
      still_ready: is_ready
    end

invariant
  contact_count_non_negative: contact_count >= 0
end
```

The bodies of functions have been removed, this is deliberate! Your reasoning should be based entirely on the pre- and post-conditions and the class invariant.

To do

Answer the following questions, and include justification for your answer.

1. Given that the command *sleep* has just been completed successfully, does its post-condition allow us to run *unlock*?

2. Given that the command *add* (**"Fred"**) has just been completed successfully, does its post-condition allow us to run *sleep*?

3. Given that the command *remove* (**"Jen"**) has just been completed successfully, does its post-condition allow us to run *add* (**"Jen"**)?

For the following features in a client of *PHONE*, does the precondition allow the body to execute successfully, and also guarantee the postcondition to be satisfied? If the precondition is not strong enough to ensure the correct completion of the routine, then propose another precondition, which can ensure this.

4.
```
use (phone: PHONE)
  require
    two_contacts: phone.contact_count = 2
  do
    phone.remove ("Ulrich")
  ensure
    one_contact: phone.contact_count = 1
  end
```

5.
```
use (phone: PHONE)
  require
    phone_exists: phone /= Void
    ready: phone.is_ready
    no_contacts: phone.contact_count = 0
  do
    phone.add ("Ulrich")
  ensure
    one_contact: phone.contact_count = 1
    has_ulrich: phone.has ("Ulrich")
  end
```

6.
```
use (phone: PHONE)
  require
    phone_exists: phone /= Void
    ready: phone.is_ready
  do
    phone.add ("Ulrich")
  ensure
    positive_contacts: phone.contact_count > 0
  end
```

To hand in

Hand in your answers to the questions above.

4 Board game: Part 1

In this task you will start a small project from scratch. We will proceed in iterations, starting with a simplified problem and then progressively enriching it. This first part will focus on choosing the right classes.

The idea is to program a prototype of a board-game[1]. It comes with a *board*, divided into 40 *squares*, a pair of *dice*, and can accommodate 2 to 6 *players*. It works as follows:

- All players start from the first square.

[1] We draw inspiration from a case study in the excellent book by Craig Larman "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)".

- One at the time, players take a *turn*: roll the dice and advance their respective *tokens* on the board.
- A *round* consists of all players taking their turns once.
- The winner will be the player that first advances beyond the 40th square.

To do

Make up a list of classes that you would use to model the board game.

Choosing right abstractions

To suggest classes, you have to ask yourself: *What are the relevant abstractions in the problem domain?* A good source of abstractions is the problem description: names (nouns) in the description often identify important concepts in the problem domain.

Unfortunately, some names in the problem description might not deserve to become classes (like "idea" or "program"); also there might be relevant abstractions that are not expressed as names in the specific text we are looking at.

Whether an entity in the problem domain deserves its own class, depends on its relevant properties and behavior. If you are modeling a door, whose only relevant property is being open or closed, use a boolean variable. If you are programming a game with trapdoors and magic doors that trigger special behavior, then you might need a class for it. Thus the second question you should ask yourself about each candidate abstraction is: *Is there any meaningful data (attributes) and behavior (routines) associated with the abstraction?*

Finally, you should take into account that the problem description (the requirements) is almost never final. When reading the description think about things that are likely to change and new functionality that is likely to be added. Sometimes a concept doesn't have enough associated behavior in the present version of the requirements, but if you think it is likely to gain more in the future, it might still deserve its own class.

To hand in

Hand in your candidate list of class names together with short descriptions of their associated properties and behavior.