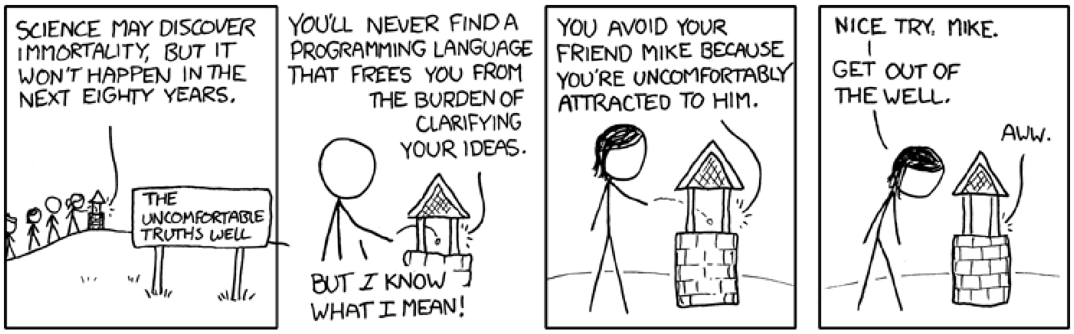
Assignment 7: Inheritance and polymorphism

ETH Zurich

Hand-out: Friday, 4 November 2011
Due: Tuesday, 15 November 2011


SCIENCE MAY DISCOVER IMMORTALITY, BUT IT WON'T HAPPEN IN THE NEXT EIGHTY YEARS.
THE UNCOMFORTABLE TRUTHS WELL
YOU'LL NEVER FIND A PROGRAMMING LANGUAGE THAT FREES YOU FROM THE BURDEN OF CLARIFYING YOUR IDEAS.
BUT I KNOW WHAT I MEAN!
YOU AVOID YOUR FRIEND MIKE BECAUSE YOU'RE UNCOMFORTABLY ATTRACTED TO HIM.
NICE TRY, MIKE.
GET OUT OF THE WELL.
AWW.

Well 2 © Randall Munroe (http://xkcd.com/568/)

Goals

- Understand polymorphism and dynamic binding.
- Practice inheritance.
- Continue the design and implementation of the board game.

1 Polymorphism and dynamic binding

Review polymorphic attachment and dynamic binding (Touch of Class, sections 16.2 and 16.3).

The following classes represent various kinds of traffic participants. Figure 1 shows the class hierarchy. The listing below shows the source code of the classes.

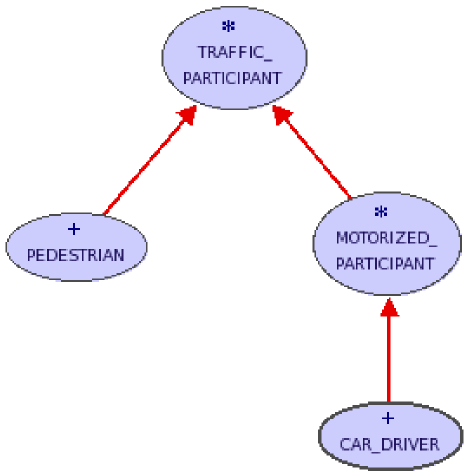


Figure 1: Class diagram for *TRAFFIC_PARTICIPANT* and its descendants.

Listing 1: Class *TRAFFIC_PARTICIPANT*

```
deferred class
  TRAFFIC_PARTICIPANT

feature -- Access
  name: STRING
      -- Name.

feature {NONE} -- Initialization
  make (a_name: STRING)
      -- Initialize with 'a_name'.
    require
      a_name_valid: a_name /= Void and then not a_name.is_empty
    do
      name := a_name
    ensure
      name_set: name = a_name
    end

feature -- Basic operations
  move (distance: REAL)
      -- Move 'distance' km.
    require
      distance_geq_zero: distance >= 0.0
    deferred
    end

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

Listing 2: Class MOTORIZED_PARTICIPANT

```
deferred class
  MOTORIZED_PARTICIPANT

inherit
  TRAFFIC_PARTICIPANT
    rename
      move as ride
    end

feature {NONE} -- Initialization
  make_with_device (a_name, a_device: STRING)
      -- Initialize with 'a_name' and 'a_device'.
    require
      a_device_valid: a_device /= Void and then not a_device.is_empty
      a_name_valid: a_name /= Void and then not a_name.is_empty
    do
      make (a_name)
      device := a_device
    ensure
      device_set: device = a_device
      name_set: name = a_name
    end

feature -- Access
  device: STRING
      -- Device name.

feature -- Basic operations
  ride (distance: REAL)
      -- Ride 'distance' km.
    do
      io.put_string (name + " rides on a " + device + " " + distance.out + " km")
    end

invariant
  device_valid: device /= Void and then not device.is_empty
end
```

Listing 3: Class CAR_DRIVER

```
class
  CAR_DRIVER

inherit
  MOTORIZED_PARTICIPANT
    rename
      make_with_device as make_with_car,
      ride as drive
    redefine
      drive
    end
```

```
create
   make_with_car

feature -- Basic operations
   drive (distance: REAL)
         -- Drive car for 'distance' km.
      do
         io.put_string (name + " drives " + device + " " + distance.out + " km")
      end
end
```

Listing 4: Class *PEDESTRIAN*

```
class
   PEDESTRIAN

inherit
   TRAFFIC_PARTICIPANT
      rename
         move as walk
      end

create make

feature -- Basic operations
   walk (distance: REAL)
         -- Walk 'distance' km.
      do
         io.put_string (name + " walks " + distance.out + " km")
      end
end
```

To do

Given the variable declarations

```
traffic_participant: TRAFFIC_PARTICIPANT
motorized_participant: MOTORIZED_PARTICIPANT
car_driver: CAR_DRIVER
pedestrian: PEDESTRIAN
```

for each of the code fragments below decide whether it compiles. If not, why? If yes, what does it print? This is a pen-and-paper task; you are not supposed to use EiffelStudio.

Example:

```
create {CAR_DRIVER} traffic_participant.make ("Bob", "Seat")
traffic_participant.drive (7.8)
```

The code does not compile, because the feature *make* is not a creation procedure of class *CAR_DRIVER*. Additionally, the static type of *traffic_participant* offers no feature *drive*.

1.
   ```
   create {CAR_DRIVER} motorized_participant.make_with_device ("Louis", "BMW")
   motorized_participant.ride (3.2)
   ```

2.
   ```
   create motorized_participant.make_with_device ("Sue", "bus")
   motorized_participant.ride (4.2)
   ```

3.
   ```
   create {PEDESTRIAN} traffic_participant.make ("Julie")
   traffic_participant.move (0.5)
   ```

4.
   ```
   create {MOTORIZED_PARTICIPANT} car_driver.make_with_car ("Ben", "Audi")
   car_driver.drive (12.3)
   ```

5.
   ```
   create {PEDESTRIAN} traffic_participant.make ("Jim")
   pedestrian := traffic_participant
   pedestrian.walk (1.9)
   ```

6.
   ```
   create {CAR_DRIVER} traffic_participant.make_with_car ("Anna", "Mercedes")
   traffic_participant.drive (3.1)
   ```

7.
   ```
   create car_driver.make_with_car ("Megan", "Renault")
   motorized_participant := car_driver
   motorized_participant.ride (17.8)
   ```

To hand in

Hand in your answers for the code fragments above.

2 Ghosts in Zurich

Ghosts are taking over Zurich! In this task you will implement a special kind of mobile object: a *GHOST*. Ghosts in Traffic have the following behavior: they choose a station of the city and then fly around it in circles.

To do

1. Download `http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/07/traffic.zip`, unzip it and open `assignment_7.ecf`.

2. Create a new class *GHOST* and make it inherit from *MOBILE*. The latter has three deferred features: *position*, *speed* and *move_distance*, which you have to implement before you can successfully compile your class. For the first two features you have a choice of making them into either an attribute or a function. The third one should be implemented as a procedure that calculates where the ghost ends up when it moves from the current position by *d* meters. You can assume that all ghosts always move at the same speed (e.g. 10 meters per second).

   You'll probably also want to add new features to *GHOST*, for example to store the station that it is flying around and the distance it keeps from the station (the radius of its circular trajectory). Additionally you'll need a creation procedure that takes the station and the radius as arguments.

   In order to make the implementation as simple as possible, first think about the easiest way to represent circular motion (maybe you would want to draw it on paper first). **Hint:** It's convenient to represent the ghost position at any point in time as a sum of two vectors.

3. In the class *GHOST_INVASION* implement a feature *add_ghost* (*s*: *STATION*; *r*: *REAL_64*) that creates a ghost flying around a station *s* at a distance *r* and adds it to Zurich (using the feature *add_custom_mobile*). Don't forget to update the map in order to create the view for the new ghost. After that, modify the view so that the ghost is depicted as an icon instead of the default black dot; you can use "ghost.png" from the "images" directory for the icon. The expression *Zurich_map.custom_mobile_view* (*ghost*) gives you access to the view of the object *ghost*.

   Test the *add_ghost* feature by calling it from *invade* with arguments of your choice. To make the ghost move, double-click on the map.

4. Modify the feature *invade* so that it generates 10 ghosts flying around random stations of Zurich at a random distance between 10 and 100 meters (you don't have to check that all stations are different). To access stations by integer index, create a cursor that iterates through the stations and call the command *go_to* on that cursor.

To hand in

Hand in classes *GHOST* and *GHOST_INVASION*.

3 Board game: Part 3

In this task you will extend the implementation of the board game. You will find an updated problem description below.

The board game comes with a *board*, divided into 40 *squares*, a pair of six-sided *dice*, and can accommodate 2 to 6 *players*. It works as follows:

- All players start from the first square.
- One at a 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.
- Players have *money*. Each player starts with 7 CHF.
- The amount of money changes when a player lands on a special square:
  - Squares 5, 15, 25, 35 are *bad investment* squares: a player has to pay 5 CHF. If the player cannot afford it, he gives away all his money.
  - Squares 10, 20, 30, 40 are *lottery win* squares: a player gets 10 CHF.
- The winner is the player with the most money after the first player advances beyond the 40th square. Ties (multiple winners) are possible.

To do

Modify the implementation of the board game in such a way that it accommodates the changes in the problem description (money, special squares, new winning criterion). We recommend that you start from the master solution to the assignment 6: `http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/07/board_game.zip`[1].

[1] The solution depends on the EiffelBase2 library being located in the standard library directory. If you did **not** put it there while solving the previous assignment, the project will not compile. In this case go to `Project->Project Settings`, choose `Groups->Libraries->base2` in the tree on the left and then change the `Location` property on the right to wherever you put the library. If you did not download the library at all, refer to Assignment 6, Task 2 for instructions.

Hints

Are there entities in the problem domain that didn't have enough properties and behavior to deserve their own classes in the previous version of the game, but that gained some properties or behavior in the current version? You might want to introduce new classes for such entities.

Bad investment and lottery win squares are special cases of squares, which differ in a way they affect players. To model this you can introduce class *SQUARE* and then use inheritance and feature redefinition to implement the behavior of special squares. You can store squares of all kinds in a single polymorphic container (e.g. *V_ARRAY* [*SQUARE*]) and let dynamic binding take care of which special behavior applies for each square.

To hand in

Hand in the code of your classes.