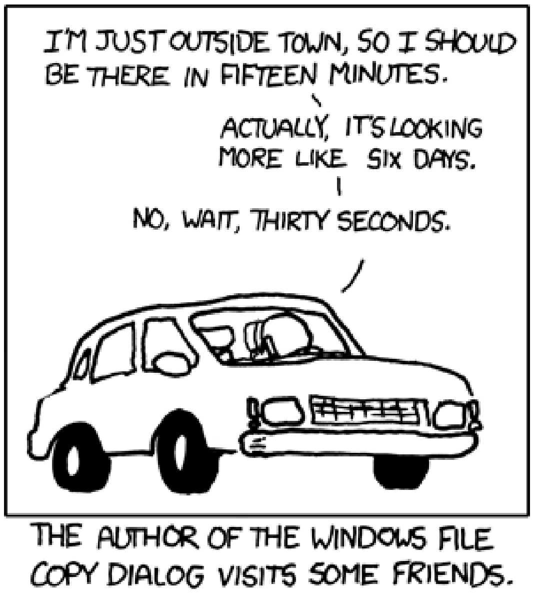
Assignment 8: Inheritance and polymorphism

ETH Zurich

Hand-out: 12 November 2010
Due: 23 November 2010


I'M JUST OUTSIDE TOWN, SO I SHOULD BE THERE IN FIFTEEN MINUTES.
ACTUALLY, IT'S LOOKING MORE LIKE SIX DAYS.
NO, WAIT, THIRTY SECONDS.
THE AUTHOR OF THE WINDOWS FILE COPY DIALOG VISITS SOME FRIENDS.

Estimation © Randall Munroe (xkcd.com)

Goals

- Understand polymorphic assignment, polymorphic creation and dynamic binding.
- Practice inheritance.
- Continue the design and implementation of the board game.

1 Dynamic binding and polymorphic attachment

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.

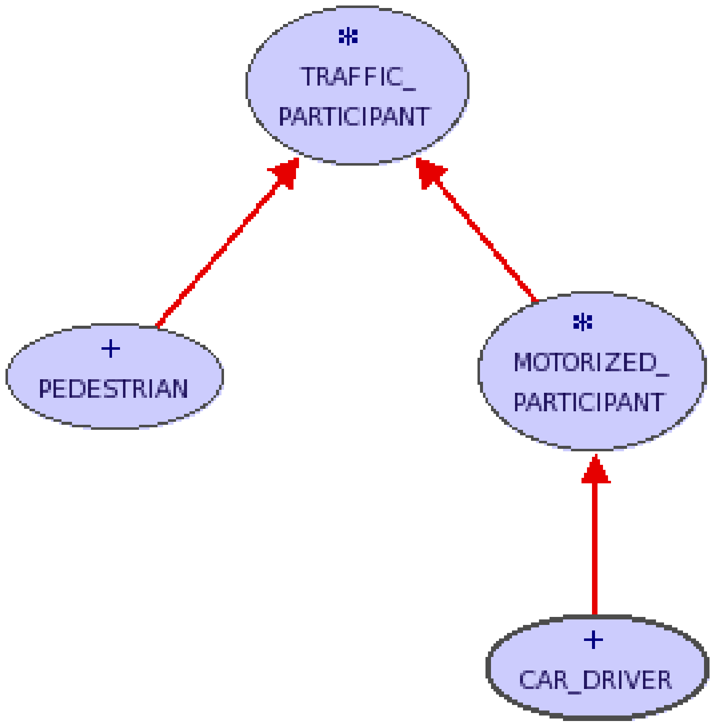


Figure 1: Class diagram for *TRAFFIC_PARTICIPANT* and its descendants.

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 using 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 Paris

Ghosts are taking over Paris! In this task you will implement a special kind of free moving objects: a *TRAFFIC_GHOST*. Ghosts in Traffic have the following behavior: they choose a station of the city and then move on a square around this station.

To do

1. Download http://se.ethz.ch/teaching/2010-H/eprog-0001/assignments/08/assignment_8.zip and extract it in `traffic/example`. You should now have a new directory `traffic/example/assignment_8` with assignment_8.ecf directly in it.

2. Open and compile this new project.

3. Create a class *TRAFFIC_GHOST* inheriting from *TRAFFIC_FREE_MOVING*. Implement a creation procedure *make* with two arguments: a station around which the ghost is flying and a side length for its square path.

   Call the procedure *make_with_points* from *make*. First you will need to create a list of points containing the edges of the square to pass into *make*. This list should be of type *DS_ARRAYED_LIST* [*TRAFFIC_POINT*]. Note that you will have to add the first point twice: once at the beginning of the list and once at the end. For the speed argument of *make_with_points* choose a value between 5.0 and 30.0. Make the ghost reiterate.

   Test your implementation by creating an instance of *TRAFFIC_GHOST* and adding it to Paris using the feature *put_free_moving*. Don't forget to call *start* on it.

4. The implementation of *TRAFFIC_FREE_MOVING* makes a reiterating object move backwards through the set of points once the last point is reached. Instead we want the ghosts to move around the station always in the same direction.

   When a free moving starts to move, the feature *move_next* is called. It takes the first point from the list (available through the list cursor *poly_cursor*) as *origin* and the second as *destination*. The feature *advance* then takes over and lets the object move stepwise from origin to destination until the destination is reached. At this point the feature *move_next* is called again. It updates the *origin* to be the former *destination* and the next point in the list becomes the new *destination*. When the end of the list is reached *move_next* of *TRAFFIC_FREE_MOVING* begins to iterate through the list in the reverse order.

   Redefine feature *move_next* in *TRAFFIC_GHOST* in such a way that when the end of the list is reached it will start at the beginning again.

5. Implement the feature *invade* of class *GHOST_INVASION*. It should generate 10 ghosts set to randomly selected stations of Paris. For this you will need to generate a random number that is within the bounds of the indices of stations of Paris. To access stations by index, convert the table of stations into an array using *Paris.stations.to_array*.

To hand in

Hand in classes *TRAFFIC_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.ethz.ch/teaching/2010-H/eprog-0001/assignments/08/board_game.zip.

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. *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.