Assignment 8: Polymorphic Behaviors

ETH Zurich

Hand-out: 13. November 2008
Due: 24. November 2008

Goals

- Understand polymorphic assignment, polymorphic creation and dynamic binding.
- Practice inheritance within Traffic.
- Continue the design and implementation of the boardgame exercise again with some inheritance included.

1 Dynamic binding and polymorphic attachment

Things you need to know

Polymorphism is the ability of an element of the software text to denote, at run time, objects of two or more possible types. Polymorphism appears in several forms. In this task, we will have a look at polymorphic assignment and polymorphic creation: Two mechanisms that result in polymorphic attachment.

Polymorphic attachment denotes the ability of an entity that is declared to be of a certain type T to become attached to objects of various types (descendants of T). The type T given in the declaration of an entity is called its *static type*, while the generating type of the object that the entity is attached to at run time is called its *dynamic type*.

For the following explanations, assume that you have an entity x declared of type T and class S that is a proper descendant of T.

Polymorphic assignment allows assignments where the dynamic type of the object returned by the source expression (right-hand side) is different from the static type of the target (left-hand side). The assignment instruction is written as

$x := e$

where x is a writable entity and e an expression of compatible type. Non-polymorphic assignment requires the result of e to be of the same type as x. With polymorphic assignments this restriction can be weakened and e needs to return an object of a descendant type of x (this includes the static type of x and all classes that inherit from it).

Polymorphic creation allows to directly create an object of a descendant type and polymorphically attach it to an entity of a certain static type. The general form is written as

create {S} x.*creation_procedure*

where x is a writable entity, S the class name of a descendant of x's static type T and *creation_procedure* is a creation procedure of class S.

Note that generally the static type T of an entity like x defines the features that can be applied to the entity as target. The polymorphic creation instruction above however requires a valid creation procedure of the dynamic type S of x. This is a special case that you should remember.

Another thing that you should remember is that although the static type defines the available features, it is the dynamic type that decides the proper version to be called if for example a feature has been redefined in the descendants T.

Task description

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.

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
```

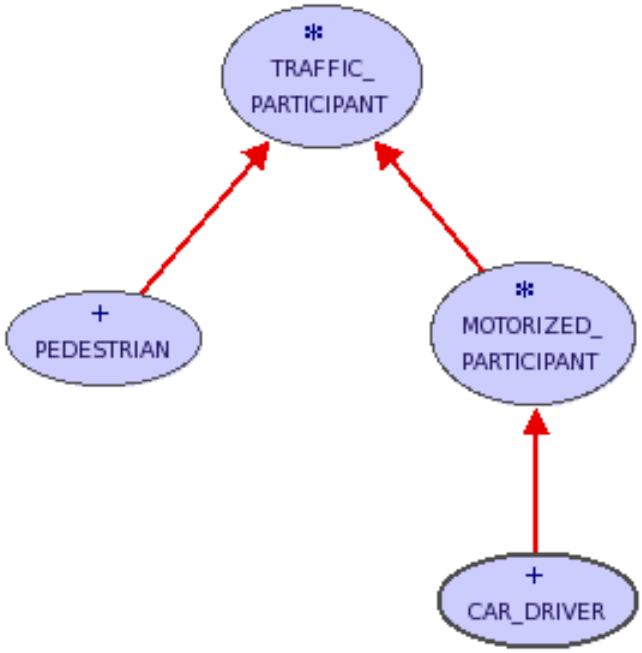

Figure 1: Class diagram for class *TRAFFIC_PARTICIPANT* and its descendants.

```
18     end

20 feature -- Basic operations

22   move (distance: REAL)
         -- Move 'distance' km.
24     require
         distance_geq_zero : distance >= 0.0
26     deferred
       end
28
   invariant
30
     name_valid: name /= Void and then not name.is_empty
32 end
```

Listing 2: Class *MOTORIZED_PARTICIPANT*

```
   deferred class MOTORIZED_PARTICIPANT
 2
   inherit
 4
     TRAFFIC_PARTICIPANT
 6     rename
         move as ride
 8     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 in Tasks 1.1 - 1.7 below analyze the code and do the following:

- If you think the code compiles, put a checkmark in the corresponding box and state what message (if any) will be printed to the console when it is executed.
- If you think the code does not compile, put a checkmark in the corresponding box and explain why it is invalid.

This is a pen-and-paper task and the idea is that you analyze the code by hand without using EiffelStudio.

Example:

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

Does the code compile? ☐ Yes ☒ No
Output/error description: 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*.

Task 1.1

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

Does the code compile? ☐ Yes ☐ No
Output/error description .....

Task 1.2

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

Does the code compile? ☐ Yes ☐ No
Output/error description .....

Task 1.3

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

Does the code compile? ☐ Yes ☐ No
Output/error description .....

Task 1.4

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

Does the code compile? ☐ Yes ☐ No
Output/error description .....

Task 1.5

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

Does the code compile? ☐ Yes ☐ No
Output/error description .....

Task 1.6

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

Does the code compile? ☐ Yes ☐ No
Output/error description .....

Task 1.7

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

Does the code compile? ☐ Yes ☐ No
Output/error description .....

To hand in

Submit your answers to your assistant.

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 (somewhat erratic) 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/2009-H/eprog-0001/exercises/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 (it is important that the location corresponds to the description here!).

2. Open and compile this new project.

3. In a first step, write a class *TRAFFIC_GHOST* inheriting from *TRAFFIC_FREE_MOVING* that will move around a station. To do this, define a creation feature called *make* that has two arguments: a station and a side length for calculating the square path. Call the creation procedure *make_with_points* of *TRAFFIC_FREE_MOVING* in *make*. Before calling *make_with_points*, you will need to create a list of points containing the edges of the square to pass it as argument. The local variable for the list of points 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 sure to let the ghost reiterate (call *set_reiterate (True)* from the creation procedure). Test your implementation by creating an instance of *TRAFFIC_GHOST* and adding it to Paris using its feature *put_free_moving*. Don't forget to call *start* on it.

4. The implementation of *TRAFFIC_FREE_MOVING* lets an object that is reiterating move backwards through the set of points if the last one is reached, but we want the ghosts to move around the station always with the same orientation. The moving of an object happens as follows: when starting to move, it takes the first point of the list (available through the list cursor *poly_cursor*) as *origin* and the second as *destination*. This is done in the feature *move_next*. 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 and it updates the *origin* to be the former *destination* and the next point in the list becomes the new *destination*. If the end of the list is reached then *move_next* of *TRAFFIC_FREE_MOVING* begins to iterate through the list again in reverse order. Redefine this behavior for *TRAFFIC_GHOST* such 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 get access to a station by index, use the feature *to_array* of *TRAFFIC_ITEM_HASH_TABLE* to convert the hash table to an array.

To hand in

Hand in class *TRAFFIC_GHOST* and *GHOST_INVASION*.

3 Programming a boardgame: Part 3

To do

In this task you will extend the implementation of the system by adding new classes and features. Here is the new description, updated with the new specifications:

Problem Description

The idea is to program a prototype of a board-game[1]. It comes with a board, divided into 40 squares, a pair of six-sided dice, and can accommodate 2 to 6 players. It works like this: all players start from the first square. One at the time, players take a turn. This includes rolling the dice and advance their respective tokens on the board. When all players are done with their turn, it is called a round. Players have now money. Each player starts with an amount of 7 CHF. There are the following special squares:

- Squares 5, 15, 25, 35 are "Bad Investment" squares. The effect on a player that lands on them is to subtract 5 CHF from that player's current owned amount of money, if the player can afford it, otherwise as much as it is possible is subtracted. The amount of money a player owns cannot go below zero.
- Squares 10, 20, 30, 40 are "Lottery Win" squares. The effect on a player that lands on them is too add 10 CHF to that player's amount.

The winner will be the player that has more money after the first player advances beyond the 40th square.

Hints

To have a common starting point, You will start from the classes provided in the solution to assignment 6. Download http://se.ethz.ch/teaching/2009-H/eprog-0001/exercises/boardgame1.zip.

- Players have money, so an attribute should be added to class *PLAYER*. An *INTEGER* will work just fine in this case. Also, don't forget to add features to debit or credit the player for a certain amount.
- The new squares look like a specialization of class *SQUARE*, so inheritance is a viable option. A specific behavior is required when a player lands on them.
- If you have doubts on how to write the code to determine the winner, you may have a look at Touch of Class, section 7.5. The example on finding the maximum should help.

To hand in

Submit the code of your classes.

[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)