

Assignment 8: Ghosts

ETH Zurich

Hand-out: 14. November 2008
Due: 20. November 2008

Goals

- Understand polymorphic assignment, polymorphic creation and dynamic binding.
- Implement a class that inherits from another class and redefines some of its features.
- Work with a deferred class.

1 Dynamic binding and polymorphic attachment

Things you need to know

Polymorphism is the ability for 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.

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.

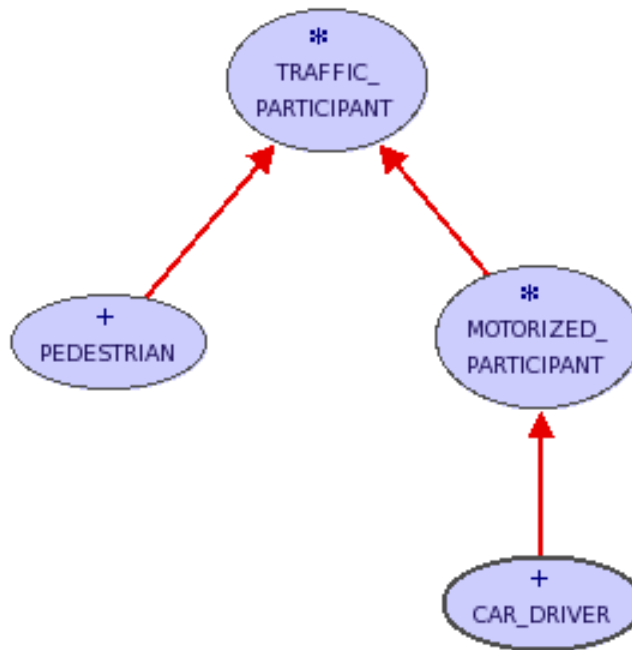


Figure 1: Class diagram for class *TRAFFIC_PARTICIPANT* and its descendants.

Listing 1: Class *TRAFFIC_PARTICIPANT*

```
deferred class TRAFFIC_PARTICIPANT
2
feature -- Access
4
  name: STRING
6   -- Name

8 feature {NONE} -- Initialization
10 make (a_name: STRING) is
    -- Initialize with 'a_name'.
12   require
    a_name_valid: a_name /= Void and then not a_name.is_empty
14   do
```

```
    name := a_name
16  ensure
    name_set: name = a_name
18  end

20 feature -- Basic operations

22  move (distance: REAL) is
    -- 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

10 feature {NONE} -- Initialization

12  make_with_device (a_name, a_device: STRING) is
    -- Initialize with 'a_name' and 'a_device'.
14  require
    a_device_valid: a_device /= Void and then not a_device.is_empty
16  a_name_valid: a_name /= Void and then not a_name.is_empty
    do
18  make (a_name)
    device := a_device
20  ensure
    device_set: device = a_device
22  name_set: name = a_name
    end
24
feature -- Access
26  device: STRING
28  -- Device name

30 feature -- Basic operations
32  ride (distance: REAL) is
```

```
    -- Ride 'distance' km.
34  do
    io.put_string (name + " rides on a " + device + " " + distance.out + " km")
36  end

38 invariant

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

Listing 3: Class *CAR_DRIVER*

```
class CAR_DRIVER
2
inherit
4
  MOTORIZED_PARTICIPANT
6  rename
  make_with_device as make_with_car,
8  ride as drive
  redefine
10  drive
  end
12
create
14  make_with_car

16 feature -- Basic operations

18  drive (distance: REAL) is
  -- Drive car for 'distance' km.
20  do
  io.put_string (name + " drives " + device + " " + distance.out + " km")
22  end

24 end
```

Listing 4: Class *PEDESTRIAN*

```
class PEDESTRIAN
2
inherit
4
  TRAFFIC_PARTICIPANT
6  rename
  move as walk
8  end

10 create make

12 feature -- Basic operations

14  walk (distance: REAL) is
```

```
16  -- Walk 'distance' km.  
17  do  
18    io.put_string (name + " walks " + distance.out + " km")  
19  end  
20 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 that 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 that 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/2008-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 with the same orientation always. 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 Inherited Fraction

Your task is to implement class *FRACTION* which represents fractions of the form $\frac{\text{numerator}}{\text{denominator}}$. *FRACTION* inherits from *NUMERIC*. The class *FRACTION_TEST* shown in Listing 5 should work with your class without any changes.

NUMERIC is a deferred class with the following deferred features:

- *one*: like *Current*
- *zero*: like *Current*
- *divisible* (*other*: like *Current*): *BOOLEAN*
- *exponentiable* (*other*: like *Current*): *BOOLEAN*
- infix "+" (*other*: like *Current*): like *Current*
- infix "-" (*other*: like *Current*): like *Current*
- infix "/" (*other*: like *Current*): like *Current*
- infix "*" (*other*: like *Current*): like *Current*
- prefix "+": like *Current*
- prefix "-": like *Current*

A deferred class is a class that may contain deferred features. The feature declaration of a deferred feature defines its signature but leaves its implementation open so that proper descendants may provide it. Deferred classes can not be instantiated, since parts of their behavior is not specified. Since you will need to create objects of type *FRACTION*, you need to implement all the deferred features of the class *NUMERIC* in *FRACTION*.

like *Current* in the feature signatures above states that the return type or argument type of a feature is the same as the type of *Current*. In the case of *FRACTION* this means that all the like *Currents* in fact denote the type *FRACTION*.

To do

1. Create a new project with the root class *FRACTION_TEST*.
2. Copy and paste the class text in the root class from http://se.ethz.ch/teaching/2008-H/eprog-0001/exercises/fraction_test.e.
3. Add a new class *FRACTION* that inherits from *NUMERIC*, and implement the missing features. The fraction should always be reduced (gekürzt). To reduce a fraction, you can use a Greatest Common Divisor (GCD) algorithm, for example the Euclidian algorithm available at <http://se.ethz.ch/teaching/2008-H/eprog-0001/exercises/gcd.txt>. Other hints:
 - In Eiffel, integer division is done with '//', integer remainder (modulo) with '\\'.
 - Do not forget the contracts. *FRACTION* has an obvious invariant.
 - The feature *exponentiable* should always return *False*.
 - The feature *divisible* should only return *True* if the division is valid (i.e. does not result in a division through zero).
 - The feature *zero* returns an identity object for addition and subtraction, so that for fractions *f* and *zero* $(f + zero).is_equal(f)$ and $(f - zero).is_equal(f)$

- The feature `one` returns an identity object for multiplication and division, so that for fractions f and `one` $(f * one).is_equal(f)$ and $(f / one).is_equal(f)$

Listing 5: Class `FRACTION_TEST`

```
class
2  FRACTION_TEST

4 create
   make

6

8  feature -- Initialization
   a, b, c: FRACTION
10  make is
12  -- Creation procedure.
   do
14  create a.make (1, 2)
      create b.make (3, 4)
16  io.put_string ("Calculating with fractions:" + "%N%N")
      io.put_string ("a: " + a.out)
18  io.put_new_line
      io.put_string ("b: " + b.out)
20  io.put_new_line

22  c := a + b
      io.put_string ("a + b: " + c.out)
24  io.put_new_line

26  c := a - b
      io.put_string ("a - b: " + c.out)
28  io.put_new_line

30  c := a * b
      io.put_string ("a * b: " + c.out)
32  io.put_new_line

34  c := a / b
      io.put_string ("a / b: " + c.out)
36  io.put_new_line

38  io.read_line
   end
40 end -- class FRACTION_TEST
```

To hand in

Hand in the source code of your class `FRACTION`.