Assignment 8: Ghosts

ETH Zurich

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.

![Class diagram](image)

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

**Listing 1**: Class `TRAFFIC_PARTICIPANT`

```plaintext
defered class TRAFFIC_PARTICIPANT

feature -- Access

name: STRING
   -- Name

feature {NONE} -- Initialization

make (a_name: STRING) is
   -- Initialize with ‘a_name’.
require
   a_name_valid: a_name /= Void and then not a_name.is_empty
```

2

4

6

8

10

12

14
name := a_name
ensure
  name_set: name = a_name
end

feature -- Basic operations
move (distance: REAL) is
  -- Move ‘distance’ km.
require
  distance_geq_zero: distance >= 0.0
deferred
delay
invariant
  name_valid: name /= Void and then not name.is_empty
end

delay class MOTORIZED_PARTICIPANT
inhibit
inherit
  TRAFFIC_PARTICIPANT
rename
  move as ride
end

feature {NONE} -- Initialization
make_with_device (a_name, a_device: STRING) is
  -- 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
declare
feature -- Access
  device: STRING
  -- Device name
feature -- Basic operations
ride (distance: REAL) is
−− Ride ‘distance’ km.
34  do
36   io. put_string (name + ” rides on a ” + device + ” ” + distance.out + ” km”)
38 end

Listing 3: Class CAR\_DRIVER

class CAR\_DRIVER
2 inherit
4   \textit{MOTORIZED\_PARTICIPANT}
6   rename
   make\_with\_device as make\_with\_car,
8   ride as drive
10 redefine
12   drive
14 end

create
16   make\_with\_car

feature −− Basic operations
18   drive (distance: REAL) is
20     −− Drive car for ‘distance’ km.
22   do
24     io. put_string (name + ” drives ” + device + ” ” + distance.out + ” km”)
26 end

Listing 4: Class PEDESTRIAN

class PEDESTRIAN
2 inherit
4   \textit{TRAFFIC\_PARTICIPANT}
6   rename
   move as walk
8 end

10 create make

12 feature −− Basic operations
14   walk (distance: REAL) is
-- Walk ‘distance’ km.
16   do
18      io.put_string (name + ” walks ” + distance.out + ” km”)
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 \textit{FRACTION} which represents fractions of the form \( \frac{\text{numerator}}{\text{denominator}} \). \textit{FRACTION} inherits from \textit{NUMERIC}. The class \textit{FRACTION\_TEST} shown in Listing 5 should work with your class without any changes.

\textit{NUMERIC} is a deferred class with the following deferred features:

- \texttt{one}: like \texttt{Current}
- \texttt{zero}: like \texttt{Current}
- \texttt{divisible} (\texttt{other}: like \texttt{Current}): \texttt{BOOLEAN}
- \texttt{exponentiable} (\texttt{other}: like \texttt{Current}): \texttt{BOOLEAN}
- \texttt{infix ”+”}(\texttt{other}: like \texttt{Current}): like \texttt{Current}
- \texttt{infix ”−”}(\texttt{other}: like \texttt{Current}): like \texttt{Current}
- \texttt{infix ”/”}(\texttt{other}: like \texttt{Current}): like \texttt{Current}
- \texttt{infix ”∗”}(\texttt{other}: like \texttt{Current}): like \texttt{Current}
- \texttt{prefix ”+”}: like \texttt{Current}
- \texttt{prefix ”−”}: like \texttt{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 \textit{FRACTION}, you need to implement all the deferred features of the class \textit{NUMERIC} in \textit{FRACTION}.

like \texttt{Current} in the feature signatures above states that the return type or argument type of a feature is the same as the type of \texttt{Current}. In the case of \texttt{FRACTION} this means that all the like \texttt{Current}s in fact denote the type \texttt{FRACTION}.

To do

1. Create a new project with the root class \textit{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 \textit{FRACTION} that inherits from \textit{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. \textit{FRACTION} has an obvious invariant.
- The feature \texttt{exponentiable} should always return \texttt{False}.
- The feature \texttt{divisible} should only return \texttt{True} if the division is valid (i.e. does not result in a division through zero).
- The feature \texttt{zero} returns an identity object for addition and subtraction, so that for fractions \( f \) and \texttt{zero} \( (f + \text{zero}).is\_equal(f) \) and \( (f - \text{zero}).is\_equal(f) \).
The feature `one` returns an identity object for multiplication and division, so that for fractions \( f \) and \( \text{one} \) \((f \times \text{one}).\text{is_equal}(f)\) and \((f/\text{one}).\text{is_equal}(f)\)


Listing 5: Class `FRACTION_TEST`

class `FRACTION_TEST`
4 create
6 make
8 feature -- -- Initialization
10 a, b, c: FRACTION
12 make is
14 -- -- Creation procedure.
16 do
18 create a.make (1, 2)
20 create b.make (3, 4)
22 io.put_string ("Calculating with fractions:" + "%%N%%N")
24 io.put_string ("a: " + a.out)
26 io.put_string ("b: " + b.out)
28 end
30 c := a + b
32 io.put_string ("a + b: " + c.out)
34 io.put_new_line
36 c := a - b
38 io.put_string ("a - b: " + c.out)
40 io.put_new_line
42 c := a * b
44 io.put_string ("a * b: " + c.out)
46 io.put_new_line
48 c := a / b
50 io.put_string ("a / b: " + c.out)
52 io.put_new_line
54 end -- -- class FRACTION_TEST

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