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.
Listing 1: Class \textit{TRAFFIC\_PARTICIPANT}

\begin{verbatim}
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
\end{verbatim}
deferred class
  
  \textit{MOTORIZED\_PARTICIPANT}

inherit
  \textit{TRAFFIC\_PARTICIPANT}
  
  rename
    move as ride
  end

feature \{NONE\} -- Initialization
  
  make\_with\_device \(\langle a\_name, a\_device: STRING\rangle\)
    -- 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 \(\langle distance: REAL\rangle\)
    -- 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 \textit{CAR\_DRIVER}

class
  \textit{CAR\_DRIVER}

inherit
  \textit{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


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 \texttt{GHOST\_INVASION} implement a feature \texttt{add\_ghost} \((s: \texttt{STATION}; r: \texttt{REAL}\_64)\) that creates a ghost flying around a station \(s\) at a distance \(r\) and adds it to Zurich (using the feature \texttt{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 \texttt{Zurich\_map.custom\_mobile\_view(ghost)} gives you access to the view of the object \texttt{ghost}.

Test the \texttt{add\_ghost} feature by calling it from \texttt{invade} with arguments of your choice. To make the ghost move, double-click on the map.

4. Modify the feature \texttt{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 \texttt{go\_to} on that cursor.

To hand in

Hand in classes \texttt{GHOST} and \texttt{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 \texttt{board}, divided into 40 \texttt{squares}, a pair of six-sided \texttt{dice}, and can accommodate 2 to 6 \texttt{players}. It works as follows:

- All players start from the first square.
- One at a time, players take a \texttt{turn}: roll the dice and advance their respective \texttt{tokens} on the board.
- A \texttt{round} consists of all players taking their turns once.
- Players have \texttt{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 \texttt{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 \texttt{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: \texttt{http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/07/board_game.zip}\footnote{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 \texttt{Project->Project Settings}, choose \texttt{Groups->Libraries->base2} in the tree on the left and then change the \texttt{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.