Solution 8: Inheritance and polymorphism

ETH Zurich

1 Dynamic binding and polymorphic attachment

1. The code does not compile. Feature `make_with_device` is unknown in `CAR_DRIVER` (it is renamed into `make_with_car`).

2. The code does not compile. Creation instruction applies to a deferred type `MOTORIZED_PARTICIPANT`.

3. The code compiles and prints “Julie walks 0.5 km”. Feature `make` is a valid creation procedure of class `PEDESTRIAN` (note the clause `create make`). Feature `move` is known in class `TRAFFIC_PARTICIPANT`. The dynamic type of `traffic_participant` is `PEDESTRIAN`; that is why the implementation of `move` from `PEDESTRIAN` (where it’s renamed into `walk`) is executed.

4. The code does not compile. First, creation instruction applies to a deferred type `MOTORIZED_PARTICIPANT`. Second, explicit creation type `MOTORIZED_PARTICIPANT` does not conform to the static type of the target `CAR_DRIVER`.

5. The code does not compile. Static type of the assignment source `TRAFFIC_PARTICIPANT` does not conform to the static type of the target `PEDESTRIAN`.

6. The code does not compile. Feature `drive` is unknown in `TRAFFIC_PARTICIPANT`.

7. The code compiles and prints “Megan drives Renault 17.8 km”. Feature `make_with_car` is a valid creation procedure of the class `CAR_DRIVER`. Static type of the assignment source `CAR_DRIVER` conforms to the static type of the target `MOTORIZED_PARTICIPANT`. Feature `ride` is known in `MOTORIZED_PARTICIPANT`. The dynamic type of `motorized_participant` is `CAR_DRIVER`; that is why the implementation of `ride` from `CAR_DRIVER` (where it’s renamed into `drive`) is executed.

2 Ghosts in Paris

Listing 1: Class `TRAFFIC_GHOST`

class
  TRAFFIC_GHOST

inherit
  TRAFFIC_FREE_MOVING
  redefine
    move
    next
end

create
make

feature -- Initialization
make (a_station: TRAFFIC_STATION; a_side: REAL_64)
   -- Create a ghost that moves around 'a_station'
   -- along a square with side 'a_side'.
require
   a_station_exists: a_station /= Void
   a_side_positive: a_side > 0.0
local
   l: DS_ARRAYED_LIST [TRAFFIC_POINT]
   p: TRAFFIC_POINT
   x, y: REAL_64
   do
      create l.make (5)
      x := a_station.location.x
      y := a_station.location.y
      create p.make (x - a_side/2, y - a_side/2)
      l.put_last (p)
      create p.make (x + a_side/2, y - a_side/2)
      l.put_last (p)
      create p.make (x + a_side/2, y + a_side/2)
      l.put_last (p)
      create p.make (x - a_side/2, y + a_side/2)
      l.put_last (p)
      create p.make (x - a_side/2, y - a_side/2)
      l.put_last (p)
      make_with_points (l, 10.0)
      set_reiterate (True)
   ensure
      reiterating: is_reiterating
   end

feature {NONE} -- Implementation
move_next
   -- Move to the next point.
   do
      -- Set the locations to the corresponding ones of the line segment.
      origin := poly_cursor.item
      location := poly_cursor.item
      if is_reiterating then
         poly_cursor.forth
         if poly_cursor.after then
            poly_cursor.start
            move_next
         else
            destination := poly_cursor.item
         end
      else
         poly_cursor.forth
      end
if poly_cursor.after then
    has_finished := True
else
    destination := poly_cursor.item
end
end
end

Listing 2: Class GHOST_INVASION

class
    GHOST_INVASION

inherit
    TOURISM

feature  -- Explore Paris
    invade  -- Invade Paris with 10 ghosts.
local
    g: TRAFFIC_GHOST
    r: RANDOM
    t: TIME
    i: INTEGER
    a: ARRAY [TRAFFIC_STATION]
do
    Paris.display
    create t.make_now
    create r.set_seed (t.milli_second)
from
    i := 1
    r.start
    a := Paris.stations.to_array
until
    i > 10
loop
    create g.make (a[i.item \ a.count + 1], 50.0)
    g.start
    Paris.put_free_moving (g)
    r.forth
    i := i + 1
end
end

3 Board game: Part 3

You can download a complete solution from

Below you will find listings of classes that changed since assignment 6.
Listing 3: Class SQUARE

class SQUARE

feature -- Basic operations
  affect (p: PLAYER)
    -- Apply square’s special effect to ‘p’.
    do
      -- For a normal square do nothing.
    end
end

Listing 4: Class BAD_INVESTMENT_SQUARE

class BAD_INVESTMENT_SQUARE

inherit SQUARE
  redefine affect
end

feature -- Basic operations
  affect (p: PLAYER)
    -- Apply square’s special effect to ‘p’.
    do
      p.transfer (-5)
    end
end

Listing 5: Class LOTTERY_WIN_SQUARE

class LOTTERY_WIN_SQUARE

inherit SQUARE
  redefine affect
end

feature -- Basic operations
  affect (p: PLAYER)
    -- Apply square’s special effect to ‘p’.
    do
      p.transfer (10)
    end
end

Listing 6: Class BOARD

class
BOARD

create
make

feature {NONE} -- Initialization
make
  -- Initialize squares.
  local
  i: INTEGER
  do
  create squares.make (1, Square_count)
  from
  i := 1
  until
  i > Square_count
  loop
  if i \ 10 = 5 then
  squares [i] := create {BAD_INVESTMENT_SQUARE}
  elseif i \ 10 = 0 then
  squares [i] := create {LOTTERY_WIN_SQUARE}
  else
  squares [i] := create {SQUARE}
  end
  i := i + 1
  end
end

feature -- Access
squares: ARRAY [SQUARE]
  -- Container for squares

feature -- Constants
Square_count: INTEGER = 40
  -- Number of squares.

invariant
  squares_exists: squares /= Void
  squares_count_valid: squares.count = Square_count
end

Listing 7: Class PLAYER

class
PLAYER
create
make

feature {NONE} -- Initialization
make (n: STRING; b: BOARD)
  -- Create a player with name ‘n’ playing on board ‘b’.
  require
```plaintext
name_exists: n /= Void and then not n.is_empty

board_exists: b /= Void

do
    name := n.twin
    board := b
    position := b.squares.lower

ensure
    name_set: name ~ n
    board_set: board = b
    at_start: position = b.squares.lower
end

feature -- Access
name: STRING
    -- Player name.

board: BOARD
    -- Board on which the player in playing.

position: INTEGER
    -- Current position on the board.

money: INTEGER
    -- Amount of money.

feature -- Moving
move (n: INTEGER)
    -- Advance 'n' positions on the board.

require
    not_beyond_start: n >= board.squares.lower - position

do
    position := position + n

ensure
    position_set: position = old position + n
end

feature -- Money
transfer (amount: INTEGER)
    -- Add 'amount’ to ‘money’.

do
    money := (money + amount).max (0)

ensure
    money_set: money = (old money + amount).max (0)
end

feature -- Basic operations
play (d1, d2: DIE)
    -- Play a turn with dice ‘d1’, ‘d2’.

require
dice_exist: d1 /= Void and d2 /= Void

do
d1.roll
```
d2.roll
move (d1.face_value + d2.face_value)
if position <= board.squares.upper then
  board.squares[position].affect (Current)
end
print (name + " rolled " + d1.face_value.out + " and " + d2.face_value.out + ", Moves to " + position.out + "", Now has " + money.out + " CHF.%N")
end

Listing 8: Class GAME

class GAME
  create
    make
      feature {NONE} -- Initialization
        make (n: INTEGER) -- Create a game with 'n' players.
          require
            n_in_bounds: Min_player_count <= n and n <= Max_player_count
          local
            i: INTEGER
            p: PLAYER
          do
            create board.make
            create players.make (1, n)
            from
              i := 1
            until
              i > players.count
            loop
              create p.make ("Player" + i.out, board)
              p.transfer (Initial_money)
              players[i] := p
              i := i + 1
            end
            create die_1.roll
            create die_2.roll
          end

        feature -- Basic operations
          play
Start a game.

local
  i: INTEGER

do
  from
  winners := Void
  until
  winners /= Void
loop
  from
  i := 1
  until
  winners /= Void or else i > players.count
loop
  players[i].play(die_1, die_2)
  if players[i].position > board.Square.count then
    select_winners
  end
  i := i + 1
end

ensure
  has_winners: winners /= Void and then not winners.is.empty
end

feature -- Constants
Min_player_count: INTEGER = 2
  -- Minimum number of players.

Max_player_count: INTEGER = 6
  -- Maximum number of players.

Initial_money: INTEGER = 7
  -- Initial amount of money of each player.

feature -- Access
board: BOARD
  -- Board.

players: ARRAY [PLAYER]
  -- Container for players.

die_1: DIE
  -- The first die.

die_2: DIE
  -- The second die.

winners: LIST [PLAYER]
  -- Winners (Void if the game if not over yet).

feature {NONE} -- Implementation
select_winners
   -- Put players with most money into ‘winners’.
local
   i, max: INTEGER
do
   create {LINKED_LIST [PLAYER]} winners.make
   from
   i := 1
   until
   i > players.count
   loop
   if players[i].money > max then
      max := players[i].money
      winners.wipe_out
      winners.extend (players[i])
   elseif players[i].money = max then
      winners.extend (players[i])
   end
   i := i + 1
end
ensure
   has_winners: winners /= Void and then not winners.is_empty
end

invariant
   board_exists: board /= Void
   players_exist: players /= Void
   number_of_players_consistent: Min_player_count <= players.count and players.count <= Max_player_count
   dice_exist: die_1 /= Void and die_2 /= Void
end

We introduced class BOARD because in the new version of the game the board has a more complicated structure (arrangement of squares of different kinds).

We went for a flexible solution that introduces class SQUARE and lets squares affect players that land on them in an arbitrary way. Classes BAD_INVESTMENT_SQUARE and LOTTERY_WIN_SQUARE define specific effects. This design would be easily extensible if other types of special squares are added, that affect not only the player’s amount of money, but also other properties (e.g. position).

A simpler solution would be not to create class SQUARE; instead of array of squares in class BOARD introduce an array of integers that represent how much money a square at certain position gives to a player. This solution is not flexible with respect to adding other kinds of special squares.

Another simpler solution would be to add a procedure affect (p: PLAYER) directly to class BOARD (instead of creating a class SQUARE and an array of squares):

affect (p: PLAYER)
do
   if p.position \ 10 = 5 then
      p.transfer (-5)
   elseif p.position \ 10 = 0 then
      p.transfer (10)
end
The disadvantage of this approach is that the logic behind all different kinds of special squares is concentrated in a single feature; it isn’t decomposed. Adding new kinds of special squares will make this feature large and complicated.