Solution 7: Inheritance and polymorphism

ETH Zurich

1 Polymorphism and dynamic binding

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 cannot be applied 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 cannot be applied 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 Zurich

Listing 1: Class *GHOST*

```
note
  description: "Ghost that flies around a station."

class
  GHOST

inherit
  MOBILE

create
```

```
  make

feature {NONE} -- Initialization

  make (a_station: STATION; a_radius: REAL_64)
      -- Create ghost flying around 'a_station' at distance 'a_radius'.
    require
      station_exists: a_station /= Void
      radius_positive: a_radius > 0.0
    do
      station := a_station
      radius := a_radius
    ensure
      station_set: station = a_station
      radius_set: radius = a_radius
    end

feature -- Access

  position: VECTOR
      -- Current position in the city.
    do
      Result := station.position + create {VECTOR}.make_polar (radius, angle)
    end

  station: STATION
      -- Station around which the ghost flies.

  radius: REAL_64
      -- Distance from 'station'.

  speed: REAL_64 = 10.0
      -- Motion speed (meters/second).

feature {NONE} -- Movement

  angle: REAL_64
      -- Angle of the current position (with respect to eastwards direction).

  move_distance (d: REAL_64)
      -- Move by 'd' meters.
    do
      angle := angle + d / radius
    end

invariant
  station_exists: station /= Void
  radius_positive: radius > 0.0
  circular_trajectory: approx_equal (position.distance (station.position), radius)
end
```

Listing 2: Class GHOST_INVASION

```
note
   description: ”Creating new objects for Zurich.”

class
   GHOST_INVASION

inherit
   ZURICH_OBJECTS

feature -- Explore Zurich

   invade
         -- Add ghosts to random stations.
      local
         i: INTEGER
         cursor: like Zurich.stations.new_cursor
         random: V_RANDOM
      do
         from
            i := 1
            cursor := Zurich.stations.new_cursor
            create random
         until
            i > 10
         loop
            cursor.go_to (random.bounded_item (1, Zurich.stations.count))
            random.forth
            add_ghost (cursor.item, random.bounded_item (10, 100))
            random.forth
            i := i + 1
         end
         Zurich_map.animate
      end

   add_ghost (a_station: STATION; a_radius: REAL_64)
         -- Add a ghost going around ‘a_station’.
      require
         a_station_exists: a_station /= Void
         a_radius_positive: a_radius > 0.0
      local
         ghost: GHOST
      do
         create ghost.make (a_station, a_radius)
         Zurich.add_custom_mobile (ghost)
         Zurich_map.update
         Zurich_map.custom_mobile_view (ghost).set_icon (”../image/ghost.png”)
      end

end
```

3 Board game: Part 3

You can download a complete solution from http://se.inf.ethz.ch/courses/2011b_fall/eprog/assignments/07/board_game_solution.zip.

Below you will find listings of classes that changed since assignment 6.

Listing 3: Class *SQUARE*

```
class
  SQUARE

inherit
  ANY
    redefine
      out
    end

feature -- Basic operations

  affect (p: PLAYER)
      -- Apply square's special effect to 'p'.
    do
      -- For a normal square do nothing.
    end

feature -- Output

  out: STRING
      -- Textual representation.
    do
      Result := "."
    end

end
```

Listing 4: Class *BAD_INVESTMENT_SQUARE*

```
class
  BAD_INVESTMENT_SQUARE

inherit
  SQUARE
    redefine
      affect,
      out
    end

feature -- Basic operations

  affect (p: PLAYER)
      -- Apply square's special effect to 'p'.
    do
      p.transfer (-5)
    end
```

```
feature -- Output

  out: STRING
      -- Textual representation.
    do
      Result := ”#”
    end

end
```

Listing 5: Class *LOTTERY_WIN_SQUARE*

```
class
  LOTTERY_WIN_SQUARE

inherit
  SQUARE
    redefine
      affect,
      out
    end

feature -- Basic operations

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

feature -- Output

  out: STRING
      -- Textual representation.
    do
      Result := ”$”
    end

end
```

Listing 6: Class *BOARD*

```
class
  BOARD

inherit
  ANY
    redefine
      out
    end

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: V_ARRAY [SQUARE]
      -- Container for squares

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

feature -- Output
  out: STRING
    do
      Result := ""
      across
        squares as c
      loop
        Result.append (c.item.out)
      end
    end

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
			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

invariant
  name_exists: name /= Void and then not name.is_empty
  board_exists: board /= Void
  position_valid: position >= board.squares.lower -- Token can go beyond the finish position,
      but not the start
  money_non_negative: money >= 0
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
        print (p.name + ” joined the game.%N”)
        i := i + 1
      end
      create die_1.roll
      create die_2.roll
    end

feature -- Basic operations

  play
      -- Start a game.
    local
      round, i: INTEGER
    do
      from
        winners := Void
        round := 1
        print (”The game begins.%N”)
        print_board
      until
        winners /= Void
      loop
        print (”%NRound #” + round.out + ”%N%N”)
        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
        print_board
        round := round + 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: V_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

  print_board
      -- Output players positions on the board.
    local
      i, j: INTEGER
    do
      io.new_line
      print (board)
      io.new_line
      from
        i := 1
      until
        i > players.count
      loop
        from
          j := 1
        until
          j >= players [i].position
        loop
          print (" ")
          j := j + 1
        end
        print (i)
        io.new_line
        i := i + 1
      end
    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
    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.