Solution 7: Inheritance and polymorphism

ETH Zurich

1 Polymorphism and dynamic binding

Task 1

```java
create warrior.make ("Thor")
warrior.level_up
```

Does the code compile?  Yes  No
“Thor is now a level 2 warrior”.

Task 2

```java
create hero.make ("Althea")
hero.level_up
```

Does the code compile?  Yes  No
Creation instruction applies to target of a deferred type.

Task 3

```java
create warrior.make ("Thor")
create healer.make ("Althea")
warrior.do_action (healer)
```

Does the code compile?  Yes  No
Class WARRIOR does not have a feature do_action.

Task 4

```java
create {HEALER} warrior.make ("Diana")
warrior.level_up
```

Does the code compile?  Yes  No
Explicit creation type HEALER does not conform to the target type WARRIOR.

Task 5

```java
create {WARRIOR} hero.make ("Thor")
hero.do_action (hero)
create {HEALER} hero.make ("Althea")
hero.do_action (hero)
```

Does the code compile?  Yes  No
“Thor attacks Thor. Does 5 damage
Althea heals Althea by 0 points”.

Task 6

```java
create {WARRIOR} hero.make ("Thor")
warrior := hero
warrior.attack (hero)
```
Does the code compile? □ Yes  ☒ No
The source of assignment (of type \textit{HERO}) does not conform to target (of type \textit{WARRIOR}).

2 Ghosts in Zurich

Listing 1: Class \textit{GHOST}

\textbf{note}
\textit{description: ”Ghost that flies around a station.”}

\textbf{class}
\textit{GHOST}

\textbf{inherit}
\textit{MOBILE}

\textbf{create}
\textit{make}

\textbf{feature} \{ \textit{NONE} \} -- Initialization

\textit{make} \{ \textit{a_station}: \textit{STATION}; \textit{a_radius}: \textit{REAL_{64}} \}
  -- Create ghost flying around ‘\textit{a_station}’ at distance ‘\textit{a_radius}’.
\textbf{require}
\textit{station_exists}: \textit{a_station} \neq \textit{Void}
\textit{radius_positive}: \textit{a_radius} > 0.0
\textbf{do}
  \textit{station} := \textit{a_station}
  \textit{radius} := \textit{a_radius}
\textbf{ensure}
  \textit{station_set}: \textit{station} = \textit{a_station}
  \textit{radius_set}: \textit{radius} = \textit{a_radius}
\textbf{end}

\textbf{feature} -- Access

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

\textit{station}: \textit{STATION}
  -- Station around which the ghost flies.

\textit{radius}: \textit{REAL_{64}}
  -- Distance from ‘\textit{station}’.

\textit{speed}: \textit{REAL_{64}} = 10.0
  -- Motion speed (meters/second).

\textbf{feature} \{ \textit{NONE} \} -- Movement
angle: \texttt{REAL}_64
   \quad\text{--- Angle of the current position (with respect to eastwards direction).}

\textit{move\_distance (d: REAL}_64)\textit{)
   \quad\text{--- Move by ‘d’ meters.}
\textbf{do}
\quad\text{angle} := \text{angle} + \text{d} / \text{radius}
\textbf{end}

\textbf{invariant}
\begin{align*}
\text{station\_exists:} \quad \text{station} &\neq \text{Void} \\
\text{radius\_positive:} \quad \text{radius} &> 0.0 \\
\text{circular\_trajectory:} \quad \text{approx\_equal} (\text{position\_distance (station\_position), radius})
\end{align*}
\textbf{end}

Listing 2: Class \textit{GHOST\_INVASION}

\textbf{note}
\begin{itemize}
   \item \textit{description: “Adding ghost to Zurich.”}
\end{itemize}

\textbf{class}
\begin{itemize}
   \item \textit{GHOST\_INVASION}
\end{itemize}

\textbf{inherit}
\begin{itemize}
   \item \textit{ZURICH\_OBJECTS}
\end{itemize}

\textbf{feature} \quad \text{--- Explore Zurich}

\textbf{invade}
\begin{itemize}
   \item \textit{--- Add ghosts to random stations.}
\end{itemize}
\textbf{local}
\begin{itemize}
   \item i: \texttt{INTEGER}
   \item cursor: \textit{like Zurich\_stations\_new\_cursor}
   \item random: \texttt{V\_RANDOM}
\end{itemize}
\textbf{do}
\begin{itemize}
   \item \textbf{from} \quad i := 1
   \item cursor := Zurich\_stations\_new\_cursor
   \item create random
   \item \textbf{until} \quad i > 10
\end{itemize}
\textbf{loop}
\begin{itemize}
   \item cursor\_go\_to (random\_bounded\_item (1, Zurich\_stations\_count))
   \item random\_forth
   \item add\_ghost (cursor\_item, random\_bounded\_item (10, 100))
   \item random\_forth
   \item i := i + 1
\end{itemize}
\textbf{end}
\textbf{Zurich\_map\_animate}
\textbf{end}

\textbf{add\_ghost (a\_station: \texttt{STATION}; a\_radius: \texttt{REAL}_64)}
\begin{itemize}
   \item \textit{--- Add a ghost going around ‘a\_station’.
\end{itemize}
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

3 Code review

There is no master solution for this task.

4 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
inherit
  ANY
  redefine
    out
  end

feature  --  Basic operations

  affect (p: PLAYER)
    --  Apply square’s special effect to ‘p’.
    require
      p_exists: p /= Void
    do
      --  For a normal square do nothing.
    end

feature  --  Output

  out: STRING
    --  Textual representation.
    do
      Result := ","
    end
Listing 4: Class \texttt{BAD_INVESTMENT_SQUARE}

class \texttt{BAD_INVESTMENT_SQUARE}

inhibit \texttt{SQUARE}
  redefine
    affect, out
  end

feature --- Basic operations

\texttt{affect (p: PLAYER)}
  --- Apply square's special effect to 'p'.
  do
    p.transfer (-5)
  end

feature --- Output

\texttt{out: STRING}
  --- Textual representation.
  do
    Result := "#"
  end

end

Listing 5: Class \texttt{LOTTERY_WIN_SQUARE}

class \texttt{LOTTERY_WIN_SQUARE}

inhibit \texttt{SQUARE}
  redefine
    affect, out
  end

feature --- Basic operations

\texttt{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.
do
        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 is 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")
      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
  winners_are_players: across winners as w all players.has (w.item) end
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: V.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 {V.LINKED_LIST [PLAYER]} winners from
  i := 1
  until
  i > players.count
  loop
if players[i].money > max then
    max := players[i].money
    winners.wipe_out
    winners.extend_back (players[i])
elseif players[i].money = max then
    winners.extend_back (players[i])
end
i := i + 1
end

ensure
has_winners: winners /= Void and not winners.is_empty
winners_are_players: across winners as w all players.has (w.item) end
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
all_players_exist: across players as p all p.item /= Void end
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):

```plaintext
affect (p: PLAYER)
require
    p_exists: p /= Void
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.

## 5 MOOC: Single Inheritance

The order in which the questions and the answers appear here in the solution may vary because they are randomly shuffled at each attempt.

### Single Inheritance Quiz

- Assume classes `MAMMAL` and `REPTILE` inheriting from a class `VERTEBRATE`. Furthermore assume the following declarations:

```plaintext
v: VERTEBRATE
m: MAMMAL
r: REPTILE
```

Assuming that every reference above is attached to an object of the exact same type, the corresponding snippets compile:

```plaintext
v := m
v := r
```

- Assume concrete classes `CAT` and `DOG` inheriting from deferred class `MAMMAL`. Furthermore assume the following declarations (implementation of routine `print_info` is omitted):

```plaintext
m: MAMMAL
c: CAT
d: DOG
print_info (mam: MAMMAL) do ... end
```
The following are true: I can pass to routine `print_info` an object of type `CAT` attached to m; I can pass to routine `print_info` an object of type `DOG` attached to m; I can pass to routine `print_info` an object of type `CAT` attached to c; I can pass to routine `print_info` an object of type `DOG` attached to d.

- Assume concrete classes `CAT` and `DOG` inheriting from deferred class `MAMMAL`. Furthermore assume the following declarations:

```
m: MAMMAL
c: CAT
d: DOG
```

The code snippets that compile are:

```
create {DOG}m
create {DOG}d
create {CAT}m
```

- Assume classes `MAMMAL` and `CAT` as follows:

```
deferred class MAMMAL

feature
    eat
    do
        print ("Mammal eating.")
    end
end

class CAT
    inherit MAMMAL
redefine
    eat end

feature
    eat
    do
        print ("Cat eating.")
    end
end
```

Assume the following references have been defined:

```
m: MAMMAL
c: CAT
```

The code snippets that compile and print the suggested text at the console are:

```
create {CAT}m
m.eat
-- Prints "Cat eating."
```

```
create {CAT}c
c.eat
-- Prints "Cat eating."
```
• Assume classes $MAMMAL$ and $CAT$ as follows:

```plaintext
defered class MAMMAL
  feature
    eat do
      print ("Mammal eating.")
    end
  end
end

class CAT inherit MAMMAL redefine eat end
feature
  eat do
    Precursor
    print ("Cat eating.")
  end
end
```

Assume the following references have been defined:

$m$: MAMMAL
c: CAT

The code snippets that compile and print the suggested text at the console are

```plaintext
create {CAT} m
m.eat
Prints "Mammal eating. Cat eating."
create {CAT} c
c.eat
Prints "Mammal eating. Cat eating."
```