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 HERO) does not conform to target (of type WARRIOR).

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
\texttt{angle: \texttt{REAL}\_64}
\hfill\texttt{-- Angle of the current position (with respect to eastwards direction).}

\texttt{move\_distance (d: REAL\_64)}
\hfill\texttt{-- Move by ‘d’ meters.}
\begin{verbatim}
do
  angle := angle + d / radius
end
\end{verbatim}

\texttt{invariant}
\begin{itemize}
  \item \texttt{station\_exists: station \neq Void}
  \item \texttt{radius\_positive: radius > 0.0}
  \item \texttt{circular\_trajectory: approx\_equal (position\_distance (station\_position), radius)}
\end{itemize}
\texttt{end}

\texttt{Listing 2: Class GHOST\_INVASION}

\texttt{note}
\begin{itemize}
  \item \texttt{description: "Adding ghost to Zurich."}
\end{itemize}

\texttt{class}
\begin{itemize}
  \item \texttt{GHOST\_INVASION}
\end{itemize}

\texttt{inherit}
\begin{itemize}
  \item \texttt{ZURICH\_OBJECTS}
\end{itemize}

\texttt{feature}
\begin{itemize}
  \item \texttt{-- Explore Zurich}
\end{itemize}

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

\texttt{end}

\texttt{add\_ghost (a\_station: STATION; a\_radius: REAL\_64)}
\begin{itemize}
  \item \texttt{-- 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
end

3 Code review

There is no master solution for this task.

4 Board game: Part 3

You can download a complete solution from
http://se.inf.ethz.ch/courses/2012b_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’.
    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 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

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

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
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.%.2N")
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_winner
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 then 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

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 \textit{SQUARE} and lets squares affect players that land on them in an arbitrary way. Classes \textit{BAD\_INVESTMENT\_SQUARE} and \textit{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 \textit{SQUARE}; instead of array of squares in class \textit{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 \textit{affect} ($p$: \textit{PLAYER}) directly to class \textit{BOARD} (instead of creating a class \textit{SQUARE} and an array of squares):

\begin{verbatim}
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
\end{verbatim}

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.