Solution 10: Agents and board games

ETH Zurich

1 Navigating in Zurich

Listing 1: Class *NAVIGATOR*

```
note
  description: "Finding routes in Zurich."

class
  NAVIGATOR

inherit
  ZURICH_OBJECTS

feature -- Explore Zurich

  add_event_handlers
      -- Add handlers to mouse-click events on stations
      -- to allow the user to select start and end points of his route.
    do
      across
        Zurich.stations as i
      loop
        Zurich_map.views [i.item].on_left_click_no_args.extend_back (agent set_origin (i.item))
        Zurich_map.views [i.item].on_left_click_no_args.extend_back (agent show_route)
        Zurich_map.views [i.item].on_right_click_no_args.extend_back (agent set_destination (i.
            item))
        Zurich_map.views [i.item].on_right_click_no_args.extend_back (agent show_route)
      end
    end

feature -- Access

  origin: STATION
      -- Currently selected start point.
      -- (Void if no start point selected).

  destination: STATION
      -- Currently selected end point.
      -- (Void if no end point selected).

  last_route: ROUTE
      -- Route calculated by the latest call to 'show_route'.
```

```
  finder: ROUTE_FINDER
      -- Route finder.
    once
      create Result.make (Zurich)
    end

feature {NONE} -- Implementation

  set_origin (s: STATION)
      -- Set 'origin' to 's'.
    do
      origin := s
    ensure
      origin_set: origin = s
    end

  set_destination (s: STATION)
      -- Set 'destination' to 's'.
    do
      destination := s
    ensure
      destination_set: destination = s
    end

  show_route
      -- If both 'origin' and 'destination' are set, show the route from 'origin' to 'destination
         ' on the map
      -- and output directions to the console.
      -- Otherwise do nothing.
    local
      i: INTEGER
    do
      if origin /= Void and destination /= Void then
        if last_route /= Void then
          Zurich.remove_route (last_route)
        end
        last_route := finder.shortest_route (origin, destination)
        Zurich.add_route (last_route)
        Zurich_map.update

        Console.output ("From " + origin.name + " to " + destination.name + ":")
        from
          i := 1
        until
          i > last_route.lines.count
        loop
          Console.append_line ("Take " + last_route.lines[i].kind.name + " " + last_route.
              lines[i].number.out +
            " until " + last_route.stations[i + 1].name)
          i := i + 1
        end
      end
```

```
    ensure
      last_route_exists: origin /= Void and destination /= Void implies last_route /= Void
    end

invariant
  finder_exists: finder /= Void
end
```

2 Home automation

Listing 2: Class *TEMPERATURE_SENSOR*

```
class
  TEMPERATURE_SENSOR

inherit
  ANY
    redefine
      default_create
    end

feature {NONE}-- Initialization

  default_create
      -- Initialize the set of observers.
    do
      create {V_HASH_SET [PROCEDURE [ANY, TUPLE [REAL_64]]]} observers
    ensure then
      no_observers: observers.is_empty
    end

feature -- Access

  temperature: REAL_64
      -- Temperature value in degrees Celcius.

feature -- Status report

  valid_temperature (a_value: REAL_64): BOOLEAN
      -- Is 'a_value' a valid temperature?
    do
      Result := a_value >= -273.15
    end

feature -- Basic operations

  set_temperature (a_temperature: REAL_64)
      -- Set 'temperature' to 'a_temperature' and notify observers.
    require
      valid_temperature: valid_temperature (a_temperature)
    do
      temperature := a_temperature
```

```
      across
        observers as c
      loop
        c.item.call ([temperature])
      end
    ensure
      temperature_set: temperature = a_temperature
    end

feature -- Subscription

  subscribe (an_observer: PROCEDURE [ANY, TUPLE [REAL_64]])
      -- Add 'an_observer' to observers list.
    do
      observers.extend (an_observer)
    ensure
      present: observers.has (an_observer)
    end

  unsubscribe (an_observer: PROCEDURE [ANY, TUPLE [REAL_64]])
      -- Remove 'an_observer' from observers list.
    do
      observers.remove (an_observer)
    ensure
      absent: not observers.has (an_observer)
    end

feature {NONE} -- Implementation

  observers: V_SET [PROCEDURE [ANY, TUPLE [REAL_64]]]
      -- Set of observing agents.

invariant
  valid_temperature: valid_temperature (temperature)
  observers_exists: observers /= Void
  all_observers_exist: not observers.has (Void)
end
```

Listing 3: Class *APPLICATION*

```
class
  APPLICATION

create
  make

feature {NONE} -- Initialization
  make
      -- Run application.
    local
      s: TEMPERATURE_SENSOR
      d: DISPLAY
      c: HEATING_CONTROLLER
```

```
    do
      create s
      create d
      create c.set_goal (21.5)

      s.subscribe (agent d.show)
      s.subscribe (agent c.adjust)

      s.set_temperature (22)
      s.set_temperature (22.8)
      s.set_temperature (20.0)

      s.set_temperature (−273.14276764)
      s.set_temperature (1000)
      s.set_temperature (0)
    end
end
```

3 The final project. Board game: part 4

You can download a complete solution from http://se.inf.ethz.ch/courses/2013b_fall/eprog/assignments/10/board_game_solution.zip.

4 MOOC: Selective exports, multiple inheritance, and agents

Selective exports and deferred classes

- Suppose to have the following class *ITEM*:

```
class
    ITEM

feature -- Basic operations

    set_price (p: INTEGER)
            -- Set price for current object.
        do
            price := p
        end

feature {STATS, ORDER\_LINE} -- Access

    description: STRING
        -- Item description.

    price: INTEGER
        -- Item price.
end
```

The true statements are: features description and price are available to classes STATS, ORDER_LINE, and their descendants; feature set_price is available to all classes.

- Suppose to have the following class *ITEM*:

```
class
    ITEM

create {ORDER\_LINE}
    set_description

feature {NONE} -- Initialition

    set_description (d: STRING)
            -- Set description for current object.
        do
            description := d
        end

feature -- Basic operations

    set_price (p: INTEGER)
            -- Set price for current object.
        do
            price := p
        end

feature -- Access

    description: STRING
        -- Item description.

    price: INTEGER
        -- Item price.
end
```

  The true statements are: Objects of class ITEM can be created from within objects of class ORDER_LINE; Feature set_description can be used as a creation procedure, but cannot be invoked normally (that is, not as a creation procedure) on an object of type ITEM from another class.

- Suppose to have the following class *ITEM*:

```
class
    ITEM

create
    set_description

feature {NONE} -- Initialition

    set_description (d: STRING)
            -- Set description for current object.
        do
            description := d
        end
```

```
feature -- Basic operations

    set_price (p: INTEGER)
            -- Set price for current object.
        do
            price := p
        end

feature -- Access

    description: STRING
        -- Item description.

    price: INTEGER
        -- Item price.
end
```

The true statements are: Objects of class ITEM can be created from within another class; Feature set_description can be used as a creation procedure, but cannot be invoked normally on an object of type ITEM from another class.

- Suppose to have the following class *ITEM*:

```
class
    ITEM

feature -- Basic operations

    set_price (p: INTEGER)
            -- Set price for current object.
        do
            price := p
        end

feature {ITEM, ORDER\_LINE} -- Access

    description: STRING
        -- Item description.

    price: INTEGER
        -- Item price.
end
```

The true statements are: features description and price are available to classes ITEM, ORDER_LINE, and their descendants; Making features description and price available to class ITEM means that I can use them from within a class different from ITEM, when applying features description and price to objects of type ITEM.

- Which of the following sentences about deferred (abstract) classes is true (more answers are possible)? You can have a deferred class whose features are all implemented; Deferred classes are useful when designing an object-oriented system; You can have a deferred class whose features are all deferred; To be useful, a deferred class has to be inherited from.

- A deferred class can have non-deferred ancestor classes: true.

- If you write a deferred feature in a non-deferred class you will get a compilation error: true.

Multiple inheritance

- Assume the following code:

```
class A
feature
    f
        do
            -- implementation omitted
        end
    g
        do
            -- implementation omitted
        end
end
class B
feature
    f
        do
            -- implementation omitted
        end
    h
        do
            -- implementation omitted
        end
end
```

  Assume that in class C (inheriting from both classes A and B) you want to keep the implementation of f coming from B. Which of the following class C implementations provides the correct answer?

```
class C
inherit
    A
        undefine f
        end
    B
end
```

- What does it mean that a class C inherits from A and, in a non-conforming way, from B? That you can declare a reference of type B and attach to it an object of type C; That polymorphism does not apply when there is a reference of type B to which there is an object of type C attached.

- Assume the following code:

```
class A
feature
    f
        do
            -- implementation omitted
```

```
        end
    g
        do
            -- implementation omitted
        end
end

class B
feature
    f
        do
            -- implementation omitted
        end
    h
        do
            -- implementation omitted
        end
end
```

Assume to have class C inheriting from both classes A and B. Which of the following class implementations correctly compile?

```
class C
inherit
    A
        rename f as a_f
    end
    B
end
```

```
class C
inherit
    A
    B
        rename f as b_f
    end
end
```

```
class C
inherit
    A
        rename f as a_f
    B
        rename f as b_f
    end
end
```

- Assume the following code:

```
deferred class A
feature
    f
        do
```

```
            -- implementation omitted
        end
    g
        deferred
        end
end

deferred class B
feature
    f
        deferred
        end
    h
        do
            -- implementation omitted
        end
end
```

Assume to have class C inheriting from both classes A and B. Which of the following class implementations correctly compile?

```
deferred class C
inherit
    A
    B
end
```

```
class C
inherit
    A
    B
       rename f as b_f
    end
feature
    g
        do
            -- implementation omitted
        do
    b_f
        do
            -- implementation omitted
        do
end
```

```
class C
inherit
    A
       rename f as a_f
    B
       rename f as b_f
    end
feature
    g
```

```
        do
            -- implementation omitted
        do
    b_f
        do
            -- implementation omitted
        do
end
```

- Assume the following code:

```
class A
feature
    f
        do
            -- implementation omitted
        end
    g
        do
            -- implementation omitted
        end
end
class B
feature
    f
        do
            -- implementation omitted
        end
    h
        do
            -- implementation omitted
        end
end
```

Assume to have class C inheriting from both classes A and B. Which of the following class implementations correctly compile?

```
class C
inherit
    A
        rename f as a_f redefine a_f
        end
    B
feature
    a_f
        do
            -- implementation omitted.
        end
end
```

```
class C
inherit
    A
```

```
        rename f as a_f redefine a_f,g
        end
    B
feature

   g
        do
            -- implementation omitted.
        end

   a_f
        do
            -- implementation omitted.
        end
end
```

```
class C
inherit
    A
        rename f as a_f
        end
    B
        redefine h end
feature

   h
        do
            -- implementation omitted.
        end
end
```

- In a multiple inheritance scenario, indicate a case in which it makes sense to inherit twice from the same class. Answer: When the ancestor has an implemented feature whose implementation we want to preserve, while at the same time provide another implementation of the same feature in the descendant.
- Assume the following code:

```
deferred class A
feature
    f
        deferred
        end
end

class B
inherits
    A
feature
    f
        do
            -- implementation omitted
        end
```

```
end

class C
inherits
    A
        rename f as c_f end
feature
    c_f
        do
            -- implementation omitted
        end
end
```

Assume to have the following declarations:

```
a: A
d: D
```

Assume further that the following code is executed:

```
create d
a := d
a.f
```

Which of the following declarations for class D works (more answers possible)?

```
class D
inherit
    B

    C
        select c_f
        end
end
```

```
class D
inherit
    B
        select f
        end
    C
end
```

Agents

- The true statements about the Model View Controller (MVC) pattern are the following: it should be straightforward to switch between views in an application using MVC; It should be straightforward to switch between models in an application using MVC; An application using two different databases, an HTML view and a command-line view can be an example of an application that can benefit from MVC; The computer memory can be an example of a model in the MVC.

- Complete the code of the following class implementing part of the observer pattern by choosing the correct instructions.

```
deferred class
    SUBSCRIBER

feature -- Basic operations

    subscribe (p: NEWS_BROADCASTER)
            -- Subscribe to 'p'.
        require
            p_exists: p /= Void
        do
            p.attach (Current)
        end

    unsubscribe (p: NEWS_BROADCASTER)
        -- Unsubscribe to 'p'.
        require
            p_exists: p /= Void
        do
            p.detach (Current)
        end

feature {NEWS_BROADCASTER} -- Implementation

    update (s: STRING)
            -- Action triggered by broadcaster.
        deferred
        end
end
```

- Complete the code of the following class implementing part of the observer pattern by choosing the correct instructions.

```
deferred class
    NEWS_BROADCASTER

feature -- Initialization

    make
            -- Initialize Current.
        do
            create subscribers.make
        end

feature {SUBSCRIBER} -- Addition

    attach (s: SUBSCRIBER)
        -- Subscribe 's'.
        require
            s_exists: s /= Void
  do
            if not subscribers.has (s) then subscribers.extend (s) end
  end
```

```
feature {SUBSCRIBER} -- Removal

    detach (s: SUBSCRIBER)
            -- Unsubscribe 's'.
        require
            s_exists: s /= Void
        do
            subscribers.start
            subscribers.search(s)
            if not subscribers.after then subscribers.remove end
     end

feature -- Basic operations

    publish
            -- Publish news to subscribers.
        deferred
        end

feature {NONE} -- Implementation

    subscribers: LINKED_LIST [SUBSCRIBER]

invariant
    subscribers_exist: subscribers /= Void

end
```

- Complete the code of the following class implementing part of the observer pattern by choosing the correct instructions.

```
class
    EVENT_MANAGER [EVENT_DATA -> TUPLE]
create
    make

feature -- Initialization

    make
            -- Initialize Current.
        do
            create subscribers.make
        end

feature -- Basic operations

    publish (args: EVENT_DATA)
            -- Trigger an event of this type.
        do
            from
                subscribers.start
```

```
			until
				subscribers.after
			loop
				subscribers.item.call (args)
				subscribers.forth
			end
		end

	subscribe (action: PROCEDURE [ANY, EVENT_DATA])
			-- Register 'action' to be executed for events of this type.
		require
			action_exists: action /= Void
		do
			if not subscribers.has (action) then
				subscribers.extend (action)
			end
		ensure
			action_added: subscribers.has (action)
		end

	unsubscribe (action: PROCEDURE [ANY, EVENT_DATA])
			-- Deregister 'action' to be executed for events of this type.
		do
				subscribers.compare_objects
				subscribers.start
				subscribers.search(action)
				if not subscribers.after then subscribers.remove end
		ensure
				action_removed: not subscribers.has (action)
		end

feature {NONE} -- Implementation

	subscribers: LINKED_LIST [PROCEDURE [ANY, EVENT_DATA]]

invariant
	subscribers_exist: subscribers /= Void
end

class
	INDIVIDUAL

create
	make

feature {NONE} -- Initialization

	make (n: STRING)
			-- Initialization for 'Current'.
		require
			n_exists: n /= Void and not n.is_empty
		do
```

```
            name := n
        ensure
            name_set: name = n
        end

feature -- Access

    name: STRING
            -- Subscriber's name

    reaction_behavior
            -- Individual's reaction behavior.
        do
            print (name + " is reacting.")
        end
end

class
APPLICATION
create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        local
            i: INDIVIDUAL
            p: EVENT_MANAGER [TUPLE []]
        do
                create i.make ("Ted")
                create p.make
                p.subscribe (agent i.reaction_behavior)
        end
```