Eiffel: Analysis, Design and Programming Exam

ETH Zürich

Date: 4 December 2008

Surname, first name: ..............................................................................................

Student number: ..............................................................................................

I confirm with my signature, that I was able to take this exam under regular circumstances and that I have read and understood the directions below.

Signature: ..............................................................................................

Directions:

• Exam duration: 90 minutes.

• Except for a dictionary you are not allowed to use any supplementary material.

• Use a pen (not a pencil)!

• All solutions can be written directly on the exam sheets. If you need more space for your solution ask the supervisors for a sheet of official paper. You are not allowed to use other paper. Please write your student number on each additional sheet.

• Only one solution can be handed in per question. Invalid solutions need to be crossed out clearly.

• Please write legibly! We will only correct solutions that we can read.

• Manage your time carefully (take into account the number of points for each question).

• Don’t forget to include header comments in features.

• Please immediately tell the exam supervisors if you feel disturbed during the exam.

Good luck!
<table>
<thead>
<tr>
<th>Question</th>
<th>Number of possible points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>50</td>
</tr>
</tbody>
</table>
1 Object-oriented principles, Design by Contract and Eiffel mechanisms (20 points)

Consider the following 5-class Eiffel system with root class APPLICATION and root procedure ‘make’ where some details related to creation procedures have been omitted. The style of classes RECTANGLE, SQUARE, LINE_SEGMENT and APPLICATION does not show good use of Eiffel (and O-O) design principles.

class POINT

create
default_create, make

feature -- Creation

make (r1, r2: REAL)
-- Set (x, y) to (r1, r2).
do
  x := r1
  y := r2
ensure
  x_set: x = r1
  y_set: y = r2
end

feature -- Access

x: REAL
-- The x-coordinate.
y: REAL
-- The y-coordinate.

feature -- Element change

move (p: POINT)
-- Move (x, y) to (x + p.x, y + p.y).
do
  x := x + p.x
  y := y + p.y
ensure
  x_updated: x = old x + p.x
  y_updated: y = old y + p.y
end

class RECTANGLE

...
class SQUARE
...
feature -- Access
  upper_left : POINT
    -- The upper left corner.
  side_length : REAL
    -- The side length.
end

indexing
description: "Line segments between points p1 and p2."
class LINE_SEGMENT
...
feature -- Access
  p1 : POINT
  p2 : POINT
end

class APPLICATION
create
  make
feature
  make -- Create some shapes and move them.
local
  r : RECTANGLE
  s : SQUARE
  l : LINE_SEGMENT
  io
  do
    create r
    create s
    create l
    io. put_string ("Moved the " + move_and_get_name (r, create {POINT}.
      make (2, 2)) + 
    io. put_string ("Moved the " + move_and_get_name (s, create {POINT}.
      make (3, 5)) + 
    io. put_string ("Moved the " + move_and_get_name (l, create {POINT}.
      make (2.5, 4)) + 
  end
move_and_get_name (a : ANY, p : POINT) : STRING
  -- Move the shape stored in ‘a’ by the vector ‘p’.
  -- ‘Result’ will be the name of the shape.
  do
    if {r : RECTANGLE} a then
      r. upper_left. move (p)
      r. lower_right. move (p)
      Result := "rectangle"
    elseif {s : SQUARE} a then
      s. upper_left. move (p)
    else
      io. put_string ("Shape not found")
    end
Result := "square"

else if \{l : LINE_SEGMENT\} a then
  l.p1.move(p)
  l.p2.move(p)
  Result := "line segment"
else
  Result := "unknown"
end

end

Rewrite the program using Eiffel and O-O principles and Design by Contract. Your solution may use class POINT as given above. Explain the changes: which principles you applied, and which language mechanisms facilitate your solution.
2 Genericity, agents, patterns and components
(20 Points)

A principal goal of the Eiffel method is the creation of reusable components. The pattern of publishing some type of object on an event channel that forwards it to a list of subscribers is a common idiom that can be reused across applications. Consider the following (artificial but concise) client code:

class APPLICATION
create
make
feature -- Creation
make
local
ec: EVENT_CHANNEL [INTEGER]
do
create ec
  -- 'ec' should now have an empty list of subscribers.
ce.subscribe (agent subscriber1)
  -- 'ec' should now have exactly one subscriber.
ce.publish (2)
  -- '2' should now have appeared on the console.
ce.subscribe (agent subscriber2)
  -- 'ec' should now have two subscribers.
ce.publish (3)
  -- '3' and '4' should now have appeared on the console.
end
feature -- Subscriber
subscriber1 (i: INTEGER)
do
  io.put_integer (i)
end
subscriber2 (i: INTEGER)
do
  io.put_integer (i + 1)
end
end

The task is to implement class EVENT_CHANNEL. You can make use of class LINKED_LIST whose interface is given here:

class interface LINKED_LIST[G]
create
make
  -- Create an empty list.
feature -- Element change
extend (v: G)
  -- Add 'v' to the end.
feature -- Access
item: G
  -- Item at current cursor position.

feature -- Cursor movement
  start
    -- Move cursor to first position.
  forth
    -- Move cursor to next position.

feature -- Status report
  after: BOOLEAN
    -- Is there no valid cursor position to the right of the cursor?

(Hint: an agent that can be called with one argument of type G has type
PROCEDURE [ANY, TUPLE [G]])
3 Multiple inheritance (10 Points)

The following program with root class APPLICATION and root procedure 'make' uses multiple inheritance:

```eiffel
class APPLICATION
  create
    make
  feature
    make
      local
        a: A
        c: C
        d: D
      do
        create c
        create d
        a := c
        a.f
        c.g
        a := d
        a.f
        c := d
        c.f
        d.f
      end
    end
end

class A
  create
    default_create
  feature
    f
      do
        io.put_string ("A.f\%N")
      end
    g
      do
        io.put_string ("A.g\%N")
      end
  end
end

deferred class B
  inherit A
    rename
      f as h
    undefine
      g
  end
end
```
class C
inherit A redefine f end
create default_create
feature f
do io.puts_string ("C.f%N")
end
end

class D
inherit B select h end
C redefine g end
create default_create
feature g
do io.puts_string ("D.g%N")
end
end

What will be printed on the console if the program is executed?