Mock Exam 2

ETH Zurich

December 5, 2012

Name: 

Group: 

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
</tr>
</tbody>
</table>

Grade
1 Terminology (10 Points)

Goal
This task will test your understanding of the object-oriented programming concepts presented so far in the lecture. This is a multiple-choice test.

Todo
Place a check-mark in the box if the statement is true. There may be multiple true statements per question; 0.5 points are awarded for checking a true statement or leaving a false statement un-checked, 0 points are awarded otherwise.

1. A class...
   a. is the description of a set of possible run-time objects to which the same features are applicable. □
   b. can only exist at runtime. □
   c. cannot be declared as expanded; only objects can be expanded. □
   d. may have more than one creation procedure. □

2. Procedures, functions and attributes.
   a. A query needs to be a function. □
   b. A function cannot modify any objects. □
   c. An attribute is stored directly in memory. □
   d. A procedure can return values that are computed. □

3. What are the possible changes in a function redefinition?
   a. To change the implementation. □
   b. To change the list of argument types. □
   c. To change the contract. □
   d. To change the result type. □

4. Clients and suppliers.
   a. A supplier of a software mechanism is a system that uses the mechanism. □
   b. A client of a software mechanism cannot be a human. □
   c. A client of a software mechanism is a system of any kind, software or not, that uses the mechanism. For its clients, the mechanism is a supplier. □
   d. A supplier of a set of software mechanisms provides an interface to its clients. □
5. Polymorphism.
   a. A data structure is polymorphic if it may contain references to objects of different types. □
   b. An assignment or argument passing is polymorphic if its target variable and source expression have different types. □
   c. Polymorphism is the capability of objects to change their types at run time. □
   d. An entity or expression is polymorphic if, as a result of polymorphic attachments, it may at run time become attached to objects of different types. □
2 Design by Contract (10 Points)

2.1 Task

Your task is to fill in the contracts (preconditions, postconditions, class invariants, loop variants and invariants) of the class CAR according to the given specification. You are not allowed to change the class interface or the given implementation. Note that the number of dotted lines does not indicate the number of missing contracts.

2.2 Solution

class
2
    CAR
4
create
  make
6
  feature {NONE} -- Creation
8
    make
 10      -- Creates a default car.
 12      require
 14
 16
 18      do
 20  create {LINKED_LIST [CAR_DOOR]} doors.make
 22      ensure
 24
 26
 28      end

28      feature {ANY} -- Access
30
  is_convertible : BOOLEAN
 32      -- Is the car a convertible (cabriolet)? Default: no.
34  doors: LIST [CAR_DOOR]
 36      -- The doors of the car. Number of doors must be 0, 2 or 4. Default: 0.
38  color: COLOR
 40      -- The color of the car. ‘Void’ if not specified. Default: ‘Void’.

40      feature {ANY} -- Element change
42 set_convertible (a_is_convertible : BOOLEAN)
  require
  .................................................................
  .................................................................
  .................................................................
  do
    is_convertible := a_is_convertible
  ensure
  .................................................................
  .................................................................
  .................................................................
  end

set_doors (a_doors: ARRAY [CAR.DOOR])
  require
  .................................................................
  .................................................................
  .................................................................
  local door_index: INTEGER
  do
    doors.wipe_out
    if a_doors /= Void then
      from
        door_index := 1
      invariant
      .................................................................
      .................................................................
      .................................................................
      until
        door_index > a_doors.count
    loop
      doors.extend (a_doors [door_index])
      door_index := door_index + 1
    variant
    .................................................................
    .................................................................
  .................................................................
begin
end
ensure

set_color (a_color: COLOR)
require

do
  color := a_color
ensure

end

invariant
end
3 Inheritance and polymorphism (14 Points)

Classes PRODUCT, COFFEE, ESPRESSO, CAPPUCINO and CAKE given below are part of the software system used by a coffee shop to keep track of the products it has.

1 deferred class PRODUCT

3 feature -- Main operations

5 set_price (r: REAL)
   -- Set ‘price’ to ‘r’.
7 require
8    r_non_negative: r >= 0
9 do
10   price := r
11 ensure
12    price_set: price = r
13 end

15 feature -- Access

17 price: REAL
   -- How much the product costs
19 description: STRING
   -- Brief description
23 deferred
25 invariant
   non_negative_price: price >= 0
27 valid_description: description /= Void and then not description.is_empty
29 end

1 deferred class COFFEE

3 inherit
   PRODUCT
5 feature -- Main operations
7 make
9   -- Prepare the coffee.
10 do
11   print ("I am making you a coffee.")
13 end
15 end
class ESPRESSO
  inherit COFFEE
  create
    set_price
  feature -- Access
    description: STRING
      do
        Result := "A small strong coffee"
      end
  end

class CAPPuccino
  inherit COFFEE
  create
    set_price
  feature -- Access
    description: STRING
      do
        Result := "A coffee with milk and milk foam"
      end
  end

class CAKE
  inherit PRODUCT
    rename set_price as make
  end
  create
    make
  feature -- Access
    description: STRING
      do
        Result := "A sweet dessert"
      end
  end


Given the following variable declarations:

- \texttt{product: PRODUCT}
- \texttt{coffee: COFFEE}
- \texttt{espresso: ESPRESSO}
- \texttt{cappuccino: CAPPUCCINO}
- \texttt{cake: CAKE}

specify, for each of the code fragments below, if it compiles. If it does not compile, explain why this is the case. If it compiles, specify the text that is output to the screen when the code fragment is executed.

1. \texttt{create product}
   \texttt{io.put_string \{product.description\}}

2. \texttt{create \{ESPRESSO\} product.set_price (5.20)}
   \texttt{io.put_string \{product.description\}}

3. \texttt{create cappuccino.make}
   \texttt{io.put_string \{cappuccino.description\}}

4. \texttt{create \{ESPRESSO\} cappuccino.set_price (5.20)}
   \texttt{io.put_string \{cappuccino.description\}}

5. \texttt{create cake.make (6.50)}
   \texttt{product := cake}
   \texttt{io.put_string \{product.description\}}
6. create \{ESPRESSO\} product.set_price(5.20)
   espresso := product
   io.put_string (espresso.description)

7. create \{CAPPUCCINO\} coffee.set_price(5.50)
   coffee.make
4 Recursion: Deleting directories (10 Points)

In this question you will work with the FILE class, which represents both directories and regular files. You can iterate through the files contained in a directory using an internal cursor:

```plaintext
from directory.start until directory.after loop
   -- Do something with 'directory.item'
   directory.forth
end
```

The delete command of class FILE physically deletes the file from disk and changes the value of the exists query on the corresponding FILE object to False. For a directory this command only works if the directory is physically empty (i.e. no files physically exist in the directory).

4.1 Task 1

Take a look at the following procedure delete_all. It deletes a given directory with all its content using recursion:

```plaintext
delete_all (directory: FILE)
   require
directory /= Void and then (directory.exists and directory.is_directory)
   do
      from directory.start until directory.after loop
         if directory.item.is_directory then
            delete_all (directory.item)
         else -- regular file
            directory.item.delete
         end
      end
      directory.forth
   end
   directory.delete
   ensure
   not directory.exists
end
```

Your task is to rewrite delete_all so that it does not use recursion (the procedure is not allowed to call itself). You are not allowed to add new features. You are only allowed to call those features of class FILE that are already used in the recursive implementation of delete_all.

You can use the class ARRAYED_LIST for this task. An excerpt is given at the end of the question.

```plaintext
delete_all (directory: FILE)
   require
   directory /= Void and then (directory.exists and directory.is_directory)
   local
```
do
ensure not directory. exists
end

4.2 Task 2

With the following example directory and the invocation

```
del_all (create {FILE}.make ("C:\Temp\to_del")
```

please give the order in which the files will be deleted for (a) the given recursive algorithm and (b) your non-recursive algorithm (e.g.: 3, 6, 7, 8, 9, 2, 5, 4, 1).

1  C:\Temp\to_del
2  C:\Temp\to_del\1
3  C:\Temp\to_del\1\foo.txt
4  C:\Temp\to_del\2
5  C:\Temp\to_del\2\3
6  C:\Temp\to_del\2\3\foobar.txt
7  C:\Temp\to_del\2\bar.txt
8  C:\Temp\to_del\another_file.txt
9  C:\Temp\to_del\file.txt

a) .................................................................
b) .................................................................
4.3 ARRAYED_LIST [G] (Excerpt)

class
  ARRAYED_LIST [G]

feature {NONE} -- Initialization

feature -- Access

  first : like item
  -- Item at first position

  item: G
  -- Current item

  last : like item
  -- Item at last position

feature -- Status report

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

  before : BOOLEAN
  -- Is there no valid cursor position to the left of cursor?

feature -- Cursor movement

  back
  -- Move to previous item.

  finish
  -- Move cursor to last position.
  -- (Go before if empty)

  forth
  -- Move cursor to next position.

  start
  -- Move cursor to first position.

feature -- Element change

  extend (v: like item)
  -- Add ‘v’ to end.
  -- Do not move cursor.

  put_front (v: like item)
  -- Add ‘v’ to beginning.
  -- Do not move cursor.

  put_left (v: like item)
-- Add 'v' to the left of cursor position.
-- Do not move cursor.

\texttt{put_right (v: like item)}
\hspace{1cm}
-- Add 'v' to the right of cursor position.
-- Do not move cursor.

\textbf{feature} -- Removal

\texttt{remove}
\hspace{1cm}
-- Remove current item.
-- Move cursor to right neighbor
-- (or after if no right neighbor).

\texttt{remove_left}
\hspace{1cm}
-- Remove item to the left of cursor position.
-- Do not move cursor.

\texttt{remove_right}
\hspace{1cm}
-- Remove item to the right of cursor position.
-- Do not move cursor.

\textbf{end} -- class ARRAYED_LIST