Mock Exam 2

ETH Zurich

December 5, 2012

Name: _____

Group: _____

Question	Points
1	
2	
3	
4	
Total	
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. ☐

Solution

1. a, d
2. c
3. a, b, c, d
4. c, d
5. a, b, d

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
    CAR

create
    make

feature {NONE} -- Creation

    make
            -- Creates a default car.
        require
            .....
            .....
            .....
        do
            create {LINKED_LIST [CAR_DOOR]} doors.make
        ensure
            .....
            .....
            .....
        end

feature {ANY} -- Access

    is_convertible : BOOLEAN
        -- Is the car a convertible (cabriolet)? Default: no.

    doors: LIST [CAR_DOOR]
        -- The doors of the car. Number of doors must be 0, 2 or 4. Default: 0.

    color: COLOR
        -- The color of the car. 'Void' if not specified. Default: 'Void'.

feature {ANY} -- Element change
```

```
42    set_convertible ( a_is_convertible : BOOLEAN)
        require
44
            ..........
46
            ..........
48
            ..........
50      do
            is_convertible := a_is_convertible
52      ensure
54          ..........
56          ..........
58          ..........
        end
60
      set_doors (a_doors: ARRAY [CAR_DOOR])
62      require
64          ..........
66          ..........
68          ..........
        local
70          door_index: INTEGER
        do
72          doors.wipe_out
            if a_doors /= Void then
74              from
                    door_index := 1
76              invariant
78                  ..........
80                  ..........
82                  ..........
                until
84                  door_index > a_doors.count
                loop
86                  doors.extend (a_doors [door_index])
                    door_index := door_index + 1
88              variant
90                  ..........
92                  ..........
```

```
94                 .....
               end
96           end
         ensure
98
             .....
100
             .....
102
             .....
104      end

106    set_color (a_color: COLOR)
         require
108
             .....
110
             .....
112
             .....
114      do
             color := a_color
116      ensure
118
             .....
120
             .....
122
             .....
         end
124
  invariant
126
     .....
128
     .....
130
     .....
132
  end
```

Solution

```
1 class
    CAR
3
  create
5   make

7 feature {NONE} -- Creation

9   make
```

```
        -- Creates a default car.
    require
    do
        create {LINKED_LIST [CAR_DOOR]} doors.make
    ensure
        not is_convertible
        doors /= Void and then doors.count = 0
        color = Void
    end

feature {ANY} -- Access

    is_convertible : BOOLEAN
        -- Is the car a convertible (cabriolet)? Default: no.

    doors: LIST [CAR_DOOR]
        -- The doors of the car. Number of doors must be 0, 2 or 4. Default: 0.

    color: COLOR
        -- The color of the car. 'Void' if not specified. Default: 'Void'.

feature {ANY} -- Element change

    set_convertible (a_is_convertible : BOOLEAN)
        require
        do
            is_convertible := a_is_convertible
        ensure
            is_convertible = a_is_convertible
        end

    set_doors (a_doors: ARRAY [CAR_DOOR])
        require
            a_doors /= Void implies (a_doors.count = 0 or a_doors.count = 2 or a_doors.count
                = 4)
        local
            door_index: INTEGER
        do
            doors.wipe_out
            if a_doors /= Void then
                from
                    door_index := 1
                invariant
                    doors.count + 1 = door_index
                    door_index >= 1 and door_index <= a_doors.count + 1
                until
                    door_index > a_doors.count
                loop
                    doors.extend (a_doors [door_index])
                    door_index := door_index + 1
                variant
                    a_doors.count + 1 - door_index
```

```
61              end
            end
63      ensure
            (a_doors = Void and doors.count = 0) or (a_doors /= Void and then a_doors.
                count = doors.count)
65      end

67  set_color (a_color: COLOR)
      require
69      do
            color := a_color
71      ensure
            color = a_color
73      end

75 invariant
    doors /= Void
77  doors.count = 0 or doors.count = 2 or doors.count = 4

79 end
```

3 Inheritance and polymorphism (14 Points)

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

```
deferred class PRODUCT

feature -- Main operations

  set_price (r: REAL)
      -- Set 'price' to 'r'.
    require
      r_non_negative: r >= 0
    do
      price := r
    ensure
      price_set: price = r
    end

feature -- Access

  price: REAL
    -- How much the product costs

  description: STRING
    -- Brief description
    deferred
    end

invariant
  non_negative_price: price >= 0
  valid_description: description /= Void and then not description.is_empty

end
```

```
deferred class COFFEE

inherit
  PRODUCT

feature -- Main operations

  make
      -- Prepare the coffee.
    do
      print ("I am making you a coffee.")
    end

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:

```
product: PRODUCT
coffee: COFFEE
espresso: ESPRESSO
cappuccino: CAPPUCCINO
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. **create** *product*
   io.put_string (product.description)

   .....

   .....

   The code does not compile, because it is not possible to create an instance of a deferred type.

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

   .....

   .....

   The code compiles. Output: "A small strong coffee"

3. **create** *cappuccino.make*
   io.put_string (cappuccino.description)

   .....

   .....

   The code does not compile. *make* is not a creation procedure of class *CAPPUCCINO*.

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

   .....

   .....

   The code does not compile. The explicit creation type *ESPRESSO* does not conform to *CAPPUCCINO*.

5. **create** *cake.make* (6.50)
   product := *cake*
   io. *put_string* (*product*. *description*)

   .....

   .....

   The code compiles. Output: "A sweet dessert"

6. **create** {*ESPRESSO*} *product.set_price* (5.20)
   espresso := *product*
   io. *put_string* (*espresso*. *description*)

   .....

   .....

   The code does not compile. The static type of *product* (*PRODUCT*) does not conform to the static type of *espresso* (*ESPRESSO*).

7. **create** {*CAPPUCCINO*} *coffee.set_price* (5.50)
   coffee . *make*

   .....

   .....

   The code compiles. Output: "I am making you a coffee."

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:

```
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:

```
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
      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.

```
delete_all (directory: FILE)
  require
    directory /= Void and then (directory.exists and directory.is_directory)
  local
```

```
6       .....
8       .....
    do
10
        .....
12
        .....
14
        .....
16
        .....
18
        .....
20
        .....
22
        .....
24
        .....
26
        .....
28
        .....
30
        .....
32
        .....
34
        .....
36
        .....
38
        .....
40
        .....
42
        .....
44
        .....
46
        .....
48
        .....
50
        .....
52
        .....
54
        .....
56
        .....
```

```
58
        .....
60
        .....
62
        .....
64
        .....
66
        .....
68
        .....
70
        .....
72
        .....
74
        .....
76
        .....
78
        .....
80    ensure
        not directory.exists
82    end
```

Solution

```
    delete_all (directory: FILE)
2     require
        directory /= Void and then (directory.exists and directory.is_directory)
4     local
        directories: ARRAYED_LIST [FILE]
6       cur_directory: FILE
      do
8       -- delete all files
        from
10        create directories.make
          directories.extend (directory)
12        directories.start
        until
14        directories.after
        loop
16        cur_directory := directories.item
          from
18          cur_directory.start
          until
20          cur_directory.after
          loop
22          if cur_directory.item.is_directory then
              directories.extend (cur_directory.item)
```

```
24                else -- normal file
                      cur_directory.item.delete
26                end
                  cur_directory.forth
28              end
                directories.forth
30          end
            -- delete all directories
32          from
                directories.finish
34          until
                directories.before
36          loop
                directories.item.delete
38              directories.back
            end
40      ensure
            not directory.exists
42      end
```

4.2 Task 2

With the following example directory and the invocation

```
delete_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) .....

Solution

a) 3, 2, 6, 5, 7, 4, 8, 9, 1
b) 8, 9, 3, 7, 6, 5, 4, 2, 1

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.

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

feature -- Removal

    remove
        -- Remove current item.
        -- Move cursor to right neighbor
        -- (or after if no right neighbor).

    remove_left
        -- Remove item to the left of cursor position.
        -- Do not move cursor.

    remove_right
        -- Remove item to the right of cursor position.
        -- Do not move cursor.

end -- class ARRAYED_LIST
```