Mock Exam 2

ETH Zurich

December 6,7 2010

Name: ________________________________

Group: ________________________________

Grading

Maximum points: 48 (+ 1 bonus point from task 4)
Points required for a passing grade (4.0): 24
Points required for a maxmimum grade (6.0): 43
The above required points can be entered in the grading spreadsheet along with the students' individual grades and their points will be calculated automatically.

Question	Points
1	
2	
3	
4	
Total	
Grade	

1 Terminology (12 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.

Example:

1. **Which of the following statements are true?**
   a. Classes exist only in the software text; objects exist only during the execution of the software. ☒
   b. Each object is an instance of its generic class. ☐
   c. An object is deferred if it has at least one deferred feature. ☐

---

1. Classes and objects.
   a. A class is the description of a set of possible run-time objects to which the same features are applicable. ☐
   b. If an object x is an instance of class C, then C is the generating class of x and x is described by C. ☐
   c. A class represents a category of things. An object represents one of these things. ☐
   d. An object represents a category of things. A class represents one of these things. ☐
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 *all* the possible changes in a function redefinition?
   a. To change the implementation and the name. ☐
   b. To change the list of argument types, the result type, the contract, and the implementation. ☐
   c. To change the list of argument types, the result type, the contract, the name, and the implementation. ☐
   d. To change the list of argument types, the result type, and the implementation. ☐

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. Information hiding...
   - a. ...is the technique of presenting client programmers with an interface that only contains the public features of a class. ☐
   - b. ...is the technique of presenting client programmers with an interface that includes only features that have built-in security controls. ☐
   - c. ...is the technique of presenting client programmers with an interface that includes a superset of the properties of a software element. ☐
   - d. ...is the technique of presenting client programmers with an interface that includes only a subset of the properties of a software element. ☐

6. 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. ☐

1.1 Grading

1. Classes and objects.
   - a. A class is the description of a set of possible run-time objects to which the same features are applicable. ☒
   - b. If an object x is an instance of class C, then C is the generating class of x and x is described by C. ☒
   - c. A class represents a category of things. An object represents one of these things. ☒
   - d. An object represents a category of things. A class represents one of these things. ☐
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 *all* the possible changes in a function redefinition?
   - a. To change the implementation and the name. ☐
   - b. To change the list of argument types, the result type, the contract, and the implementation. ☒
   - c. To change the list of argument types, the result type, the contract, the name, and the implementation. ☐
   - d. To change the list of argument types, the result type, and the implementation. ☐
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. Information hiding...

a. ...is the technique of presenting client programmers with an interface that includes only features that have been explicitly defined as public. ☐

b. ...is the technique of presenting client programmers with an interface that includes only features that have built-in security controls. ☐

c. ...is the technique of presenting client programmers with an interface that includes a superset of the properties of a software element. ☐

d. ...is the technique of presenting client programmers with an interface that includes only a subset of the properties of a software element. ☒

6. 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. ☒

1. a, b, c
2. c
3. b
4. c, d
5. d
6. 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
```

```
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

                ...................................................................

                ...................................................................
```

```
                    ..........................................................................
                end
            end
        ensure

            ..........................................................................

            ..........................................................................

            ..........................................................................
        end

    set_color (a_color: COLOR)
        require

            ..........................................................................

            ..........................................................................

            ..........................................................................
        do
            color := a_color
        ensure

            ..........................................................................

            ..........................................................................

            ..........................................................................
        end

invariant

    ..........................................................................

    ..........................................................................

    ..........................................................................

end
```

2.3 Grading

```
class
    CAR

create
    make

feature {NONE} -- Creation

    make
```

```
			-- Creates a default car.
		require
			-- 0.5 points for nothing
		do
			create {LINKED_LIST [CAR_DOOR]} doors.make
		ensure
			not is_convertible -- 0.5 points
			doors /= Void and then doors.count = 0 -- 0.5 points
			color = Void -- 0.5 points
		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
			-- 0.5 points for nothing
		do
			is_convertible := a_is_convertible
		ensure
			is_convertible = a_is_convertible -- 0.5 points
		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) -- 1 point
			-- a_doors /= Void and then (a_doors.count = 0 or a_doors.count = 2 or a_doors.
				count = 4) -> 0.5 points
		local
			door_index: INTEGER
		do
			doors.wipe_out
			if a_doors /= Void then
				from
					door_index := 1
				invariant
					doors.count + 1 = door_index -- 1 point
					door_index >= 1 and door_index <= a_doors.count + 1 -- 1 point
				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 −− 1 point
                end
            end
        ensure
            (a_doors = Void and doors.count = 0) or (a_doors /= Void and then a_doors.count
                = doors.count) −− 1 point
            −− a_doors.count = doors.count −> 0.5 points if there is the "a_doors /= Void"
                precondition
        end

    set_color (a_color: COLOR)
        require
            −− 0.5 points for nothing
        do
            color := a_color
        ensure
            color = a_color −− 0.5 points
        end

invariant
    doors /= Void −− 0.5 points
    doors.count = 0 or doors.count = 2 or doors.count = 4 −− 0.5 points

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 Tree Iteration (12 Points)

The following class *TREE* [*G*] represents n-ary trees. A tree consists of a root node, which can have arbitrarily many children nodes. Each child node itself can have arbitrarily many children. In fact each child node itself is a tree, with itself as a root node.

```
class TREE [G]

create
    make

feature {NONE} -- Initialization

    make (v: G)
            -- Create new cell with value 'v'.
        require
            v_not_void: v /= Void
        do
            value := v
            create {LINKED_LIST [TREE [G]]} children.make
        ensure
            value_set: value = v
        end

feature -- Access

    value: G
            -- Value of node

    children: LIST [TREE [G]]
            -- Child nodes of this node

feature -- Insertion

    put (v: G)
            -- Add child cell with value 'v' as last child.
        require
            v_not_void: v /= Void
        local
            c: TREE [G]
        do
            create c.make (v)
            children.extend (c)
        ensure
            one_mode: children.count = old children.count + 1
            inserted: children.last.value = v
        end

invariant
    children_not_void: children /= Void
    value_not_void: value /= Void
```

```
end
```

The following gives relevant aspects of the interface of class *LIST* [*G*]. Class *LINKED_LIST* [*G*] is a descendant of class *LIST* [*G*].

```
deferred class interface LIST [G]

feature -- Access

    index: INTEGER
            -- Index of current position.

    item: G
            -- Item at current position.
        require
            not_off: not off

feature -- Measurement

    count: INTEGER
            -- Number of items.

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?

    off: BOOLEAN
            -- Is there no current item?

    is_empty: BOOLEAN
            -- Is structure empty?

feature -- Cursor movement

    back
            -- Move to previous position.
        require
            not_before: not before
        ensure
            moved_back: index = old index - 1

    finish
            -- Move cursor to last position.
            -- (No effect if empty)
        ensure
            not_before: not is_empty implies not before

    forth
            -- Move to next position.
```

```
        require
            not_after: not after
        ensure
            moved_forth: index = old index + 1

    start
            -- Move cursor to first position.
            -- (No effect if empty)
        ensure
            not_after: not is_empty implies not after

feature -- Element change

    extend (v: G)
            -- Add a new occurrence of 'v'.
        ensure
            one_more: count = old count + 1

invariant
    before_definition: before = (index = 0)
    after_definition: after = (index = count + 1)
    non_negative_index: index >= 0
    index_small_enough: index <= count + 1
    off_definition: off = ((index = 0) or (index = count + 1))
    not_both: not (after and before)
    before_constraint: before implies off
    after_constraint: after implies off
    empty_definition: is_empty = (count = 0)
    non_negative_count: count >= 0
end
```

4.1 Traversing the tree

Class *APPLICATION* below first builds a tree and then prints the values of the tree in two different ways: pre-order and post-order.

Fill in the missing source code of the features *print_pre_order* and *print_post_order* so they will print the node values of an arbitrary tree. For example, a call of feature *make* in class *APPLICATION* should print out the following:

```
1
1.1
1.1.1
1.1.2
1.2
1.3
1.3.1
---
1.1.1
1.1.2
1.1
1.2
1.3.1
1.3
1
```

```
class APPLICATION

create
    make

feature

    make
            -- Run program.
        local
            root: TREE [STRING]
            cell: TREE [STRING]
        do
            create root.make ("1")
            root.put ("1.1")
            cell := root.children.last
            cell.put ("1.1.1")
            cell.put ("1.1.2")
            root.put ("1.2")
            root.put ("1.3")
            cell := root.children.last
            cell.put ("1.3.1")

            print_pre_order (root)
            io.put_string ("---")
            io.put_new_line
            print_post_order (root)
        end
```

```
print_pre_order (t: TREE [STRING])
        -- Print tree in pre-order.
    require
        t_not_void: t /= Void
    local

        ...........................................................................

        ...........................................................................

        ...........................................................................
    do

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................

        ...........................................................................
    end
```

```
print_post_order (t: TREE [STRING])
        -- Print tree in post-order.
    require
        t_not_void: t /= Void
    local

        ..........................................................................

        ..........................................................................

        ..........................................................................
    do

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................

        ..........................................................................
    end

end
```

4.2 Solution

```
class APPLICATION

create
    make

feature

    make
            -- Run program.
        local
            root: TREE [STRING]
            cell: TREE [STRING]
        do
            create root.make ("1")
            root.put ("1.1")
            cell := root.children.last
            cell.put ("1.1.1")
            cell.put ("1.1.2")
            root.put ("1.2")
            root.put ("1.3")
            cell := root.children.last
            cell.put ("1.3.1")

            print_pre_order (root)
            io.put_string ("---")
            io.put_new_line
            print_post_order (root)
        end

    print_pre_order (t: TREE [STRING])
            -- Print tree in pre-order.
        require
            t_not_void: t /= Void
        do
            io.put_string (t.value)
            io.put_new_line
            from
                t.children.start
            until
                t.children.off
            loop
                print_pre_order (t.children.item)
                t.children.forth
            variant
                t.children.count - t.children.index + 1
            end
        end


    print_post_order (t: TREE [STRING])
```

```
			-- Print tree in post-order.
		require
			t_not_void: t /= Void
		do
			from
				t.children.start
			until
				t.children.off
			loop
				print_post_order (t.children.item)
				t.children.forth
			variant
				t.children.count - t.children.index + 1
			end
			io.put_string (t.value)
			io.put_new_line
		end

end
```

4.3 Grading

Correction per routine:

- Loop present: 1 Point
- From part: 1 Point
- Exit condition: 1 Point
- Iteration: 1 Point
- Print (at correct place): 1 Point
- Recursive call (at correct place): 1 Point
- Loop variant: 0.5 Bonus points