Mock Exam 2

ETH Zurich

December 5,6 2011

Name: ______________________________

Group: ______________________________

Question	Max Points	Points
1	10	
2	11	
3	15	
4	12	
Total	48	

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.

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. ☐

1.1 Solution

1. Classes and objects.
   a. A class may be created at run-time. ☐
   b. A class may be deferred or effective. ☒
   c. An object may be created at run-time. ☒
   d. An object may be deferred or effective. ☐

2. Features.
   a. Every feature is either a routine or a procedure. ☐
   b. Every query is either an attribute or a function. ☒
   c. The result value of commands is always computed. ☐
   d. Every command is implemented as a procedure. ☒

3. Inheritance and polymorphism.
   a. A class can always call all features of its immediate parent classes. ☒
   b. When different parents of a class have features with the same name, you always have to rename all but one of them. ☐
   c. An object attached to a polymorphic entity can change its type at runtime. ☐
   d. If the target variable and source expression of an attachment have different types, then the attachment is polymorphic. ☒

4. Generics.
   a. Different generic derivations of the same generic class always conform to each other. ☐
   b. A generic class is a class that has one or more generic parameters. ☒
   c. Only non-generic classes can be used as generic parameters. ☐
   d. Genericity is used to specialize a class and inheritance is used to parametrize a class. ☐

5. Contracts.
   a. It is the responsability of the caller of a routine that the precondition of the routine is satisfied. ☒
   b. It is the responsability of the caller of a routine that the class invariant of the target object is satisfied. ☐
   c. If a loop is never executed (the exit condition is true from the beginning) then the loop invariant does not have to hold. ☐
   d. If a routine redefinition contains a new postcondition, this condition has to hold in addition to the inherited postcondition. ☒

2 Design by Contract (11 Points)

Classes *CARD* and *DECK* are part of a software system that models a card game. The following is an extract from the game rules booklet:

1. A deck is initially made of 36 cards.
2. Every card represents a value in the range 2..10. Furthermore, every card represents one color out of four possible colors.
3. The colors represented in the game cards are red ('R'), white ('W'), green ('G') and blue ('B').
4. The players can look at the top card and if there are cards left remove the top card.

Your task is to fill in the contracts of the two classes *CARD* and *DECK* (preconditions, postconditions and class invariants), according to the specification given. You are not allowed to change the interfaces of the classes or any of the already given implementations. Note that the number of dotted lines does not indicate the number of assertions that you have to provide, or if you have to provide a contract at all.

2.1 Solution

```
class
    CARD

create
    make

feature -- Creation

    make (a_color: CHARACTER; a_value: INTEGER)
            -- Create a card given a color and a value.
        require
            is_valid_color (a_color)
            is_in_range (a_value)
        do
            color := a_color
            value := a_value
        ensure
            color_set: color = a_color
            value_set: value = a_value
        end

feature -- Status report

    color: CHARACTER
            -- The card color

    value: INTEGER
            -- The card value

    is_valid_color (c: CHARACTER): BOOLEAN
            -- Is 'c' a valid color?
```

```
        do
            Result := (c = 'R' or c = 'B' or c = 'W' or c = 'G')
        ensure
            Result = (c = 'R' or c = 'B' or c = 'W' or c = 'G')
        end

    is_in_range (n: INTEGER): BOOLEAN
            -- Is 'n' in the acceptable range of values?
        do
            Result := (2 <= n and n <= 10)
        ensure
            Result = (2 <= n and n <= 10)
        end

invariant
    valid_color: is_valid_color (color)
    valid_range: is_in_range (value)

end
```

```
class
    DECK

create
    make

feature -- Creation

    make
            -- Create deck.
        do
            create card_list
            across << 'R', 'B', 'W', 'G' >> as c loop
                across 2 |..| 10 as n loop
                    card_list.extend_back (create {CARD}.make (c.item, n.item))
                end
            end
        ensure
            deck_filled: count = 36
        end

feature -- Status report

    is_empty: BOOLEAN
            -- Is this deck empty?
        do
            Result := card_list.is_empty
        ensure
            Result = card_list.is_empty
        end

    count: INTEGER
```

```
			-- Number of remaining cards in deck.
		do
			Result := card_list.count
		ensure
			Result = card_list.count
		end

feature -- Access

	top_card: CARD
			-- Top card of deck.
		do
			if not card_list.is_empty then
				Result := card_list.last
			end
		ensure
			no_card_when_empty: is_empty implies Result = Void
			right_card_when_not_empty: not is_empty implies Result = card_list.last
		end

feature -- Basic operations

	remove_top_card
			-- Remove top card from deck.
		require
			not_empty: not is_empty
		do
			card_list.remove_back
		ensure
			one_card_less_in_deck: count = old count - 1
			remaining_cards_still_there :
				across card_list as i all i.item = (old card_list).item (i.index) end
		end

	shuffle
			-- Shuffle remaining cards.
		local
			l_new_list : V_LINKED_LIST [CARD]
			l_random: V_RANDOM
			i: INTEGER
		do
			from
				create l_random
				create l_new_list
			until
				card_list.is_empty
			loop
				l_random.forth
				i := l_random.bounded_item (1, card_list.count)
				l_new_list.extend_back ( card_list.item (i))
				card_list.remove_at (i)
			variant
```

```
                    card_list .count
                end
                card_list := l_new_list
            ensure
                count_unchanged: count = old count
                cards_unchanged: across old card_list as c all card_list .has (c.item) end
            end

feature {NONE} -- Implementation

    card_list : V_LINKED_LIST [CARD]
            -- Implementation of the card list

invariant
    is_legal_deck : 0 <= count and count <= 36
    card_list_attached : card_list /= Void
    count_empty_relation: is_empty = (count = 0)
    cards_attached: not card_list .has (Void)

end
```

3 Inheritance (15 points)

Below you see the class *GAME_CHARACTER*. The class represents game characters. There are three types of game characters: dragon, marshmallow man and zombie. Every character has a health level in the range of 0 to 100, where 0 means that the character is dead and 100 that it has full strength. Since zombies are dead by definition, their health level stays at 0 at all times. Each of the character types has a damage potential that it can inflict on others. For all of them the damage doubles if the character is angry.

Listing 1: Class *GAME_CHARACTER*

```
class
  GAME_CHARACTER

create
  make

feature -- Initialization

  make (t: INTEGER)
      -- Initialize with type 't'.
    require
      t_valid: (t = marshmallow_man xor t = dragon xor t = zombie) and not
             (t = marshmallow_man and t = dragon and t = zombie)
    do
      type := t
      if type = zombie then
        health := 0
      else
        health := 100
      end
    ensure
      type_set: type = t
    end

feature -- Access

  type: INTEGER
      -- Type of character

  health: INTEGER
      -- Health of character (0: dead, 100: full strength)

  damage: INTEGER
      -- Damage that the character can do
    do
      if type = zombie then
        Result := zombie_damage
      elseif type = marshmallow_man then
        Result := marshmallow_man_damage
      else
        Result := dragon_damage
      end
```

```
      if is_angry then
        Result := Result * 2
      end
    ensure
      zombie: not is_angry and type = zombie implies Result = zombie_damage
      angry_zombie: is_angry and type = zombie implies Result = 2*zombie_damage
      dragon: not is_angry and type = dragon implies Result = dragon_damage
      angry_dragon: is_angry and type = dragon implies Result = 2*dragon_damage
      marshmallow_man: not is_angry and type = marshmallow_man implies Result =
          marshmallow_man_damage
      angry_marshmallow_man: is_angry and type = marshmallow_man implies Result = 2*
          marshmallow_man_damage
    end

feature -- Status report

  is_dead: BOOLEAN
      -- Is the character dead?
    do
      Result := (health = 0)
    ensure
      Result_set: Result = (health = 0)
    end

  is_angry: BOOLEAN
      -- Is the character angry?
      -- (Then it can do more damage!)

feature -- Element change

  set_health (h: INTEGER)
      -- Set 'health' to 'h'.
    require
      h_valid: h >= 0 and h <= 100
      h_for_zombie: type = zombie implies h = 0
    do
      health := h
    ensure
      health_set: health = h
    end

  set_angry (b: BOOLEAN)
      -- Set 'is_angry' to 'b'.
    do
      is_angry := b
    ensure
      is_angry_set: is_angry = b
    end

feature -- Constants

  marshmallow_man: INTEGER = 1
```

```
        -- Marshmallow man

    dragon: INTEGER = 2
        -- Dragon

    zombie: INTEGER = 3
        -- Zombie (is always dead)

    zombie_damage: INTEGER = 1
        -- Damage that a zombie does

    dragon_damage: INTEGER = 2
        -- Damage that a dragon does

    marshmallow_man_damage: INTEGER = 3
        -- Damage that a marshmallow man does

invariant

    type_valid : (type = marshmallow_man xor type = dragon xor type = zombie) and not (
        type = marshmallow_man and type = dragon and type = zombie)
    health_valid : health >= 0 and health <= 100
    zombie_always_dead: type = zombie implies health = 0

end
```

The above code does not exhibit a nice object-oriented design and it can hardly be called reusable. Redesign the code such that it uses inheritance instead of the *type* attribute to represent the three types of game characters. Write a **deferred** ancestor class *NEW_GAME_CHARACTER* and effective descendants *ZOMBIE*, *MARSHMALLOW_MAN*, and *DRAGON* that inherit from *NEW_GAME_CHARACTER*.

Your design should

- result in the deletion of the *type* attribute.
- result in the same behavior for the three types of game characters as the original code of class *GAME_CHARACTER*.
- include semantically equivalent contracts as the original code of class *GAME_CHARACTER*.

If a feature stays the same in your re-factored code as in the original code, please indicate it by giving the full feature signature and adding a comment `-- See original`.

Example:

```
    is_dead: BOOLEAN
        -- See original.
```

Listing 2: Class *NEW_GAME_CHARACTER*

```
deferred class
  NEW_GAME_CHARACTER

feature -- Access

  health: INTEGER
      -- Health of character (0: dead, 100: full strength)

  damage: INTEGER
      -- Damage that the character can do
    do
      Result := damage_constant
      if is_angry then
        Result := Result * 2
      end
    ensure
      not_angry: not is_angry implies Result = damage_constant
      angry: is_angry implies Result = 2*damage_constant
    end

feature -- Status report

  is_dead: BOOLEAN
      -- Is the character dead?
    do
      Result := (health = 0)
    ensure
      Result_set: Result = (health = 0)
    end

  is_angry: BOOLEAN
      -- Is the character angry?
      -- (Then it can do more damage!)

  is_valid_health (h: INTEGER): BOOLEAN
      -- Is 'h' a valid health for the character?
    deferred
    ensure
      Result implies (h >= 0 and h <= 100)
      -- other possiblilty: no postcondition
    end

feature -- Element change

  set_health (h: INTEGER)
      -- Set 'health' to 'h'.
    require
      h_valid: is_valid_health (h)
    do
      health := h
    ensure
```

```
      health_set: health = h
    end

  set_angry (b: BOOLEAN)
      -- Set 'is_angry' to 'b'.
    do
      is_angry := b
    ensure
      is_angry_set: is_angry = b
    end

feature -- Constants

  damage_constant: INTEGER
      -- Damage that a character does
    deferred
    end

invariant

  health_valid: is_valid_health (health)
  -- other possiblilty: health >= 0 and health <= 100

end
```

Listing 3: Class *ZOMBIE*

```
class
  ZOMBIE

inherit

  NEW_GAME_CHARACTER

create
  make

feature -- Initialization

  make
      -- Initialize health 0.
    do
      health := 0
    ensure
      health_set: health = 0
    end

feature -- Status report

  is_valid_health (h: INTEGER): BOOLEAN
      -- Is 'h' a valid health for the character?
    do
      Result := (h = 0)
```

```
    ensure then
      Result = (h = 0)
    end

feature -- Constants

  damage_constant: INTEGER = 1

invariant

  zombie_always_dead: health = 0

end
```

Listing 4: Class *DRAGON*

```
class
  DRAGON

inherit

  NEW_GAME_CHARACTER

create
  make

feature -- Initialization

  make
      -- Initialize with health 100.
    do
      health := 100
    ensure
      health_set: health = 100
    end

feature -- Status report

  is_valid_health (h: INTEGER): BOOLEAN
      -- Is 'h' a valid health for the character?
    do
      Result := (h >= 0 and h <= 100)
    ensure then
      Result = (h >= 0 and h <= 100)
    end

feature -- Constants

  damage_constant: INTEGER = 2

end
```

Listing 5: Class *MARSHMALLOW_MAN*

```
class
  MARSHMALLOW_MAN

inherit

  NEW_GAME_CHARACTER

create
  make

feature -- Initialization

  make
      -- Initialize with health 100.
    do
      health := 100
    ensure
      health_set: health = 100
    end

feature -- Status report

  is_valid_health (h: INTEGER): BOOLEAN
      -- Is 'h' a valid health for the character?
    do
      Result := (h >= 0 and h <= 100)
    ensure then
      Result = (h >= 0 and h <= 100)
    end

feature -- Constants

  damage_constant: INTEGER = 3

end
```

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 children
        ensure
            value_set: value = v
        end

feature -- Access

    value: G
            -- Value of node

    children: V_LINKED_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_back (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 *V_LINKED_LIST [G]* and *V_LINKED_LIST_ITERATOR [G]*.

```
class interface V_LINKED_LIST [G]

feature -- Access

    first: G
            -- First element.
        require
            not_empty: not is_empty

    last: G
            -- Last element.
        require
            not_empty: not is_empty

    item (i: INTEGER): G
            -- Value at position 'i'.
        require
            has_key: has_index (i)


feature -- Status report

    is_empty: BOOLEAN
            -- Is container empty?

feature -- Extension

    extend_back (v: G)
            -- Insert 'v' at the back.

    extend_front (v: G)
            -- Insert 'v' at the front.

feature -- Measurement

    count: INTEGER
            -- Number of elements.

feature -- Iteration

    new_cursor: V_LINKED_LIST_ITERATOR [G]
            -- New iterator pointing to the first position.
end
```

```
class interface V_LINKED_LIST_ITERATOR [G]

create
    default_create

feature -- Access
```

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

    index: INTEGER_32
            -- Current position.

feature -- Status report

    off: BOOLEAN
            -- Is current position off scope?

    after: BOOLEAN
            -- Is current position after the last container position?

    before: BOOLEAN
            -- Is current position before the first container position?

feature -- Cursor movement

    start
            -- Go to the first position.
        ensure
            index_effect: index = 1

    finish
            -- Go to the last position.
        ensure
            index_effect: index = sequence.count

    forth
            -- Move one position forward.
        require
            not_off: not off

    back
            -- Go one position backwards.
        require
            not_off: not off

invariant
    not_both: not (after and before)
    before_constraint: before implies off
    after_constraint: after implies off

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
```

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
                -- using across
            io.put_string (t.value)
            io.put_new_line

            across
                t.children as i
            loop
                print_pre_order (i.item)
            end
        end


    print_post_order (t: TREE [STRING])
            -- Print tree in post-order.
        require
            t_not_void: t /= Void
        local
            i: V_LINKED_LIST_ITERATOR [TREE [STRING]]
        do
                -- using normal loop
            from
                i := t.children.new_cursor
            until
                i.off
            loop
                print_post_order (i.item)
                i.forth
            variant
                t.children.count - i.index + 1
            end

            io.put_string (t.value)
            io.put_new_line
        end

end
```