Mock Exam 2

ETH Zurich

December 5,6 2011

Name: ____________________________________________

Group: ____________________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Max Points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>48</strong></td>
<td></td>
</tr>
</tbody>
</table>
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. 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. ☐
   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 \emph{CARD} and \emph{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 \emph{CARD} and \emph{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.

class CARD

create

make

feature -- Creation

make (a_color: CHARACTER; a_value: INTEGER)

--- Create a card given a color and a value.

require

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

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

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

ensure

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

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

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

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?
require

ensure

end

is_in_range (n: INTEGER): BOOLEAN
  -- Is 'n' in the acceptable range of values?
require

ensure

end

invariant

end
class
  DECK

create
  make

feature -- Creation

  make
    -- Create deck.
    require

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

    do
      create  card_list
      across << 'R', 'B', 'W', 'G' >> as c loop
      across << 2, 3, 4, 5, 6, 7, 8, 9, 10 >> as n loop
      card_list . extend_back (create {CARD}.make (c.item, n.item))
    end
  end

ensure

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

end

feature -- Status report

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

  count: INTEGER
    -- Number of remaining cards in deck.
    do
      Result := card_list.count
    end
feature -- Access

    top_card: CARD
        -- Top card of deck.
    require

    do
        if not card_list.is_empty then
            Result := card_list.last
        end
    ensure

    end

feature -- Basic operations

    remove_top_card
        -- Remove top card from deck.
    require

    do
        card_list.remove_back
    ensure

    end
shuffle
   -- Shuffle remaining cards.
require
   ...
local
   \texttt{l\_new\_list} : \texttt{V\_LINKED\_LIST} [\texttt{CARD}]
   \texttt{l\_random} : \texttt{V\_RANDOM}
   \texttt{i} : \texttt{INTEGER}
do
   from
   create \texttt{l\_random}
   create \texttt{l\_new\_list}
until
   \texttt{card\_list\_is\_empty}
loop
   \texttt{l\_random.forth}
   \texttt{i := l\_random.bounded\_item (1, card\_list\_count)}
   \texttt{l\_new\_list.extend\_-back (card\_list\_item (i))}
   \texttt{card\_list.remove\_at (i)}
variant
   ...
end
\texttt{card\_list := l\_new\_list}
ensure
   ...
end
feature \{ \texttt{NONE} \} -- Implementation
   \texttt{card\_list} : \texttt{V\_LINKED\_LIST} [\texttt{CARD}]
   -- Implementation of the card list
invariant
   ...
end
3 Inheritance (15 points)

Below you see the class \textit{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 \textit{GAME\_CHARACTER}

```plaintext
1 class
2 GAME\_CHARACTER
3
4 create
5 make

7 feature -- Initialization

9 make (t: INTEGER)
10 -- Initialize with type ‘t’.
11 require
12 t_valid: (t = marshmallow\_man xor t = dragon xor t = zombie) and not
13 (t = marshmallow\_man and t = dragon and t = zombie)
14 do
15 type := t
16 if type = zombie then
17 health := 0
18 else
19 health := 100
20 end
21 ensure
22 type_set: type = t
23 end

25 feature -- Access

27 type: INTEGER
28 -- Type of character
29
30 health: INTEGER
31 -- Health of character (0: dead, 100: full strength)
32
33 damage: INTEGER
34 -- Damage that the character can do
35 do
36 if type = zombie then
37 Result := zombie\_damage
38 elseif type = marshmallow\_man then
39 Result := marshmallow\_man\_damage
40 else
41 Result := dragon\_damage
42 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
93  -- Marshmallow man

95  dragon: INTEGER = 2
    -- Dragon

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

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

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

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

109 invariant

111 type_valid : (type = marshmallow_man xor type = dragon xor type = zombie) and not (type = marshmallow_man and type = dragon and type = zombie)

113 health_valid : health >= 0 and health <= 100

115 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 NEWGAMECHARACTER and effective descendants ZOMBIE, MARSHMALLOW_MAN, and DRAGON that inherit from NEWGAMECHARACTER.

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.
deferred class NEW_GAME_CHARACTER

...
class ZOMBIE

end
class MARSHMALLOW_MAN

end
class DRAGON

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
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

\[
\text{first : } G \\
\quad \text{-- First element.} \\
\quad \text{require} \\
\quad \quad \text{not} \_\text{empty: not} \ is\_\text{empty}
\]

\[
\text{last : } G \\
\quad \text{-- Last element.} \\
\quad \text{require} \\
\quad \quad \text{not} \_\text{empty: not} \ is\_\text{empty}
\]

\[
\text{item} \ (i : \ \text{INTEGER}) : G \\
\quad \text{-- Value at position ‘i’}. \\
\quad \text{require} \\
\quad \quad \text{has} \_\text{key: has} \_\text{index} \ (i)
\]

feature -- Status report

\[
\text{is} \_\text{empty: BOOLEAN} \\
\quad \text{-- Is container empty?}
\]

feature -- Extension

\[
\text{extend}\_\text{back} \ (v : G) \\
\quad \text{-- Insert ‘v’ at the back.}
\]

\[
\text{extend}\_\text{front} \ (v : G) \\
\quad \text{-- Insert ‘v’ at the front.}
\]

feature -- Measurement

\[
\text{count: INTEGER} \\
\quad \text{-- Number of elements.}
\]

feature -- Iteration

\[
\text{new}\_\text{cursor: V\_LINKED\_LIST\_ITERATOR} \ [G] \\
\quad \text{-- New iterator pointing to the first position.}
\]
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
```

class *APPLICATION*

create

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)
```


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

end
end