Mock Exam 2

ETH Zurich

December 6, 7 2010

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>
<tr>
<td>Grade</td>
<td></td>
</tr>
</tbody>
</table>
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.
   □
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

```plaintext
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. do
    is_convertible := a_is_convertible
    ensure

50. end

52. set_doors (a_doors: ARRAY [CAR.DOOR])
    require

54. local
    door_index: INTEGER
    do
      doors.wipe_out
      if a_doors /= Void then
        from
        door_index := 1
        invariant

58. until
    door_index > a_doors.count
    loop
      doors.extend (a_doors [door_index])
      door_index := door_index + 1
      variant

60.
set_color (a_color: COLOR)

require

do
  color := a_color

ensure

end

invariant
3 Inheritance and polymorphism (14 Points)

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

1 \texttt{deferred class PRODUCT}

3 \texttt{feature -- Main operations}

5 \texttt{set\_price (r: REAL)}
   \hspace{1cm} -- Set ‘price’ to ‘r’.
7 \texttt{require}
   \hspace{1cm} \texttt{r\_non\_negative: r >\geq 0}
9 \texttt{do}
11 \texttt{price := r}
13 \texttt{ensure}
15 \texttt{price\_set: price = r}
17 \texttt{end}

15 \texttt{feature -- Access}

17 \texttt{price: REAL}
   \hspace{1cm} -- How much the product costs
19 \texttt{description: STRING}
   \hspace{1cm} -- Brief description
21 \texttt{deferred}
23 \texttt{end}
25 \texttt{invariant}
27 \texttt{non\_negative\_price: price \geq 0}
27 \texttt{valid\_description: description /= Void and then not description.is\_empty}
29 \texttt{end}
31 \texttt{deferred class COFFEE}

33 \texttt{inherit}
35 \texttt{PRODUCT}
37 \texttt{feature -- Main operations}
39 \texttt{make}
41 \texttt{-- Prepare the coffee.}
43 \texttt{do}
45 \texttt{print ("I am making you a coffee.")}
47 \texttt{end}
49 \texttt{end}
51 \texttt{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:

\[
\begin{align*}
\text{product: PRODUCT} \\
\text{coffee: COFFEE} \\
\text{espresso: ESPRESSO} \\
\text{cappuccino: CAPPUCCINO} \\
\text{cake: CAKE}
\end{align*}
\]

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}
\[
\text{io.put_string (product.description)}
\]

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

3. \texttt{create cappuccino.make}
\[
\text{io.put_string (cappuccino.description)}
\]

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

5. \texttt{create cake.make (6.50)}
\[
\text{product := cake} \\
\text{io.put_string (product.description)}
\]
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 Tree Iteration (12 Points)

The following class \( \text{TREE}[\text{G}] \) 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 \( \text{TREE}[\text{G}] \)

create

make

feature \{NONE\} -- Initialization

\begin{align*}
\text{make } (v: \text{G}) & \quad -- \text{Create new cell with value ‘v’}. \\
\text{require} & \\
\text{v\_not\_void: } v \neq \text{Void} & \\
\text{do} & \\
\text{value := v} & \\
\text{create } \{\text{LINKED\_LIST } [\text{TREE}[\text{G}]]\} & \text{children.make} \\
\text{ensure} & \\
\text{value\_set: value = v} & \\
\text{end}
\end{align*}

feature -- Access

\begin{align*}
\text{value: } \text{G} & \quad -- \text{Value of node} \\
\text{children: } \text{LIST}[\text{TREE}[\text{G}]] & \quad -- \text{Child nodes of this node}
\end{align*}

feature -- Insertion

\begin{align*}
\text{put } (v: \text{G}) & \quad -- \text{Add child cell with value ‘v’ as last child}. \\
\text{require} & \\
\text{v\_not\_void: } v \neq \text{Void} & \\
\text{local} & \\
\text{c: } \text{TREE}[\text{G}] & \\
\text{do} & \\
\text{create } c.\text{make } (v) & \\
\text{children.\text{extend } (c)} & \\
\text{ensure} & \\
\text{one\_mode: children.count = old children.count + 1} & \\
\text{inserted: } \text{children.\text{last:value} = v} & \\
\text{end}
\end{align*}

invariant

\begin{align*}
\text{children\_not\_void: children } \neq \text{Void} & \\
\text{value\_not\_void: value } \neq \text{Void}
\end{align*}
The following gives relevant aspects of the interface of class \textit{LIST}\( [G] \). Class \textit{LINKED\_LIST}\( [G] \) is a descendant of class \textit{LIST}\( [G] \).

\textbf{deferred class interface \textit{LIST}\( [G] \)}

\textbf{feature} \textit{−− Access}

\begin{itemize}
  \item \textit{index: INTEGER} \textit{−− Index of current position.}
  \item \textit{item: G} \textit{−− Item at current position.}
    \textbf{require}
    \textit{not\_off: not off}
\end{itemize}

\textbf{feature} \textit{−− Measurement}

\begin{itemize}
  \item \textit{count: INTEGER} \textit{−− Number of items.}
\end{itemize}

\textbf{feature} \textit{−− Status report}

\begin{itemize}
  \item \textit{after: BOOLEAN} \textit{−− Is there no valid cursor position to the right of cursor?}
  \item \textit{before: BOOLEAN} \textit{−− Is there no valid cursor position to the left of cursor?}
  \item \textit{off: BOOLEAN} \textit{−− Is there no current item?}
  \item \textit{is\_empty: BOOLEAN} \textit{−− Is structure empty?}
\end{itemize}

\textbf{feature} \textit{−− Cursor movement}

\begin{itemize}
  \item \textit{back} \textit{−− Move to previous position.}
    \textbf{require}
    \textit{not\_before: not before}
    \textbf{ensure}
    \textit{moved\_back: index = old index – 1}
  \item \textit{finish} \textit{−− Move cursor to last position.}
    \textit{−− (No effect if empty)}
    \textbf{ensure}
    \textit{not\_before: not is\_empty implies not before}
  \item \textit{forth} \textit{−− Move to next position.}
\end{itemize}
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
1.2
1.1
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)
  end
```
print_pre_order \( (t: \text{TREE}[\text{STRING}]) \)
\(--\) Print tree in pre-order.

\textbf{require}  
\textit{t\_not\_void: t \neq Void}

\textbf{local}  

\textbf{do}  

\textbf{end}
print_post_order (t: TREE[STRING])

-- Print tree in post-order.

require

\[ t \neq Void \]

local

do

end

end