Mock Exam 1

ETH Zurich

November 7, 8 2011

Name: 

Group: 

1 Terminology (10 points)

Solution

1. A command...
   ✓ a. call is an instruction.
   ✓ b. may modify an object.
       c. may appear in the precondition and the postcondition of another command but not in the
          precondition or the postcondition of a query.
       d. may appear in the class invariant.

2. The syntax of a program...
   a. is the set of properties of its potential executions.
   b. can be derived from the set of its objects.
   ✓ c. is the structure and the form of its text.
   d. may be violated at run-time.

3. 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.

4. Immediately before a successful execution of a creation instruction with target $x$ of type $C$...
   a. $x = Void$ must hold.
   b. $x /= Void$ must hold.
   ✓ c. the postcondition of the creation procedure may not hold.
   ✓ d. the precondition of the creation procedure must hold.

5. Void references...
✓ a. cannot be the target of a successful call.
b. are not default values for any type.
c. indicate expanded objects.
✓ d. can be used to terminate linked structures (e.g. linked lists).

2 Design by Contract (10 Points)

Solution

class
  CAR

create
  make

  feature {NONE} -- Creation

  make
    -- Creates a default car.
    require
      -- nothing
    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
        -- nothing
      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
doors_index: INTEGER
do
  doors.wipe_out
  if a_doors /= Void then
    from
    doors_index := 1
  invariant
    doors.count + 1 = doors_index
    doors_index >= 1 and doors_index <= a_doors.count + 1
  until
    doors_index > a_doors.count
  loop
    doors.extend (a_doors [doors_index])
    doors_index := doors_index + 1
  variant
    a_doors.count + 1 - doors_index
  end
  end
end
ensure
(a_doors = Void and doors.count = 0) or (a_doors /= Void and then a_doors.count = doors.count)
end

set_color (a_color: COLOR)
require
  -- nothing
do
  color := a_color
ensure
  color = a_color
end

invariant
  doors /= Void
  doors.count = 0 or doors.count = 2 or doors.count = 4
end

3 Inheritance: A Persistence Framework (12 Points)

Solution

1. manager_1: SERIALIZATION_MANAGER
   manager_2: BASIC_SERIALIZATION_MANAGER
   an_object: STRING
...
create manager_1.make
create manager_2.make
create an_object.make_from_string ("test")
manager_1 := manager_2
manager_1.store (an_object)
1. Suppose you want the framework to provide support for XML stored in a text file. Which of the following solutions seems more appropriate to you?

- a. Add one new class, namely `XML_FORMAT`, and make it inherit from `PERSISTENCE_FORMAT`.  
- b. Add the necessary code to handle the XML format to class `PERSISTENCE_FORMAT`. In addition, add a new class named `XML_SERIALIZATION_MANAGER` and make it inherit from `SERIALIZATION_MANAGER`.  
- c. Add three new classes, namely `XML_FORMAT`, `TEXTUAL_FORMAT` and `XML_SERIALIZATION_MANAGER`. The first of them, `XML_FORMAT`, will inherit from the second, `TEXTUAL_FORMAT`. In addition, `TEXTUAL_FORMAT` will inherit from `PERSISTENCE_FORMAT` and `XML_SERIALIZATION_MANAGER` will inherit from `SERIALIZATION_MANAGER`.  
- d. Add one new class, `TEXTUAL_FORMAT`, including the necessary code to serialize data in XML format, and make it inherit from `PERSISTENCE_FORMAT`.  
- e. Add two new classes, `XML_FORMAT` and `XML_SERIALIZATION_MANAGER`. Make `XML_FORMAT` inherit from `PERSISTENCE_FORMAT`, and make `XML_SERIALIZATION_MANAGER` inherit from `SERIALIZATION_MANAGER`.  
- f. Add two new classes, `XML_FORMAT` and `XML_SERIALIZATION_MANAGER`. Then add to class `SERIALIZATION_MANAGER` two attributes having types `XML_FORMAT` and `XML_SERIALIZATION_MANAGER`.

2. Suppose you have to write the code for feature `store` in a new class `ADVANCED_SERIALIZATION_MANAGER` that inherits from `BASIC_SERIALIZATION_MANAGER`. What do you have to do to be able to reuse the same implementation of feature `store` in `BASIC_SERIALIZATION_MANAGER`, but adding some code to it? The new code should be placed after the reused code.

- a. In `ADVANCED_SERIALIZATION_MANAGER`, use the keyword `redefine` after the clause `inherit from BASIC_SERIALIZATION_MANAGER`, and specify the new implementation in the body of feature `store`.  
- b. In `BASIC_SERIALIZATION_MANAGER`, specify the new implementation in the body of feature `store`. Nothing else is necessary because feature `store` is not implemented in class `SERIALIZATION_MANAGER`.  
- c. In `ADVANCED_SERIALIZATION_MANAGER`, use the keyword `undefine` after the clause `inherit from BASIC_SERIALIZATION_MANAGER`, and specify the new implementation in the body of feature `store`.  
- d. In `BASIC_SERIALIZATION_MANAGER`, use the keyword `redefine` after the clause `inherit from SERIALIZATION_MANAGER`, and specify the new implementation in the body of feature `store`. In addition, use the keyword `Precursor` to reuse the implementation from `SERIALIZATION_MANAGER`.  
- e. In `ADVANCED_SERIALIZATION_MANAGER`, use the keyword `redefine` after the clause `inherit from BASIC_SERIALIZATION_MANAGER`, and specify the new implementation in the body of feature `store`. In addition, use the keyword `Precursor` to reuse the implementation from `BASIC_SERIALIZATION_MANAGER`.  
- f. In `ADVANCED_SERIALIZATION_MANAGER`, use the keyword `undefine` after the clause `inherit from BASIC_SERIALIZATION_MANAGER`, and specify the new implementation in the body of feature `store`. In addition, use the keyword `Precursor` to reuse the implementation from `BASIC_SERIALIZATION_MANAGER`.

*It does not compile. You cannot create an object of class `SERIALIZATION_MANAGER` as it is a deferred class.*

2. `manager_1: SERIALIZATION_MANAGER
an_object: STRING
`
create \{BASIC\_SERIALIZATION\_MANAGER\}manager\_1.make
create an\_object.make\_from\_string ("test")
manager\_1.store (an\_object)

*It does compile and prints: Creating a basic serialization manager. Serialization an object.*

3. \textbf{manager}$_1$: \textsc{SERIALIZATION\_MANAGER}
manager\_2: \textsc{BASIC\_SERIALIZATION\_MANAGER}
\textsc{an}\_object: \textsc{STRING}

... create manager\_2.make
create an\_object.make\_from\_string ("test")
manager\_1 := manager\_2
manager\_1.store (an\_object)

*It does compile and prints: Creating a basic serialization manager. Serialization an object.*

4. \textbf{manager}$_1$: \textsc{SERIALIZATION\_MANAGER}
manager\_2: \textsc{BASIC\_SERIALIZATION\_MANAGER}
\textsc{an}\_object: \textsc{STRING}

... create manager\_2.make
create an\_object.make\_from\_string ("test")
manager\_2 := manager\_1
manager\_2.store (an\_object)

*It does not compile. You cannot assign a reference of a ancestor type to a reference of a descendant type.*

4 Inversion of Linked List (10 Points)

\textbf{Solution}

\texttt{invert}

\begin{verbatim}
2  -- Invert the order of the elements of the list.
3  -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6]
4 local L\_old\_list, L\_old\_list\_first, L\_new\_list: like first
6 do
7   from
8     L\_old\_list := first
9 until
10   -- Until the old list (L\_old\_list) is empty ...
11     L\_old\_list = Void
12 loop
13   -- ... remove the first element (L\_old\_list\_first) from the old list and ...
14     L\_old\_list\_first := L\_old\_list
15     L\_old\_list := L\_old\_list\_next
16   -- ... prepend it to the new list (L\_new\_list).
17     L\_old\_list\_first.set\_next (L\_new\_list)
18     L\_new\_list := L\_old\_list\_first
20 end
\end{verbatim}
-- Replace the old list by the new one.
first := new_list

ensure
count_remains_the_same: count = old count

end