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
        door_index: INTEGER
    do
        doors.wipe_out
        if a_doors /= Void then
            from
                door_index := 1
            invariant
                doors.count + 1 = door_index
                door_index >= 1 and door_index <= a_doors.count + 1
            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
            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.Serializing an object.*

3.
```
manager_1: SERIALIZATION_MANAGER
manager_2: BASIC_SERIALIZATION_MANAGER
an_object: 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.Serializing an object.*

4.
```
manager_1: SERIALIZATION_MANAGER
manager_2: BASIC_SERIALIZATION_MANAGER
an_object: 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)

Solution

```
invert
    -- Invert the order of the elements of the list.
    -- E.g. the list [6, 2, 8, 5] should be become [5, 8, 2, 6]
local
    l_old_list, l_old_list_first, l_new_list: like first
do
    from
        l_old_list := first
    until
            -- Until the old list ('l_old_list') is empty ...
        l_old_list = Void
    loop
            -- ... remove the first element ('l_old_list_first') from the old list and ...
        l_old_list_first := l_old_list
        l_old_list := l_old_list.next

            -- ... prepend it to the new list ('l_new_list').
        l_old_list_first.set_next (l_new_list)
        l_new_list := l_old_list_first
    end
```

```
22          -- Replace the old list by the new one.
        first := l_new_list
24  ensure
        count_remains_the_same: count = old count
26  end
```