Prova di esame del 16 giugno 2016

Esercizio 1) [10 punti]

Marcare le affermazioni che si ritengono vere. Ogni domanda può avere un qualunque numero naturale di affermazioni vere. Vengono **assegnati** 0.5 punti sia per ogni affermazione *vera che viene marcata* che per ogni affermazione *falsa che viene lasciata non marcata*. Analogamente vengono **sottratti** 0.5 punti sia per ogni affermazione *falsa che viene marcata* che per ogni affermazione *vera che viene lasciata non marcata*.

1. Ricordiamo che una classe C che è antenato sia di A che di B viene definita "*antenato comune minimo*" di A e B se non esiste un'altra classe X diversa da C che è antenato sia di A che di B ed è anche discendente di C. Si definisce **ereditarietà multipla** il caso di una classe discendente di due classi distinte A e B solo quando l'antenato comune minimo di A e B …
   a. … esiste
   b. … è unico e corrisponde ad *ANY*
   c. … esiste ed è diverso sia da A che da B
   d. … esiste ed è unico

2. Ridefinire una feature *f* ereditata vuol dire …
   a. … cambiare il nome della feature senza cambiarne l'implementazione
   b. … cambiare l'implementazione della feature senza cambiarne il nome
   c. … cambiare sia il nome che l'implementazione della feature
   d. … rendere la feature *effective*

3. Se il nome di una feature *f* compare nella clausola di dichiarazione **create** in una classe *C* allora …
   a. … può essere usata per creare istanze di *C*
   b. … deve essere usata per creare istanze di *C*
   c. … non può essere usata se non per creare istanze di *C*
   d. … può essere invocata per le istanze di *C*

4. Se la classe *C* esporta la feature *x* verso *ANY* allora …
   a. Nessuna istanza può invocare in modo qualificato la feature *x* di un'istanza di *C*
   b. Una qualunque istanza può invocare in modo qualificato la feature *x* di un'istanza di *C*
   c. Solo un'istanza di *C* può invocare in modo qualificato la feature *x* di un'altra istanza di *C*
   d. Solo la stessa istanza di *C* può invocare in modo non qualificato la feature *x* di sé stessa

5. Una classe *deferred* …
   a. Deve avere almeno una feature *deferred*
   b. Deve avere tutte le feature *deferred*
   c. Può avere tutte le feature *deferred*
   d. Può avere tutte le feature *effective*

SOLUZIONE:

1. Ricordiamo che una classe C che è antenato sia di A che di B viene definita "*antenato comune minimo*" di A e B se non esiste un'altra classe X diversa da C che è antenato sia di A che di B ed è anche discendente di C. Si definisce **ereditarietà multipla** il caso di una classe discendente di due classi distinte A e B solo quando l'antenato comune minimo di A e B …
   a. … esiste
   b. … è unico e corrisponde ad *ANY*
   c. … esiste ed è diverso sia da A che da B
   d. … esiste ed è unico

2. Ridefinire una feature *f* ereditata vuol dire …
   a. … cambiare il nome della feature senza cambiarne l'implementazione
   b. … cambiare l'implementazione della feature senza cambiarne il nome
   c. … cambiare sia il nome che l'implementazione della feature
   d. … rendere la feature *effective*

3. Se il nome di una feature *f* compare nella clausola di dichiarazione **create** in una classe *C* allora …
   a. … può essere usata per creare istanze di *C*
   b. … deve essere usata per creare istanze di *C*
   c. … non può essere usata se non per creare istanze di *C*
   d. … può essere invocata per le istanze di *C*

4. Se la classe *C* esporta la feature *x* verso *ANY* allora …
   a. Nessuna istanza può invocare in modo qualificato la feature *x* di un'istanza di *C*
   b. Una qualunque istanza può invocare in modo qualificato la feature *x* di un'istanza di *C*
   c. Solo un'istanza di *C* può invocare in modo qualificato la feature *x* di un'altra istanza di *C*
   d. Solo la stessa istanza di *C* può invocare in modo non qualificato la feature *x* di sé stessa

5. Una classe *deferred* …
   a. Deve avere almeno una feature *deferred*
   b. Deve avere tutte le feature *deferred*
   c. Può avere tutte le feature *deferred*
   d. Può avere tutte le feature *effective*

Si noti, a proposito della domanda 3, che il testo non dice che *f* sia la l'unica *feature* a comparire nella clausola **create**: pertanto, l'affermazione (a) è vera mentre la (b) è falsa. Se il testo invece dicesse che *f* è l'unica *feature* nella clausola **create**., viceversa (a) sarebbe falsa e (b) vera.

Esercizio 2) [10 punti]

La classe *INT_LINKABLE* modella un elemento di una lista che può rappresentare valori interi. La sua implementazione è la seguente:

```
class
   INT_LINKABLE
create
   make

feature -- accesso
   value : INTEGER
      -- L'intero memorizzato in questo elemento

   next : INT_LINKABLE
      -- Il successivo elemento della lista

feature -- operazioni fondamentali
   make (i : INTEGER)
      -- crea l'elemento
      do
         value := i
      end

   link_to (other: INT_LINKABLE)
         -- collega questo elemento con `other'
      do
         next := other
      ensure
         next = other
      end

   link_after (other: INT_LINKABLE)
         -- inserisce questo elemento dopo `other' conservando quello che c'era dopo di esso
      require
         other /= Void
      do
         link_to (other.next)
         other.link_to (Current)
      ensure
         other.next = Current
         other.next.next = old other.next
      end

end
```

La classe *INT_LINKED_LIST* modella una lista di interi. Una parte della sua implementazione è fornita qua sotto:

```
class
   INT_LINKED_LIST

feature -- accesso
   first_element: INT_LINKABLE
         -- Il primo elemento della lista

   last_element: INT_LINKABLE
         -- L'ultimo elemento della lista

   count: INTEGER
         -- Il numero di elementi della lista
```

Aggiungere alla classe *INT_LINKED_LIST* sia una feature *append* (*a_value*: *INTEGER*) che inserisce un nuovo elemento di valore *a_value* dopo l'ultimo elemento che una feature *prepend* (*a_value*: *INTEGER*) che inserisce un nuovo elemento di valore *a_value* prima del primo elemento. Il numero di righe vuote non ha alcuna relazione con il numero di istruzioni/dichiarazioni da scrivere.

```
feature -- operazioni fondamentali
   append (a_value: INTEGER)
        -- Aggiunge `a_value' dopo l'ultimo elemento

      require
         ____________________________________________________________________________
         ____________________________________________________________________________
      local
         ____________________________________________________________________________
         ____________________________________________________________________________
      do
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
      ensure
         ____________________________________________________________________________
         ____________________________________________________________________________
end
```

```
feature -- operazioni fondamentali
   prepend (a_value: INTEGER)
        -- Aggiunge `a_value' prima del primo elemento

      require
         ____________________________________________________________________________
         ____________________________________________________________________________
      local
         ____________________________________________________________________________
         ____________________________________________________________________________
      do
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
         ____________________________________________________________________________
      ensure
         ____________________________________________________________________________
         ____________________________________________________________________________
end
```

SOLUZIONE:

```
feature -- operazioni fondamentali
  append (a_value: INTEGER)
        -- Aggiunge `a_value' dopo l'ultimo elemento
     local
       new_element: like first_element
     do
       create new_element.make (a_value)
       if count = 0 then
         first_element := new_element
       else
         last_element.link_to (new_element)
       end
       last_element := new_element
       count := count + 1
     ensure
       uno_in_piu: count = old count + 1
       accodato: last_element.value = a_value
       collegato: (old last_element /= Void) implies (old last_element).next = last_element
     end
```

```
feature -- operazioni fondamentali
  prepend (a_value: INTEGER)
        -- Aggiunge `a_value' prima del primo elemento
     local
       new_element: like first_element
     do
       create new_element.make (a_value)
       if count /= 0 then
         new_element.link_to (first_element)
       else
         last_element := new_element
       end
       first_element := new_element
       count := count + 1
     ensure
       uno_in_piu: count = old count + 1
       intestato: first_element.value = a_value
       collegato: first_element.next = old first_element
     end
```

Si noti che i test che usano il valore di *count* possono essere sostituiti da test che usano il valore di *first_element*.

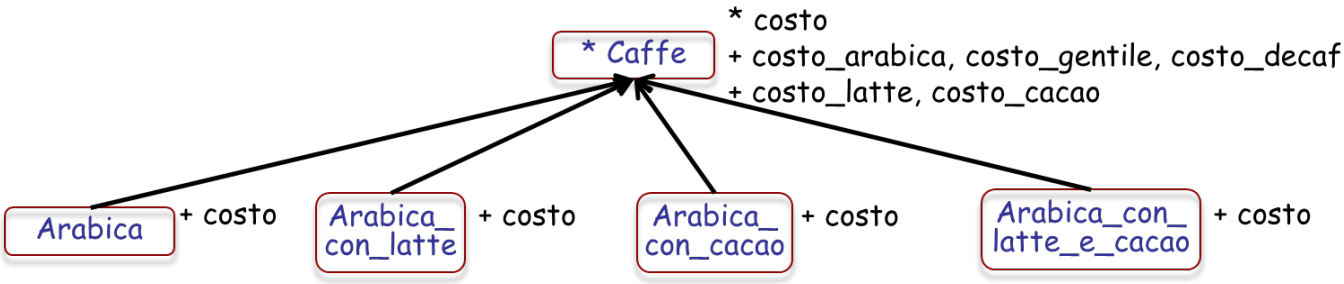
Esercizio 3) [10 punti]

Il software di gestione di una catena di bar specializzati nel servire caffè espresso di vari tipi deve modellare tre differenti specie di caffè (*arabica*, *gentile*, *decaf*) ognuna delle quali può essere servita con nessuna, una o due modifiche a scelta (*con_latte*, *con_cacao*). Nel caso di richiesta di due modifiche queste devono essere diverse tra loro. Il costo di un caffè espresso è determinato dal costo della specie usata per prepararlo più l'eventuale costo di tutte le modifiche che il cliente richiede.

Un'implementazione proposta per tale software suggerisce di implementare una classe *deferred* per rappresentare il caffè generico con una feature *deferred* per calcolare il costo del caffè, una serie di features *effective* che tengono traccia del costo base di ogni specie e di ogni modifica. Si veda qua sotto una traccia di tale implementazione.

```
deferred class
   CAFFE

feature {ANY}
   costo : REAL
      -- calcola il costo del caffè con le aggiunte desiderate.
      deferred
      end

   costo_arabica : REAL = 2.1
      -- il costo base di un espresso della specie arabica

   costo_gentile : REAL = 1.7
      -- il costo base di un espresso della specie gentile

   costo_decaf : REAL = 1.9
      -- il costo base di un espresso della specie decaf

   costo_latte : REAL = 0.1
      -- il costo base di una modifica con aggiunta di latte

   costo_cacao : REAL = 0.2
      -- il costo base di una modifica con aggiunta di cacao

end
```

Inoltre, la proposta suggerisce di avere una sottoclasse per tutte le possibili combinazioni di specie e modifiche. Ognuna di tali sottoclassi rende *effective* la feature per il calcolo del costo del caffè. Si veda qua sotto un frammento della struttura di classi che deriverebbe da questa proposta.



Il sistema proposto avrebbe come *root* la classe *SERVI_CAFFE* sotto descritta, la cui *root procedure* è *servi*. La scelta da parte dell'utente della specie "nessuna" viene interpretato come richiesta di terminare l'esecuzione del sistema. La scelta come prima modifica di "nessuna" viene interpretata come richiesta dell'utente di non aggiungere niente.

```
class
   SERVI_CAFFE

feature {NONE}
   caffe_da_servire : CAFFE

feature {ANY}
   servi
      local
         specie_scelta, modifica_1, modifica_2 : STRING
         somma_da_incassare : REAL
   do
      from
         -- "acquisizione dall'utente dei valori di specie_scelta, modifica_1, modifica_2"
      until
         specie_scelta ~ "nessuna"
      loop
         if specie_scelta ~ "arabica" then
            if modifica_1 ~ "nessuna" then
               caffe_da_servire := create {ARABICA}
            elseif modifica_1 ~ "latte" then
               if modifica_2 ~ "nessuna" then
                  caffe_da_servire := create {ARABICA_CON_LATTE}
               else -- qui modifica_2 vale "cacao"
                  caffe_da_servire := create {ARABICA_CON_LATTE_E_CACAO}
               end
            else -- qui modifica_1 vale "cacao"
               if modifica_2 ~ "nessuna" then
                  caffe_da_servire := create {ARABICA_CON_CACAO}
               else -- qui modifica_2 vale "latte"
                  caffe_da_servire := create {ARABICA_CON_LATTE_E_CACAO}
               end
            end
         elseif specie_scelta ~ "gentile" then
            "lo stesso tipo di codice del caso dell'arabica"
         else -- qui specie_scelta vale "decaf"
            "lo stesso tipo di codice del caso dell'arabica"
         end
         somma_da_incassare := caffe_da_servire.costo
         -- "acquisizione dall'utente dei valori di specie_scelta, modifica_1, modifica_2"
      end
   end

end
```

Dire quante sarebbero in totale, in base a tale implementazione proposta, le classi risultanti (a parte *SERVI_CAFFE*) e motivare la risposta. Scrivere inoltre l'implementazione della feature *costo* (senza contratti) per ognuna delle quattro classi del frammento della struttura di classi precedentemente raffigurato:

Discutere eventuali problemi dell'implementazione proposta

SOLUZIONE:

Con l'implementazione proposta si otterrebbero 12 classi, cioè 4 classi per ognuna delle 3 possibili specie di caffè.

Le implementazioni della feature *costo* per le 4 classi del frammento proposto sono le seguenti:

```
class
   ARABICA
inherit
   CAFFE
feature {ANY}
   costo : REAL
   do
      Result := costo_arabica
   end
end

class
   ARABICA_CON_LATTE
inherit
   CAFFE
feature {ANY}
   costo : REAL
   do
      Result := costo_arabica + costo_latte
   end
end

class
   ARABICA_CON_CACAO
inherit
   CAFFE
feature {ANY}
   costo : REAL
   do
      Result := costo_arabica + costo_cacao
   end
end

class
   ARABICA_CON_LATTE_E_CACAO
inherit
   CAFFE
feature {ANY}
   costo : REAL
   do
      Result := costo_arabica + costo_latte + costo_cacao
   end
end
```

I problemi di questa implementazione sono essenzialmente legati alla difficoltà di manutenzione:

- il codice viene replicato inutilmente perché la gestione delle modifiche viene replicata per ognuna delle specie
- in caso di rimozione di una modifica vanno modificate tutte le classi in cui è presente la modifica
- in caso di aggiunta di una modifica vanno modificate le classi di tutte le specie.