Compito di *Fondamenti della Programmazione: Metodi Evoluti* - A.A. 2013-14
Prova di esame del 16 giugno 2014

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. …
   a. Una funzione è una query implementata mediante memorizzazione
   b. Un attributo è una query implementata mediante memorizzazione
   c. Una routine è l'implementazione di una feature mediante computazione
   d. Un attributo è l'implementazione di una feature mediante computazione

2. …
   a. Una struttura dati è polimorfica se può contenere riferimenti a oggetti di tipi diversi
   b. Un'assegnazione o un passaggio di argomenti sono polimorfici se la variabile di destinazione (*target variable*) e l'espressione sorgente (*source expression*) hanno tipi diversi
   c. Il polimorfismo è la capacità degli oggetti di cambiare il loro tipo a tempo di esecuzione (*run-time*)
   d. Un'entità è polimorfica se durante l'esecuzione può riferirsi ad oggetti di tipi diversi

3. La ridefinizione di una feature che è una procedura può …
   a. Cambiare l'implementazione
   b. Cambiare arbitrariamente l'implementazione, purché rispetti i contratti ereditati
   c. Cambiare la lista dei tipi degli argomenti
   d. Cambiare arbitrariamente la lista dei tipi degli argomenti

4. Se la classe *C* esporta l'attributo *x* verso una classe *T* allora …
   a. Un'istanza di *T* può leggere il valore di *x* direttamente da un'istanza di *C*
   b. Un'istanza di *T* può ottenere il valore di *x* da un'istanza di *C* solo mediante un'apposita feature
   c. Un'istanza di *T* può assegnare direttamente il valore di *x* in un'istanza di *C*
   d. Un'istanza di *T* non può assegnare in alcun modo il valore di *x* in un'istanza di *C*

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. …
   a. Una funzione è una query implementata mediante memorizzazione
   b. Un attributo è una query implementata mediante memorizzazione
   c. Una routine è l'implementazione di una feature mediante computazione
   d. Un attributo è l'implementazione di una feature mediante computazione

2. …
   a. Una struttura dati è polimorfica se può contenere riferimenti a oggetti di tipi diversi
   b. Un'assegnazione o un passaggio di argomenti sono polimorfici se la variabile di destinazione (*target variable*) e l'espressione sorgente (*source expression*) hanno tipi diversi
   c. Il polimorfismo è la capacità degli oggetti di cambiare il loro tipo a tempo di esecuzione (*run-time*)
   d. Un'entità è polimorfica se durante l'esecuzione può riferirsi ad oggetti di tipi diversi

3. La ridefinizione di una feature che è una procedura può …
   a. Cambiare l'implementazione
   b. Cambiare arbitrariamente l'implementazione, purché rispetti i contratti ereditati
   c. Cambiare la lista dei tipi degli argomenti
   d. Cambiare arbitrariamente la lista dei tipi degli argomenti

4. Se la classe *C* esporta l'attributo *x* verso una classe *T* allora …
   a. Un'istanza di *T* può leggere il valore di *x* direttamente da un'istanza di *C*
   b. Un'istanza di *T* può ottenere il valore di *x* da un'istanza di *C* solo mediante un'apposita feature
   c. Un'istanza di *T* può assegnare direttamente il valore di *x* in un'istanza di *C*
   d. Un'istanza di *T* non può assegnare in alcun modo il valore di *x* in un'istanza di *C*

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*

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

feature -- operazioni fondamentali
    has (a_value: INTEGER): BOOLEAN
            -- La lista contiene `a_value'?
        local
            temp, pre_temp: INT_LINKABLE
        do
            from
                temp := first_element;
                pre_temp := Void
            invariant
                not Result implies (pre_temp /= Void implies pre_temp.value /= a_value)
            until
                (temp = Void) or Result
            loop
                if temp.value = a_value then
                    Result := True
                end
                pre_temp := temp
                temp := temp.next
            end
        end

    get_item (a_value: INTEGER): INT_LINKABLE
            -- Ritorna l'elemento contenente `a_value', se esiste
        local
            temp, pre_temp: INT_LINKABLE
        do
            from
                temp := first_element;
                pre_temp := Void
            invariant
                Result = Void implies (pre_temp /= Void implies pre_temp.value /= a_value)
            until
                (temp = Void) or (Result /= Void)
            loop
                if temp.value = a_value then
                    Result := temp
                end
                pre_temp := temp
                temp := temp.next
            end
        ensure
            (Result /= Void) implies Result.value = a_value
        end

invariant
    count >= 0
    last_element /= Void implies last_element.next = Void
    count = 0 implies (first_element = last_element) and (first_element = Void)
    count = 1 implies (first_element = last_element) and (first_element /= Void)
    count > 1 implies (first_element /= last_element) and (first_element /= Void) and (last_element /= Void) and
    then (first_element.next /= Void)

end
```

Aggiungere alla classe *INT_LINKED_LIST* una feature *insert_after* (*new*, *target*: *INTEGER*) che inserisce un nuovo elemento di valore *new* dopo l'elemento di valore *target*, se quest'ultimo è presente. Altrimenti il nuovo elemento viene inserito alla fine della lista.

feature -- operazioni fondamentali
insert_after (*new, target*: *INTEGER*)
-- Inserisce elemento con valore `*new*' dopo elemento con valore `*target*', se quest'ultimo esiste
-- Altrimenti inserisce elemento con valore `*new*' alla fine della lista
local
_____
_____
do
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
ensure
_____
_____
_____
_____
end

SOLUZIONE:

Qua sotto una prima versione che usa le feature già esistenti *has*() e *get_item*():

```
feature -- operazioni fondamentali
    insert_after (new, target: INTEGER)
            -- Inserisce elemento con valore `new' dopo elemento con valore `target', se quest'ultimo esiste
            -- Altrimenti inserisce elemento con valore `new' alla fine della lista
        local
            new_item: INT_LINKABLE
        do
            create new_item.make (new)
            if  has (target)  then
                new_item.link_after (get_item (target))
            else
                if count = 0 then
                    first_element := new_item
                    last_element := new_item
                else
                    new_item.link_after (last_element)
                    last_element := item
                end
            end
            count := count + 1
        end
        ensure
            count = old count + 1
            not (old has(target)) implies last_element.value = new;
            old has(target) implies get_item(target).next.value = new;
end
```

Si noti che non si può scrivere una post-condizione che assicuri che il nuovo elemento inserito punta a quello che era successivo a *target* prima dell'inserimento, se *target* esiste, come ad esempio

```
    old has(target) implies get_item(new).next = old (get_item(target).next)
```

perché il valore degli elementi prefissi con **old** viene sempre e comunque pre-calcolato all'ingresso nella routine e nel caso di *target* che non esiste si avrebbe l'errore *feature call on void target*.

Si ricorda inoltre che non si possono scrivere post-condizioni relative alle variabili locali.

Un'implementazione alternativa più efficiente si ottiene senza usare le feature già esistenti:

```
feature -- operazioni fondamentali
    insert_after (new, target: INTEGER)
            -- Inserisce elemento con valore `new' dopo elemento con valore `target', se quest'ultimo esiste
            -- Altrimenti inserisce elemento con valore `new' alla fine della lista
        local
            new_item, temp: INT_LINKABLE
        do
            create new_item.make (new)
            from
                temp := first_element
            until
                temp = Void or else temp.value = target
            loop
                temp := temp.next
            end
```

```
            if temp /= Void then
                new_item.link_after (temp)
            else
                if count=0 then
                    first_element := new_item
                    last_element := new_item
                else
                    new_item.link_after (last_element)
                    last_element := new_item
                end
            end
            count := count + 1
        end
        ensure
            count = old count + 1
            not (old has(target)) implies last_element.value = new;
            old has(target) implies get_item(target).next.value = new;
end
```

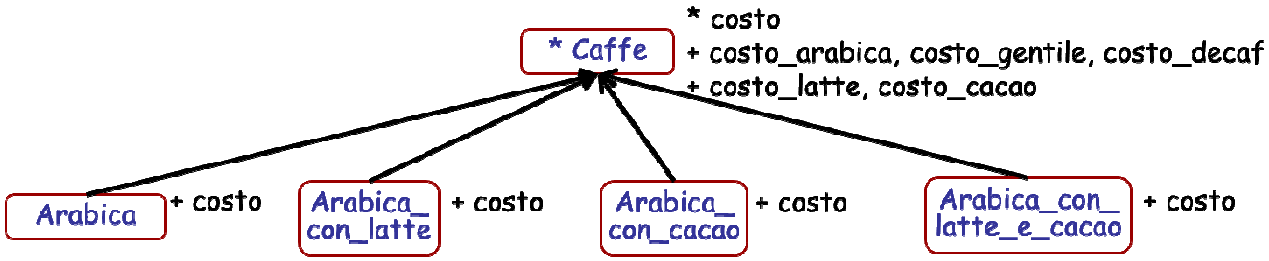
Esercizio 3) [10 punti]

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 aggiunte a scelta (*con_latte*, *con_cacao*). Nel caso di richiesta di due aggiunte 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 aggiunte 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 aggiunta. 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 un aggiunta di latte

    costo_cacao : REAL = 0.2
        -- il costo base di un aggiunta di cacao

end
```

Inoltre, la proposta suggerisce di avere una sottoclasse per tutte le possibili combinazioni di specie e aggiunte. 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 aggiunta 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, aggiunta_1, aggiunta_2 : STRING
            somma_da_incassare : REAL
    do
        from
            -- "acquisizione dall'utente dei valori di specie_scelta, aggiunta_1, aggiunta_2"
        until
            specie_scelta ~ "nessuna"
        loop
            if specie_scelta ~ "arabica" then
                if aggiunta_1 ~ "nessuna" then
                    caffe_da_servire := create {ARABICA}
                elseif aggiunta_1 ~ "latte" then
                    if aggiunta_2 ~ "nessuna" then
                        caffe_da_servire := create {ARABICA_CON_LATTE}
                    else -- qui aggiunta_2 vale "cacao"
                        caffe_da_servire := create {ARABICA_CON_LATTE_E_CACAO}
                    end
                else -- qui aggiunta_1 vale "cacao"
                    if aggiunta_2 ~ "nessuna" then
                        caffe_da_servire := create {ARABICA_CON_CACAO}
                    else -- qui aggiunta_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, aggiunta_1, aggiunta_2"
        end
    end

end
```

Dire quante sarebbero in totale, in base a tale implementazione proposta, le classi risultanti 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

OPZIONALE
Definire una nuova struttura di ereditarietà (modificando e/o eliminando a piacere le classi esistenti) che superi i problemi individuati.

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
```

Per ottenere un migliore riuso del codice ed una migliore manutenibilità va separata l'implementazione relativa alla specie di caffè da quella relativa alle aggiunte. Inoltre, le due implementazioni vanno rese componibili.

Questo si può ottenere evitando di dichiarare le classi relative alle specie con tutte le possibili aggiunte e mantenendo le sole classi relative alle specie base. L'implementazione della classi per la specie *ARABICA* è uguale alla precedente implementazione e quelle per le altre specie si derivano per analogia.

L'implementazione relativa alle aggiunte si ottiene definendo una classe per ognuno dei tipi di aggiunte. Per rendere tali classi componibili con le classi che modellano le specie di caffè, le classi relative alle aggiunte devono

essere sottoclassi di una stessa classe *deferred* che deve ereditare dalla superclasse comune a tutte le specie di caffè. Qua sotto è presente l'implementazione di tale classe *deferred*.

```
deferred class
    CAFFE_SERVITO
inherit
    CAFFE
feature {NONE}
    caffe_base : CAFFE
feature {ANY}
    set_base (base : CAFFE)
    do
        caffe_base := base
    end
end
```

Poi, per ognuna delle possibili aggiunte si definisce una specifica sottoclasse di *CAFFE_SERVITO* che implementa la feature *costo* che è rimasta *deferred* nella catena di ereditarietà a partire da *CAFFE*. Qua sotto è presente la classe nel caso di aggiunta *CAFFE_CON_LATTE* e per estensione si possono derivare le implementazioni per le altre aggiunte.

```
class
    CAFFE_CON_LATTE
inherit
    CAFFE_SERVITO
create
    set_base
feature {ANY}
    costo : REAL
    do
        Result := costo_latte + caffe_base.costo
    end
end
```

Per semplicità di gestione conviene anche aggiungere una classe che modella il caso di nessuna aggiunta, implementata come segue:

```
class
    CAFFE_CON_NIENTE
inherit
    CAFFE_SERVITO
create
    set_base
feature {ANY}
    costo : REAL
    do
        Result := caffe_base.costo
    end
end
```

Le classi così definite possono essere composte all'atto della creazione delle istanze, in modo che una nuova istanza prenda come base una precedente istanza ed aggiunga ad essa il suo specifico comportamento. Così facendo, l'invocazione della feature *costo* causerà l'esecuzione in successione delle relative versioni della stessa feature di ognuna delle istanze che sono state composte.

La root class e la root procedure del sistema diventano quindi le seguenti:

```
class
    SERVI_CAFFE

feature {NONE}
    caffe_da_servire : CAFFE
    specie_scelta, aggiunta_1, aggiunta_2 : STRING

feature {NONE}
    aggiungi (aggiunta : STRING; caffe_base: CAFFE) : CAFFE
        do
            if aggiunta ~ "nessuna" then
                Result := create {CAFFE_CON_NIENTE}. set_base (caffe_base)
            elseif aggiunta ~ "latte" then
                Result := create {CAFFE_CON_LATTE}. set_base (caffe_base)
            else -- qui aggiunta vale "cacao"
                Result := create {CAFFE_CON_CACAO}. set_base (caffe_base)
            end
        end

feature {ANY}
    servi
        local
            somma_da_incassare : REAL
    do
        from
            -- "acquisizione dall'utente dei valori di specie_scelta, aggiunta_1, aggiunta_2"
        until
            specie_scelta ~ "nessuna"
        loop
            if specie_scelta ~ "arabica" then
                caffe_da_servire := create {ARABICA}
            elseif specie_scelta ~ "gentile" then
                caffe_da_servire := create {GENTILE}
            else -- qui specie_scelta vale "decaf"
                caffe_da_servire := create {DECAF}
            end
            if aggiunta_1 /~ "nessuna" then
                caffe_da_servire := aggiungi (aggiunta_1, caffe_da_servire)
                caffe_da_servire := aggiungi (aggiunta_2, caffe_da_servire)
            end
        somma_da_incassare := caffe_da_servire. costo
        -- "acquisizione dall'utente dei valori di specie_scelta, aggiunta_1, aggiunta_2"
        end
    end
end
```

In questa implementazione il codice è riusato meglio dal momento che le aggiunte vengono gestite in un'unica procedura (la nuova feature *aggiungi*).

Inoltre la manutenzione è facilitata: nel caso di variazioni sulle specie di caffe si deve intervenire solo sulla classe variata o definendo una nuova classe (se viene introdotta una nuova specie di caffè). Anche nel caso di variazioni sulle aggiunte si deve intervenire solo sulla procedura *aggiungi* e/o definire una nuova classe (se viene introdotta una nuova aggiunta)