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 classe esiste solo a tempo di esecuzione (*run-time*)
   b. Una funzione non dovrebbe modificare alcun oggetto
   c. Una query è sempre una funzione
   d. Una procedura non può ritornare valori come risultato della sua esecuzione

2. …
   a. Una classe *deferred* non può ereditare da una classe *effective*
   b. Una classe *C* non può ereditare da due classi differenti *B1* e *B2*, se sia *B1* che *B2* hanno una stessa classe A come antenato comune
   c. Ad un entità di tipo statico *C* si può assegnare un oggetto di un tipo che è un antenato di *C*
   d. In una classe *C* una feature *f* ereditata dalla classe *A* può essere ridefinita soltanto se *f* è *deferred* in *A*

3. …
   a. Se un loop non definisce un invariante il programma non compila correttamente
   b. Se la clausola *from* del loop è vuota il programma compila correttamente
   c. Il variante del loop deve essere uguale a 0 dopo l'ultima iterazione
   d. Se la clausola *until* del loop è falsa si esce dal loop

4. …
   a. Il variante del loop può anche non decrementare ad ogni iterazione del loop
   b. Se la clausola *until* del loop è vuota il programma compila correttamente
   c. L'invariante del loop deve essere vero anche immediatamente prima della prima iterazione
   d. Il variante del loop ha sempre un valore intero positivo

5. …
   a. Il parametro di una classe generica può essere istanziato sia da una classe non-generica che da una derivazione di una classe generica
   b. Se *C* è una classe *deferred* allora non possono esistere nel programma entità di tipo statico *C*
   c. La genericità viene usata per parametrizzare una classe e l'ereditarietà viene usata per specializzare una classe
   d. Un'istruzione *b.f* può dar luogo a tempo di esecuzione (*run-time*) all'esecuzione di differenti feature

SOLUZIONE:

1. …
   a. Una classe esiste solo a tempo di esecuzione (*run-time*)
   b. Una funzione non dovrebbe modificare alcun oggetto
   c. Una query è sempre una funzione
   d. Una procedura non può ritornare valori come risultato della sua esecuzione

2. …
   a. Una classe *deferred* non può ereditare da una classe *effective*
   b. Una classe *C* non può ereditare da due classi differenti *B1* e *B2*, se sia *B1* che *B2* hanno una stessa classe A come antenato comune
   c. Ad un entità di tipo statico *C* si può assegnare un oggetto di un tipo che è un antenato di *C*
   d. In una classe *C* una feature *f* ereditata dalla classe *A* può essere ridefinita soltanto se *f* è *deferred* in *A*

3. …
   a. Se un loop non definisce un invariante il programma non compila correttamente
   b. Se la clausola *from* del loop è vuota il programma compila correttamente
   c. Il variante del loop deve essere uguale a 0 dopo l'ultima iterazione
   d. Se la clausola *until* del loop è falsa si esce dal loop

4. …
   a. La variante del loop può anche non decrementare ad ogni iterazione del loop
   b. Se la clausola *until* del loop è vuota il programma compila correttamente
   c. L'invariante del loop deve essere vero anche immediatamente prima della prima iterazione
   d. Il variante del loop ha sempre essere un valore intero positivo

5. …
   a. Il parametro di una classe generica può essere istanziato sia da una classe non-generica che da una derivazione di una classe generica
   b. Se *C* è una classe *deferred* allora non possono esistere nel programma entità di tipo statico *C*
   c. La genericità viene usata per parametrizzare una classe e l'ereditarietà viene usata per specializzare una classe
   d. Un'istruzione *b.f* può dar luogo a tempo di esecuzione (*run-time*) all'esecuzione di differenti feature

Esercizio 2) [10 punti]
La classe *LIST* [*G*] e *ELEM* [*G*] realizzano una lista semplice. Il primo elemento della lista viene memorizzato nell'attributo *first* della classe *LIST* [*G*]. L'attributo *next* della classe *ELEM* [*G*] dà accesso all'elemento successivo della lista. L'invocazione di *next* sull'ultimo elemento di una lista ritorna **Void**.

Implementare la feature *invert* della classe *LIST* [*G*] che realizza l'inversione dell'ordine con cui gli elementi sono presenti nella lista. Ad esempio, invertire la lista [1, 3, 2, 9] risulta nella lista [9, 2, 3, 1].

Non devono essere creati nuovi oggetti della classe *ELEM* [*G*] né modificate le feature esistenti o introdotte nuove feature nelle classi *LIST* [*G*] e *ELEM* [*G*]. Non è necessario conoscere altro codice all'infuori di quello fornito.

```
class
    LIST [G]

feature -- accesso
    first : ELEM [G]
        -- Il primo elemento di questa lista, oppure Void se questa lista è vuota

feature -- operazioni fondamentali
    invert
        -- Inverti l'ordine degli elementi in questa lista
        -- Ad esempio, invertire la lista [1, 3, 2, 9] risulta nella lista [9, 2, 3, 1]
        local
            ____________________________________________________________________________
            ____________________________________________________________________________
        do
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
        end
        ensure
            ____________________________________________________________________________
            ____________________________________________________________________________
end


class
    ELEM [G]

feature -- accesso
    next : ELEM [G]
        -- Dà accesso all'elemento successivo a questo. Vale Void se non esiste l'elemento successivo.

feature -- modifica
    set_next (an_element : ELEM [G])
        -- Assegna an_element come elemento successivo di questo
        ensure
            next = an_element
end
```

SOLUZIONE:

```
class
    LIST [G]

feature -- operazioni fondamentali
    invert
        -- Inverti l'ordine degli elementi in questa lista
        -- Ad esempio, invertire la lista [1, 3, 2, 9] risulta nella lista [9, 2, 3, 1]
        local
            old : ELEM [G] – punta alla parte di lista ancora da invertire
            old_first : ELEM [G] -- il primo elemento della parte di lista ancora da invertire
            rev : ELEM [G] – punta alla parte di lista già invertita
        do
            from
                old := first
            until
                old := Void
            loop
                old_first := old
                old := old.next
                old_first.set_next := rev
                rev := old_first
            end
        end
        ensure
            count = old count
end


class
    ELEM [G]

feature -- accesso
    next : ELEM [G]
        -- Dà accesso all'elemento successivo a questo. Vale Void se non esiste l'elemento successivo.

feature -- modifica
    set_next (an_element : ELEM [G])
        -- Assegna an_element come elemento successivo di questo
        ensure
            next = an_element
end
```

Esercizio 3) [10 punti]
Completare i contratti (pre-condizioni, post-condizioni, invarianti di classe) della porzione di classe *PERSONA* sotto presentata che modella un tipo di contratto di matrimonio:

1. Ogni persona ha un nome non vuoto
2. Una persona non può essere sposata a sé stessa
3. Se X è una persona sposata alla persona Y, allora Y è sposata a X
4. Affinché la persona X possa sposare la persona Y, né X né Y devono essere già sposati
5. Non è permesso annullare il contratto di matrimonio

Non c'è relazione tra il numero di righe vuote ed il numero di contratti da scrivere. Non è permesso cambiare le interfacce né il codice già fornito. Non è necessario conoscere altro codice all'infuori di quello fornito.

```
class
    PERSONA

feature   -- creazione
    make (a_name : STRING)
        -- Crea una persona con il nome "a_name".
        require
            _____________________________________________________________________
            _____________________________________________________________________
        do
            name ≔ a_name
        end
        ensure
            _____________________________________________________________________
            _____________________________________________________________________
        end

feature -- accesso
    name : STRING
        -- Il nome di questa persona
    spouse : PERSON
        -- Il coniuge di questa persona, se esiste. Altrimenti è Void

feature   -- stato
    is_married : BOOLEAN
        -- Questa persona è sposata?
        do
            Result ≔ (spouse /= Void)
        ensure
            _____________________________________________________________________
            _____________________________________________________________________
        end

feature -- implementazione
    accept_marriage_with (a_person : PERSON)
        -- Assegna a questa persona come coniuge a_person, che è già sposata con questa persona
        require
            _____________________________________________________________________
            _____________________________________________________________________
        do
            spouse ≔ a_person
        ensure
            _____________________________________________________________________
            _____________________________________________________________________
        end
```

feature -- operazioni fondamentali
marry_with (*a_person* : *PERSON*)
-- Sposa questa persona con *a_person*
require
_____
_____
do
spouse ≔ *a_person*
a_person.accept_marriage_with (**Current**)
ensure
_____
_____
end

invariant
_____
_____
end

SOLUZIONE:

class
PERSONA

feature -- creazione
make (*a_name* : *STRING*)
-- Crea una persona con il nome "*a_name*".
require
a_name /= **Void and then not** *a_name.is_empty*
do
name ≔ *a_name*
end
ensure
name = *a_name*
not *is_married*
end

feature -- accesso
name : *STRING*
-- Il nome di questa persona
spouse : *PERSON*
-- Il coniuge di questa persona, se esiste. Altrimenti è **Void**

feature -- stato
is_married : *BOOLEAN*
-- Questa persona è sposata?
do
Result ≔ (*spouse* /= **Void**)
ensure
Result = (*spouse* /= **Void**)
end

```
feature -- implementazione
    accept_marriage_with (a_person : PERSON)
        -- Assegna a questa persona come coniuge a_person, che è già sposata con questa persona
        require
            a_person /= Void  and then  a_person /= Current
            not is_married
            a_person.spouse = Current
        do
            spouse := a_person
        ensure
            spouse = a_person
            is_married
        end

feature -- operazioni fondamentali
    marry_with (a_person : PERSON)
        -- Sposa questa persona con a_person
        require
            a_person /= Void  and then  a_person /= Current
            not is_married
            not a_person.is_married
        do
            spouse := a_person
            a_person.accept_marriage_with (Current)
        ensure
            spouse = a_person
            is_married
        end

invariant
    a_name /= Void  and then  not a_name.is_empty
    spouse /= Current
    is_married = (spouse /= Void)
    is_married  implies  spouse.spouse = Current
end
```

N.B.
Il contratto evidenziato in giallo non è necessario, dal momento che lo stesso contratto è presente tra gli invarianti di classe. Non è stato considerato un errore l'averlo inserito nella soluzione fornita al compito.