Prova di esame del 19 giugno 2017

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. Il risultato ritornato da una query è fornito sempre da una funzione
   b. Il risultato ritornato da una query è fornito sempre da un attributo
   c. Il risultato ritornato da una query è fornito da una funzione o da un attributo
   d. Il risultato ritornato da un comando è fornito da una procedura o da un attributo

2. La ridefinizione di una *feature* che è una funzione può …
   a. Cambiare arbitrariamente l'implementazione purché vengano rispettati i contratti ereditati
   b. Cambiare arbitrariamente la lista dei tipi degli argomenti
   c. Cambiare arbitrariamente il contratto
   d. Cambiare arbitrariamente il tipo del risultato… può essere usata come procedura di creazione

3. Una query …
   a. … può essere usata come procedura di creazione
   b. … può essere implementata come una *routine*
   c. … può apparire nelle precondizioni e postcondizioni di una qualunque *routine*
   d. … può apparire tra gli invarianti di classe

4. …
   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`**

5. …
   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

SOLUZIONE:

1. …
   a. Il risultato ritornato da una query è fornito sempre da una funzione
   b. Il risultato ritornato da una query è fornito sempre da un attributo
   c. Il risultato ritornato da una query è fornito da una funzione o da un attributo
   d. Il risultato ritornato da un comando è fornito da una procedura o da un attributo

2. La ridefinizione di una *feature* che è una funzione può …
   a. Cambiare arbitrariamente l'implementazione purché vengano rispettati i contratti ereditati
   b. Cambiare arbitrariamente la lista dei tipi degli argomenti
   c. Cambiare arbitrariamente il contratto
   d. Cambiare arbitrariamente il tipo del risultato

3. Una query …
   a. … può essere usata come procedura di creazione
   b. … può essere implementata come una *routine*
   c. … può apparire nelle pre-condizioni e post-condizioni di una qualunque *routine*
   d. … può apparire tra gli invarianti di classe

4. …
   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`**

5. …
   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

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

   insert_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_element (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_multiple_before* (*new*, *target*: *INTEGER*) che inserisce un nuovo elemento di valore *new* prima di ogni occorrenza dell'elemento di valore *target*, se quest'ultimo è presente. Altrimenti il nuovo elemento viene inserito all'inizio della lista.

```
feature -- operazioni fondamentali
  insert_multiple_before (new, target: INTEGER)
       -- Inserisce elemento con valore `new' prima di ogni elemento con valore `target',
se ne esistono
       -- Altrimenti inserisce elemento con valore `new' all'inizio della lista
     local
        ____________________________________________________________________________
        ____________________________________________________________________________
     do
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
     ensure
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
        ____________________________________________________________________________
end
```

SOLUZIONE:

```
feature -- operazioni fondamentali
   insert_multiple_before (new, target: INTEGER)
         -- Inserisce elemento con valore `new' prima di ogni elemento con valore `target',
se ne esistono
         -- Altrimenti inserisce elemento con valore `new' all'inizio della lista
      local
         new_element, previous_element, current_element: INT_LINKABLE
      do
         if has(target) then
            from
               previous_element := Void
               current_element := first_element
            until
               current_element = Void
            loop
               if current_element.value = target then
                  create new_element.make (new)
                  if current_element = first_element then
                     new_element.link_to (first_element)
                     first_element := new_element
                  else
                     new_element.link_to(current_element)
                     previous_element.link_to(new_element)
                  end
                  count := count + 1
               end
               previous_element := current_element
               current_element := current_element.next
            end
         else -- la lista non contiene `target'
            create new_element.make (new)
            if count=0 then
               first_element := new_element
               last_element := new_element
            else
               new_element.link_to(first_element)
               first_element := new_element
            end
            count := count + 1
         end
      end
      ensure
         count > old count
         not (old has(target)) implies count = old count + 1
         not (old has(target)) implies first_element.value = new;
         not (old has(target)) implies first_element.next = old first_element;
         old has(target) implies get_element(new).next.value = target;
end
```

Si noti che l'ultima post-condizione garantisce soltanto che sia stato fatto l'inserimento di *new* prima della prima occorrenza di *target* nella lista.

Esercizio 3) [10 punti]
Il software usato da un editore multimediale modella libri, riviste, DVD e libri digitali (*e-book*). Il software ha una specifica classe per ognuno di questi tipi di oggetti. I libri, le riviste e i libri digitali possono essere stampati e quindi il software fornisce un comando **stampa** per ognuna delle relative classi. Un DVD non può essere stampato ma può essere eseguito e quindi il software fornisce un comando **esegui** per la relativa classe. Anche un libro digitale può essere eseguito e quindi anche per la relativa classe il software fornisce un comando **esegui**. Ogni istanza di un tipo di oggetto ha un suo nome e quindi il software fornisce un attributo **nome** di tipo **stringa**.

Nell'attuale implementazione ognuna di queste classi è definita come sottoclasse di *ANY* e quindi non viene ottenuto alcun vantaggio dall'utilizzo di un approccio object-oriented.

Ridefinire la struttura di ereditarietà delle classi (anche introducendone di nuove, se necessario) disegnando il nuovo diagramma di ereditarietà che mostra:

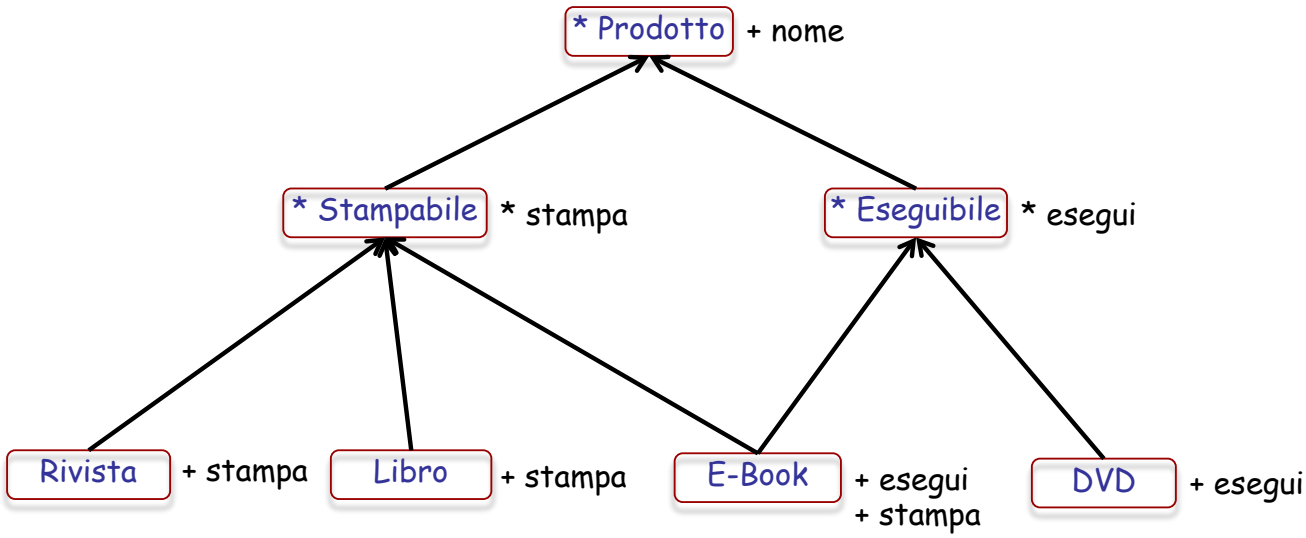
1. la struttura di ereditarietà delle classi (cioè le relazioni superclasse – sottoclasse)
2. quali classi e *feature* sono *deferred* (*), *effective* (+) o *redefined* (++).

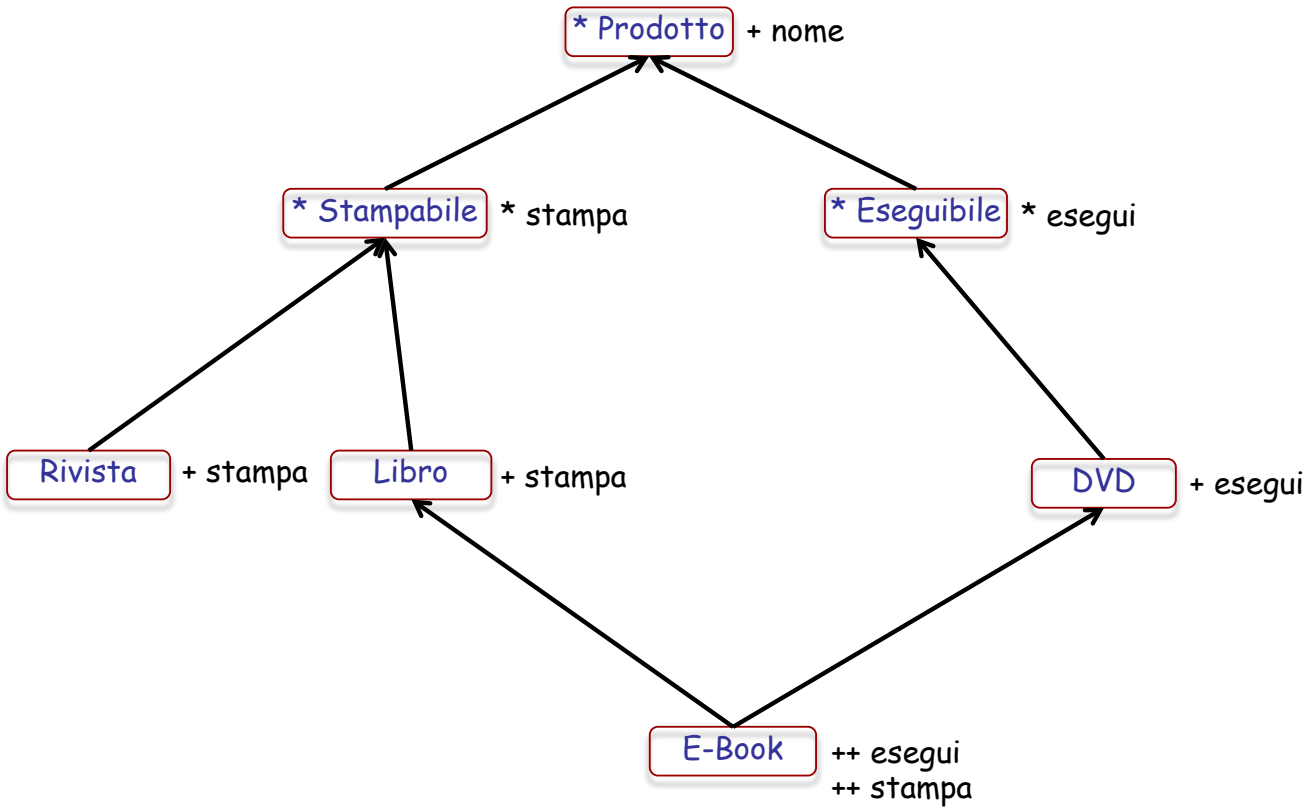
Non è necessario scrivere alcun codice.

SOLUZIONE:

La soluzione introduce una classe *deferred* per rappresentare ciò che è comune a tutti i prodotti e due sue sottoclassi (anch'esse *deferred*) per rappresentare ciò che è comune a tutti i prodotti stampabili e a tutti i prodotti eseguibili.

Si possono poi avere diverse soluzioni, a seconda di ciò che si assume sulla condivisione di caratteristiche tra libro digitale e libro, da un lato, e tra libro digitale e dvd, dall'altro. Ad un estremo non vi sono caratteristiche condivisibili, e si ottiene il primo diagramma qua sotto, all'altro vi sono caratteristiche condivisibili sia con il libro che con il dvd, e si ottiene il diagramma successivo.




* Prodotto
+ nome
* Stampabile
* stampa
* Eseguibile
* esegui
Rivista
+ stampa
Libro
+ stampa
DVD
+ esegui
E-Book
++ esegui
++ stampa