Compito di *Fondamenti della Programmazione: Metodi Evoluti* - A.A. 2012-13
Prova di esame del 16 luglio 2013

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. Un comando …
   a. … è una query che non è implementata mediante un attributo
   b. … può apparire nell'invariante di classe
   c. … non dovrebbe modificare alcun oggetto
   d. … può ritornare valori come risultato della sua esecuzione

2. 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 nell'invariante di classe

3. …
   a. Un loop deve sempre definire un invariante altrimenti il programma non compila correttamente
   b. La variante del loop deve incrementare ad ogni iterazione del loop
   c. La clausola *from* del loop non può essere vuota altrimenti il programma non compila correttamente
   d. L'invariante del loop deve essere vero anche immediatamente dopo l'ultima iterazione

4. …
   a. Se la clausola *until* del loop è falsa si esce dal loop
   b. Se la clausola *until* del loop è vuota il programma non compila correttamente
   c. L'invariante del loop deve essere vero anche immediatamente prima della prima iterazione
   d. La variante del loop deve sempre essere un intero positivo

5. …
   a. Se *C* è una classe *deferred* allora non possono esistere nel programma entità di tipo statico *C*
   b. Se *C* è una classe che usa la *feature* di un'altra classe S allora C è *supplier* (fornitore) di S
   c. La classe *C* può essere contemporaneamente un *supplier* (fornitore) ed un *client* (cliente)
   d. Se il tipo di una variabile è un tipo espanso allora il valore della variabile a tempo di esecuzione (*run-time*) è un oggetto

SOLUZIONE:

1. Un comando …
   a. … è una query che non è implementata mediante un attributo
   b. … può apparire nell'invariante di classe
   c. … non dovrebbe modificare alcun oggetto
   d. … può ritornare valori come risultato della sua esecuzione

2. 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 nell'invariante di classe

3. …
   a. Un loop deve sempre definire un invariante altrimenti il programma non compila correttamente
   b. La variante del loop deve incrementare ad ogni iterazione del loop
   c. La clausola *from* del loop non può essere vuota altrimenti il programma non compila correttamente
   d. L'invariante del loop deve essere vero anche immediatamente dopo l'ultima iterazione

4. …
   a. Se la clausola *until* del loop è falsa si esce dal loop
   b. Se la clausola *until* del loop è vuota il programma non compila correttamente
   c. L'invariante del loop deve essere vero anche immediatamente prima della prima iterazione
   d. La variante del loop deve sempre essere un intero positivo

5. …
   a. Se *C* è una classe *deferred* allora non possono esistere nel programma entità di tipo statico *C*
   b. Se *C* è una classe che usa la *feature* di un'altra classe S allora C è *supplier* (fornitore) di S
   c. La classe *C* può essere contemporaneamente un *supplier* (fornitore) ed un *client* (cliente)
   d. Se il tipo di una variabile è un tipo espanso allora il valore della variabile a tempo di esecuzione (*run-time*) è un oggetto

Esercizio 2) [7 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:

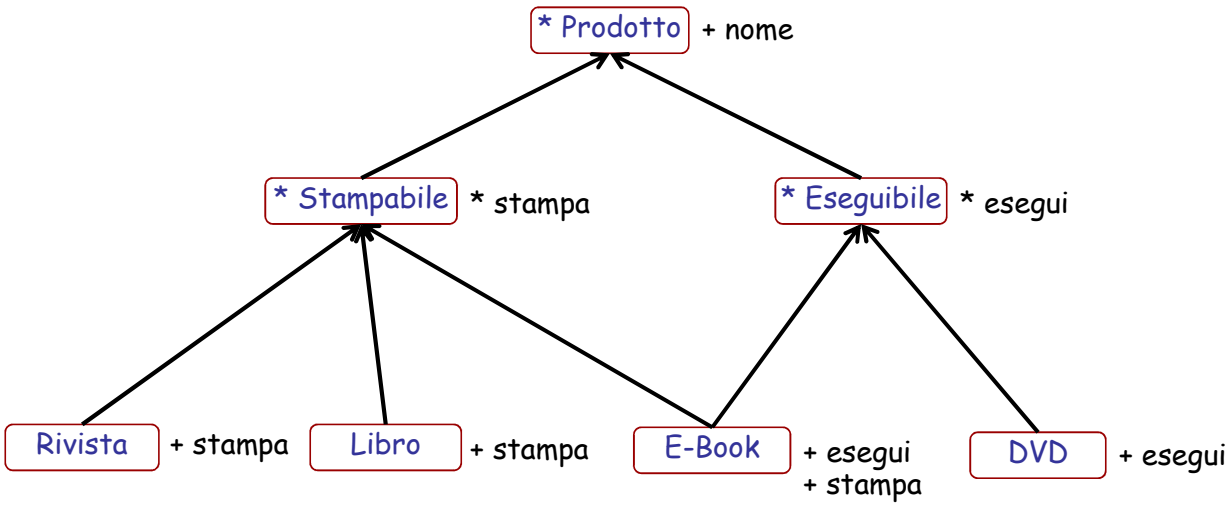
1. la struttura di ereditarietà delle classi (cioè le relazioni superclasse – sottoclasse)
2. quali classi e *feature* sono *deferred* (*), *effective* (+) o *redefined* (++).

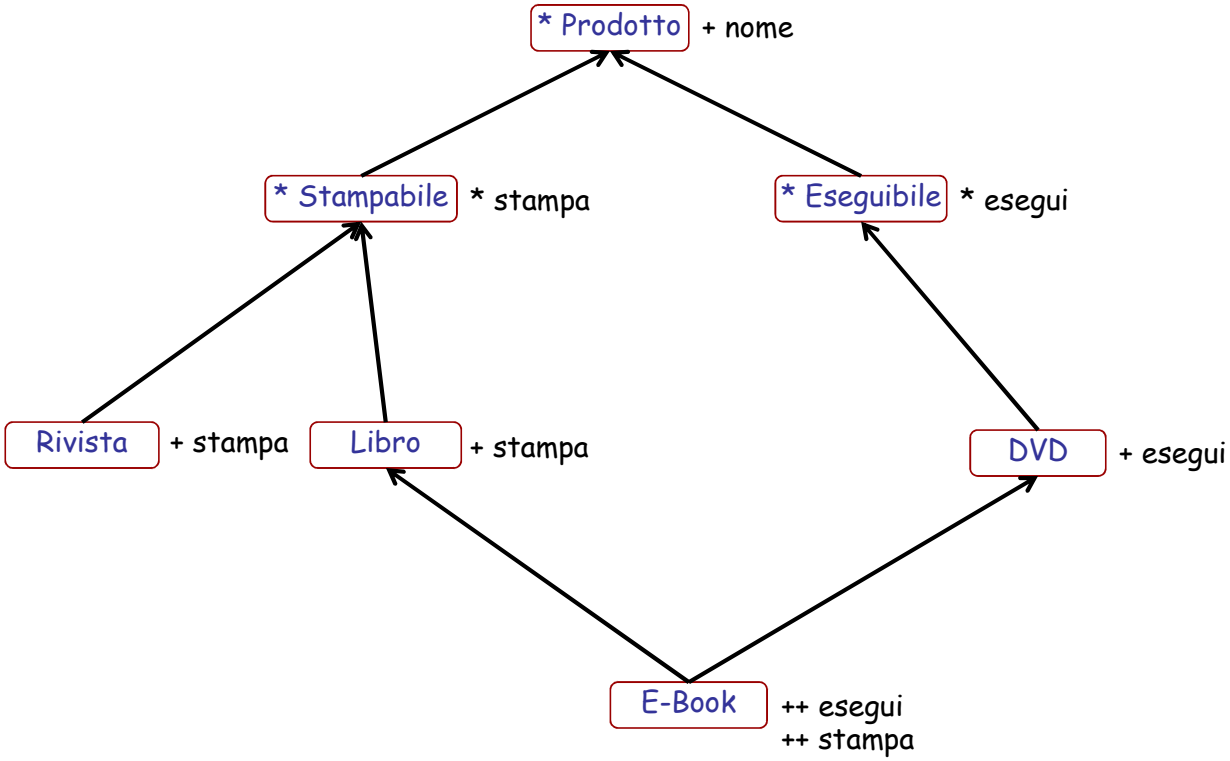
Non è necessario scrivere alcun codice.

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.

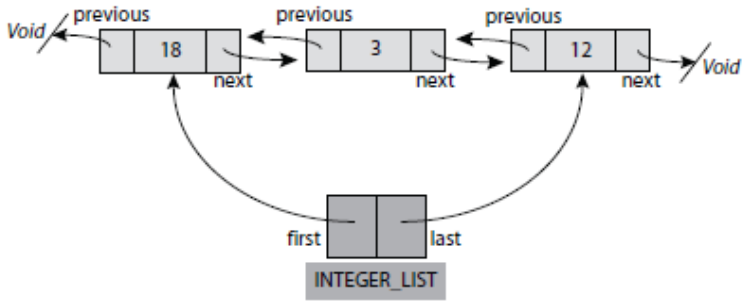
Si possono poi avere diverse varianti, a seconda di ciò che si assume sulla condivisione di caratteristiche tra il libro digitale e il libro, da un lato, e tra il libro digitale e il dvd, dall'altro. Ad un estremo non c'è nessuna condivisione, e si ottiene il primo diagramma qua sotto, all'altro c'è la condivisione da entrambe le classi, e si ottiene il secondo diagramma ancora più sotto.




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

Esercizio 3) [13 punti]

Nel programma 1 qua sotto viene fornita l'implementazione della classe INTEGER_CELL, che rappresenta un elemento con valore intero che fa parte di una lista doppia (cioè una lista che può essere scandita in ambedue i sensi). Nel programma 2 qua sotto viene fornita l'implementazione parziale della classe INTEGER_LIST, che rappresenta una lista doppia di interi. Un esempio di lista doppia con 3 elementi (il primo con valore 18, il secondo con valore 3, l'ultimo con valore 12) è disegnato qua sotto:



Leggere bene il programma 1 e completare l'implementazione del programma 2. Più specificamente:

1. Implementare la feature *extend* che riceve un INTEGER come argomento, crea un oggetto di tipo INTEGER_CELL che ha come valore l'INTEGER ricevuto e aggiunge l'oggetto creato alla fine della lista esistente. Assicurarsi che l'implementazione soddisfi le postcondizioni già scritte.
2. Implementare la feature *has* che riceve un INTEGER come argomento e verifica se nella lista esistente è già presente un elemento che ha lo stesso INTEGER come valore. Non è necessario scrivere postcondizioni.

Non c'è nessuna relazione tra il numero di righe vuote ed il numero di istruzioni da scrivere.

Programma 1: La classe INTEGER_CELL

```
class
    INTEGER_CELL

create
    set_value

feature  -- accesso
    value : INTEGER
        -- Il valore memorizzato nell'elemento.
    next : INTEGER_CELL
        -- Riferimento al successivo elemento della lista. E' Void per l'ultimo elemento di una lista.
    previous : INTEGER_CELL
        -- Riferimento al precedente elemento della lista. E' Void per il primo elemento di una lista.

feature  -- modifica
    set_value (a_value : INTEGER)
        -- Assegna all'elemento il valore 'a_value'.
        do
            value := a_value
        ensure
            value = a_value
        end
```

```
    set_next (a_cell : INTEGER_CELL)
        -- Assegna 'a_cell' come valore di 'next'.
        do
            next := a_cell
        ensure
            next = a_cell
        end


    set_previous (a_cell : INTEGER_CELL)
        -- Assegna 'a_cell' come valore di 'previous'.
        do
            previous := a_cell
        ensure
            previous = a_cell
        end
```

Programma 2: La classe INTEGER_LIST

```
class
    INTEGER_LIST

create
    make_empty

feature   -- accesso
    first : INTEGER_CELL
        -- Il primo elemento della lista. E' Void se la lista è vuota.
    last : INTEGER_CELL
        -- L'ultimo elemento della lista. E' Void se la lista è vuota.

feature   -- misura
    count : INTEGER
        -- Il numero di celle nella lista.

feature   -- inizializzazione
    make_empty
        -- Inizializza la lista come lista vuota.
        do
            first := Void
            last := Void
            count := 0
        end
```

```
feature   -- modifica
    extend (a_value : INTEGER)
        -- Aggiunge alla fine della lista un oggetto di tipo INTEGER_CELL contenente il valore 'a_value'.
        local
            el : INTEGER_CELL
        do
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
        ensure
            count = old count + 1
            count = 1 implies first.value = a_value
            last.value = a_value
        end

feature   -- stato
    empty : BOOLEAN
        -- La lista è vuota?
        do
            Result := (count = 0)
        end

    has (a_value : INTEGER) : BOOLEAN
        -- La lista contiene un elemento contenente il valore 'a_value'?
        local
            ____________________________________________________________________________
            ____________________________________________________________________________
        do
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
            ____________________________________________________________________________
        end
```

SOLUZIONE:

```
feature   -- modifica
    extend (a_value : INTEGER)
        -- Aggiunge alla fine della lista un oggetto di tipo INTEGER_CELL contenente il valore ‘a_value’.
        local
            el : INTEGER_CELL
        do
            create el.set_value (a_value)
            if count = 0 then
                first := el
            else
                el.set_previous (last )
                last.set_next (el )
            end
            last := el
            count := count + 1
        ensure
            count = old count + 1
            count = 1 implies first.value = a_value
            last.value = a_value
        end

feature   -- stato
    empty : BOOLEAN
        -- La lista è vuota?
        do
            Result := (count = 0)
        end

    has (a_value : INTEGER) : BOOLEAN
        -- La lista contiene un elemento contenente il valore ‘a_value’?
        local
            current : INTEGER_CELL
        do
            from
                current := first
            until
                current = Void or else current.value = a_value
            loop
                current := current.next
            end
            Result := not (current = Void )

        end
```