

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. Una classe...
 - a. ... è la descrizione di un insieme di possibili oggetti esistenti a tempo di esecuzione (*run-time*) a cui sono applicabili le stesse *features*
 - b. ... può esistere solo a tempo di esecuzione (*run-time*)
 - c. ... può essere dichiarata come espansa (*expanded*)
 - d. ... deve avere un'unica procedura di creazione
2. ...
 - a. Una query è sempre una funzione
 - b. Una funzione non deve modificare alcun oggetto
 - c. Un attributo è memorizzato direttamente in memoria
 - d. Una procedura può ritornare valori come risultato della sua esecuzione
3. La ridefinizione di una *feature* che è una funzione può...
 - a. ... cambiare l'implementazione
 - b. ... cambiare la lista dei tipi degli argomenti
 - c. ... cambiare il contratto
 - d. ... cambiare il tipo del risultato
4. ...
 - a. Derivazioni diverse di una stessa classe generica sono sempre conformi l'una all'altra
 - b. Una classe generica è una classe che ha uno o più parametri generici
 - c. Il parametro di una classe generica può essere istanziato sia da una classe non-generica che da una derivazione di una classe generica
 - d. La genericità viene usata per parametrizzare una classe e l'ereditarietà viene usata per specializzare una classe
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. Una classe...
 - a. ... è la descrizione di un insieme di possibili oggetti esistenti a tempo di esecuzione (*run-time*) a cui sono applicabili le stesse *features*
 - b. ... può esistere solo a tempo di esecuzione (*run-time*)
 - c. ... può essere dichiarata come espansa (*expanded*)
 - d. ... deve avere un'unica procedura di creazione
2. ...
 - a. Una query è sempre una funzione
 - b. Una funzione non deve modificare alcun oggetto
 - c. Un attributo è memorizzato direttamente in memoria
 - d. Una procedura può ritornare valori come risultato della sua esecuzione

3. La ridefinizione di una *feature* che è una funzione può...
 - a. ... cambiare l'implementazione
 - b. ... cambiare la lista dei tipi degli argomenti
 - c. ... cambiare il contratto
 - d. ... cambiare il tipo del risultato
4. ...
 - a. Derivazioni diverse di una stessa classe generica sono sempre conformi l'una all'altra
 - b. Una classe generica è una classe che ha uno o più parametri generici
 - c. Il parametro di una classe generica può essere istanziato sia da una classe non-generica che da una derivazione di una classe generica
 - d. La genericità viene usata per parametrizzare una classe e l'ereditarietà viene usata per specializzare una classe
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]

Completare i contratti (pre-condizioni, post-condizioni, invarianti di classe, invarianti e varianti di *loop*) della classe *CAR* sotto descritta che modella automobili. **Non è ammesso** né cambiare l'interfaccia della classe né le implementazioni fornite. Non c'è relazione tra il numero di righe vuote ed il numero di contratti da scrivere. Non è necessario conoscere altro codice all'infuori di quello fornito.

class

CAR

create

make

feature {NONE} -- creazione

make

-- crea un'automobile di default

require

.....

do

create *doors.make* – la feature *make* crea una *LINKED_LIST* con zero elementi

ensure

.....

end

feature {ANY} -- accesso

is_convertible : *BOOLEAN*

-- L'automobile è una cabriolet? Il valore di creazione di default è **False**.

doors : *LINKED_LIST* [*CAR_DOOR*]

-- Le porte dell'automobile. Il numero delle porte deve essere 0, 2, o 4. Il valore di creazione di default è 0.

colors : *COLOR*

-- Il colore dell'automobile. **Void** se non specificato. Il valore di creazione di default è **Void**.

feature {ANY} -- modifica dell'oggetto

set_convertible (*status* : *BOOLEAN*)

require

.....

do

is_convertible := *status*

ensure

.....

end

set_doors (*new_doors* : *ARRAY* [*CAR_DOOR*])

require

.....

local

door_index : *INTEGER*

do

doors.wipe_out -- la feature *wipe_out* cancella tutti gli elementi della *LINKED_LIST*

if *new_doors* != **Void** **then**

from

door_index := 1

invariant

.....


```

    until
        door_index > new_doors.count
    loop
        doors.extend (new_doors[door_index])
        door_index := door_index + 1
    variant
        .....
        .....
    end
end
ensure
    .....
    .....
end

set_color (new_color : COLOR)
require
    .....
    .....
do
    color := new_color
ensure
    .....
    .....
end

invariant
    .....
    .....

```

SOLUZIONE:

```

class
    CAR

create
    make

feature {NONE} -- creazione
    make
        -- crea un'automobile di default
        require
        do
            create doors.make -- la feature make crea una LINKED_LIST con zero elementi
        ensure
            not is_convertible -- assicura la correttezza del valore di creazione dell'attributo
            doors /= Void and then doors.count = 0 -- assicura la correttezza del valore di creazione dell'attributo
            color = Void -- assicura la correttezza del valore di creazione dell'attributo
        end

feature {ANY} -- accesso
    is_convertible : BOOLEAN
        -- L'automobile è una cabriolet? Il valore di creazione di default è False.
    doors : LINKED_LIST [CAR_DOOR]
        -- Le porte dell'automobile. Il numero delle porte deve essere 0, 2, o 4. Il valore di creazione di default è 0.
    colors : COLOR
        -- Il colore dell'automobile. Void se non specificato. Il valore di creazione di default è Void.

feature {ANY} -- modifica dell'oggetto

```

set_convertible (*status* : *BOOLEAN*)

```

require
do
    is_convertible := status
ensure
    is_convertible = status
end

```

set_doors (*new_doors* : *ARRAY* [*CAR_DOOR*])

require -- Le due versioni in alternativa sono da accoppiare nell'ordine corrispondente con le versioni in alternativa per le postcondizioni
v_1: *new_doors* != **Void** **implies** (*new_doors.count* = 0 **or** *new_doors.count* = 2 **or** *new_doors.count* = 4)
v_2: *new_doors* != **Void** **and then** (*new_doors.count* = 0 **or** *new_doors.count* = 2 **or** *new_doors.count* = 4)

```

local
    door_index : INTEGER
do
    doors.wipe_out -- la feature wipe_out cancella tutti gli elementi della LINKED_LIST
    if new_doors != Void then
        from
            door_index := 1
        invariant
            doors.count = door_index - 1
            1 ≤ door_index
            door_index ≤ new_doors.count
        until
            door_index > new_doors.count
        loop
            doors.extend (new_doors[door_index])
            door_index := door_index + 1
        variant
            new_doors.count - doors.count - equivalentemente: new_doors.count - doors.index + 1
        end
    end
    ensure -- Le due versioni in alternativa sono da accoppiare con le versioni in alternativa per le
    precondizioni nell'ordine corrispondente
    v_1: (new_doors = Void implies doors.count = 0) or (new_doors != Void implies doors.count = 0 or
doors.count = 2 or doors.count = 4)
    v_2: doors.count = 0 or doors.count = 2 or doors.count = 4
end

```

set_color (*new_color* : *COLOR*)

```

require
do
    color := new_color
ensure
    color = new_color
end

```

invariant

doors != **Void**
doors.count = 0 **or** *doors.count* = 2 **or** *doors.count* = 4
-- equivalente ai due contratti separati: *doors* != **Void** **and then** (*doors.count* = 0 **or** *doors.count* = 2 **or** *doors.count* = 4)

Esercizio 3) [10 punti]

Completare i contratti (pre-condizioni, post-condizioni, invarianti di classe) della classe *POTATO_OVEN* sotto descritta che modella un forno per la cottura di patate in modo da rispecchiare la seguente specifica informale:

1. L'unica porta del forno può essere chiusa o aperta
2. Il forno può contenere al più due patate
3. Per mettere una patata nel forno o toglierla la porta deve essere aperta
4. Per avviare o fermare la cottura bisogna usare il pulsante di avvio/arresto cottura
5. Il forno non inizia la cottura se la porta è aperta o se non c'è niente dentro
6. La porta non può essere aperta se la cottura è in corso: bisogna prima fermarla

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

deferred class*POTATO_OVEN***feature** {ANY}-- accesso*potatoes_to_cook* : *INTEGER*

-- Il numero di patate all'interno del forno.

feature {ANY}-- stato*is_door_open* : *BOOLEAN*

-- La porta del forno è aperta?

is_cooking : *BOOLEAN*

-- La cottura è in corso?

is_empty : *BOOLEAN*

-- Il forno è vuoto?

deferred**ensure****Result** = (*potatoes_to_cook* = 0)**end****feature** {ANY}-- operazioni fondamentali*open_door*

-- Apre la porta del forno

require

.....

.....

deferred**ensure**

.....

.....

end*close_door*

-- Chiude la porta del forno

require

.....

.....

deferred**ensure**

.....

.....

end*put_potato*

-- Mette una patata dentro al forno

require

.....

.....

```

deferred
ensure

```

```

end

```

```

remove_potato

```

```

  -- Toglie una patata dal forno

```

```

require

```

```

deferred
ensure

```

```

end

```

```

switch_on

```

```

  -- Avvia la cottura

```

```

require

```

```

deferred
ensure

```

```

end

```

```

switch_off

```

```

  -- Arresta la cottura

```

```

require

```

```

deferred
ensure

```

```

end

```

```

invariant

```

SOLUZIONE:

```

deferred class

```

```

  POTATO_OVEN

```

```

feature {ANY} -- accesso

```

```

  potatoes_to_cook : INTEGER

```

```

    -- Il numero di patate all'interno del forno.

```

```

feature {ANY} -- stato

```

```

  is_door_open : BOOLEAN

```

```

    -- La porta del forno è aperta?

```

```

  is_cooking : BOOLEAN

```

```

    -- La cottura è in corso?

```

```

  is_empty : BOOLEAN

```

```

    -- Il forno è vuoto?

```

```
deferred  
ensure  
    Result = (potatoes_to_cook = 0)  
end
```

feature {ANY} -- operazioni fondamentali

```
open_door  
    -- Apre la porta del forno  
require  
    not is_door_open  
    not is_cooking  
deferred  
ensure  
    is_door_open  
end
```

```
close_door  
    -- Chiude la porta del forno  
require  
    is_door_open  
deferred  
ensure  
    not is_door_open  
end
```

```
put_potato  
    -- Mette una patata dentro al forno  
require  
    potatoes_to_cook < 2  
    is_door_open  
deferred  
ensure  
    potatoes_to_cook = old potatoes_to_cook + 1  
end
```

```
remove_potato  
    -- Toglie una patata dal forno  
require  
    not is_empty -- equivalentemente: potatoes_to_cook > 0  
    is_door_open  
deferred  
ensure  
    potatoes_to_cook = old potatoes_to_cook - 1  
end
```

```
switch_on  
    -- Avvia la cottura  
require  
    not is_cooking  
    not is_door_open  
    not is_empty -- equivalentemente: potatoes_to_cook > 0  
deferred  
ensure  
    is_cooking  
end
```

```
switch_off  
    -- Arresta la cottura  
require
```



```
    is_cooking  
deferred  
ensure  
    not is_cooking  
end
```

invariant

```
is_cooking implies not is_door_open  
is_cooking implies not is_empty  
-- equivalente ai primi due contratti: (is_door_open or is_empty) implies not is_cooking  
 $0 \leq \textit{potatoes\_to\_cook}$   
 $\textit{potatoes\_to\_cook} \leq 2$ 
```