

# PROGRAMMAZIONE II (A, B) - a.a. 2018-19

## Prima Valutazione Intermedia – 31 Ottobre 2018

### Esercizio 1

Si vuole progettare un tipo di dato astratto `AdvResult` che intende rappresentare i risultati delle campagne pubblicitarie di una determinata azienda. L'elemento tipico di `AdvResult` ha la struttura seguente

```
<company, { < adv-0, likes-0 >, ..., < adv-n-1, likes-n-1 > }>
```

dove `company` è una stringa che descrive il nome dell'azienda, mentre le coppie `< adv-i, likes-i >` descrivono le valutazioni delle campagne pubblicitarie in termini del nome della campagna (`adv-i`) e il numero di *like* ricevuti (`likes-i`). La classe `AdvRecord` descrive la struttura di valutazione di una singola campagna pubblicitaria.

```
class AdvRecord {
    // OVERVIEW: Tipo mutabile che rappresenta il numero di like ottenuto da una campagna pubblicitaria.
    // Typical Element: < name, likes > dove
    //     name e' un valore di tipo String che descrive il nome della campagna, e
    //     likes e' il valore intero non negativo che descrive il numero di like.
    // Rep Invariant: RI(c) = c.likes >= 0 && c.name != null

    private String name;
    private int likes;

    public AdvRecord (String n){
        // THROWS: se n e' null solleva NullPointerException
        // EFFECTS: inizializza this al record con nome n e senza likes.
        if (s==null) throw new NullPointerException(); name = n; likes = 0;
    }

    public void addLikes () {
        // MODIFIES: this.
        // EFFECTS: aggiunge un like a this.
        likes++;
    }

    public String getName () {
        // EFFECTS: restituisce il nome della campagna.
        return name;
    }

    public int likes () {
        // EFFECTS: restituisce il numero di like.
        return likes;
    }
}
```

L'interfaccia di seguito rappresenta una descrizione parziale dell'astrazione `AdvResult`.

```
public interface AdvResult {
    // aggiunge la campagna cmp alla struttura.
    public void addCampaign (String cmp);

    // restituisce le informazione relative alla campagna cmp.
    public AdvRecord getRecord (String cmp);

    // incrementa di uno il valore dei like della campagna cmp.
    public void setLike (String cmp);
}
```

1. Assumendo di adottare una strategia di programmazione difensiva, si completi il progetto del tipo di dato astratto `AdvResult`, definendo le clausole `REQUIRES`, `MODIFIES`, e `EFFECTS` di ogni metodo, indicando le eccezioni eventualmente lanciate e se sono checked o unchecked.

*Si veda l'interfaccia `AdvResult.java`*

2. Nell'ipotesi in cui la struttura di implementazione concreta del tipo `AdvResult` sia

```
private String company;
private Vector<AdvRecord> results;
```

si definiscano la funzione di astrazione e l'invariante di rappresentazione.

*Si veda la classe `MyAdvResult.java`*

3. Si fornisca l'implementazione del costruttore e del metodo `addCampaign` e si dimostri che l'implementazione fornita preserva l'invariante di rappresentazione.

- Abbiamo la rappresentazione e l'invariante

```
private String company;
private Vector<AdvRecord> results;
private int elem;
```

$$RI(c) = c.company! = null \text{ and } c.results! = null \text{ and } \\ c.elem \geq 0 \text{ and } c.elem = c.results.capacity() \text{ and } \\ (\forall i \ 0 \leq i < c.elem \implies c.results.get(i)! = null) \text{ and } \\ (\forall i, j \ 0 \leq i, j < c.elem, i! = j \implies c.results.get(i).getName()! = c.results.get(j).getName()) \\ (\forall i, 0 \leq i < c.elem, c.results.get(i).getName()! = null, c.results.get(i).like() \geq 0 \text{(opzionale)})$$

- Il costruttore soddisfa l'invariante dato che le variabili `company` e `results` vengono inizializzate e non sono quindi `null`. Inoltre la variabile `dim` ha valore 0 e il vettore `results` è vuoto.
- Il metodo `addCampaign` preserva l'invariante di rappresentazione. Infatti il vettore `results` viene modificato solo se non esiste già una coppia relativa alla campagna `cmp`. Quindi la coppia `< cmp, 0 >` viene aggiunta al vettore solo se non esiste già una coppia `< cmp, likes >` in `results`. In tale caso viene anche incrementato il valore di `dim`.

4. Si vuole ora trattare un caso più generale nel quale il tipo `AdvResult` abbia oltre al numero dei like altri parametri per misurare l'effetto di una campagna. Si descriva in quale modo modificare la classe `AdvResult` per raggiungere tale effetto e se ne motivi la scelta.

*Possiamo definire una versione generica di `AdvResult` e di conseguenza anche di `AdvRecord`. Conviene considerare un tipo parametrico `V` equipaggiato con una nozione di ordinamento dei valori in modo da misurare il valore delle campagne. Il progetto della classe si generalizza come descritto nel seguito.*

```
class AdvRecord <V extends Comparable<V>> {
    // OVERVIEW: Tipo modificabile che descrive i likes di una campagna pubblicitaria.
    // Typical Element: < name, val > dove
    // name e' un valore di tipo String che descrive il nome della campagna
    // e val e' un oggetto di tipo V che misura il valore della campagna.

    private String name;
    private V val;
}
```

```
class AdvResult<V extends Comparable<V>> {
    // OVERVIEW: Tipo modificabile che descrive i risultati delle campagne pubblicitarie di una
    // determinata azienda.
```

```

//Typical Element <company, { < adv_0, val_0 >, ..., < adv_n-1, val_n-1 > } >
//dove adv_i != adv_j per ogni i,j tale che 0 <=i < j <n

    private String company;
    private Vector<AdvRecord<V>> results;
    private int elem;
}

```

## Esercizio 2

Si consideri il seguente programma in Java

```

class A {
    void f() {
        System.out.println("A f");
    }
}

class B extends A {
    void f() {
        super.f();
        System.out.println("B f");
    }
}

public static void main(String[] args) {
    B b = new B();
    b.f();
    A a = b;
    a.f();
}

```

- Motivando la risposta, si dica quale è il risultato osservabile dell'esecuzione del metodo main.

*Il risultato osservabile dell'esecuzione del metodo main è la stampa A f B f A f B f . Dynamic Dispatch per la invocazione del metodo a.f()*

## Esercizio 3

Si consideri il seguente frammento di codice Java

```

class Account {
    protected int balance;
    public int withdraw(int m) {
        if (m <= 0) throws new IllegalArgumentException();
        this.balance = this.balance - m;
        return m;
    }
}

class BetterAccount extends Account {
    public int withdraw(int m) {
        if (m <= 0) throws new IllegalArgumentException();
        if (m <= this.balance) {
            this.balance = this.balance - m;
        }
    }
}

```

```

        return m;
    } else { return 0; }
}
}

```

- Motivando la risposta si dica se la gerarchia di classi soddisfa il principio di sostituzione.

*Per rispondere alla domanda aggiungiamo una possibile specifica alle due classi. In questo caso **BetterAccount** non è sotto-tipo di **Account** dato che la pre-condizione del metodo **withdraw** del sotto-tipo è rafforzata da quella del super-tipo.*

```

class Account {
    protected int balance;
    public int withdraw(int m) {
        // REQUIRES: m > 0
        // MODIFIES: this
        // THROWS: m <= 0 solleva IllegalArgumentException
        // EFFECTS: modifica this sottraendo e restituisce m
        if (m <= 0) throws new IllegalArgumentException();
        this.balance = this.balance - m;
        return m;
    }
}
}

```

```

class BetterAccount extends Account {
    public int withdraw(int m) {
        / REQUIRES: m > 0 and m <= this.balance
        // MODIFIES: this
        // THROWS: m <= 0 solleva IllegalArgumentException
        // EFFECTS: se m >= valore di this allora modifica this sottraendo m e restituisce m
        // altrimenti restituisce 0
        if (m <= 0) throws new IllegalArgumentException();
        if (m <= this.balance) {
            this.balance = this.balance - m;
            return m;
        } else { return 0; }
    }
}
}

```