Programmazione ad Oggetti
Mod. 2

25/6/2024

Studente _____ Matricola _____

1. Si implementi in Java 8+ un sistema di classi che rappresentano punti, segmenti e poligoni nel piano cartesiano $\mathbb{R} \times \mathbb{R}$.
   (a) [1 punti] Si implementi una classe di nome `Point` che rappresenta punti bidimensionali immutabili in cui le coordinate `x` ed `y` sono di tipo `double`.
   (b) [2 punti] Si implementi una classe di nome `Segment` che rappresenta segmenti bidimensionali immutabili, il cui costruttore prende due argomenti di tipo `Point`. Essa deve fornire un metodo `length()` che restituisce la lunghezza del segmento calcolando la distanza euclidea tra i due punti.
   (c) Si prenda in considerazione la seguente classe astratta `Polygon` che rappresenta poligoni come liste di punti (minimo 3, verificato a runtime tramite un assert in costruzione). I punti nella lista determinano l'ordine di costruzione dei segmenti di cui è composto il poligono. Ad esempio, una lista contenente i seguenti 4 punti nel seguente ordine $A = (0,0), B = (2,3), C = (3,0), D = (2,-1)$ rappresenta un quadrilatero in cui il primo lato è $\overline{AB}$, il secondo è $\overline{BC}$, il terzo è $\overline{CD}$ ed il quarto è $\overline{DA}$. In altre parole, l'ultimo punto della lista crea l'ultimo segmento con il primo punto della lista, per chiudere il poligono.

```java
public abstract class Polygon implements Iterable<Segment> {
    protected final List<Point> points;

    protected Polygon(List<Point> points) {
        assert points.size() >= 3;
        this.points = points;
    }

    public Iterator<Segment> iterator() { /* da implementare */ }
    public double perimeter() { /* da implementare */ }
    public abstract double area();
}
```

      i. [3 punti] Si implementi il metodo `iterator()` che costruisce un iteratore di oggetti di tipo `Segment` creandoli al volo unendo i punti della lista `points` in maniera opportuna.
      ii. [1 punti] Si implementi il metodo `perimeter()` in funzione del metodo `iterator()`, ovvero calcolando il perimetro del poligono sommando semplicemente le lunghezze dei segmenti che lo compongono.
   (d) Estendiamo ora la gerarchia di classi introducendo tipi specializzati per alcuni poligoni noti.
      i. [2 punti] Si implementi una sottoclasse di `Polygon` di nome `Triangle` che rappresenta triangoli qualunque.

```java
public class Triangle extends Polygon {
    public Triangle(Point p1, Point p2, Point p3) { /* da implementare */ }

    @Override
    public double area() { /* da implementare */ }
}
```

Il costruttore prende 3 argomenti di tipo `Point` e deve chiamare il super-costruttore opportunamente. Si implementi il metodo `area()` in modo che calcoli l'area del triangolo senza fare assunzioni sulla sua forma[1].

ii. [2 punti] Si implementi una sottoclasse di `Polygon` di nome `Rectangle` che rappresenta rettangoli.

```
public class Rectangle extends Polygon {
    public Rectangle(Point p1, Point p3) { /* da implementare */ }

    @Override
    public double area() { /* da implementare */ }
}
```

Il costruttore prende i 2 punti delle diagonale principale del rettangolo (ovvero quella avente come primo punto il vertice in basso a sinistra e come secondo punto il vertice in alto a destra) e deve passare al super-costruttore la lista con i 4 punti che costituiscono il rettangolo, calcolandone le coordinate opportunamente e facendo attenzione all'ordine in cui compaiono nella lista. Si implementi poi il metodo `area()` in modo che calcoli l'area del rettangolo.

iii. [2 punti] Si implementi una sottoclasse di `Rectangle` di nome `Square` che rappresenta quadrati.

```
public static class Square extends Rectangle {
    public Square(Point p1, double side) { /* da implementare */ }
}
```

Il costruttore prende il punto che rappresenta il vertice in basso a sinistra del quadrato e la dimensione del lato. Deve chiamare il super-costruttore opportunamente.

iv. [2 punti (bonus)] Sarebbe necessario e/o consigliabile definire un override del metodo `area()` per la classe `Square`? Si argomenti la risposta.

(e) Si prenda ora in considerazione il seguente codice, in cui compare l'invocazione di un metodo statico `min()` da implementare.

```
Square sq1 = new Square(new Point(11.235, -8.53), 0.1),
       sq2 = new Square(new Point(1., 20.), 0.01),
       sq3 = new Square(new Point(0., 0.), 0.2);
List<Square> squares = List.of(sq1, sq2, sq3);
Square r = min(squares, (a, b) -> (int) (a.area() - b.area()));
```

i. [3 punti] Si scrivano firma e implementazione del metodo statico `min()` invocato nell'ultimo statement affinché il codice di cui sopra compili correttamente. Si renda tale metodo più generico possibile e non monomorfo rispetto ai tipi che compaiono in questa invocazione, prestando particolare attenzione ai vincoli sui generics. La semantica di `min()` è facilmente intuibile: trova l'elemento minore usando un `Comparator` per confrontare gli elementi della lista di input. Si implementi `min()` sfruttando il metodo statico `sort()` definito nella classe `Collections` del JDK, del quale riportiamo qui la firma:

```
static <T> void sort(List<T> l, Comparator<? super T> c)
```

ii. [1 punti] Assumendo il comportamento corretto del metodo `min()`, quale delle seguenti espressioni booleane in Java computerebbe `true`? Si barri la risposta corretta.

- ◯ `sq1 == r`
- ◯ `sq2 == r`
- ◯ `sq3 == r`
- ◯ `r == null`
- ◯ `r.equals(sq1)`
- ◯ `r.equals(sq2)`
- ◯ `r.equals(sq3)`

2. Si scriva in C++ (specificando quale revisione del linguaggio si intende adottare) una classe generica `matrix` che rappresenta matrici bidimensionali di valori di tipo `T`, dove `T` è un tipo templatizzato. Tale classe deve comportarsi come un *valore*, implementando lo stile della *value-oriented programming* e della *generic programming*. Inoltre deve aderire al concept denominato *Container* da STL.

   Si implementino dunque:

[1] La formula di Erone per calcolare l'area di un triangolo qualsiasi conoscendone solo la lunghezza dei lati è $\sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$ dove a, b, c sono i lati e p è il semiperimetro.

(a) [1 punti] costruttore di default;

(b) [1 punti] costruttore per copia;

(c) [1 punti] costruttore con 2 parametri + 1 opzionale: numero di righe, numero di colonne e valore iniziale di tipo `T`;

(d) [1 punti] distruttore (se necessario);

(e) [2 punti] operatore di assegnamento;

(f) [3 punti] member type `iterator`, `const_iterator`, `value_type`, `reference`, `const_reference` e `pointer`;

(g) [2 punti] operatore di accesso tramite riga e colonna: implementare 2 overload (const e non-const) di `operator()`;

(h) [3 punti] metodi `begin()` ed `end()`: 2 overload (const e non-const) per poter iterare tutta la matrice come un unico container lineare dall'angolo superiore sinistro all'angolo inferiore destro come se fosse piatta;

(i) [2 punti] altri metodi a piacere che si ritengono utili.

L'implementazione può utilizzare STL liberamente.

Si prenda a riferimento il seguente snippet per la specifica dei requisiti del tipo `matrix`:

```
matrix<double> m1;           // non inizializzata
matrix<double> m2(10, 20);   // 10 X 20 inizializzata col default constructor di double
matrix<double> m3(m2);       // costruita per copia
m1 = m2;                     // assegnamento
m3(3, 1) = 11.23;            // operatore di accesso come left-value
for (typename matrix<double>::iterator it = m1.begin(); it != m1.end(); ++it) {
    typename matrix<double>::value_type& x = *it;  // de-reference non-const
    x = m2(0, 2);  // operatore di accesso come right-value
}
matrix<string> ms(5, 4, "ciao"); // 5 X 4 inizializzata col la stringa passata come terzo argomento
for (typename matrix<string>::const_iterator it = ms.begin(); it != ms.end(); ++it)
    cout << *it;   // de-reference const
```

Question:	1	2	Total
Points:	17	16	33
Bonus Points:	2	0	2
Score:			