Piastrelle

Sezioni

La traccia

La traccia inizia con l'elencazione di alcune delle entità utili a gestire alcuni aspetti dell'attività di un piastrellista.

La piastrella

Una **piastrella** è un elemento architettonico usato per rivestire la superficie dei pavimenti; ne esistono di diversi tipi, ciascuno dei quali è caratterizzato da una serie di caratteristiche, tra le quali almeno la *superficie* e il *costo* unitario (che per semplicità assumeremo rappresentati da numeri interi).

Esempi di piastrelle possono essere: triangolari (caratterizzate da base e altezza), quadrate (caratterizzate dalla lunghezza del lato), romboidali (caratterizzate dalla lunghezza delle due diagonali) e così via…

La pavimentazione

Una **pavimentazione** è costituita da una collezione (finita e non vuota) di piastrelle, o altre pavimentazioni; la sua *superficie* è pari alla somma delle superfici di tutte le piastrelle che contiene (direttamente o indirettamente) e il suo *costo* totale è dato dalla somma dei costi di tutte le piastrelle che contiene. Evidentemente una pavimentazione non può contenere sé stessa (direttamente o indirettamente).

Se ad esempio:

Skip to main content

- la pavimentazione della cucina è fatta di 42 piastrelle quadrate di lato 2 e prezzo 3,
- quella del bagno è fatto di 65 piastrelle romboidali con diagonali di lunghezza 4 e 2 con prezzo 5 e
- la pavimentazione della casa è costituita da 1 pavimentazione della cucina più 2 pavimentazioni del bagno,

la superficie della pavimentazione della casa è pari a

```
42 x (2 x 2) + 2 x [65 x (4 x 2) / 2] = 688
```

dove il primo addendo corrisponde alla superficie della pavimentazione della cucina, mentre la quantità tra parentesi quadre corrisponde alla superficie della pavimentazione del bagno. Similmente, il costo della pavimentazione della casa è dato da

```
42 x 3 + 2 x [65 x 5] = 776
```

dove i due addendi sono rispettivamente i costi della pavimentazione della cucina e dei due bagni.

Cosa è necessario implementare

Dovrà implementare una gerarchia di oggetti utili a rappresentare piastrelle e pavimentazioni e a conoscerne superfici e costi.

In particolare dovrà essere possibile creare piastrelle di diverso tipo (a partire quanto meno dall'indicazione delle grandezze che le caratterizzano) e pavimentazioni (a partire quanto meno dalle loro componenti).

Per verificare il comportamento del suo codice le può essere utile implementare una *classe di test* che, leggendo dal flusso di ingresso un elenco di azioni, le realizzi (creando le necessarie istanze di oggetti d'appoggio).

Le azioni, indicate una per riga, sono specificate da un carattere seguito da uno o più numeri interi; ciascuna azione determina la creazione di una piastrella o pavimentazione, che si considerano pertanto indicizzate dal numero di riga in cui sono state create (le righe sono numerate a partire da 0). Le azioni sono:

- `Q` seguita da due interi, crea una piastrella quadrata di lato e costo assegnato;

Skip to main content

- `T` seguita da tre interi, crea una piastrella triangolare di base, altezza e costo assegnati;
- `P` seguita da *2n* interi, crea una pavimentazione; le *n* coppie di interi rappresentano ciascuna la quantità e l'indice di una delle *n* piastrelle o pavimentazioni di cui è costituita.

Assuma che gli indici delle piastrelle, o pavimentazioni, che seguono la lettera `P` siano distinti e strettamente minori del numero di riga in cui compaiono (questo garantisce che le pavimentazioni siano in effetti insiemi e che una pavimentazione non comprenda mai, neppure indirettamente, sé stessa); assuma anche che le dimensioni siano sempre scelte in modo tale che la superficie risultante sia intera.

Una volta lette tutte le azioni, l'esecuzione della classe di test termina stampando una linea per ciascuna pavimentazione creata (in ordine di creazione) contenente due interi (separati dal segno di tabulazione) corrispondenti alla sue superficie e costo.

L'elenco di azioni che rappresenta le pavimentazioni dell'esempio nella sezione precedente è:

```
Q 2 3
P 42 0
R 4 2 5
P 65 2
P 1 1 2 3
```

le prime due righe creano le piastrelle quadrate e la pavimentazione della cucina (che è quindi indicizzata dal numero 1), le righe di numero 2 e 3 creano le piastrelle romboidali e la pavimentazione del bagno (indicizzata dal numero 3) e l'ultima riga crea la pavimentazione della casa (data da 1 pavimentazione della cucina, di indice 1, e 2 pavimentazioni del bagno, di indice 3).

A fronte di tali azioni, la classe di test emette

```
168     126
260     325
688     776
```

nel flusso d'uscita, dove l'ultima riga riporta i numeri calcolati nella precedente sezione, mentre le prime due sono il valori relativi rispettivamente a cucina e bagno.

Nota bene: è possibile implementare anche altri tipi di piastrelle oltre a quadrati, rombi e triangoli (aggiungendo le opportune azioni alla classe di test), così come potrebbe essere

Skip to main content

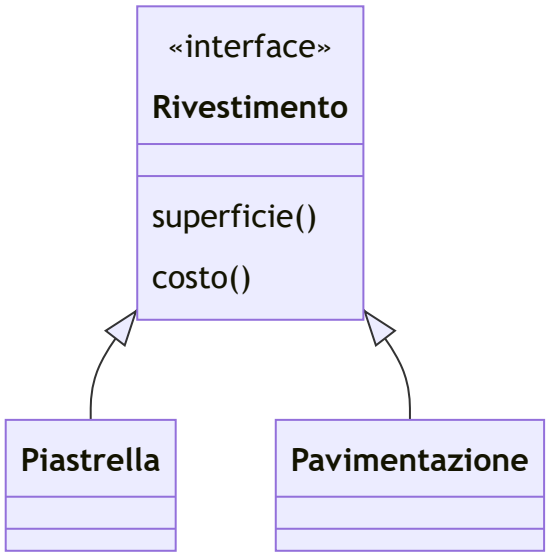
illustrato durante il corso), rispetto a quelle strettamente necessarie per implementare la classe di test.

La soluzione

Una prima domanda da porsi riguarda la *mutabilità*, che non appare necessaria dato il problema assegnato che consiste nel descrivere alcuni rivestimenti essendo in grado di calcolarne il prezzo e la superficie. Per questa ragione, le entità in gioco verranno tutte specificate come immutabili.

Il rivestimento

La prima cosa che salta all'occhio è che *superficie* e *costo* sono proprietà di entrambe le entità; questo suggerisce che potrebbero essere convenientemente raccolte in una *interfaccia* (che potremmo chiamare `Rivestimento`) questo consentirebbe, tra l'altro, di trattare le due entità in modo omogeneo grazie al *polimorfismo* (fatto che sembra particolarmente utile dal momento che una pavimentazione può essere costituita da entrambe le entità). La situazione al momento è quindi rappresentata dal seguente diagramma



la cui interfaccia corrisponde al seguente codice

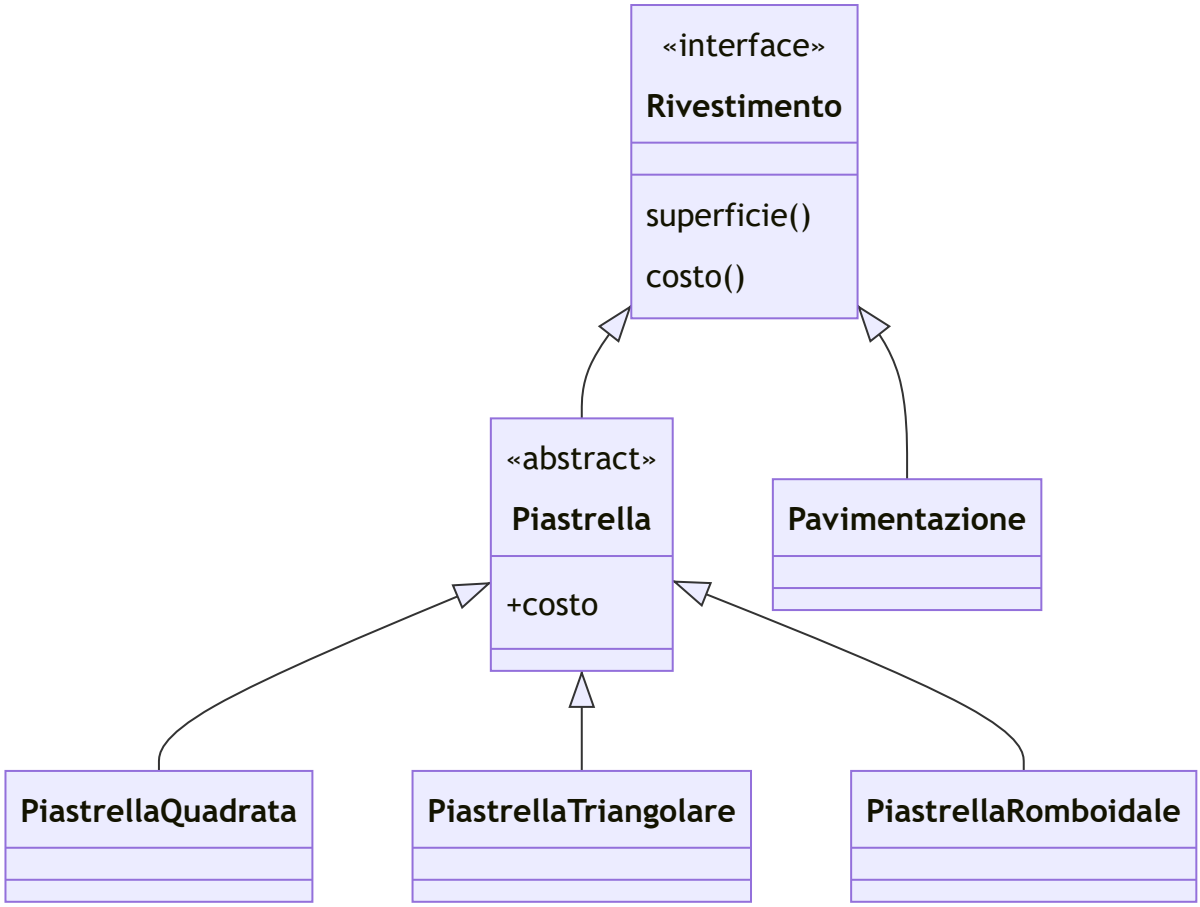
```java
public interface Rivestimento {
  int costo();

  int superficie();
}
```

Skip to main content

Le piastrelle

Per quanto concerne le piastrelle, ne sono presenti di diverso tipo, distinte quanto meno dalla forma (ma in linea di principio potrebbero essere costituite di materiali diversi, o avere proprietà diverse, ad esempio dal punto di vista termico, o di resistenza); questo sembra suggerire lo sviluppo di una gerarchia di tipi, sopratutto in considerazione del fatto che quanto meno è difficile immaginare la segnatura di un costruttore comune alle diverse forme. Una delle competenze che distingue le piastrelle è certamente quella di calcolare la propria superficie a partire dalle misure che la caratterizzano, viceversa, il prezzo appare come una caratteristica comune a tutte le sottoclassi. Queste considerazioni fanno propendere verso la realizzazione di una classe astratta il cui stato coincida col solo prezzo, da cui derivare le sottoclassi di diverso tipo. Il diagramma precedente diventa quindi:



Un primo abbozzo di implementazione della piastrella è dato dal seguente codice

```
public abstract class Piastrella implements Rivestimento {

  private final int costo;

  public Piastrella(final int costo) {
    if (costo <= 0) throw new IllegalArgumentException("Il costo dev'essere positiv
```

Skip to main content

```
  @Override
  public int costo() {
    return costo;
  }
}
```

[sorgente]

Occorre osservare che la classe è astratta in quanto il metodo `superficie` prescritto dall'interfaccia che implementa non è un suo metodo concreto (ragione per cui, tra l'altro, non c'è bisogno di aggiungerlo come metodo astratto).

Inoltre, poiché l'unico attributo della classe è il `costo` che è di tipo primitivo, è sufficiente indicare il modificatore `final` per garantire che il suo valore resti immutabile; sebbene potrebbe essere esposto come `public`, per soddisfare l'interfaccia è necessario implementare un *getter*. Unica accortezza, dal momento che il costo è sempre positivo (unico invariante di rappresentazione della classe) è necessario controllarne il valore in costruzione (come mostra la linea evidenziata nel codice precedente).

Le sottoclassi concrete sono elementari da implementare. Una bozza di codice che comprenda solo un costruttore e quanto necessario a soddisfare l'interfaccia (ereditata dalla classe astratta) è (per la sola piastrella romboidale) il seguente

```
public class PiastrellaRomboidale extends Piastrella {

  public final int minore;

  public final int maggiore;

  public PiastrellaRomboidale(final int prima, final int seconda, final int costoUn
    super(costoUnitario);
    if (prima <= 0) throw new IllegalArgumentException("La prima diagonale dev'esse
    if (seconda <= 0)
      throw new IllegalArgumentException("La seconda diagonale dev'essere positiva.
    if (prima < seconda) {
      minore = prima;
      maggiore = seconda;
    } else {
      minore = seconda;
      maggiore = prima;
    }
  }

  @Override
  public int superficie() {
    return (minore * maggiore) / 2;
  }
```

Skip to main content

[sorgente]

Osserviamo che gli attributi sono rappresentabili con variabili di tipo primitivo, l'immutabilità è quindi garantita dall'attributo `final` ed è ragionevole omettere i *getter* rendendo l'attributo `public`.

Come sopra, l'unica accortezza è garantire l'invariante che quantomeno richiede che ciascuna dimensione sia positiva; nel caso delle piastrelle romboidali è comodo che il costruttore accetti due valori per le diagonali (comunque ordinati, per facilitare l'uso del costruttore da parte degli utenti della classe), ma è ragionevole che la rappresentazione distingua la maggiore dalla minore: questo richiede che l'assegnamento dei parametri del costruttore agli attributi della classe sia fatto avvedutamente (come mostra il codice evidenziato).

La pavimentazione

La pavimentazione deve immagazzinare una collezione di rivestimenti con le relative quantità. Per ottenere questo risultato ci sono (almeno) tre possibilità:

- due *vettori* (o *liste*) "paralleli", uno di rivestimenti e uno di interi (della stessa dimensione), che in ciascuna coordinata indichino rispettivamente un rivestimento e la relativa quantità,
- una *mappa* dai rivestimenti agli interi che ne indicano la quantità,
- un *vettore* (o *lista*) di "record" ciascuno dei quali contenga una coppia rivestimento e quantità.

La prima possibilità richiede una certa attenzione nel mantenimento dell'invariante di rappresentazione (avere a che fare con due attributi che vanno mantenuti in modo coordinato può non essere del tutto banale), la seconda richiede l'uso delle mappe (che potrebbe non essere ovvio), mentre la terza sembra la più semplice.

Procediamo quindi con l'implementare un record, che chiameremo `Componente`; osserviamo che è sensato che esso implementi l'interfaccia `Rivestimento` (è infatti in grado di calcolare la sua superficie e costo, essendogli nota quella del rivestimento da cui è composto e dalla sua quantità — come mostrano le linee evidenziate).

```
public static class Componente implements Rivestimento {

  public final Rivestimento rivestimento;
```

Skip to main content

```
  public Componente(final int quantità, final Rivestimento rivestimento) {
    this.rivestimento =
        Objects.requireNonNull(rivestimento, "Il rivestimento non può essere null."
    if (quantità <= 0) throw new IllegalArgumentException("La quantità dev'essere p
    this.quantità = quantità;
  }

  @Override
  public int costo() {
    return quantità * rivestimento.costo();
  }

  @Override
  public int superficie() {
    return quantità * rivestimento.superficie();
  }

}
```

[sorgente]

Anche questa entità è immutabile, il suo stato è dato da due attributi che ne rappresentano lo stato a patto che il rivestimento sia non nullo e la quantità positiva, invariante che è controllato in costruzione.

Tale classe ha poco senso al di fuori dell'uso che ne faremo come parte di una pavimentazione, per questa ragione può essere implementata come classe interna (statica).

A questo punto lo stato della pavimentazione è semplicemente dato da una lista di componenti, tale rappresentazione è valida a patto che:

- ai componenti non corrisponda un riferimento a `null`,
- la lista non sia vuota e
- nessun componente (contenuto nella lista) sia un riferimento a `null`.

Lo stato e il costruttore (che garantisce tale invariante) sono dati dal codice seguente:

```
private final Collection<Componente> componenti;

public Pavimentazione(final Collection<Componente> componenti) {
  this.componenti = List.copyOf(componenti);
  if (componenti.isEmpty())
    throw new IllegalArgumentException("Ci deve essere sempre almeno una componente
}
```

[sorgente]

Skip to main content

Si noti la linea evidenziata in cui all'attributo della classe non è assegnato il valore del riferimento passato come parametro (che restando in possesso del chiamante renderebbe la esposta la rappresentazione), ma viene fatta una copia tramite il metodo di fabbricazione di `List.copyOf` il quale, tra l'altro, assicura che `componenti` non sia e non contenga `null`.

La rappresentazione scelta rende banale soddisfare l'interfaccia `Rivestimento`, sono infatti sufficienti poche linee di codice:

```
@Override
public int costo() {
  int totale = 0;
  for (final Rivestimento r : componenti) totale += r.costo();
  return totale;
}

@Override
public int superficie() {
  int totale = 0;
  for (final Rivestimento r : componenti) totale += r.superficie();
  return totale;
}
```

[sorgente]

La chiave di questo risultato è aver introdotto una interfaccia come supertipo di entrambe le componenti della pavimentazione, questo ha consentito di sviluppare `Componente` in modo elementare, senza dove distinguere i casi in cui il suo "contenuto" sia una piastrella, o un rivestimento.

Per finire, è plausibile rendere la pavimentazione un iterabile di componenti

```
@Override
public Iterator<Pavimentazione.Componente> iterator() {
  return componenti.iterator();
}
```

[sorgente]

in questo modo è possibile comunicare all'esterno il suo stato senza che ne venga esposta la rappresentazione.

La classe di test

Skip to main content

creati e una delle sole pavimentazioni)

```
final List<Rivestimento> rivestimento = new ArrayList<>();
final List<Pavimentazione> pavimentazione = new ArrayList<>();
```

[sorgente]

tali liste saranno popolate da un ciclo che legge tratta una linea per volta (ottenuta da uno `Scanner` che avvolge il flusso di ingresso); ciascuna linea può essere a sua volta avvolta in uno `Scanner` per suddividerla nelle sue parti (su cui agire con uno `switch`) secondo la seguente struttura

```
try (final Scanner s = new Scanner(System.in)) {
  while (s.hasNextLine())
    try (final Scanner line = new Scanner(s.nextLine())) {
      /* switch */
    }
}
```

Il corpo dello `switch` decide cosa fare a seconda del primo carattere (della prima parola) della linea, quindi consuma gli altri interi per ottenere i parametri da passare ai costruttori.

```
switch (line.next().charAt(0)) {
  case 'Q':
    rivestimento.add(new PiastrellaQuadrata(line.nextInt(), line.nextInt()));
    break;
  case 'R':
    rivestimento.add(
        new PiastrellaRomboidale(line.nextInt(), line.nextInt(), line.nextInt()));
    break;
  case 'T':
    rivestimento.add(
        new PiastrellaTriangolare(line.nextInt(), line.nextInt(), line.nextInt()));
    break;
  case 'P':
    final List<Pavimentazione.Componente> componenti = new ArrayList<>();
    while (line.hasNextInt())
      componenti.add(
          new Pavimentazione.Componente(
              line.nextInt(), rivestimento.get(line.nextInt())));
    final Pavimentazione p = new Pavimentazione(componenti);
    pavimentazione.add(p);
    rivestimento.add(p);
    break;
  default:
    throw new IllegalArgumentException("Errore nel formato.");
}
```

Skip to main content

Di particolare interesse è la parte evidenziata che costruisce una lista di componenti e la popola con un ciclo in cui vengono lette le quantità e gli indici dei rivestimenti da utilizzare per costruire la pavimentazione; l'invocazione `rivestimento.get(...)` è quella che permette di ottenere il rivestimento dato il suo indice.

> **Nota**
>
> Puoi trovare il codice integrale della soluzione nell'apposito repository.