

# Bancarelle

## Sezioni

- [La traccia](#)
- [Soluzione](#)

## La traccia

Se siete andati in vacanza al mare, vi sarà senz'altro capitato di vedere che (specialmente alla fine della stagione) i bambini organizzano sulle spiagge, o nelle piazzette, delle bancarelle per vendere i giocattoli usati. Spesso si trova lo stesso giocattolo in bancarelle diverse e ciascun bambino decide a modo suo i prezzi, talvolta inventandosi sconti e offerte per attrarre i compratori.

Scopo del progetto è modellare le entità coinvolte in questa attività in modo da poter rappresentare più giocattoli offerti in diverse bancarelle secondo diverse politiche di prezzo e dei compratori che acquistino, potenzialmente seguendo differenti strategie d'acquisto, un certo numero di giocattoli.

Ad esempio, date queste bancarelle

```
Bancarella di: Massimo
num. 2 bilia di vetro, prezzo: 2
num. 3 cane di pezza, prezzo: 10
num. 10 elastico di gomma, prezzo: 1
num. 1 soldatino di stagno, prezzo: 3

Bancarella di: Carlotta
num. 4 bilia di vetro, prezzo: 1
num. 10 braccialetti di perline, prezzo: 3
num. 1 soldatino di stagno, prezzo: 5

Bancarella di: Federico
num. 10 bilia di vetro, prezzo: 3
num. 1 cane di pezza, prezzo: 5
num. 10 soldatino di stagno, prezzo: 2
```

un compratore che volesse acquistare 11 giocattoli **soldatino di stagno** potrebbe, ad esempio, effettuare il seguente acquisto

```
Acquisto di: soldatino di stagno, per un costo di: 23, numero: 11 di cui:
10 da Federico
1 da Massimo
```

## La bancarella: giocattolo, inventario e listino prezzi

Per semplicità assumeremo che ciascun **giocattolo** abbia un nome (rappresentato con una *stringa*) e sia fatto di un dato materiale (anch'esso rappresentato da una *stringa*); ad esempio, una bambola di pezza, una bilia di vetro e una bilia di marmo sono tre giocattoli gli ultimi due differiscono nel materiale, ma non nel nome. Due giocattoli sono uguali se e solo se hanno lo stesso nome e sono fatti dello stesso materiale.

Ogni bancarella offre un certo insieme di giocattoli, per tener traccia di quanti e quali giocattoli offra in un certo momento è utile usare un **inventario**: una classe in grado di tener traccia dei giocattoli man mano aggiunti ed eliminati (ad esempio perché venduti) dalla bancarella.

Il gestore di ogni bancarella può decidere diverse *politiche di prezzo* per ciascun giocattolo: ad esempio, può fissare un prezzo unitario  $U$  per un dato giocattolo e stabilire che il prezzo di  $N$  giocattoli identici a esso sia dato *moltiplicativamente* da  $U * N$ , oppure applicare degli *sconti* (per esempio, se  $N$  supera la decina, applicare un 15% di sconto sulle unità eccedenti la decina, in modo che il prezzo finale sia  $10 * U + (N - 10) * U * 85 / 100$ ), o vendere «tre giocattoli al prezzo di due», e così via.

Un modo ragionevole di rappresentare queste politiche è definire un **listino** che, dato un giocattolo e la quantità da acquistare, restituisca il prezzo complessivo; più precisamente, descrivete una *interfaccia Listino* e almeno una *classe* che la implementi (ad esempio, quella che descriva la semplice politica moltiplicativa).

Ogni **bancarella** è identificata da un *proprietario* (che è rappresentato tramite una stringa) e ha i suoi *inventario* e *listino*; evidentemente il listino deve permettere di conoscere il prezzo di ciascun giocattolo presente nell'inventario.

Una bancarella deve poter indicare quanti giocattoli di un certo tipo è in grado di vendere e a che prezzo, nonché procedere alla vendita (aggiornando l'inventario).

## Compratore e acquisto

Se più bancarelle offrono lo stesso giocattolo, il *compratore* che intenda acquistarne una certa quantità, può comporre il suo **acquisto** in modi diversi, decidendo di acquistare un diverso numero di giocattoli dalle varie bancarelle che lo offrono, magari cercando di minimizzare il prezzo totale.

L'acquisto (di un determinato giocattolo) è pertanto caratterizzato da: il *giocattolo* stesso, la *quantità* acquistata e il *prezzo* pagato, nonché dall'elenco delle *bancarelle*, ciascuna accompagnata dal numero di giocattoli che ha venduto.

Implementate la classe **Acquisto** che consenta di gestire tali informazioni. Un esempio di acquisto potrebbe essere

```
Acquisto di: soldatino di stagno, per un costo di: 23, numero: 11 di cui:  
10 da Federico  
1 da Massimo
```

Finalmente è arrivato il momento di occuparsi del **compratore**. Questo, una volta noto l'insieme di *bancarelle* da cui fare acquisti, può comprare un certo numero di giocattoli di un dato tipo seguendo diverse strategie: comprando dalla bancarella che esibisce il minor prezzo unitario, o dalle bancarelle che hanno maggior disponibilità del giocattolo, o scegliendo a caso da quali bancarelle comprare.

### ➡ Vedi anche

Osservate che determinare una strategia «ottima» è in generale molto difficile e non è affatto richiesto per portare a termine il progetto, gli studenti curiosi possono farsi una idea della questione sfogliando ad esempio l'articolo [Allocating procurement to capacitated suppliers with concave quantity discounts](#) o il rapporto tecnico [An exact method for the Capacitated Total Quantity Discount Problem](#).

Potrebbe aver senso raccogliere alcune competenze comuni a tutti i compratori in una *classe astratta* fornendo poi delle implementazioni concrete che realizzino in modo diverso le varie strategie d'acquisto.

In ogni modo, la classe concreta dovrà avere almeno un costruttore che riceva un parametro di tipo **Set<Bancarella>** e un metodo di segnatura

```
public Acquisto compra(final int num, final Giocattolo giocattolo)
```

che sarà usata per effettuare l'acquisto.

## La classe di test

Per ottenere la classe di test di questo esercizio, partite dalla *bozza di sorgente* della funzione **main** così definita

```

public static void main(final String[] args) {

    /* Lettura dei parametri dalla linea di comando */
    final int numDaComprare = Integer.parseInt(args[0]);
    final Giocattolo giocattoloDaComprare = new Giocattolo(args[1], args[2]);

    /* Lettura del flusso di ingresso */
    final Scanner s = new Scanner(System.in);

    final int numBancarelle = s.nextInt();
    final Set<Bancarella> bancarelle = new HashSet<>(numBancarelle);
    final Map<Giocattolo, Integer> giocattolo2prezzo = new HashMap<>();
    final Inventario inventario = new Inventario();

    for (int b = 0; b < numBancarelle; b++) {
        /* Lettura di una bancarella */
        final String proprietario = s.next();
        final int numGiochi = s.nextInt();
        for (int g = 0; g < numGiochi; g++) {
            /* Lettura dei giochi della bancarella */
            final int num = s.nextInt();
            final String nome = s.next();
            final String materiale = s.next();
            final int prezzo = s.nextInt();
            final Giocattolo giocattolo = new Giocattolo(nome, materiale);
            inventario.aggiungi(num, giocattolo);
            giocattolo2prezzo.put(giocattolo, prezzo);
        }

        /*
        MODIFICARE: aggiungere l'istanziatura del Listino, es:

        final Listino listino = new ListinoConcreto(giocattolo2prezzo);
        */

        final Bancarella bancarella = new Bancarella(proprietario, inventario, listino);
        bancarelle.add(bancarella);
    }

    s.close();

    /*
    MODIFICARE: aggiungere l'istanziatura del compratore, es:

    final Compratore compratore = new Compratore(bancarelle);
    */

    final Acquisto ordine = compratore.compra(numDaComprare, giocattoloDaComprare);
    System.out.println(ordine);
}

```

L'*input* a tale classe è fornito sia dai parametri sulla linea di comando (che indicano quale giocattolo comprare e in che quantità), che dal flusso di ingresso (che contiene una descrizione delle bancarelle).

I tre parametri sulla linea di comando indicano rispettivamente il numero di giocattoli da comprare, il nome e il materiale del giocattolo. Il flusso di ingresso contiene i seguenti elementi (separati da *white-space*):

- un intero positivo corrispondente al numero di bancarelle;
- per ciascuna bancarella:
  - una stringa corrispondente al nome del proprietario,
  - il numero di giocattoli della bancarella,
  - una quaterna di valori per ciascun giocattolo corrispondenti al:
    - numero,
    - nome,
    - materiale e
    - prezzo unitario.

Una volta costruito il compratore, esso dovrà effettuare l'acquisto specificato ed emetterlo nel flusso d'uscita (secondo il formato dato dall'esempio).

## Esempio

Eseguito *soluzione 11 soldatino stagno* e avendo

```

3
Massimo 4
3 cane pezza 10
2 bilia vetro 2
10 elastico gomma 1
1 soldatino stagno 3
Carlotta 2
10 braccialetti perline 3
4 bilia vetro 1
Federico 3
10 soldatino stagno 2
10 bilia vetro 3
1 cane pezza 5

```

nel flusso d'ingresso, il programma emette

```

Acquisto di: soldatino di stagno, numero: 11, per un costo di: 23
Federico 10
Massimo 1

```

nel flusso d'uscita.

## Soluzione

### Il giocattolo

La classe più elementare è il giocattolo, potrebbe essere anche un `record`, ma seguendo un approccio più convenzionale è sufficiente una classe immutabile con due attributi il cui invariante (del tutto ovvio) può essere controllato in costruzione:

```

public final String nome, materiale;

public Giocattolo(final String nome, final String materiale) {
    this.nome = Objects.requireNonNull(nome);
    this.materiale = Objects.requireNonNull(materiale);
    if (nome.isEmpty() || materiale.isEmpty()) throw new IllegalArgumentException();
}

```

[\[sorgente\]](#)

Unica cosa degna di nota è la sovrascrittura dei metodi di `Object`:

```

@Override
public int hashCode() {
    return Objects.hash(nome, materiale);
}

@Override
public boolean equals(Object obj) {
    if (!(obj instanceof Giocattolo)) return false;
    final Giocattolo tmp = (Giocattolo) obj;
    return tmp.nome.equals(nome) && tmp.materiale.equals(materiale);
}

```

[\[sorgente\]](#)

In particolare il codice evidenziato mostra l'uso del metodo `hash` di `Objects` che consente di calcolare l'hashcode a partire da un elenco di attributi.

### L'inventario

Poco più complicato è l'*inventario*, per tener traccia del numero di giocattoli è sufficiente una `Map<Giocattolo, Integer>` il cui invariante è che non contenga chiavi o valori nulli e contenga solo numeri positivi:

```

private final Map<Giocattolo, Integer> inventario = new HashMap<>();

```

[\[sorgente\]](#)

I metodi *mutazionali* devono prestare attenzione a mantenere l'invariante. Iniziamo con l'aggiunta di giocattoli: per prima cosa è necessario accertarsi che non ce ne siano già del tipo indicato (in quel caso, il numero specificato deve aggiungersi a quello già presente nella mappa), viceversa è sufficiente aggiungere una nuova *entry*:

```
public int aggiungi(final int num, final Giocattolo giocattolo) {
    Objects.requireNonNull(giocattolo);
    if (num <= 0) throw new IllegalArgumentException("Il numero deve essere positivo");
    int totale = num;
    if (inventario.containsKey(giocattolo)) totale += inventario.get(giocattolo);
    inventario.put(giocattolo, totale);
    return totale;
}

public int aggiungi(final Giocattolo giocattolo) {
    return aggiungi(1, giocattolo);
}
```

[\[sorgente\]](#)

Si osservi che il metodo per aggiungere un singolo giocattolo delega al metodo più generale, fare il contrario sarebbe stato poco conveniente perché nel metodo generale sarebbe stato necessario effettuare un ciclo per aggiungere un giocattolo alla volta (usando il metodo di aggiunta singola).

In modo del tutto simmetrico all'aggiunta, il metodo per rimuovere un certo numero di giocattoli deve verificare che il numero rimanente di giocattoli non sia negativo (caso in cui viene sollevata una eccezione, ma lasciato immutato lo stato dell'inventario) e, nel caso sia nullo, eliminare la chiave dalla mappa (come evidenziato nel codice):

```
public int rimuovi(final int num, final Giocattolo giocattolo) {
    Objects.requireNonNull(giocattolo);
    if (num <= 0) throw new IllegalArgumentException("Il numero deve essere positivo");
    if (!inventario.containsKey(giocattolo))
        throw new NoSuchElementException("Giocattolo non presente: " + giocattolo);
    final int totale = inventario.get(giocattolo) - num;
    if (totale < 0)
        throw new IllegalArgumentException("Non ci sono abbastanza giocattoli: " +
            giocattolo);
    if (totale == 0) inventario.remove(giocattolo);
    else inventario.put(giocattolo, totale);
    return totale;
}
```

[\[sorgente\]](#)

I metodi *osservazionali* sono più elementari. Conoscere la quantità di giocattoli di un certo tipo è banale:

```
public int quantità(final Giocattolo giocattolo) {
    Objects.requireNonNull(giocattolo);
    if (!inventario.containsKey(giocattolo)) return 0;
    return inventario.get(giocattolo);
}
```

[\[sorgente\]](#)

Unica cosa degna di nota è la decisione di restituire 0 nel caso l'inventario non comprenda il giocattolo (senza sollevare eccezione); questa scelta permette di evitare l'aggiunta di un metodo per sapere esplicitamente se un giocattolo sia presente o meno nell'inventario.

Per finire, appare comodo rendere la classe un *iterabile* sui giocattoli, in modo che (con l'ausilio del metodo precedente) si possa conoscerne completamente lo stato.

```
@Override
public Iterator<Giocattolo> iterator() {
    final List<Giocattolo> giocattoli = new ArrayList<>(inventario.keySet());
    Collections.sort(
        giocattoli,
        new Comparator<Giocattolo>() {
            @Override
            public int compare(Giocattolo o1, Giocattolo o2) {
                return o1.toString().compareTo(o2.toString());
            }
        });
    return giocattoli.iterator();
}
```

[\[sorgente\]](#)

Si noti l'uso della classe anonima che implementa l'interfaccia `Comparator<Giocattolo>` utile a consentire una iterazione in ordine lessicografico (della rappresentazione testuale dei giocattoli).

## I listini

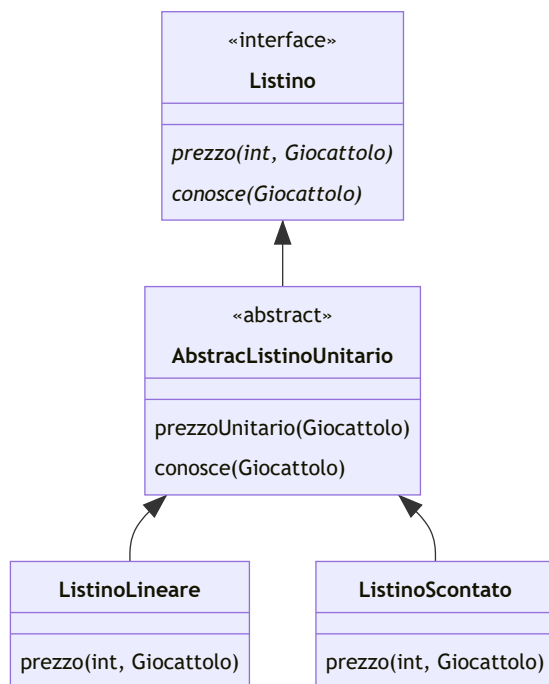
Stando alla traccia, il *listino* è una delle entità a cui è necessario provvedere la maggior variabilità possibile di comportamenti. Per questa ragione, è utile definire una interfaccia:

```
public interface Listino {
    public boolean conosce(final Giocattolo giocattolo);
    public int prezzo(final int num, final Giocattolo giocattolo);
}
```

[\[sorgente\]](#)

Il primo metodo è necessario perché il secondo solleverà eccezione se interrogato su giocattoli di cui non conosce il prezzo: in un inventario può aver senso segnalare l'assenza di un giocattolo restituendo 0 alla domanda sulla sua numerosità, viceversa non è sensato dire che una cosa ha prezzo 0 se non è compresa nel listino!

A questo punto, i due comportamenti descritti nella traccia dipendono entrambi dalla conoscenza del prezzo unitario, ragion per cui appare sensato interporre una classe astratta che offra questa competenza alle due classi concrete che realizzino i comportamenti descritti.



La classe astratta deve solo tener traccia (in una mappa tra giocattoli e interi) del prezzo unitario, l'invariante di tale rappresentazione (oltre all'ovvia richiesta che l'attributo non sia e non contenga `null`) è che i prezzi siano tutti positivi; la classe può essere immutabile e quindi l'invariante sarà controllato solo in costruzione (codice evidenziato):

```
private final Map<Giocattolo, Integer> prezzoUnitario;

public AbstractListinoUnitario(final Map<Giocattolo, Integer> prezzoUnitario) {
    this.prezzoUnitario = new HashMap<>();
    for (Map.Entry<Giocattolo, Integer> e : prezzoUnitario.entrySet()) {
        final Giocattolo giocattolo = Objects.requireNonNull(e.getKey());
        final Integer num = Objects.requireNonNull(e.getValue());
        if (num <= 0)
            throw new IllegalArgumentException("Il prezzp di " + giocattolo + " deve essere positivo");
        this.prezzoUnitario.put(giocattolo, num);
    }
}
```

[\[sorgente\]](#)

La rappresentazione scelta consente di sovrascrivere (in modo da renderlo concreto) il metodo `conosce` prescritto dall'interfaccia in modo molto elementare:

```
@Override
public boolean conosce(final Giocattolo giocattolo) {
    return prezzoUnitario.containsKey(Objects.requireNonNull(giocattolo));
}
```

[\[sorgente\]](#)

Similmente, il suo unico metodo osservazionale consente di tenere la sua rappresentazione completamente isolata da quella delle sottoclassi:

```
public int prezzoUnitario(final Giocattolo giocattolo) {
    Objects.requireNonNull(giocattolo);
    final Integer prezzo = prezzoUnitario.get(giocattolo);
    if (prezzo == null) throw new NoSuchElementException("Giocattolo non trovato: " +
        giocattolo);
    return prezzo;
}
```

[\[sorgente\]](#)

Alle sottoclassi concrete, a questo punto, resta solo l'onere di rendere concreto il metodo `prezzo` (che è l'unico metodo che resta astratto, nella classe astratta); nel caso del prezzo lineare, questo può essere fatto senza memorizzare ulteriore stato:

```
@Override
public int prezzo(final int num, final Giocattolo giocattolo) {
    if (num <= 0) throw new IllegalArgumentException("Il numero deve essere positivo");
    return prezzoUnitario(giocattolo) * num;
}
```

[\[sorgente\]](#)

Nel caso dello sconto, è necessario tener traccia di soglia e percentuale (lo stato della classe) che è immutabile e il cui banale invariante può essere controllato solo in costruzione:

```
private final int soglia, sconto;

public ListinoScontato(
    final Map<Giocattolo, Integer> prezzoUnitario, final int soglia, final int sconto) {
    super(prezzoUnitario);
    if (soglia <= 0) throw new IllegalArgumentException("La soglia deve essere positiva.");
    this.soglia = soglia;
    if (sconto < 1 || sconto > 100)
        throw new IllegalArgumentException("Lo sconto dev'essere compreso tra 1 e 100.");
    this.sconto = sconto;
}
```

[\[sorgente\]](#)

Date le due informazioni di cui sopra, l'implementazione del prezzo è semplice (la parte evidenziata nel codice segue dalla definizione nella traccia):

```
@Override
public int prezzo(int num, Giocattolo giocattolo) {
    if (num <= 0) throw new IllegalArgumentException("Il numero deve essere positivo");
    final int prezzoUnitario = prezzoUnitario(giocattolo);
    return num < soglia
        ? prezzoUnitario * num
        : soglia * prezzoUnitario
            + (int) ((num - soglia) * prezzoUnitario * (100 - sconto)) / 100.0;
}
```

[\[sorgente\]](#)

## La bancarella

La bancarella può essere facilmente ottenuta per *composizione* delle classi sviluppate sin qui; essa conterrà un inventario ed un listino (a cui delegherà il compito di rispondere a domande sui giochi disponibili e sul loro prezzo).

Oltre agli invarianti banali (concernenti la nullità, per così dire), l'unica accortezza è verificare che il listino contenga i prezzi di tutti i giocattoli

nell'inventario; dal momento che durante la vita della bancarella l'inventario può solo essere ridotto e che il listino è immutabile, tale controllo può essere fatto in costruzione (codice evidenziato):

```
public final String proprietario;

private final Listino listino;

private final Inventario inventario;

public Bancarella(final String proprietario, final Inventario inventario, final Listino listino) {
    this.proprietario = Objects.requireNonNull(proprietario);
    if (proprietario.isEmpty())
        throw new IllegalArgumentException("Il proprietario non deve essere vuoto.");
    this.listino = Objects.requireNonNull(listino);
    this.inventario = Objects.requireNonNull(inventario);
    for (final Giocattolo g : inventario)
        if (!listino.conosce(g))
            throw new IllegalArgumentException("Il listino manca del prezzo per: " + g);
}
```

[\[sorgente\]](#)

L'unico metodo mutazionale è quello che esegue una vendita (che di fatto comporta solo la riduzione del numero di beni in inventario):

```
public int vende(final int num, final Giocattolo giocattolo) {
    return inventario.rimuovi(num, giocattolo);
}
```

[\[sorgente\]](#)

I metodi osservazionali `quantità` e `prezzo` sono di banale implementazione (in quanto *delegati* all'inventario e al listino):

```
public int quantità(final Giocattolo giocattolo) {
    return inventario.quantità(giocattolo);
}

public int prezzo(final int num, final Giocattolo giocattolo) {
    return listino.prezzo(num, giocattolo);
}

@Override
public Iterator<Giocattolo> iterator() {
    return inventario.iterator();
}
```

[\[sorgente\]](#)

In aggiunta, la classe è resa un `Iterable<Giocattolo>` per consentire una ispezione completa del suo stato (anche in questo caso, attraverso una delega all'inventario).

Per concludere, siccome sarà comodo usare le bancarelle come chiavi delle mappe o come membri degli insiemi (nelle *collections*), sono stati sovrascritti i metodi `equals` e `hashCode` considerando uguali bancarelle col medesimo proprietario (indipendentemente dall'inventario e dal listino).

```
@Override
public boolean equals(Object other) {
    if (!(other instanceof Bancarella)) return false;
    return ((Bancarella) other).proprietario.equals(proprietario);
}

@Override
public int hashCode() {
    return proprietario.hashCode();
}
```

[\[sorgente\]](#)

## L'acquisto

La descrizione dell'acquisto di una determinata quantità di un giocattolo è costituita dall'informazione di quanti giocattoli di quel tipo sono stati acquistati per ciascuna bancarella e del prezzo totale pagato.



```

public final Giocattolo giocattolo;

private final Map<Bancarella, Integer> descrizione;

private int prezzo = 0, quantità = 0;

```

[\[sorgente\]](#)

La rappresentazione di tale informazioni può essere contenuta in una mappa dalle bancarelle agli interi e da due valori che memorizzino il prezzo e la quantità totali; l'invariante prescrive che tali valori siano tenuti aggiornati in dipendenza del contenuto della mappa. Poiché la classe ha un costruttore nullo, l'invariante può essere controllato nell'unico metodo mutazionale (codice evidenziato):

```

public void aggiungi(final int num, final Bancarella bancarella) {
    if (num <= 0) throw new IllegalArgumentException("Il numero deve essere positivo");
    Objects.requireNonNull(bancarella);
    if (descrizione.containsKey(bancarella))
        throw new IllegalArgumentException("La bancarella è già elencata nell'acquisto.");
    prezzo += bancarella.prezzo(num, giocattolo);
    quantità += num;
    descrizione.put(bancarella, num);
}

```

[\[sorgente\]](#)

In particolare, si osservi che per semplicità ogni bancarella può essere aggiunta una sola volta (altrimenti la mappa andrebbe aggiornata, come avviene ad esempio nel metodo `aggiungi` di `Inventario`).

I metodi osservazionali sono in parte banali:

```

public int prezzo() {
    return prezzo;
}

public int quantità() {
    return quantità;
}

```

[\[sorgente\]](#)

Inoltre, per consentire una piena conoscenza dello stato dell'acquisto, esso è reso un `Iterable<Bancarella>` e il metodo `quantità` consente di sapere, per ciascuna delle bancarelle restituite dall'iteratore, quanti giocattoli siano acquistati da essa:

```

public int quantità(final Bancarella bancarella) {
    Objects.requireNonNull(bancarella);
    if (!descrizione.containsKey(bancarella))
        throw new NoSuchElementException("L'acquisto non riguarda la bancarella specificata.");
    return descrizione.get(bancarella);
}

@Override
public Iterator<Bancarella> iterator() {
    Set<Bancarella> bancarelle = Collections.unmodifiableSet(descrizione.keySet());
    return bancarelle.iterator();
}

```

[\[sorgente\]](#)

Si osservi che l'iteratore «protegge» con `Collections.unmodifiableSet` le chiavi della mappa prima di restituirne un iteratore (che potrebbe mutare la mappa grazie al metodo `remove`).

## I compratori

La situazione del compratore richiede una variabilità di comportamenti (rispetto al modo di effettuare l'acquisto) comparabile a quella del listino. Tutti i compratori però non possono prescindere dalla conoscenza di un insieme di bancarelle da cui acquistare; dovendo condividere dello stato, appare più ragionevole mettere a capo della loro sotto-gerarchia una classe astratta.

```
protected final Set<Bancarella> bancarelle;

public AbstractCompratore(final Set<Bancarella> bancarelle) {
    Objects.requireNonNull(bancarelle);
    if (bancarelle.isEmpty())
        throw new IllegalArgumentException("Il mercatino deve contenere almeno una bancarella");
    this.bancarelle = Set.copyOf(bancarelle);
}
```

[\[sorgente\]](#)

Riguardo alla rappresentazione essa è `protected` in modo che le sottoclassi possano accedervi, ma è dichiarata `final` e l'insieme è immutabile (grazie all'uso di `Set.copyOf`, vedi codice evidenziato). Pertanto l'invariante di questa rappresentazione è controllato in costruzione e non potrà mai essere alterato dalle sottoclassi.

Resterà astratto il metodo che descrive la strategia di acquisto:

```
public abstract Acquisto compra(final int num, final Giocattolo giocattolo);
```

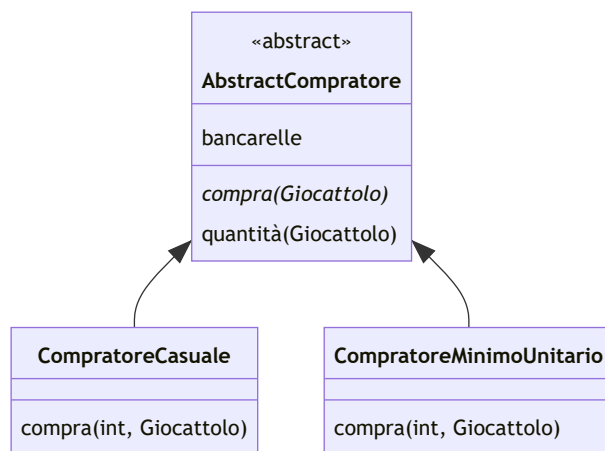
[\[sorgente\]](#)

mentre può essere comodo implementare un metodo che, dato l'insieme di bancarelle, calcoli la quantità totale di giocattoli di un dato tipo presenti nei loro inventari:

```
public int quantità(final Giocattolo giocattolo) {
    int quantità = 0;
    for (final Bancarella b : bancarelle) quantità += b.quantità(giocattolo);
    return quantità;
}
```

[\[sorgente\]](#)

Supponendo quindi di limitarci ai casi descritti nella traccia, potremmo implementare la seguente gerarchia:



Alle sottoclassi concrete non resta che implementare il metodo `compra` sulla scorta dello stato condiviso con la superclasse (l'insieme delle bancarelle).

Il caso più elementare è l'acquisto «a caso», ottenuto disponendo le bancarelle in ordine casuale (codice evidenziato) e quindi procedendo a comprare quanto più possibile da ciascuna bancarella:

```

@Override
public Acquisto compra(int num, Giocattolo giocattolo) {
    Objects.requireNonNull(giocattolo);
    if (num <= 0) throw new IllegalArgumentException("Il numero deve essere positivo");
    if (quantità(giocattolo) < num)
        throw new IllegalArgumentException("Non ci sono abbastanza: " + giocattolo);
    final Acquisto acquisto = new Acquisto(giocattolo);
    int rimanenti = num;
    final List<Bancarella> aCaso = new ArrayList<>(bancarelle);
    Collections.shuffle(aCaso, rng);
    for (final Bancarella b : aCaso) {
        if (rimanenti == 0) break;
        final int daComprare = Math.min(b.quantità(giocattolo), rimanenti);
        if (daComprare == 0) continue;
        b.vende(daComprare, giocattolo);
        acquisto.aggiungi(daComprare, b);
        rimanenti -= daComprare;
    }
    return acquisto;
}

```

[\[sorgente\]](#)


Un po' più complessa l'altra strategia. Finché restano giocattoli da comprare, si sceglie di volta in volta la bancarella che ha il minor prezzo unitario (comprando da essa tutti i giocattoli possibili):

```

@Override
public Acquisto compra(int num, Giocattolo giocattolo) {
    Objects.requireNonNull(giocattolo);
    if (num <= 0) throw new IllegalArgumentException("Il numero deve essere positivo");
    if (quantità(giocattolo) < num)
        throw new IllegalArgumentException("Non ci sono abbastanza: " + giocattolo);
    final Acquisto acquisto = new Acquisto(giocattolo);
    int rimanenti = num;
    while (rimanenti > 0) {
        int daComprare, minUnitario = Integer.MAX_VALUE;
        Bancarella min = null;
        for (final Bancarella b : bancarelle) {
            daComprare = Math.min(b.quantità(giocattolo), rimanenti);
            if (daComprare == 0) continue;
            int unitario = b.prezzo(daComprare, giocattolo) / daComprare;
            if (unitario < minUnitario) {

```

---

 Stars  16k
  DOI [10.5281/zenodo.5831781](https://doi.org/10.5281/zenodo.5831781)
 License  GPL v3
  License  CC BY-SA 4.0