Azzolini Riccardo 2019-09-24

Classi e oggetti: motivazioni

1 Codice di esempio

Per esporre le motivazioni che hanno portato alla programmazione a oggetti, consideriamo come esempio una possibile implementazione di un insieme di interi,

```
public class InsiemeDiInteri {
    public static final int CAPACITA = 10;
    public int n = 0;
    public int elenco[] = new int[CAPACITA];
}
```

e alcune classi che lo utilizzano (le quali appartengono ipoteticamente ad altre applicazioni).

```
public class GestioneTarghe {
    public InsiemeDiInteri insiemeTarghe;
    // ...
    public int cercaTarga(int numeroTarga) {
        int i = 0;
        while (i < insiemeTarghe.n
               && numeroTarga != insiemeTarghe.elenco[i]) {
            i++;
        }
        if (numeroTarga != insiemeTarghe.elenco[i]) {
            return -1;
        } else {
            return i;
        }
    }
    // ...
}

public class GestioneMatricole {
    public InsiemeDiInteri insiemeMatricole;
    // ...
    public int leggiMatricola() {
        int numeroLetto;
```

```
        // leggi numero dall'esterno...
        return numeroLetto;
    }
    public int cercaMatricola(int numeroMatricola) {
        // implementazione analoga a GestioneTarghe.cercaTarga
    }
    public void aggiungiMatricola() {
        int numero = leggiMatricola();
        int posizione = cercaMatricola(numero);
        if (posizione == -1) {
            insiemeMatricole.elenco[insiemeMatricole.n] = numero;
            insiemeMatricole.n++;
        }
    }
    // ...
}

public class GestioneContiCorrenti {
    public InsiemeDiInteri insiemeConti;
    // ...
    public int cercaConto(int numeroConto) {
        // implementazione analoga a GestioneTarghe.cercaTarga
    }
    public void eliminaConto(int numeroConto) {
        int posizione = cercaConto(numeroConto);
        if (posizione != -1) {
            insiemeConti.elenco[posizione] = 0;
        }
    }
    // ...
}
```

2 Problema: modificabilità del software

In quest'implementazione di `InsiemeDiInteri`, le variabili utilizzate per la gestione dei dati sono dichiarate `public`, quindi le applicazioni che usano tale insieme vengono implementate in base alla sua rappresentazione.

Se tale rappresentazione viene cambiata (ad esempio, usando una lista concatenata invece di un array), pur mantenendo il significato inalterato,

```
public class NodoListaInteri {
    public int dato;
```

```
    public NodoListaInteri next;
    // ...
}

public class InsiemeDiInteri {
    public NodoListaInteri primoNodo;
    // ...
}
```

sono necessarie modifiche estese al codice delle altre applicazioni:

```
 public class GestioneTarghe {
     public InsiemeDiInteri insiemeTarghe;
     // ...
-    public int cercaTarga(int numeroTarga) {
-        int i = 0;
-        while (i < insiemeTarghe.n
-                && numeroTarga != insiemeTarghe.elenco[i]) {
-            i++;
-        }
-        if (numeroTarga != insiemeTarghe.elenco[i]) {
-            return -1;
-        } else {
-            return i;
+    public NodoListaInteri cercaTarga(int numeroTarga) {
+        NodoListaInteri rifNodo = insiemeTarghe.primoNodo;
+        while (rifNodo != null && numeroTarga != rifNodo.dato) {
+            rifNodo = rifNodo.next;
         }
+        return rifNodo;
     }
     // ...
 }

 public class GestioneMatricole {
     public InsiemeDiInteri insiemeMatricole;
     // ...
     public int leggiMatricola() {
         int numeroLetto;
         // leggi numero dall'esterno...
         return numeroLetto;
     }
-    public int cercaMatricola(int numeroMatricola) {
+    public NodoListaInteri cercaMatricola(int numeroMatricola) {
```

```
        // implementazione analoga a GestioneTarghe.cercaTarga
    }
    public void aggiungiMatricola() {
        int numero = leggiMatricola();
-       int posizione = cercaMatricola(numero);
-       if (posizione == -1) {
-           insiemeMatricole.elenco[insiemeMatricole.n] = numero;
-           insiemeMatricole.n++;
+       NodoListaInteri rifNodo = cercaMatricola(numero);
+       if (rifNodo != null) {
+           NodoListaInteri nuovoNodo = new NodoListaInteri(numero);
+           nuovoNodo.next = insiemeMatricole.primoNodo;
+           insiemeMatricole.primoNodo = nuovoNodo;
        }
    }
    // ...
}

public class GestioneContiCorrenti {
    public InsiemeDiInteri insiemeConti;
    // ...
-   public int cercaConto(int numeroConto) {
+   public NodoListaInteri cercaConto(int numeroConto) {
        // implementazione analoga a GestioneTarghe.cercaTarga
    }
    public void eliminaConto(int numeroConto) {
-       int posizione = cercaConto(numeroConto);
-       if (posizione != -1) {
-           insiemeConti.elenco[posizione] = 0;
+       NodoListaInteri rifNodo = cercaConto(numeroConto);
+       if (rifNodo != null) {
+           NodoListaInteri prec;
+           // cerca il nodo precedente
+           prec.next = rifNodo.next;
        }
    }
    // ...
}
```

In particolare, vanno modificate *tutte* e *solo* le istruzioni che facevano riferimento alla vecchia struttura dati. Questo intervento richiede tempo e sforzo che sono decisamente mal spesi, e inoltre può introdurre degli errori.

3 Problema: integrità dei dati

Se più funzioni possono accedere ai dati liberamente, aumentano le possibilità di avere problemi di correttezza nell'*uso* e nella *modifica* delle informazioni.

Ad esempio, considerando la rappresentazione basata su array di `InsiemeDiInteri`,

```
public class InsiemeDiInteri {
    public static final int CAPACITA = 10;
    public int n = 0;
    public int elenco[] = new int[CAPACITA];
}
```

per eliminare un dato sarebbe necessario sfruttare l'*indice di riempimento* `n`,

```
InsiemeDiInteri insieme;
int posizione;
// ...
insieme.n--;
insieme.elenco[posizione] = insieme.elenco[insieme.n];
```

ma nella classe `GestioneContiCorrenti` viene invece usata una convenzione diversa, e *non compatibile*, che prevede di sostituire il dato eliminato con uno `0`:

```
public class GestioneContiCorrenti {
    public InsiemeDiInteri insiemeConti;
    // ...
    public void eliminaConto(int numeroConto) {
        int posizione = cercaConto(numeroConto);
        if (posizione != -1) {
            insiemeConti.elenco[posizione] = 0;
        }
    }
    // ...
}
```

Di conseguenza, eventuali altri metodi che operano su `insiemeConti` potrebbero considerare i valori `0` come conti validi.

4 Problema: riuso del software

Nell'implementazione delle classi che usano `InsiemeDiInteri`, lo stesso codice per la ricerca è stato scritto tre volte (`cercaTarga`, `cercaMatricola` e `cercaConto`). Supponendo che tale codice sia stato copiato e incollato, per evitare di doverlo riscrivere da

zero ogni volta, rimane comunque un problema: se esso contiene errori, questi dovranno essere corretti manualmente in ciascuno dei tre metodi.

La soluzione è definire `InsiemeDiInteri` come un **tipo di dato astratto**. Esso:

- è dotato di metodi che portano sempre agli stessi risultati, indipendentemente dalla rappresentazione scelta;
- può essere riutilizzato da molti programmi, senza bisogno di modificarlo.

5 Soluzione

Per risolvere questi tre problemi, è necessario separare:

- il **contenuto**, cioè l'insieme dei dati a cui si vuole accedere (con le relative politiche di gestione, che dipendono dalla rappresentazione);
- l'**interfaccia**, cioè le modalità di accesso ai dati.

Così facendo, la rappresentazione dei dati e l'implementazione dei metodi si possono cambiare liberamente, a patto che l'interfaccia venga "congelata", cioè non più modificata, una volta che essa è stata inizialmente decisa.

Inoltre, grazie al riuso, si ottengono ulteriori benefici:

- maggiore affidabilità (perché le applicazioni possono usare classi già scritte e testate, invece di crearne delle nuove che potrebbero contenere errori);
- minore costo;
- minore tempo di sviluppo.

Un esempio di buona implementazione di `InsiemeDiInteri` è:

```
public class InsiemeDiInteri {
    private static final int CAPACITA = 10;
    private int n;
    private int elenco[] = new int[CAPACITA];

    private int ricerca(int numero) {
        if (eVuoto()) {
            return -1;
        }
        int i = 0;
        while (i < n - 1 && numero != elenco[i]) {
            i++;
        }
        if (numero != elenco[i]) {
```

```
            return -1;
        } else {
            return i;
        }
    }

    public InsiemeDiInteri() {
        n = 0;
    }

    public InsiemeDiInteri(InsiemeDiInteri altroInsieme) {
        copia(altroInsieme);
    }

    public void finalize() {
        n = 0;
    }

    public boolean eVuoto() {
        return n == 0;
    }

    public boolean ePieno() {
        return n == CAPACITA;
    }

    public int cardinalita() {
        return n;
    }

    public boolean appartiene(int numero) {
        return ricerca(numero) != -1;
    }

    public void inserimento(int numero) {
        if (!ePieno() && !appartiene(numero)) {
            elenco[n] = numero;
            n++;
        }
    }

    public void eliminazione(int numero) {
        int posizione = ricerca(numero);
        if (posizione != -1) {
```

```
            n--;
            elenco[posizione] = elenco[n];
        }
    }

    public boolean contiene(InsiemeDiInteri altroInsieme) {
        for (int i = 0; i < altroInsieme.n; i++) {
            if (!appartiene(altroInsieme.elenco[i])) {
                return false;
            }
        }
        return true;
    }

    public boolean equals(InsiemeDiInteri altroInsieme) {
        return contiene(altroInsieme) && altroInsieme.contiene(this);
    }

    public void svuota() {
        n = 0;
    }

    public void copia(InsiemeDiInteri altroInsieme) {
        n = 0;
        for (int i = 0; i < altroInsieme.n; i++) {
            elenco[n] = altroInsieme.elenco[i];
            n++;
        }
    }

    public void unione(InsiemeDiInteri altroInsieme) {
        for (int i = 0; i < altroInsieme.n; i++) {
            if (ePieno()) {
                return;
            }
            inserimento(altroInsieme.elenco[i]);
        }
    }

    public void differenza(InsiemeDiInteri altroInsieme) {
        for (int i = 0; i < altroInsieme.n; i++) {
            if (eVuoto()) {
                return;
            }
```

```
            eliminazione(altroInsieme.elenco[i]);
        }
    }

    public void intersezione(InsiemeDiInteri altroInsieme) {
        int i = 0;
        while (i < n) {
            if (!altroInsieme.appartiene(elenco[i])) {
                eliminazione(elenco[i]);
            } else {
                i++;
            }
        }
    }

    public String toString() {
        String risultato = "";
        for (int i = 0; i < n; i++) {
            risultato += elenco[i] + "\n";
        }
        return risultato;
    }
}
```

5.1 Osservazioni

- Le variabili sono private, in modo che sia possibile accedere ai dati solo attraverso l'interfaccia del tipo di dato astratto.
- Il metodo `ricerca` è privato, perché restituisce un numero (l'indice del valore cercato) che ha senso solo se rapportato a una struttura dati interna (l'array `elenco`). Infatti, se si cambiasse la rappresentazione dei dati, cambierebbe anche il tipo restituito da questo metodo. Ad esempio, per una lista verrebbe restituito un riferimento a un nodo, che, se ottenuto da metodi di altre classi, permetterebbe addirittura di accedere alla lista bypassando l'interfaccia di `InsiemeDiInteri`.
- Sono definiti due costruttori:
  - un costruttore *per default*, che crea un insieme vuoto;
  - un costruttore *per copia*, che permette di ottenere una copia di un insieme esistente.

In questo caso, i costruttori sono pubblici, ma non è detto che sia sempre così: restringere l'accesso ai costruttori è necessario se si vuole controllare come e quando vengono creati gli oggetti di una classe.

- Il metodo `finalize` è un *finalizzatore*: esso esprime le "ultime volontà" dell'oggetto, e viene eseguito automaticamente subito prima che quest'ultimo venga distrutto.
- L'esistenza del metodo `ePieno` non obbliga a usare rappresentazioni con capacità limitata. Infatti, se si sostituisse l'array con una struttura dati a capacità illimitata (quale ad esempio una lista), l'implementazione di questo metodo potrebbe semplicemente diventare:

  ```
  public boolean ePieno() {
      return false;
  }
  ```

- I metodi `inserimento`, `eliminazione`, `unione`, `differenza` e `intersezione` modificano l'insieme di partenza. In alternativa, si potrebbe scegliere di lasciarlo inalterato, e restituire invece un nuovo oggetto `InsiemeDiInteri` aggiornato. Questa seconda opzione è meno efficiente, perché è necessaria una copia dell'insieme di partenza, ma ha alcuni vantaggi:
  - i metodi non hanno *effetti collaterali*;
  - la semantica delle operazioni corrisponde a quella che hanno nella teoria degli insiemi (ad esempio, $A \cup B$ costruisce un nuovo insieme, senza modificare A).

  La scelta tra le due soluzioni dipende dalle esigenze dell'applicazione, ma, una volta selezionata una strategia, essa dovrebbe essere adottata per tutti i metodi simili:
  - se, per esempio, `inserimento` modificasse l'insieme di partenza, allora dovrebbe farlo anche `eliminazione`;
  - se `unione`, ad esempio, restituisse un nuovo insieme, allora dovrebbero farlo anche `differenza` e `intersezione`.
- Molti metodi della classe sono implementati sfruttando altri suoi metodi. Infatti, il riuso del codice deve essere fatto prima di tutto nella classe stessa, oltre che in altre classi.
- Non sono definiti metodi di lettura o di stampa. Infatti, una classe di questo genere *non* deve eseguire I/O: saranno invece invece le applicazioni che la utilizzano a fare da tramite tra l'utente e questa classe (ad esempio, nei metodi `main`).