Programmazione astratta

Case study sull'uso di tipi astratti: figura

Anno 2017/2018

Indice generale

1 Introduzione

In questo tutorial utilizzo un semplice esercizio di programmazione come mezzo per affrontare le basi della programmazione astratta; nella fattispecie: definire un tipo astratto per rappresentare una insieme di oggetti di tipo diverso.

A partire dalla definizione del problema, utilizzerò dapprima un approccio OO collaudato, mettendone in evidenza i limiti. Successivamente mostrerò come superarli utilizzando la programmazione per astrazioni.

1.1 Problema

> Si vuole calcolare l'area totale di un elenco di figure geometriche: quadrati, rettangoli, triangoli equilateri e triangoli rettangoli.

L'esercizio richiede alcune premesse:

- Le figure (i loro dati) vengono gestite in memoria.
- Ogni figura è definita dai valori necessari a determinarne l'area:
  - *quadrato*: lato;
  - *rettangolo*: base e altezza;
  - *triangolo equilatero*: lato;
  - *triangolo rettangolo*: i due cateti;
- Il calcolo dell'area totale è realizzato attraverso un metodo che elabora i dati delle figure; questo metodo rappresenta il "codice applicativo" del programma.

In sostanza, vogliamo poter scrivere codice come il seguente:

```
static double AreaTotaleFigure(<elenco figure>)
{
    double areaTotale = 0;
    <per ogni figura>
      areaTotale += <area figura>;

    return areaTotale;
}
```

Le parti in verde identificano i problemi da risolvere:

- Come gestire l'elenco delle figure?
- Come calcolare l'area di ogni figura?

2 Una soluzione OO “concreta”

Il problema fondamentale è quello di gestire i vari tipi di figura, poiché ognuna è caratterizzata da dati diversi (lato, base e altezza, cateti) e richiede un diverso tipo di calcolo per l’area.

2.1 Una soluzione onnicomprensiva: classe Figura

L’idea è semplice: si definisce una classe che implementa il concetto generale di *figura*. La classe sarà in grado di eseguire il calcolo dell’area in base al tipo di figura.

```
public enum TipoFigura
{
    Quadrato,
    Rettangolo,
    TriangoloEquilatero,
    TriangoloRettangolo
}
```

```
public class Figura
{
    static double Radice3 = Math.Sqrt(3);

    public Figura(TipoFigura tipo, params double[] dati)
    {
        //... codice di validazione, basato sul tipo della figura
        // e sulle misure specificate
        this.TipoFigura = tipo;
        this.Dati = dati;
    }

    public readonly TipoFigura TipoFigura;
    public readonly double[] Dati;

    public double Area()
    {
        switch (TipoFigura)
        {
            case TipoFigura.Quadrato: return Dati[0] * Dati[0];
            case TipoFigura.Rettangolo: return Dati[0] * Dati[1];
            case TipoFigura.TriangoloEquilatero: return Dati[0] * Dati[0] / 4 * Radice3;
            case TipoFigura.TriangoloRettangolo: return Dati[0] * Dati[1] / 2;
            default: throw new InvalidOperationException("Tipo figura non definito");
        }
    }
}
```

Nota bene: 1) per brevità, nel costruttore ho omesso il codice di validazione dei dati. 2) L’uso di un parametro `params` consente semplificare la creazione degli oggetti.

La realizzazione del codice applicativo diventa banale:

```
static void Main(string[] args)
{
    var figure = new List<Figura>();

    figure.Add( new Figura(TipoFigura.Quadrato, 10) );
    figure.Add( new Figura(TipoFigura.Rettangolo, 10, 5) );
    figure.Add( new Figura(TipoFigura.TriangoloEquilatero, 10) );
    figure.Add( new Figura(TipoFigura.TriangoloRettangolo, 10, 7) );

    Console.WriteLine("Area totale = {0}", AreaTotaleFigure(figure));
    Console.ReadKey();
}

static double AreaTotaleFigure(List<Figura> figure)
{
    double areaTotale = 0;
    foreach (var f in figure)
    {
        areaTotale += f.Area();
    }
    return areaTotale;
}
```

Questa soluzione è funzionale, ma fragile; la classe `Figura` presenta infatti diversi problemi.

2.1.1 Codice complicato e scarsamente leggibile

Il codice di `Figura` è soltanto apparentemente semplice. Innanzitutto, le misure sono memorizzate in modo "anonimo" nel vettore `Dati`. La corrispondenza tra le misure – lato, base, altezza, etc – e gli elementi del vettore emerge soltanto analizzando il codice.

Ad esempio, dall'istruzione:

```
figure.Add( new Figura(TipoFigura.Rettangolo, 10, 5) );
```

non è possibile stabilire quale sia la base e quale l'altezza. (In questo scenario non è rilevante, ma in altri potrebbe esserlo.)

Il codice di calcolo delle aree è quasi incomprensibile e soggetto ad errori[1]:

```
case TipoFigura.Quadrato: return Dati[0] * Dati[0];
case TipoFigura.Rettangolo: return Dati[0] * Dati[1];
case TipoFigura.TriangoloEquilatero: return Dati[0] * Dati[0] / 4 * Radice3;
case TipoFigura.TriangoloRettangolo: return Dati[0] * Dati[1] / 2;
```

Infine, non esiste un modo elegante per rendere accessibili le misure della figura rispettando i principi OO. Per brevità, ho scelto di rendere pubblico il vettore `Dati`; pur essendo il campo `readonly`, ciò non impedisce al codice esterno di modificare direttamente i dati dopo aver creato l'oggetto.

1 Non a caso, nella prima stesura ho commesso un errore nel calcolo della area del triangolo rettangolo, utilizzando **Dati[0]** anche per il secondo cateto.

2.1.2 Codice ripetuto

Il codice che dipende dal tipo di figura deve essere eseguito in uno `switch`. Precedentemente ho omesso il codice di validazione nel costruttore; la sua implementazione richiede un secondo `switch` del tutto simile a quello utilizzato in `Area()`:

```
public class Figura
{
    static double Radice3 = Math.Sqrt(3);
    public Figura(TipoFigura tipo, params double[] dati)
    {
        switch (tipo)
        {
            case TipoFigura.Quadrato:
                if (dati.Length != 1)
                    throw new ArgumentException("Il quadrato richiede il lato");
                break;

            case TipoFigura.Rettangolo:
                if (dati.Length != 2)
                    throw new ArgumentException("Il rettangolo richiede base e altezza");
                break;

             //... le altre figure

            default: throw new ArgumentException("Tipo figura non definito");
        }
        this.TipoFigura = tipo;
        this.Dati = dati;
    }
    ...
}
```

2.1.3 Nuovi tipi di figura → modifica del codice esistente

Considera l'ipotesi di dover gestire anche i cerchi. Per farlo occorre effettuare tre modifiche:

- Aggiungere l'identificatore `Cerchio` all'*enum* `TipoFigura`.
- Aggiungere il codice di validazione del cerchio nel costruttore.
- Aggiungere il calcolo della area del cerchio nel metodo `Area()`.

È un processo da ripetere per ogni tipo di figura, e ha delle conseguenze:

- Induce il rischio di introdurre un bug in codice prima funzionante.
- Aumenta la complessità della classe.

2.2 Conclusioni

La classe `Figura` presenta un codice complicato e di difficile manutenzione. Non rispetta il principio DRY (*Don't Repeat Yourself*). Soprattutto, viola il principio OCP (*Open Closed Principle*). Quest'ultimo afferma che si dovrebbero progettare soluzioni che siano facilmente estendibili senza la necessità di modificare il codice esistente, ma soltanto mediante l'aggiunta di nuove classi.

Ebbene, esiste una soluzione più semplice, elegante e che, soprattutto, rispetta i principi della programmazione OO.

3 Elaborare e implementare figure “astratte”

Considera le cose dal punto di vista del codice applicativo:

```
static double AreaTotaleFigure(List<Figura> figure)
{
    double areaTotale = 0;
    foreach (var f in figure)
    {
        areaTotale += f.Area();
    }
    return areaTotale;
}
```

Questo non dipende in alcuno modo del tipo delle figure elaborate; *il codice elabora un “concetto astratto” di figura*, ed è applicabile a qualsiasi figura geometrica, purché questa definisca un metodo che restituisce l’area.

3.1 Concetto di figura → tipo astratto IFigura

Un concetto si definisce astratto perché non definisce delle caratteristiche concrete. Ad esempio, il termine *autoveicolo* designa un concetto astratto, poiché con esso non ci si riferisce a un modello specifico, al tipo di motorizzazione, al numero di posti, etc. Stesso discorso, nell’ambito del problema che stiamo considerando, vale per il concetto di *figura*; con questo termine si designa qualsiasi tipo di figura, nello specifico: qualsiasi oggetto che definisce un procedimento per il calcolo dell’area.

In C# i concetti astratti possono essere rappresentati mediante i *tipi astratti*, o *interfacce*, i quali definiscono le operazioni che caratterizzano i concetti, ma senza specificarne l’implementazione.

Di seguito mostro l’interfaccia `IFigura`:

```
public interface IFigura
{
    double Area();
}
```

Il suffisso `I` è convenzionale, e ha lo scopo di identificare il tipo come un’interfaccia. Molto più importante è il fatto che `IFigura` si limita a definire l’intestazione di un metodo, senza fornire la sua implementazione.

3.2 Uso del tipo IFigura

Adesso è possibile rappresentare un elenco di figure mediante un `List<IFigura>`:

```
static double AreaTotaleFigure(List<IFigura> figure)
{
    double areaTotale = 0;
    foreach (var f in figure)
    {
```

```
        areaTotale += f.Area();
    }
    return areaTotale;
}
```

3.2.1 Creare le figure

Il tipo `IFigura` consente di scrivere del codice che elabori un elenco di figure senza dipendere da uno specifico tipo, ma, di per sé, non permette di creare nessuna figura concreta.

Considera un ipotetico codice che crea l'elenco e inserisce alcune figure:

```
var figure = new List<IFigura>();
figure.Add( <figura concreta> );
figure.Add( <figura concreta> );
...
```

Tra parentesi non possiamo certo specificare il tipo `IFigura`, poiché non rappresenta un oggetto concreto, non definisce cioè le caratteristiche di una specifica figura. Per ritornare alla metafora precedente: le automobili sono *autoveicoli*, ma qualsiasi auto che esce da uno stabilimento appartiene sempre a un modello specifico, che ne definisce concretamente tutte le caratteristiche.

3.3 Implementare IFigura: definire i tipi concreti di figura

Perché il programma possa elaborare quadrati, rettangoli, etc, occorre definire oggetti che rappresentino quadrati, rettangoli, etc, con tutte le loro caratteristiche. Inoltre, occorre che tali oggetti siano compatibili con il tipo astratto `IFigura`.

Di seguito mostro i tipi `Quadrato` e `Rettangolo`:

```
public class Quadrato : IFigura
{
    public Quadrato(double lato)
    {
        Lato = lato;
    }
    public readonly double Lato;
    public double Area()
    {
        return Lato * Lato;
    }
}
```

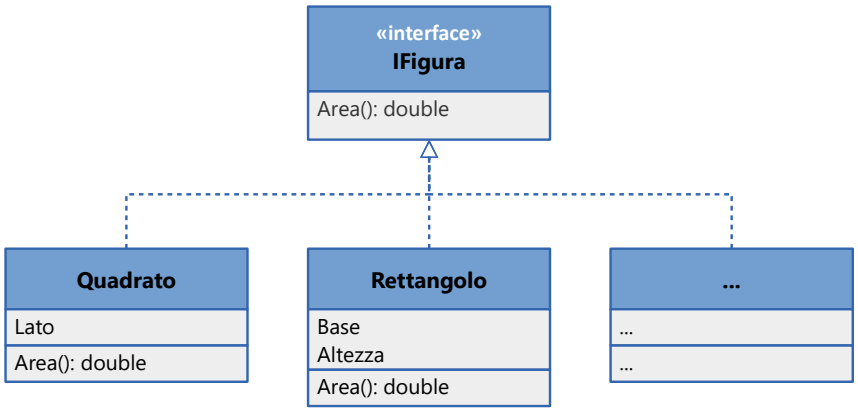
```
public class Rettangolo : IFigura
{
    public Rettangolo(double @base, double altezza)
    {
        Base = @base;
        Altezza = altezza;
    }
```

```
    public readonly double Base;
    public readonly double Altezza;

    public double Area()
    {
        return Base * Altezza;
    }
}
```

L'intestazione delle classi stabilisce che quadrati e rettangoli sono (anche) oggetti di tipo `IFigura`.

Formalmente: `Quadrato`, `Rettangolo`, etc, implementano l'interfaccia `IFigura`. In UML, questa relazione può essere rappresentata in questo modo[2]:



3.3.1 Creare figure concrete e elaborarle come oggetti IFigura

Non c'è niente di nuovo: basta creare oggetti del tipo appropriato e inserirli nella lista:

```
static void Main()
{
    var figure = new List<IFigura>();

    figure.Add( new Quadrato(10) );
    figure.Add( new Rettangolo(10, 5) );
    figure.Add( new TriangoloEquilatero(10) );
    figure.Add( new TriangoloRettangolo(10, 7) );

    Console.WriteLine("Area totale = {0}", AreaTotaleFigure(figure));
}
```

Nel metodo `AreaTotaleFigure()` non occorre cambiare niente.

```
static double AreaTotaleFigure(List<IFigura> figure)
{
    double areaTotale = 0;
    foreach (var f in figure)
    {
        areaTotale += f.Area();  //-> calcola l'area in base al tipo dell'oggetto!
    }
```

2 Il diagramma è incompleto e non rispetta completamente la notazione formale UML.

```
    return areaTotale;
}
```

La “magia” avviene nell’invocazione del metodo `Area()`. Il metodo effettivamente invocato dipende dal tipo effettivo referenziato da `f`:

- `figure[0]` - > `Quadrato`.
- `figure[1]` - > `Rettangolo`
- `figure[2]` - > `TriangoloEquilatero`
- `figure[3]` - > `TriangoloRettangolo`

4 Aggiungere nuove figure (estendere il programma)

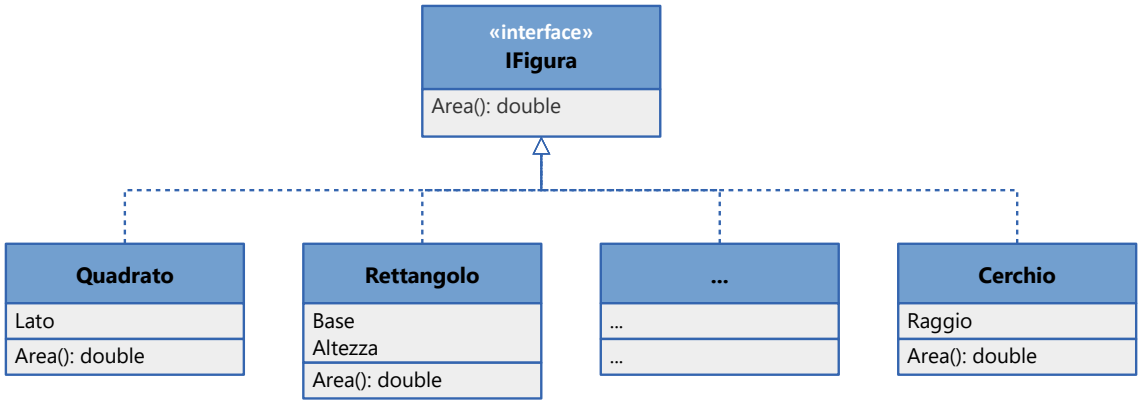
Supponi di voler estendere il funzionamento del programma, rendendolo in grado di elaborare anche i cerchi. Ebbene, non esiste alcun bisogno di modificare il codice esistente: basta aggiungere la classe `Cerchio`:

```
public class Cerchio : IFigura
{
    public Cerchio(double raggio)
    {
        Raggio = raggio;
    }

    public readonly double Raggio;

    public double Area()
    {
        return Raggio * Raggio * Math.PI;
    }
}
```

Con l'aggiunta di `Cerchio`, lo schema UML diventa il seguente:



Dopodiché, il metodo `AreaTotaleFigure()` potrà gestire anche i cerchi:

```
var figure = new List<IFigura>();

figure.Add( new Quadrato(10) );
figure.Add( new Rettangolo(10, 5) );
figure.Add( new TriangoloEquilatero(10) );
figure.Add( new TriangoloRettangolo(10, 7) );
figure.Add( new Cerchio(4) );

Console.WriteLine("Area totale = {0}", AreaTotaleFigure(figure));
```

Nota bene: non esiste alcun bisogno di modificare il metodo, perché *non dipende dal tipo effettivo delle figure, ma soltanto dal tipo astratto* `IFigura`.

5 Aggiungere funzioni al tipo IFigura

Anche se l'uso di `IFigura` consente di estendere il programma senza modificarlo (ma aggiungendo nuove classi), non significa che qualsiasi cambiamento ai requisiti del problema sia indolore.

Considera l'ipotesi di aggiungere il calcolo del perimetro totale. In sostanza vogliamo poter scrivere il seguente codice applicativo:

```
static double PerimetroTotaleFigure(List<IFigura> figure)
{
    double perimetroTotale = 0;
    foreach (var f in figure)
    {
        perimetroTotale += f.Perimetro();  // questo metodo non esiste!
    }
    return perimetroTotale;
}
```

Rappresenta una modifica al concetto astratto di figura, che ora include anche il calcolo del perimetro. Evidentemente, ciò significa dover modificare l'interfaccia `IFigura`.

5.1 Aggiornamento dell'interfaccia IFigura: metodo Perimetro()

L'aggiunta di operazioni a un'interfaccia ha delle conseguenze notevoli. La modifica dell'interfaccia in sé è banale:

```
public interface IFigura
{
    double Area();
    double Perimetro();
}
```

Ma questa semplice modifica rappresenta un *breaking change*, e cioè un cambiamento che rende le classi `Quadrato`, `Rettangolo`, etc, non più valide. Infatti, *un'interfaccia può essere considerata come un "contratto formale" con le classi che la implementano*. Se quel contratto cambia, le classi si devono necessariamente adeguare.

Occorre aggiungere il metodo `Perimetro()` a tutte le classi che implementano `IFigura`. Di seguito mostro il cambiamento delle classi `Quadrato` e `Rettangolo`:

```
public class Quadrato : IFigura
{
    ...
    public double Perimetro()
    {
        return Lato * 4;
    }
}
```

```
public class Rettangolo : IFigura
{
    ...
    public double Perimetro()
    {
        return (Base + Altezza) * 2;
    }
}
```

6 Riepilogo

Esistono scenari nei quali un procedimento deve elaborare degli oggetti sulla base di una o più caratteristiche comuni. È il caso delle figure geometriche: tutte le figure hanno un'area, ma ogni figura è diversa dalle altre, e definisce un diverso procedimento per il calcolo. Ci troviamo di fronte a un "concetto astratto", definito attraverso un'operazione che può avere differenti implementazioni.

Un tipo astratto rappresenta l'equivalente C# di un concetto astratto. `Ifigura` stabilisce che una figura è caratterizzata dall'area, ma non stabilisce in che modo l'area sia calcolata; questo dipende dal tipo concreto di figura, il quale viene rappresentato dalle classi `Quadrato`, `Rettangolo`, etc.

L'uso di tipi astratti permette di risolvere un problema fondamentale della programmazione:

> *realizzare procedimenti che non dipendono dalle operazioni effettivamente eseguite.*

L'affermazione può apparire contraddittoria, ma considera il metodo `AreaTotaleFigure()`:

```
static double AreaTotaleFigure(List<IFigura> figure)
{
    double areaTotale = 0;
    foreach (var f in figure)
    {
       areaTotale += f.Area(); // il metodo eseguito dipende dal tipo effettivo di figura!
    }
    return areaTotale;
}
```

Il metodo esegue un'operazione, `Area()`, senza "conoscere" il calcolo che sarà eseguito; infatti, il metodo effettivamente eseguito dipende dal tipo degli oggetti memorizzati nella lista.

In questo senso, emerge una netta distinzione tra *tipo statico* e *tipo run-time*:

- *tipo statico*: è il tipo delle variabili. Nell'esempio, il *tipo statico* degli elementi memorizzati in figure è `IFigura`.
- *tipo run-time*: è il tipo degli oggetti referenziati da tali variabili. (`Quadrato`, etc)

Il codice dipende formalmente dal *tipo statico*, ma durante l'esecuzione del programma, i metodi realmente eseguiti sono stabiliti dal *tipo run-time*.

La progettazione di tipi astratti (e dunque la loro implementazione mediante classi) deve essere effettuata con cura. Infatti, eventuali modifiche al tipo astratto costringono a modificare tutte le classi che lo implementano.

Si parla in questo caso di *breaking change*; Nell'esempio, un *breaking change* è stata l'aggiunta del metodo `Perimetro()` al tipo `IFigura`, cosa che mi ha costretto ad aggiornare le classi `Quadrato`, `Rettangolo`, etc.