Programmazione astratta

Case study sull'uso di tipi astratti: poligono

Anno 2017/2018

Indice generale

1 Introduzione

In questo tutorial parto da un esercizio simile a quello proposto in **Astrazioni – Figura** e metto a confronto l'uso di interfacce con quello di classi astratte. L'obiettivo è fare emergere le diversità tra i due tipi di astrazioni, partendo da un caso concreto.

1.1 Problema

Si vuole gestire un elenco di figure geometriche appartenenti alla categoria dei poligoni. Per ogni poligono si vogliono visualizzare i dati caratteristici: *nome*, *n° lati*, *area*, *perimetro*.

In sostanza, vogliamo poter scrivere codice come il seguente:

```
static void VisualizzaPoligoni(<elenco poligoni>)
{
    <per ogni poligono>
      Visualizzara <nome> <n° lati> <area> <perimetro>
}
```

2 Uso delle interfacce

L'obiettivo è quello di poter elaborare quadrati, rettangoli, etc, mediante un unico tipo di dato. Occorre pertanto definire un tipo astratto in grado di rappresentare qualsiasi poligono; questo sarà implementato da tipi concreti, a ognuno dei quali corrisponde a uno specifico poligono.

L'interfaccia `IFigura` utilizzata in **Astrazioni – Figura** è un buon punto di partenza; definisco dunque l'interfaccia `IPoligono`, che aggiunge le proprietà `Nome` e `NumeroLati`.

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

```
public interface IPoligono
{
    string Nome { get; }
    int NumeroLati { get; }
    double Area();
    double Perimetro();
}
```

Seguono le classi `Quadrato` e `TriangoloEquilatero`; le altre sono del tutto simili:

```
public class Quadrato : IPoligono
{

    public Quadrato(double lato)
    {
        Lato = lato;
    }
    public readonly double Lato;

    public string Nome
    {
        get { return "Quadrato"; }
    }

    public int NumeroLati
    {
        get { return 4; }
    }

    public double Area()
    {
        return Lato * Lato;
    }

    public double Perimetro()
    {
        return Lato * 4;
    }
}
```

```
public class TriangoloEquilatero : IPoligono
{
    static double Radice3 = Math.Sqrt(3);
    public TriangoloEquilatero(double lato)
    {
        Lato = lato;
    }
    public readonly double Lato;

    public string Nome
    {
        get { return "Triangolo ..."; }
    }

    public int NumeroLati
    {
        get { return 4; }
    }

    public double Area()
    {
        return Lato * Lato / 4 * Radice3;
    }

    public double Perimetro()
    {
        return Lato * 3;
    }
}
```

2.1 Creare ed elaborare i poligoni

Il codice è praticamente identico a quello utilizzato nel precedente tutorial:

```
static void Main()
{
    var poligoni = new List<IPoligono>();

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

    VisualizzaPoligoni(poligoni);
}
```

Posso dunque implementare il metodo `VisualizzaPoligoni()`:

```
static void VisualizzaPoligoni(List<IPoligono> poligoni)
{
    foreach (var p in poligoni)
    {
        Console.WriteLine($"Nome:{p.Nome}\nN° lati:{p.NumeroLati}\n...");
    }
}
```

2.2 Limite delle interfacce

Questa soluzione è perfettamente funzionale; ciò nonostante mette in evidenza un limite delle interfacce, le quali non sono in grado di catturare una relazione fondamentale che collega alcuni tipi tra loro: l'*ereditarietà*.

Considera le quattro caratteristiche comuni ai poligoni: *nome*, *n° lati*, *area* e *perimetro*. Le ultime due sono caratteristiche astratte: ogni poligono definisce un procedimento di calcolo distinto[1]. Ma *nome* e *n° lati* sono dei semplici valori, e non dipendono dal tipo di poligono; non a caso, se fossero gli unici due attributi da elaborare, non avremmo alcun bisogno di definire un tipo astratto: potremmo gestire il tutto mediante la semplice classe `Poligono`:

```
public class Poligono
{
    public Poligono(string nome, int numeroLati) { //...}

    public readonly string Nome;
    public readonly int NumeroLati;
}
```

Le interfacce non consentono di definire dei campi; dunque: ogni classe deve fornire la propria versione di `NumeroLati` e `Nome`, nonostante siano praticamente tutte uguali tra loro.

1 Nel contesto che sto considerando. I poligoni regolari utilizzano gli stessi procedimenti.

3 Ereditare le caratteristiche comuni: classi astratte

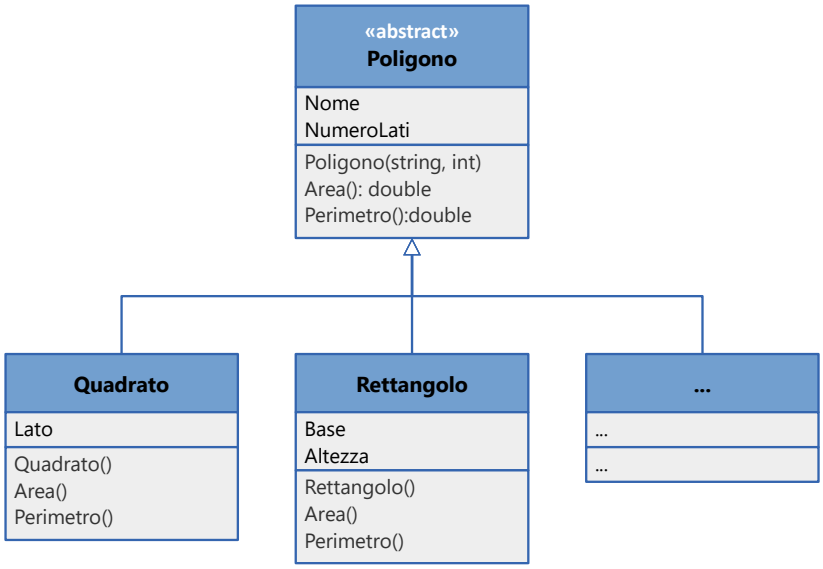
La classe `Poligono` rappresenta un buon punto di partenza come prototipo di poligono, ma non è sufficiente, perché ogni poligono definisce le proprie misure (lato, base e altezza, cateti) e i propri metodi di calcolo. Sarebbe utile poter definire contestualmente sia le caratteristiche concrete che quelle astratte, in modo che le classi `Quadrato`, `Rettangolo`, etc, possano ereditare le prime, fornire la propria versione delle seconde e aggiungere le caratteristiche specifiche di ogni poligono.

La soluzione consiste nel definire una classe astratta.

3.1 Classi astratte

Una classe astratta può essere vista come un "ibrido" tra un'interfaccia e una normale classe. Può definire campi e operazioni concrete, ma anche operazioni astratte. Diversamente dalle interfacce, consente di centralizzare le caratteristiche concrete comuni in una sola classe, facendo sì che le classi concrete possano "ereditarle".

Relativamente al nostro esempio, rendere `Poligono` astratta ci consente di realizzare il seguente schema:



Nota bene: le classi concrete definiscono i campi che dipendono dal tipo di poligono (lato, base, etc) e forniscono la propria versione (*ridefiniscono*) delle operazioni astratte; ma non definiscono `Nome` e `NumeroLati`, poiché ereditano questi campi dal tipo `Poligono`.

3.1.1 Classe astratta Poligono

Per designare una classe come astratta si usa la parola chiave `abstract`, sia per decorare la classe, che per decorare le singole operazioni astratte.

```
public abstract class Poligono
{
    public Poligono(string nome, int numeroLati)
    {
        Nome = nome;
```

```
        NumeroLati = numeroLati;
    }

    public readonly string Nome;
    public readonly int NumeroLati;

    public abstract double Area();
    public abstract double Perimetro();
}
```

Nota bene: i metodi `Area()` e `Perimetro()` non definiscono un corpo, poiché sono astratti.

3.1.2 Usare la classe Poligono

Il metodo `VisualizzaPoligoni` non cambia affatto, eccetto per il tipo degli elementi della lista :

```
static void VisualizzaPoligoni(List<Poligono> poligoni)
{
    foreach (var p in poligoni)
    {
        Console.WriteLine($"Nome:{p.Nome}\nN° lati:{p.NumLati}\n...");
    }
}
```

Creare oggetti di tipo Poligono

È impossibile, esattamente come è impossibile creare oggetti di tipo `IPoligono`.

```
var p = new Poligono("Quadrato", 4);  // errore!
```

È interessante notare che sarebbe proibito anche se `Poligono` definisse soltanto operazioni concrete. La parola `abstract` qualifica la classe come tipo astratto e dunque ne impedisce l'uso per la creazione di oggetti.

3.2 Derivare le classi astratte

Derivare una classe astratta è un'operazione simile a quella di *implementare* un'interfaccia, ma presenta alcune differenze, nella forma e nella sostanza. Dire che `Quadrato` deriva da `Poligono` significa affermare che:

- `Quadrato` *è un tipo* di `Poligono` e *deve* ridefinire tutte le operazioni astratte definite in `Poligono`.
- `Quadrato` viene convenzionalmente definita ***classe derivata***, (o *sottoclasse*) mentre `Poligono` viene definita ***classe base*** (o *superclasse*).
- `Quadrato` eredita tutti i campi e le operazioni concrete definite in `Poligono`, eccetto gli eventuali costruttori.

Segue la nuova versione della classe `Quadrato`:

```
public class Quadrato : Poligono
{
    public Quadrato(double lato): base("Quadrato", 4)  // -> invoca costruttore Poligono
    {
        Lato = lato;
    }

    public readonly string Nome;      //ereditato
    public readonly int NumeroLati;   //ereditato

    public readonly double Lato;

    public override double Area()  // ridefinisce Area()
    {
        return Lato * Lato;
    }

    public override double Perimetro() // ridefinisce Perimetro()
    {
        return Lato * 4;
    }
}
```

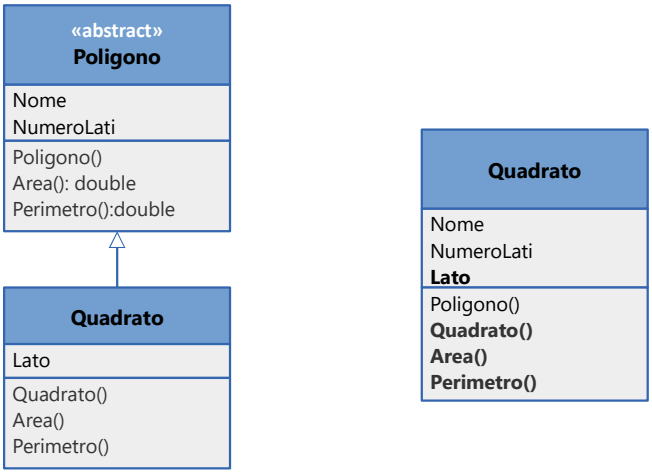
Alcune considerazioni:

- I metodi `Area()` e `Perimetro()` sono decorati con `override`; significa che ridefiniscono i metodi omologhi definiti in `Poligono`.
- Mediante la parola chiave `base` viene invocato il costruttore di `Poligono`.
- In grigio ho riportato i campi ereditati da `Poligono`, non perché debbano essere dichiarati, ma soltanto per puntualizzare che, a tutti gli effetti, questi campi fanno parte degli oggetti di tipo `Quadrato`.

4 Vincoli e implementazioni alternative

Derivare una classe è un'operazione più complessa rispetto a quella di implementare un'interfaccia; infatti, la classe derivata eredita l'implementazione della classe base.

Considera la relazione tra `Quadrato` e `Poligono`; di fatto è come se esistesse una classe che contiene sia i membri della prima che della seconda. Da un altro punto di vista, è come se l'implementazione di un oggetto fosse "spalmata" in due classi, con la differenza che l'accesso ai membri ereditati dalla classe base è sottoposto a dei vincoli.



Tutto ciò ha delle conseguenze, che dipendono dalle regole del linguaggio e dalle decisioni prese nell'implementare la classe base.

4.1 Chiamare il costruttore della classe base

Le regole del linguaggio stabiliscono che una classe deve avere un costruttore (in mancanza d'esso, il linguaggio definisce un costruttore vuoto), che sarà invocato nella creazione degli oggetti. In presenza di ereditarietà, i costruttori in gioco sono minimo due, uno per la classe base e uno per quella derivata.

`Poligono` definisce un costruttore con parametri; ciò pone un preciso vincolo sulle classi derivate, *che devono chiamare quel costruttore*.

```
public class Quadrato : Poligono
{
    public Quadrato(double lato)   //deve essere chiamato il costruttore base!
    {
        Lato = lato;
    }
    ...
}
```

4.1.1 Ordine di esecuzione dei costruttori

Durante la creazione di un oggetto viene eseguito prima il costruttore della classe base, poi quello della classe derivata. Ad esempio, la seguente istruzione:

```
var q = new Quadrato(3);
```

produce, nell'ordine, le seguenti assegnazioni:

```
q.Nome = "Quadrato";  -> costruttore Poligono
q.NumeroLati = 4;     -> costruttore Poligono
q.Lato = 3;           -> costruttore Quadrato
```

4.1.2 Assegnazione a campi readonly ereditati

Gli eventuali campi *readonly* ereditati *devono essere inizializzati nel costruttore base*. Ciò ha delle conseguenze, che possono produrre delle implementazioni incoerenti.

Considera l'ipotesi che `Poligono` definisca anche un costruttore vuoto.

```
public abstract class Poligono
{
    public Poligono(string nome, int numeroLati)
    {
        Nome = nome;
        NumeroLati = numeroLati;
    }

    public Poligono() { }
    ...
}
```

Ciò consente di realizzare classi derivate che non invocano il costruttore con parametri, poiché il linguaggio aggiunge automaticamente la chiamata al costruttore vuoto.

```
public class Quadrato : Poligono
{
    public Quadrato(double lato):base()   //OK, viene invocato Poligono()
    {
        Lato = lato;
    }
    ...
}
```

Così facendo, però, diventa possibile costruire dei poligoni che non specificano il nome e hanno zero lati.

```
var q = new Quadrato(3); //->Nome:null; NumeroLati:0
```

Si tratta ovviamente di un risultato incoerente, ma che non può essere corretto modificando le classi derivate:

```
public class Quadrato : Poligono
{
    public Quadrato(double lato):base()   //OK, viene invocato Poligono()
    {
        Nome = "Quadrato"   //impossibile inizializzare i campi readonly ereditati!
        NumeroLati = 4;
        Lato = lato;
    }
    ...
}
```

Questo esempio non riguarda l'uso dei campi *readonly*, ma intende dimostrare che le scelte di progettazione di una classe base hanno delle conseguenze sulle classi derivate.

In questo scenario, la definizione di un costruttore vuoto in `Poligono` è scorretta.

4.2 Imporre delle precondizioni

La possibilità, per le classi astratte, di fornire un'implementazione concreta, consente di imporre dei vincoli che le classi derivate sono costrette a rispettare. Ad esempio, possiamo garantire che ogni poligono abbia un nome e un numero di lati valido:

```
public abstract class Poligono
{
    public Poligono(string nome, int numeroLati)
    {
        if (string.IsNullOrEmpty(nome))
            throw new ArgumentException("Il nome del poligono non deve essere vuoto");

        if (numeroLati < 3)
            throw new ArgumentException("Un poligono deve avere almeno 3 lati");

        Nome = nome;
        NumeroLati = numeroLati;
    }
    ...
}
```

Con questa soluzione le classi derivate sono obbligate a inizializzare i due campi. Si tratta di un risultato impossibile da ottenere con l'uso di un'interfaccia.

4.3 Soluzioni alternative: uso delle proprietà

Di seguito esploro alcune soluzione alternative. Lo scopo non è quello di migliorare la classe (già valida), ma di mostrare che determinate scelte progettuali influenzano le classi derivate.

4.3.1 Proprietà readonly (prive di set accessor)

Non cambia niente rispetto all'uso di campi *readonly*:

```
public abstract class Poligono
{
    public Poligono(string nome, int numeroLati)  //necessario!
    {
        ...
    }

    public string Nome { get; }
    public int NumeroLati { get; }

    ...
}
```

4.3.2 Proprietà get-only (set accessor privato)

Si tratta di uno scenario interessante, poiché mostra che, nonostante i membri di `Poligono` siano ereditati da `Quadrato`, restano comunque soggetti al livello di accessibilità stabilito nella classe base.

```
public abstract class Poligono
{
    public Poligono(string nome, int numeroLati)  //necessario!
    {
        ...
    }

    public string Nome { get; private set; }      //non modificabili in Quadrato!
    public int NumeroLati { get; private set; }

   ...
}
```

Le due proprietà sono modificabili ovunque in `Poligono`, ma non in `Quadrato`; in sostanza è ancora necessario che `Poligono` definisca il costruttore con parametri.

4.3.3 Proprietà protette (set accessor protected)

Questo scenario sfrutta una funzionalità specificatamente progettata per favorire l'ereditarietà: la possibilità di rendere accessibile un membro alle classi derivate ma non al codice esterno.

```
public abstract class Poligono
{
    public Poligono(string nome, int numeroLati)  //non più obbligatorio!
    {
        ...
    }
    public Poligono() {}  //ha senso definire un costruttore vuoto
```

```
    public string Nome { get; protected set; }     //modificabili in Quadrato.
    public int NumeroLati { get; protected set;}

   ...
}
```

In sostanza, con questa modifica si consente alle classi derivate di impostare direttamente il nome e il numero dei lati:

```
public class Quadrato : Poligono
{
    public Quadrato(double lato)
    {
        Nome = "Quadrato";  //OK
        NumeroLati = 4;     //OK
        Lato = lato;
    }
    ...
}
```

Si tratta soltanto di un esempio; in questo scenario non esiste alcun motivo per lasciare alle classi derivate il compito di impostare direttamente `Nome` e `NumeroLati`, infatti, così facendo:

- Si duplica il codice.
- Non si sfrutta la possibilità di centralizzare (nella classe base) la logica di validazione dei due attributi.

5 Riepilogo

Classi astratte e interfacce permettono di implementare concetti astratti, e cioè oggetti o processi dei quali esistono, o possono esistere, più implementazioni. Tra i due costrutti ci sono differenze formali e sostanziali, anche se esistono scenari nei quali decidere di usare l'uno o altro può essere indifferente.

Diversamente dalle interfacce, le classi astratte consentono di definire sia campi che operazioni concrete; ciò è utile quando alcune caratteristiche del concetto astratto hanno un'implementazione comune, che può essere ereditata dai tipi derivati.

È il caso del concetto di *poligono*, caratterizzato da *nome*, *n° lati*, *area* e *perimetro*. Ebbene: non esiste un modo specifico per ogni poligono di rappresentare *nome* e il *n° lati*, sono sufficienti due campi, di tipo `string` e `int`. Diversamente: il calcolo di *area* e *perimetro* dipendono dal tipo di di poligono, sulla base di misure specifiche, lato, base e altezza, etc. In questo scenario, l'uso di una classe astratta consente di implementare in una sola volta *nome* e *n° lati*, facendo sì che le classi concrete ereditino questa implementazione.

La classe astratta consente di sfruttare il meccanismo dell'*ereditarietà* e dunque di riutilizzare lo stesso codice, dichiarativo o esecutivo, in più classi. L'ereditarietà, oltre a evitare ripetizioni di codice, consente di imporre dei vincoli "a monte", facendo sì che le classi derivate non siano costretti a implementarli (oppure semplicemente a ignorarli). La classe `Poligono` ne fornisce un esempio, definendo un costruttore con parametri che valida *nome* e il *n° lati*. Poiché è l'unico costruttore della classe, e poiché le classi derivate devono invocare un costruttore della classe base, la validazione dei due attributi è garantita.