Programmazione Fluente (Fluent Programming Style) in C#

Nell'ambito dell'ingegneria del software, il termine "Interfaccia Fluente" (coniato da Eric Evans e Martin Fowler) indica un modo di programmazione orientato agli oggetti che si basa sulla concatenazione delle chiamate ai metodi ("method chaining") con lo scopo di rendere il codice maggiormente leggibile, rendendolo più vicino ad espressioni in lingua naturale.

La tecnica denominata "Method Chaining" prevede l'esecuzione di metodi in cascata senza in modo da poter leggere il codice in modo "fluente" come se fosse una frase.

Tecnicamente, questo comporta che ciascun metodo ritorni un oggetto in modo che si possano concatenare le chiamate senza dover riscrivere l'oggetto chiamante, come in

x.f().g().h()

Si ottiene una maggiore fluidità nella lettura del codice a spese di una maggiore difficoltà nel debugging in quanto i debugger attuali non consentono di collocare punti di interruzione (breakpoint) in un punto preciso di una riga di codice.

Nel C# questa tecnica di programmazione fluente è ampiamente utilizzata dalla libreria LINQ per effettuare ricerche in una collezione di oggetti.

(Traduzione e adattamento da http://en.wikipedia.org/wiki/Fluent_interface)

Nell'articolo "Fluent interfaces and Method Chaining in C#" di Shivprasad Koirala, 21 Aug 2013 pubblicato in www.codeproject.com, l'autore mostra una tecnica basata sulla creazione di un "adapter", ovvero di una classe che incorpora l'oggetto di interesse e che funge da interfaccia fluente per lo stesso, implementando gli appositi metodi di accesso all'oggetto stesso.

L'approccio da me seguito, invece, semplifica l'implementazione di uno stile di programmazione fluente ed è basato sulla semplice riscrittura dei metodi di accesso alle proprietà dell'oggetto; in particolare si dovranno modificare soltanto i "setter method".

Per apprezzare meglio le caratteristiche di uno stile di programmazione fluente, viene ora proposto a titolo di esempio l'utilizzo di oggetti di classe Persona:

```
Persona q = Persona.New().diNome("gianni").diEtà(22).diProfessione("poliziotto")
              .conAmici("oscar", "alex").conAmico("osvaldo");

q.conAmico("beniamino").conAmico("tony");  // aggiunta di altri amici
// notare che non c'è l'obbligo di assegnare il risultato dell'espressione

q = q.conAmico("willy");  // aggiunta di un altro amico

// non è previsto un preciso ordine delle chiamate ai metodi
Persona p = Persona.New().diNome("maria").diProfessione("babysitter").diEtà(18);
```

```
// se si preferisce, si può anche creare l'oggetto nel modo tradizionale
// e poi utilizzare i suddetti metodi di assegnazione
var z = new Persona();
z.diNome("zoe").diEtà(10);

q.conPartner(p);  // assegnare o cambiare il partner

// esempio di scrittura dei dati
label1.Text = q.Nome + q.Età + q.Professione + q.NumeroAmici +
              String.Join(",", q.Amici) + q.Partner.Nome;
```

// Si ottiene

gianni22poliziotto6oscar,alex,osvaldo,beniamino,tony,willymaria

---

Le medesime operazioni scritte in modo tradizionale sarebbero state le seguenti:

```
Persona q = new Persona2() { Nome = "gianni", Età = 22, Professione = "poliziotto" };
q.AggiungiAmici("oscar", "alex");
q.AggiungiAmico("osvaldo");

q.AggiungiAmico("beniamino");  // aggiunta di altri amici
q.AggiungiAmico("tony");
q.AggiungiAmico("willy");

Persona p = new Persona2();
p.Nome = "maria";
p.Professione = "babysitter";
p.Età=18;

q.Partner = p;  // assegnazione o cambio del partner
```

Ora vediamo il codice della classe Persona (scritta con metodi Fluenti)

```
class Persona
{
    // versione fluente con method chaining
    // ci sono modifiche minime rispetto alla classe scritta in modo tradizionale

    // public properties (come al solito)
    public string Nome { get; set; }
    public int Età { get; set; }
    public string Professione { get; set; }
    public List<string> Amici { get; set; }
    public Persona Partner { get; set; }

    // proprietà calcolate (come al solito)
    public int NumeroAmici
    {
        get { return Amici.Count; }
    }

    // costruttore senza parametri
    public Persona()
```

```
    {
        // inizializzo la lista di amici
        Amici = new List<string>();
    }

    // costuttore static
    public static Persona New()
    {
        return new Persona();
    }

    // fluent setter methods
    // essi ritornano l'oggetto chiamante stesso (è di tipo Persona)
    // oltre a assegnare il valore alla proprietà di turno (come al solito)

    public Persona diNome(string unNome)
    { Nome = unNome; return this; }

    public Persona diEtà(int unEtà)
    { Età = unEtà; return this; }

    public Persona diProfessione(string unaProfessione)
    { Professione = unaProfessione; return this; }

    // altri metodi fluenti
    // per cambiare lo stato dell'oggetto stesso
    public Persona haCompleanno()
    { Età++; return this; }

    // uso di un numero variabile di argomenti
    public Persona conAmici(params string[] nomi)
    {
        foreach (string nome in nomi)
        { Amici.Add(nome); }
        return this;
    }

    public Persona conAmico(string unNome)
    {
        Amici.Add(unNome);
        return this;
    }

    public Persona conPartner(Person unPartner)
    { Partner = unPartner; return this; }

    // metodi ordinari
    public string Informazioni()
    { return Nome + Età + Professione + NumeroAmici; }

}
```

La classe Persona è un tipico esempio di una classe composta di alcune proprietà semplici e di una collezione di dati (stringhe), che potrebbe essere agevolmente sostituita da una collezione di oggetti.

Convenzioni sui Nomi

Nel suddetto esempio è stata seguita una particolare convenzione sui nomi dei metodi:

- I metodi usati per assegnare una proprietà dell'oggetto hanno il nome che inizia con la preposizione "di". In generale sono del tipo `diNomeProprietà(valore)`

- I metodi usati per assegnare una associazione (possesso) di altri oggetti hanno il nome che inizia con la preposizione "con". In generale sono del tipo `conOggetto(oggetto)`

Questa distinzione è fondata su questioni semantiche: essa ha lo scopo di esprimere il significato delle proprietà dell'oggetto e ha finalità MNEMONICHE.

E' preferibile evitare nomi basati su azioni/verbi e utilizzare piuttosto nomi che esprimono caratteristiche/proprietà dell'oggetto. Questo è un modo per supportare le intenzioni "dichiarative" di questo stile di programmazione.
L'aspetto attraente di questo stile di programmazione è la leggibilità del codice, che è legata alla scelta di queste convenzioni.

Si può anche esporre lo stesso metodo con nomi diversi per migliorare la fluidità espressiva delle istruzioni di programmazione. Fa così LINQ che prevede di scrivere

```
collezione.OrderBy(x=>x.Età).ThenBy(x=>Nome);
```

invece di

```
collezione.OrderBy(x=>x.Età).OrderBy(x=>Nome);
```

Per completezza ecco il codice dell'omologa classe denominata Persona2, scritta in modo tradizionale:

```
class Persona2  // versione tradizionale
{
    // public properties
    public string Nome { get; set; }
    public int Età { get; set; }
    public string Professione { get; set; }
    public List<string> Amici { get; set; }
    public Persona2 Partner { get; set; }
    // proprietà calcolate
    public int NumeroAmici
    { get { return Amici.Count; } }

    // costruttore senza parametri
    public Persona2()
    {
        // inizializzo la lista di amici
        Amici = new List<string>();
    }

    // metodi che cambiano lo stato dell'oggetto
```

```
    public void HaCompleanno()
    { Età++; }

    // uso di un numero variabile di argomenti
    public void AggiungiAmici(params string[] nomi)
    {
        foreach (string nome in nomi)
        { Amici.Add(nome); }
    }

    public void AggiungiAmico(stringe unNome)
    {
        Amici.Add(unNome);
    }
}
```

Questo stile di programmazione è diventato familiare ai programmatori grazie all'uso di LINQ e di alcune librerie di unit testing.

Con la conoscenza delle tecniche sottostanti l'implementazione di questo stile di programmazione abbiamo la possibilità di applicarlo alle nostre classi scritte in C#.