

Programmazione astratta III

Esempi di polimorfismo mediante l'uso e l'implementazione di tipi astratti predefiniti

Compatibilità: dotNET 2.0+

Anno 2017/2018

Indice generale

1	Introduzione.....	3
2	Formattazione in stringa: <code>object.ToString()</code>.....	4
2.1	Tipo <code>Object</code>	4
2.2	Funzionamento del metodo <code>string.Format()</code>	6
2.3	Rendere un tipo “Formattabile”.....	6
3	Formattazione personalizzata: <code>IFormattable</code>.....	7
3.1	Caratteristica della “formattabilità”.....	7
3.2	Interfaccia <code>IFormattable</code>	8
3.2.1	Tipo <code>TimeRange</code> : applicazione di <code>IFormattable</code>	8
3.3	Uso dell’interfaccia nel metodo <code>Format()</code>	9
3.4	Conclusioni.....	10
4	Processare sequenze: <code>IEnumerable</code>.....	11
4.1	Elaborare sequenze di oggetti: usare <code>IEnumerable</code>	12
4.2	Implementare l’interfaccia <code>IEnumerable</code>	12
4.3	Conclusioni.....	14
5	Testare in isolamento: classe astratta <i>stream</i>.....	15
5.1	Rendere <code>Channel</code> indipendente dallo stream di rete.....	16
5.2	Testare <code>Channel</code> in “isolamento”.....	16
5.3	Conclusioni.....	17
6	Decorare una funzione: classe astratta <code>Stream</code>.....	18
6.1	Decorare uno stream: comprimere i dati.....	19
6.2	Conclusioni.....	20
7	Riutilizzare l’implementazione: <code>Collection</code>.....	21
7.1	Realizzare una collezione “osservabile”.....	21
7.1.1	Uso della collezione.....	22

1 Introduzione

In questo tutorial propongo alcuni esempi di programmazione che prevedono l'uso di tipi già definiti in .NET: interfacce, classi astratte, classi concrete con metodi virtuali. L'obiettivo è quello di affrontare scenari che mostrino l'importanza dei tipi astratti in generale e come questi siano integrati, insieme ai tipi concreti, all'interno del .NET Framework.

Ogni esempio proposto avrà come obiettivo la realizzazione di una classe che implementa (o deriva) uno o più tipi astratti allo scopo di essere utilizzabile in determinati scenari. Prenderò in considerazione i seguenti tipi:

- Classe (concreta) `Object`.
- Interfaccia `IFormattable`.
- Interfaccia `IEnumerable`.
- Classe (astratta) `Stream`.
- Classe (concreta) `Collection`.

2 Formattazione in stringa: object.ToString()

Considera il seguente codice:

```
double n = 12.5;
DateTime adesso = DateTime.Now;
string s = string.Format("[{0}] [{1}]", n, adesso);
Console.WriteLine(s);
```

che produce il seguente output:

```
[12,5] [05/03/2018 21:44:41]
```

Appare evidente che il metodo `Format()` è in grado di ottenere una stringa da ognuno dei parametri. La questione è in che modo ci riesca. Non si tratta di una questione banale; infatti, nel seguente caso l'output non è quello che ci aspetteremmo:

```
class Persona
{
    public string Nome;
    public string Cognome;
}
...
var p = new Persona { Nome = "Pippo", Cognome = "Spada"};
DateTime adesso = DateTime.Now;
var s = string.Format("[{0}] [{1}]", p, adesso);
Console.WriteLine(s);
```

```
[Polimorfismo.ConsoleUI.Persona] [05/03/2018 21:46:32]
```

La questione si complica se si considera la *signature* del metodo¹:

```
public static string Format(string format, object arg0, object arg1);
```

Ricapitolando:

1. I parametri sono di tipo `object`, eppure `Format()` è in grado di elaborare qualsiasi tipo di dato.
2. Per ogni parametro, `Format()` ottiene una stringa, ma soltanto in alcuni casi è una stringa che rappresenta il valore del parametro.

Entrambe le questioni richiedono una spiegazione, la cui radice risiede nel ruolo del tipo `Object`.

2.1 Tipo Object

`Object` (alias `object`) è il tipo fondamentale dell'intero *type system* di .NET; infatti, qualsiasi tipo di dato deriva da `Object` e dunque ne eredita i metodi. Segue lo scheletro della classe (non riporto i metodi statici):

¹ In realtà esistono 8 versioni del metodo.

```

public class Object
{
    public Object() { }
    public virtual bool Equals(Object obj) { }
    public virtual int GetHashCode() { }
    public Type GetType() { }

    public virtual string ToString()
    {
        return GetType().ToString();
    }

    protected Object MemberwiseClone() { }
}

```

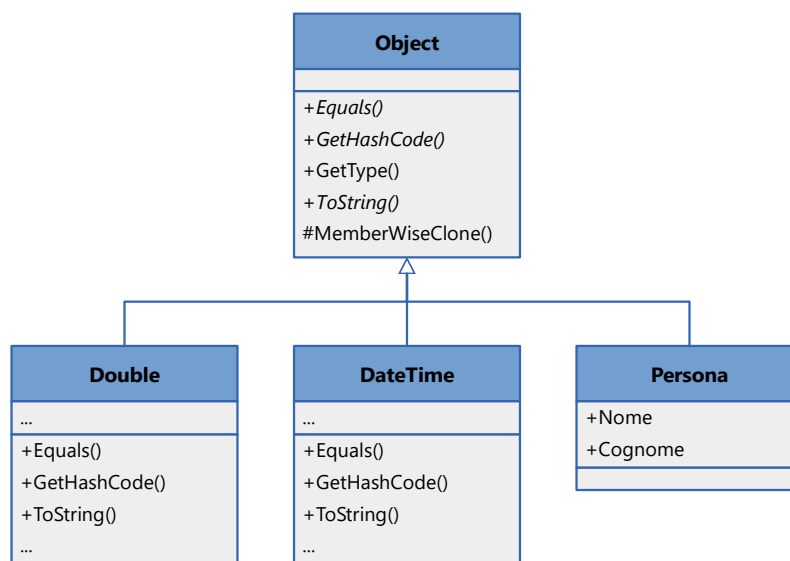
Nota bene: ho evidenziato i metodi virtuali e riportato l'implementazione del metodo `ToString()`. Quest'ultimo visualizza il nome, completo di *namespace*, del tipo *run-time* dell'oggetto, come mostra il seguente frammento di codice:

```

var persona = new Persona { Nominativo = "Pippo, Spada" };
DateTime adesso = DateTime.Now;
string nomeTipoAdesso = adesso.GetType().ToString(); //-> "System.DateTime"
string nomeTipoPersona = persona.GetType().ToString(); //-> "Polimorfismo.ConcoleUI.Persona"

```

Lo schema UML mostra la relazione tra `double`, `DateTime`, `Persona` e `Object`:²



`double` e `DateTime` ridefiniscono i metodi virtuali di `Object`, mentre `Persona` si limita a ereditarli.

2 La relazione tra `int`, `double`, `DateTime`, etc e `object` è più “articolata” di quella presentata in questo tutorial; ma ciò non cambia la sostanza del discorso.

2.2 Funzionamento del metodo `string.Format()`

Segue uno pseudo codice semplificato del metodo `Format()`³.

```
static string Format(string format, object arg0, object arg1)
{
    // per ogni carattere di "format"
    // se trova un segnaposto
    // formatta argomento corrispondente: (arg0 o arg1).ToString()
    // e aggiungi risultato alla stringa finale
    // ...
}
```

La parte evidenziata spiega il funzionamento del metodo: viene eseguito `ToString()` sul parametro corrispondente al segnaposto.

In conclusione, `Format()` ignora completamente il tipo effettivo degli argomenti; si limita a invocare `ToString()` su di essi. Il metodo effettivamente eseguito è quello implementato dal tipo dell'argomento, oppure quello ereditato da `Object`.

2.3 Rendere un tipo “Formattabile”

Per farlo è sufficiente ridefinire il metodo `ToString()`:

```
class Persona
{
    public string Nome;
    public string Cognome;

    public override string ToString()
    {
        return $"{Cognome}, {Nome}";
    }
}
```

Dopo questa modifica, l'output rispetta le attese, poiché il metodo eseguito sugli oggetti di tipo `Persona` non è più quello ereditato da `Object`, ma quello definito nella classe:

```
[Spada, Pippo] [05/03/2018 21:46:32]
```

3 Il codice reale è piuttosto lungo e complicato, e coinvolge diversi metodi.

3 Formattazione personalizzata: IFormattable

Considera il codice proposto all’inizio del precedente capitolo, con qualche modifica:

```
double n = 12.5;
DateTime adesso = DateTime.Now;
string s = string.Format("{0:F2} [{1:d}] [{1:D}]", n, adesso);
Console.WriteLine(s);
```

Ecco l’output:

```
[12,50] [06/03/2018] [martedì 6 marzo 2018]
```

Questo è regolato dalle *stringhe di formato* (`F2`, `d`, `D`), che specificano la formattazione da applicare ai parametri. Ebbene, la spiegazione fornita nel precedente capitolo non può giustificare questo risultato, poiché `ToString()` non accetta parametri e dunque non può produrre un risultato diverso in base a un fattore esterno.

Infatti, `Format()` funziona secondo un meccanismo diverso da quello presentato.

3.1 Caratteristica della “formattabilità”

Considera il seguente problema (lo stesso davanti al quale si sono trovati i progettisti del .NET Framework): vogliamo dare la possibilità di creare dei tipi di dati capaci di produrre una stringa in base a un parametro fornito dall’esterno (la *stringa di formato*). Il metodo virtuale `ToString()` definito in `Object` non è utilizzabile, poiché non accetta parametri. Si possono immaginare due soluzioni:

- Modificare `ToString()`, aggiungendo un parametro. Si tratta di una soluzione problematica, poiché in molti scenari non esiste la necessità di specificare una stringa di formato, ma saremmo costretti a farlo. Ad esempio, saremmo costretti a scrivere:

```
lblNumeroOrdini.Text = numOrdini.ToString(null);
```

- Definire in `Object` due metodi `ToString()`, di cui uno con parametro. A questo punto, però, tutti i tipi erediterebbero due metodi `ToString()`, di cui uno inutile per la maggior parte dei tipi.⁴

La questione centrale è: soltanto per alcuni tipi è significativo implementare una formattazione personalizzabile. In sostanza è utile introdurre il concetto astratto di *formattabilità*, cioè un *tratto distintivo che caratterizza i tipi in grado di produrre una conversione in stringa personalizzabile*.

4 Per inciso, è ciò che effettivamente accade per tutti i metodi virtuali di `Object`, e alcuni ingegneri Microsoft sostengono che non sia stata una buona scelta definire in `Object` tali metodi, poiché la maggiore parte dei tipi non ne hanno bisogno.

3.2 Interfaccia IFormattable

L'interfaccia `IFormattable` definisce il concetto astratto di *formattabilità*. Un tipo che implementa questa interfaccia è in grado di fornire una rappresentazione stringa personalizzabile.

```
public interface IFormattable
{
    string ToString(string format, IFormatProvider formatProvider);
}
```

Il metodo `ToString()` accetta come primo parametro una *stringa di formato* (sorvoliamo sul secondo parametro). È compito dei tipi che implementano l'interfaccia utilizzare `format` per stabilire il tipo di formattazione da applicare.

3.2.1 Tipo TimeRange: applicazione di IFormattable

Il tipo `TimeRange` implementa un intervallo di tempo (vedi tutorial **TimeRange**) ed è un buon candidato per mostrare in azione il concetto di *formattabilità*. L'attuale versione ridefinisce già il metodo `ToString()` ereditato da `Object`:

```
public struct TimeRange
{
    ...
    public override string ToString()
    {
        string sign = (Hours < 0 || Minutes < 0 || Seconds < 0) ? "-" : "";
        return string.Format("{0}{1}:{2}:{3}", sign, Math.Abs(Hours), Math.Abs(Minutes),
                               Math.Abs(Seconds));
    }
}
```

Ciò consente di scrivere codice come il seguente:

```
var tr = new TimeRange(1, 2, 3); // (1 ora, 2 minuti e 3 secondi)
Console.WriteLine(tr);
```

e ottenere l'output:

```
1:2:3
```

Desidero aggiungere a `TimeRange` la possibilità di utilizzare un diverso formato, che, nell'esempio, produrrebbe il seguente output⁵: **1 ore 2 minuti 3 secondi**.

Il formato è specificato dalla lettera `T`; pertanto mi aspetto che il codice:

```
var tr = new TimeRange(1, 2, 3); //->1:2:3
var s = string.Format("[{0}] [{0:T}]", tr);
Console.WriteLine(s);
```

produca l'output:

⁵ Formato senz'altro migliorabile, ma rappresenta soltanto un esempio per introdurre un concetto.


```
[1:2:3] [1 ore 2 minuti 3 secondi]
```

Ecco la nuova versione di `TimeRange`:

```
public struct TimeRange: IFormattable
{
    ...
    public override string ToString() {...}

    public string ToString(string format, IFormatProvider formatProvider)
    {
        if (format == "T")
        {
            string sign = (Hours < 0 || Minutes < 0 || Seconds < 0) ? "-" : "";
            return string.Format("{0}{1} ore {2} minuti {3} secondi", sign, Math.Abs(Hours),
                                Math.Abs(Minutes), Math.Abs(Seconds));
        }
        return ToString(); //qualunque altro formato produce l'output standard
    }
}
```

3.3 Uso dell'interfaccia nel metodo `string.Format()`

Il tratto della *formattabilità* ha una caratteristica importante: è opzionale. Mentre `Format()` può invocare `ToString()` per ogni argomento, poiché qualsiasi tipo definisce questo metodo (lo eredita da `Object`), non può fare altrettanto con `ToString(string format...)`, poiché soltanto i tipi che implementano `IFormattable` lo definiscono.

`Format()` deve dunque verificare se il tipo *run-time* dell'oggetto implementa `IFormattable`. Ecco una nuova versione di `Format()` che gestisce questo problema:

```
static string Format(string format, object arg0, object arg1)
{
    // per ogni carattere di "format"
    // se trova un segnaposto
    string formatString = <estrae stringa di formato da segnaposto>;
    object arg = <seleziona arg0 o arg1>;
    string ris = FormatArg(formatString, arg)
    // aggiungi risultato alla stringa finale
    // ...
}
```

Dopo aver estratto l'eventuale stringa di formato e aver selezionato il parametro corrispondente al segnaposto, il lavoro di formattazione viene eseguito da `FormatArg()`:

```
static string FormarArg(string formatString, object arg)
{
    IFormattable fmt = arg as IFormattable;
    if (fmt != null) // il tipo run-time di arg implementa IFormattable?
        return fmt.ToString(formatString, null);
}
```

```

    return arg.ToString();
}

```

Le prime due righe sono fondamentali: l'operatore di cast `as` converte `arg` nel tipo `IFormattable`. Se l'operazione fallisce (il tipo *run-time* di `arg` non implementa l'interfaccia), la variabile `fmt` viene inizializzata a `null`; se ha successo viene invocato `ToString(string format...)`, altrimenti il metodo `ToString()` ereditato (ed eventualmente ridefinito) da `Object`.

Nota bene, in tutto questo il valore della *stringa di formato* non è rilevante: se `arg` implementa `IFormattable` sarà comunque invocato il metodo dell'interfaccia e non quello ereditato da `object`.

3.4 Conclusioni

La definizione in `Object` del metodo virtuale `ToString()` e, soprattutto, l'esistenza dell'interfaccia `IFormattable` mostrano in azione il fondamentale risultato prodotto dell'uso di astrazioni: *poter scrivere un procedimento che non dipende dal tipo effettivo degli oggetti che elabora*.

Il metodo `string.Format()` produce una stringa sulla base del valore di alcuni argomenti; *ma non conosce il tipo degli argomenti*, né implementa un procedimento per convertire un argomento in stringa. D'altra parte, `Format()` "sa":

- Che ogni argomento definisce il metodo `ToString()`, poiché qualunque tipo di dato deriva da `Object`.
- Che un argomento può eventualmente implementare l'interfaccia `IFormattable`, e dunque definire il metodo `ToString(string format...)`.

Su queste basi è stato possibile scrivere del codice in grado di formattare in stringa qualsiasi tipo di oggetto.

Naturalmente, l'intera questione non riguarda soltanto `Format()`, ma l'intero "ecosistema" che si basa sulla conversione in stringa degli oggetti.

```

var tr = new TimeRange(1, 2, 3);           // (1 ora, 2 minuti, 3 secondi)
Console.WriteLine(tr);                    //->1:2:3
Console.WriteLine("{0}", tr);              //->1:2:3
Console.WriteLine("{0:T}", tr);            //->1 ore  2 minuti  3 secondi
var s = $"{tr} [{tr:T}]";                  // (1 ora, 2 minuti, 3 secondi)
Console.WriteLine(s);                      //->[1:2:3] [1 ore  2 minuti  3 secondi]

```

```

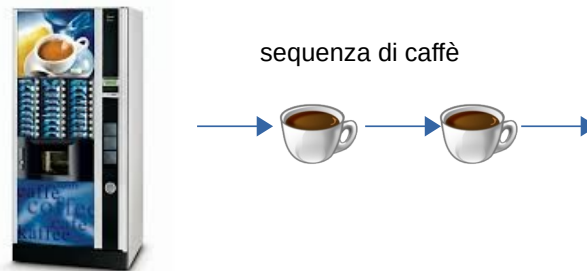
1:2:3
1:2:3
1 ore 2 minuti 3 secondi
[1:2:3] [1 ore 2 minuti 3 secondi]

```

4 Processare sequenze: IEnumerable

In molti scenari il codice deve rispondere alla necessità di processare una sequenza di oggetti. Per comprendere l'importanza di questi scenari occorre innanzitutto comprendere la differenza tra i concetti di *sequenza* e *collezione*, che spesso vengono usati in modo intercambiabile. Per farlo ricorrerò a una metafora.

Considera il distributore di caffè:



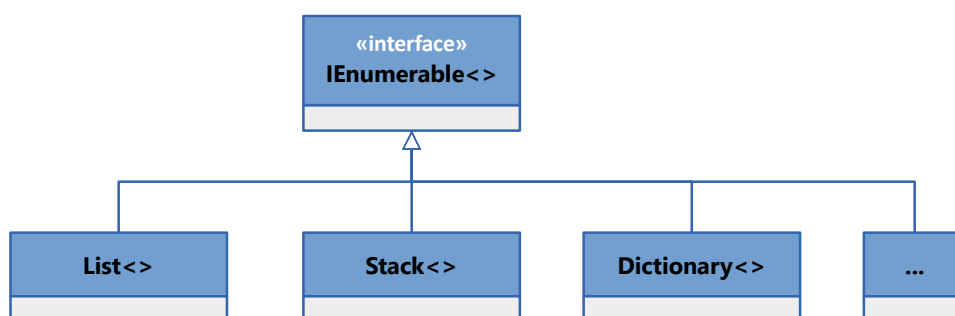
È in grado di produrre una *sequenza di caffè*, ma tutti noi sappiamo che dentro non nasconde una collezione di caffè già pronti: questi vengono preparati, e quindi consumati, man mano che sono richiesti. In sostanza: *la sequenza di caffè non è una cosa che esiste da qualche parte, né prima (dentro il distributore), né dopo (i caffè sono consumati, man mano che sono erogati).*

D'altra parte, possiamo "collezionare" i caffè in un vassoio per portarli agli amici (e ottenere così una *collezione di caffè*). Sappiamo, inoltre, che il distributore delle bibite, pur comportandosi come quello dei caffè (produce una *sequenza di bibite*), contiene effettivamente una *collezione di bibite*.

In conclusione, *sequenza* è un concetto più astratto di *collezione*. Il secondo suggerisce l'idea di una struttura che contiene degli oggetti; il primo, invece, fa semplicemente riferimento alla capacità di ottenere un oggetto per volta, man mano che viene richiesto.

.NET implementa il concetto di sequenza attraverso l'interfaccia generica `IEnumerable<>`. Il termine può essere tradotto in "iterabile" e cioè: un oggetto è *iterabile* se è in grado di produrre una sequenza di oggetti, uno per volta.

Naturalmente, tutti i tipi di collezione sono *iterabili*, sono cioè in grado di produrre in sequenza gli elementi che contengono.



Per accedere agli elementi di una sequenza (dunque, di un oggetto iterabile), occorre usare il costrutto `foreach`. Questo "scorre" gli elementi uno per volta, indipendentemente dal tipo effettivo dell'oggetto iterabile: vettore, lista, pila, dizionario, etc.

4.1 Elaborare sequenze di oggetti: usare IEnumerable

Il concetto di sequenza è indipendente dal modo con il quale i dati sono gestiti; è dunque vantaggioso scrivere codice che elabori sequenze senza dipendere dalla loro origine. Lo si fa utilizzando esplicitamente il tipo `IEnumerable`.

Il seguente metodo visualizza una sequenza di stringhe:

```
static void Visualizza(IEnumerable<string> listaNomi)
{
    foreach (var nome in listaNomi)
    {
        Console.WriteLine(nome);
    }
}
```

Poiché `Visualizza()` non dipende dal tipo effettivo di `listaNomi`, è utilizzabile con qualsiasi oggetto iterabile, come mostra il seguente codice:

```
string[] nomi = { "filippo", "andrea", "sara" };

List<string> listaNomi = new List<string>(nomi);
Stack<string> pilaNomi = new Stack<string>(nomi);
Queue<string> codaNomi = new Queue<string>(nomi);

Visualizza(nomi);           // -> filippo, andrea, sara
Visualizza(listaNomi);      // -> filippo, andrea, sara
Visualizza(pilaNomi);       // -> sara, andrea, filippo
Visualizza(codaNomi);       // -> filippo, andrea, sara
```

Nota bene: gli elementi della sequenza non vengono prodotti sempre nello stesso ordine; dipende dal tipo effettivo che li memorizza.

4.2 Implementare l'interfaccia IEnumerable

Il meccanismo sottostante all'*iterabilità* è piuttosto complicato da comprendere appieno; qui mi limito a introdurre quanto basta per poterla implementare in classi che gestiscono i dati in una collezione.

Segue un frammento della classe `AlbumFoto`, che gestisce un elenco di fotografie:

```
public class AlbumFoto
{
    List<Foto> fotoList = new List<Foto>();
    ...
    public AlbumFoto(string folder) {...}

    public int Count => fotoList.Count

    public Foto this[int index] => fotoList[index];
}
```

La classe incapsula una lista e fornisce l'accesso alle foto mediante un indicizzatore. Questa implementazione ha un limite: un oggetto `AlbumFoto` non può essere processato come una sequenza di foto. Perché ciò sia possibile deve implementare `IEnumerable<Foto>`:

```
public class AlbumFoto : IEnumerable<Foto>
{
    List<Foto> fotoList = new List<Foto>();
    ...
    public IEnumerator<Foto> GetEnumerator()
    {
        return fotoList.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator() // obsoleto
    {
        throw new NotImplementedException();
    }
}
```

L'interfaccia definisce il solo metodo `GetEnumerator()`⁶, il quale restituisce un `IEnumerator`; questo è un *iteratore*, e cioè un oggetto che ha la funzione di produrre le foto in sequenza.

Comprendere il funzionamento degli *iteratori* non è semplice, e in questo caso inutile; infatti, `AlbumFoto` si limita a restituire l'*iteratore* utilizzato da `fotoList`, il quale produrrà una sequenza di foto nell'ordine con il quale sono memorizzate nella lista.

Segue un frammento di codice che sfrutta l'iterabilità di `AlbumFoto`:

```
AlbumFoto album = new AlbumFoto("Immagini");
...
void LoadGallery()
{
    pnlGallery.Controls.Clear();
    foreach (Foto foto in album)
    {
        var fotoView = CreateFotoView(foto);
        pnlGallery.Controls.Add(fotoView);
    }
}
```

4.3 Conclusioni

Ci sarebbe molto da aggiungere sull'implementazione degli *iteratori*, ma qui è importante comprendere il fondamentale concetto dell'*iterabilità*, e cioè la capacità di produrre una sequenza di oggetti; fondamentale perché, in molti scenari, tutto ciò che serve è processare i dati in sequenza. La sua applicazione implica la comprensione di due punti:

- Il codice che elabora i dati mediante un `foreach` dovrebbe usare il tipo `IEnumerable`; in questo modo non dipenderà da un particolare tipo concreto.

⁶ In realtà definisce due metodi `GetEnumerator()`, ma il secondo è obsoleto e può essere ignorato (ma non cancellato).

- Qualunque tipo che gestisce, o è in grado di produrre, una collezione di oggetti, dovrebbe implementare l'interfaccia `IEnumerable`.

La sua implementazione in classi che gestiscono una collezione è estremamente semplice, poiché è sufficiente, nel metodo `GetEnumerator()`, chiamare l'omologo metodo della collezione.

5 Testare in isolamento: classe astratta *stream*

Di seguito disegno uno scenario tipico nel quale la possibilità di usare astrazioni è molto importante. Si vuole realizzare un'applicazione client-server che usa il protocollo di trasporto TCP. Il processo di comunicazione viene realizzato mediante un oggetto, `Channel`, che definisce i metodi di scrittura/lettura sullo *stream* di rete. Inoltre, le richieste del client e le risposte del server sono incapsulate in due record, `Request` e `Response`, i quali definiscono i campi contenuti nei messaggi.

Di seguito mostro il codice che incapsula le funzioni di scrittura/lettura delle richieste:

```
public class Request
{
    public string Type;
    public string Body;
}

...

public class Channel
{
    NetworkStream stream;
    public Channel(NetworkStream stream)
    {
        this.stream = stream;
    }

    public Request ReadRequest()
    {
        string[] items = ReadLine().Split(' ');
        return new Request { Type = items[0], Body = items[1] };
    }

    public void WriteRequest(Request req)
    {
        WriteLine($"{req.Type} {req.Body}");
    }

    private string ReadLine() {...} // legge una riga da stream
    private void WriteLine(string text) {...} // scrive il testo su stream
}
```

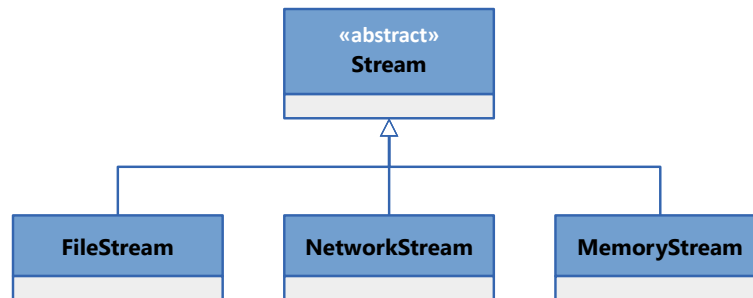
`Channel` viene usata nel client per inviare le richieste, nel server per leggerle; il problema è: come testare il suo funzionamento. È un problema tipico, quando un modulo è inserito in un contesto che ha dipendenze architetturali (database, *filesystem*, rete, etc).

Nell'esempio occorre eseguire i due processi client e server, inviare una richiesta dal client e verificare il risultato sul server; ma così facendo non si testa il singolo componente, ma l'intera sequenza di operazioni che hanno luogo durante la comunicazione TCP tra i due processi. L'obiettivo, al contrario, è verificare il corretto funzionamento di `Channel` isolandolo dal contesto in cui viene utilizzato. Ciò semplifica il test e garantisce che il resto del sistema possa fare affidamento su un componente funzionante.

5.1 Rendere Channel indipendente dallo stream di rete

L'attuale versione di `Channel` dipende da un `NetworkStream`, il quale viene ottenuto da un oggetto `TcpClient`; ma si tratta di una dipendenza non necessaria, perché tutto ciò che fa `Channel` è leggere/scrivere su uno *stream*, e non necessariamente uno *stream* di rete.

Stream è un concetto astratto: rappresenta un flusso di byte che può essere scritto e letto. In .NET è implementato attraverso la classe astratta `Stream`, dalla quale derivano vari tipi concreti, che si differenziano per la loro origine/destinazione: file, rete, memoria.



Il primo passo, dunque, è sostituire il tipo concreto `NetworkStream` con la classe astratta `Stream`:

```
public class Channel
{
    Stream stream;
    public Channel(Stream stream)
    {
        this.stream = stream;
    }
    ...
}
```

5.2 Testare Channel in “isolamento”

Poiché `Channel` non dipende più da una connessione di rete, ma da un tipo astratto, è possibile utilizzarlo con qualsiasi tipo di *stream*; in questo caso un `MemoryStream` si presta perfettamente, poiché gestisce il flusso di byte in memoria. Di seguito mostro un metodo che:

- Crea un `Channel` che agisce su un `MemoryStream`.
- Scrive una richiesta sul `Channel`.
- Resetta la posizione dello *stream* (necessario per rileggere i dati appena scritti).
- Legge la richiesta dal `Channel`.
- Visualizza la richiesta scritta e quella letta, verificando che siano uguali.

(L'approccio corretto sarebbe quello di creare uno *unit test*.)


```

static void TestChannel()
{
    var ms = new MemoryStream();
    var ch = new Channel(ms);
    var write = new Request { Type = "LOGIN", Body = "sara" };

    ch.WriteRequest(write);
    ms.Position = 0;    // fa sì che la lettura cominci dal primo byte
    var read = ch.ReadRequest();

    Console.WriteLine($"{write.Type} {write.Body}"); //-> "LOGIN sara"
    Console.WriteLine($"{read.Type} {read.Body}");    //-> "LOGIN sara"
}

```

5.3 Conclusioni

Questo semplice esempio mostra le potenzialità nell'uso di tipi astratti. Ci sono anni luce di distanza tra l'implementazione delle classi `NetworkStream` e `MemoryStream`. La prima deve gestire dei *socket* per la comunicazione bidirezionale di rete, la seconda rappresenta un semplice wrapper di un vettore di byte; ma, dal punto di vista di `Channel`, questa differenza è irrilevante, poiché sono entrambe degli *stream*. Ciò consente di applicare una strategia largamente utilizzata nello sviluppo del software: testare il funzionamento di un componente in un contesto diverso da quello reale.

6 Decorare una funzione: classe astratta Stream

Quando si implementa una funzionalità vale la pena renderla utilizzabile in un contesto più ampio possibile. In alcuni casi ciò non richiede ulteriore progettazione, ma soltanto l'uso di tipi astratti.

Segue una classe che implementa la copia di un file, un blocco di byte per volta; un evento notifica l'avanzamento del processo di copia riportando il numero di byte copiati⁷.

```
class FileCopy
{
    public event EventHandler<int> DataCopied;

    public void Copy(string sourceFileName, string destFileName, int blockSize)
    {
        var source = File.OpenRead(sourceFileName);
        var dest = File.Create(destFileName);
        byte[] buffer = new byte[blockSize];
        int bytesRead = source.Read(buffer, 0, blockSize);
        int byteCopied = 0;
        while (bytesRead > 0)
        {
            dest.Write(buffer, 0, bytesRead);
            byteCopied += bytesRead;
            DataCopied?.Invoke(this, byteCopied);
            bytesRead = source.Read(buffer, 0, blockSize);
        }
        source.Close();
        dest.Close();
    }
}
```

Segue un frammento di codice che usa la classe:

```
var fileCopy = new FileCopy();
fileCopy.DataCopied += FileCopy_DataCopied;
fileCopy.Copy("Relazione.pdf", "RelazioneCopia.pdf", 1024);
```

La classe è semplice da usare, ma la sua implementazione è rigida; infatti, il processo di copia elabora degli *stream*, ma sorgente e destinazione devono essere dei file. Si tratta di un vincolo inutilmente restrittivo che è molto semplice da rimuovere:

```
class FileCopy
{
    public event EventHandler<int> DataCopied;

    public void Copy(string sourceFileName, string destFileName, int blockSize)
    {
        FileStream source = File.OpenRead(sourceFileName);
        FileStream dest = File.Create(destFileName);
```

⁷ Per mantenere il codice più semplice possibile, non gestisco la chiusura degli *stream* in caso di eccezione. Inoltre ho implementato l'evento senza una classe dedicata.

```

        Copy(source, dest, blockSize);
    }

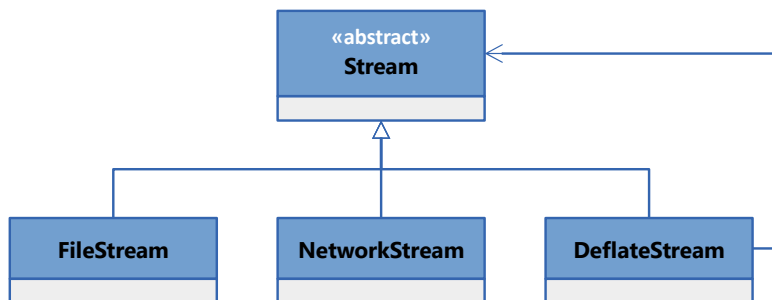
    public void Copy(Stream source, Stream dest, int blockSize)
    {
        byte[] buffer = new byte[blockSize];
        int bytesRead = source.Read(buffer, 0, blockSize);
        int byteCopied = 0;
        while (bytesRead > 0)
        {
            dest.Write(buffer, 0, bytesRead);
            byteCopied += bytesRead;
            DataCopied?.Invoke(this, byteCopied);
            bytesRead = source.Read(buffer, 0, blockSize);
        }
        source.Close();
        dest.Close();
    }
}

```

Ho estratto il processo di copia in un secondo metodo, più generale, che accetta degli *stream* e che viene usato dal metodo originale.

6.1 Decorare uno stream: comprimere i dati

Ora che la classe è in grado di processare qualunque *stream*, è possibile sfruttare un particolare pattern, chiamato *decorator*. Questo è applicato da alcune classi che derivano da *stream*, le quali sono in grado estendere il comportamento di uno *stream* esistente. Un esempio è `DeflateStream`, che consente di comprimere/decomprimere i dati:



Il diagramma mostra che `DeflateStream` è uno *stream* che, internamente, ne usa un altro. Segue un frammento della classe, che mostra come questa utilizzi uno *stream* esistente:

```

public class DeflateStream : Stream
{
    Stream stream;
    public DeflateStream(Stream stream, CompressionMode mode)
    {
        this.stream = stream;
        ...
    }
}

```

```
...
}
```

Segue un frammento di codice che mostra come usare `FileCopy` per copiare un file applicando la compressione (e successivamente la decompressione):

```
FileCopy fileCopy = new FileCopy();

//-> comprime il file "Relazione.pdf" in "Relazione.pdf.cmp"
fileCopy.DataCopied += FileCopy_DataCopied;
var src = File.OpenRead("Relazione.pdf");
var dst = File.Create("Relazione.pdf.cmp");
var deflateStream = new DeflateStream(dst, CompressionMode.Compress);

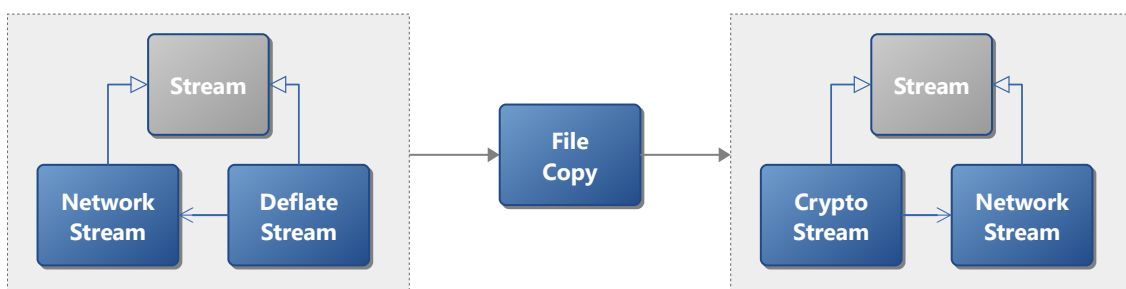
fileCopy.Copy(src, deflateStream, 1024);
...
//-> decomprime il file "Relazione.pdf.cmp" in "Relazione.pdf"
var src = File.OpenRead("Relazione.pdf.cmp");
var dst = File.Create("Relazione.pdf");
var deflateStream = new DeflateStream(src, CompressionMode.Decompress);

fileCopy.Copy(deflateStream, dst, 1024);
```

6.2 Conclusioni

L'esempio proposto mostra l'applicazione di un doppio livello di astrazione. Da una parte, il metodo `Copy()` esegue un processo di copia completamente indipendente da sorgente e destinazione dei dati: `MemoryStream`, `FileStream`, `NetworkStream`. Dall'altra, l'uso di `DeflateStream`, e cioè di un *decoratore*, mostra che, a loro volta, sorgente e destinazione possono essere manipolate.

Siamo partiti da un metodo in grado soltanto di copiare un file, per arrivare a un metodo (con lo stesso codice!) che può essere usato, ad esempio, per estrarre dati compressi da un *stream* di rete e scriverli criptati in un altro *stream* di rete:



Ciò è stato possibile perché il metodo dipende soltanto da un tipo astratto e dunque può processare qualunque tipo che sia uno *stream*.

7 Riutilizzare l'implementazione: Collection

L'ereditarietà è uno dei due meccanismi usati in .NET per l'implementazione di astrazioni (l'altro è l'uso di interfacce). Alla base di questo meccanismo c'è la possibilità di riutilizzare un'implementazione esistente, estendendola e/o modificandola per adattarla alle proprie esigenze. Questo evita di "dover reinventare la ruota" ogni volta.

Considera l'ipotesi di una classe che gestisce internamente una collezione di oggetti e che, mediante un evento, notifichi eventuali modifiche. Oppure considera l'ipotesi di una collezione che registra in un *log* i propri cambiamenti. Questa avrebbe una funzione simile a quella ipotizzata in precedenza, ma reagirebbe in modo diverso alle eventuali modifiche.

In questi e altri scenari è utile avere a disposizione un tipo che implementi una funzione generale, ma che possa essere adattato a esigenze specifiche, evitando di dover riscrivere un nuovo componente da zero. Un esempio è dato dalla classe generica `Collection`.

`Collection` definisce le tipiche funzioni di una collezione, con una particolarità: i metodi che eseguono le modifiche sono virtuali, in modo che i tipi derivati possano "decorarli" con ulteriori operazioni, o, eventualmente, modificarne il comportamento.

7.1 Realizzare una collezione "osservabile"

Si dice "osservabile" una collezione che notifica i propri cambiamenti, in genere attraverso un evento. Per realizzarla, il modo più semplice è derivarla da `Collection`, poiché basta ridefinire i quattro metodi che modificano gli elementi.

Innanzitutto implemento i tipi necessari per definire l'evento, il quale notifica quattro tipi di cambiamento:

```
public enum ChangeAction
{
    Reset,    // -> la collezione è stata svuotata
    Add,      // -> è stato inserito un elemento
    Remove,   // -> è stato rimosso un elemento
    Replace   // -> è stato modificato un elemento
}

public class ListChangedEventArgs
{
    public ChangeAction Action { get; set; }
    public int Index { get; set; } // -> indice elemento coinvolto (-1 se Reset)
}
```

Dopodiché implemento una classe generica che deriva da `Collection` e che ridefinisce i metodi virtuali `ClearItems()`, `InsertItem()`, `RemoveItem()` e `SetItem()`:

```
class ObservableList<T> : Collection<T>
{
    public event EventHandler<ListChangedEventArgs> ListChanged;

    protected override void ClearItems()
    {
        base.ClearItems();
    }
}
```

```

        RaiseEvent(ChangeAction.Reset);
    }

    protected override void InsertItem(int index, T item)
    {
        base.InsertItem(index, item);
        RaiseEvent(ChangeAction.Add, index);
    }

    protected override void RemoveItem(int index)
    {
        base.RemoveItem(index);
        RaiseEvent(ChangeAction.Remove, index);
    }

    protected override void SetItem(int index, T item)
    {
        base.SetItem(index, item);
        RaiseEvent(ChangeAction.Replace, index);
    }

    private void RaiseEvent(ChangeAction action, int index = -1)
    {
        var e = new ListChangedEventArgs { Action = action, Index = index };
        ListChanged?.Invoke(this, e);
    }
}

```

Alcune considerazioni:

- In `Collection` i metodi virtuali sono protetti e dunque accessibili soltanto nelle classi derivate; vengono chiamati dai metodi pubblici (non virtuali) soltanto dopo la validazione degli argomenti. Tutto questo semplifica l'implementazione della classe.
- In ogni metodo invoco innanzitutto il metodo base; il mio scopo non è modificare il funzionamento della classe base, ma aggiungere un nuovo comportamento.

7.1.1 Uso della collezione

Segue un frammento di codice che usa la classe:

```

ObservableList<string> list = new ObservableList<string>();
list.ListChanged += List_ListChanged;
list.Add("sara");           // -> Add 0
list.Add("filippo");        // -> Add 1
list[1] = "andrea";         // -> Replace 1
list.RemoveAt(0);           // -> Remove 0
list.Clear();               // -> Reset -1

private static void List_ListChanged(object sender, ListChangedEventArgs e)
{
    Console.WriteLine($"{e.Action} {e.Index}");
}

```