

1	6
Panoramica generale	6
1 Programmazione procedurale	6
1.1 Un primo passo verso la programmazione orientata ad oggetti	8
2 Programmazione orientata agli oggetti	9
2.1 Esempio di una soluzione ad oggetti di un problema	10
3 Aspetti principali della programmazione orientata agli oggetti	12
3.1 Incapsulamento (astrazione e interfaccia pubblica)	12
3.2 Ereditarietà	13
3.3 Polimorfismo	15
4 Il principio di entità "Aperta/Chiusa"	15
4.1 La versione di Mayer	15
4.2 La versione polimorfica	16
2	18
Tipi di dati definiti dal programmatore	18
1 Classi che definiscono "tipi"	18
1.1 Tipi definiti dal programmatore (UDT: User Defined Types)	18
1.2 Rappresentare i tipi con l'UML (Unified Modelling Language)	18
1.3 I diagrammi UML	19
2 Definizione di una classe	20
2.1 Rappresentazione di una classe in UML	20
2.2 Un primo esempio di classe	21
2.3 Classe parziale: suddividere una classe in più file	23
2.4 Campi membro	23
2.5 Rappresentazione degli attributi in UML	24
2.6 Funzioni membro	26
2.7 Riferimento all'oggetto: parola chiave "this"	26
2.8 Rappresentazione dei metodi in UML	28
3 Classi e oggetti	29
3.1 "Campo di azione" di classe e "Ciclo di vita" di un oggetto	30
3.2 Membri statici e non statici (d'istanza)	33
3.3 Rappresentazione di membri statici in UML	34
3.4 Classi statiche	35
3.5 Costruttori delle classi statiche	35
4 Livello di accesso ai membri di una classe	36
4.1 Modificatori del livello di accesso (o protezione)	36
4.2 Interfaccia pubblica e implementazione di una classe	38
4.3 Stabilire il livello di accesso dell'intera classe	40
5 Metodi	40
5.1 Metodi di accesso	41
5.2 <i>Overloading</i> di metodi	42
5.3 "Risoluzione" di un metodo sovraccaricato	44
5.4 <i>Overloading</i> di operatori	45
6 Costruttori	46
6.1 Codice ammissibile all'interno di un costruttore	47
6.2 <i>Overloading</i> di costruttori	48
6.3 Inizializzatori di costruttore	48
6.4 Costruttore di default	49
7 Il tipo "struttura"	51
7.1 Tipi struttura come semplici aggregati	51

7.2 Tipi struttura come oggetti.....	52
8 Migliorare l'accesso agli attributi della classe.....	55
8.1 "Proprietà"	56
8.2 Definizione e uso di una proprietà.....	56
8.3 Descrizione delle proprietà in UML	58
8.4 Modificatori di accesso separati per ciascun accessor di una proprietà	58
8.5 Funzionamento del "set accessor" e del "get accessor"	60
8.6 Codice ammissibile negli <i>accessor</i> di una proprietà.....	61
8.7 Proprietà "automatiche"	63
8.8 Inizializzatore di oggetto	63
8.9 Proprietà e variabili a confronto	64
8.10 <i>Overloading</i> di proprietà	64
8.11 Linee guida nella scelta dei nomi delle proprietà	64
9 Un esempio completo: Classe Nave.....	65
9.1 Classe "Nave"	65
9.2 Accesso agli attributi della classe "Nave"	66
9.3 Costruttori della classe "Nave".....	67
9.4 Metodi della classe "Nave".....	67
9.5 Operatori della classe "Nave"	69
9.6 (De)Serializzazione della classe "Nave"	71
3.....	74
Relazioni tra classi.....	74
1 Tipi di relazione tra classi	74
2 Dipendenza	75
3 Associazione.....	76
3.1 Navigabilità.....	77
3.2 Molteplicità.....	78
3.3 Classi di Associazione	80
4 Aggregazione.....	81
5 Composizione.....	82
6 Generalizzazione.....	83
4.....	84
Ereditarietà	84
1 "Tipi" e "tipi derivati"	84
1.1 Esempio concreto di ereditarietà.....	86
2 Ereditarietà: "classi base" e "classi derivate"	89
2.1 Membri derivati e nuovi membri di una classe derivata	90
2.2 Rendere accessibili i membri definiti nella classe base	92
2.3 Invocare i costruttori della classe base	93
2.4 Invocazione implicita del costruttore di default della classe base.....	94
3 Ridefinire i membri della classe base.....	95
3.1 Definire un campo membro con lo stesso nome di un campo ereditato.....	95
3.2 Definire funzioni membro con lo stesso nome di funzioni ereditate.....	97
3.3 Sovraccaricare il metodo ereditato.....	98
3.4 Conclusioni.....	99
5.....	100
Polimorfismo	100
1 Premessa al polimorfismo.....	100
1.1 Relazione "un tipo di" applicata agli oggetti	101
2 Funzioni membro virtuale e "invocazione ritardata"	105
2.1 Definizione e ridefinizione (<i>override</i>) di una funzione membro virtuale	106

2.2 "Invocazione ritardata" (collegamento ritardato) di una funzione	106
2.3 Invocare la funzione virtuale della classe base.....	108
2.4 Ridefinizione di funzioni virtuali sovraccaricate	109
3 Polimorfismo	110
3.1 Funzioni virtuali e funzioni non virtuali a confronto	110
3.2 Polimorfismo in .NET	111
3.3 Classe "Object"	111
3.4 Conversione implicita.....	113
3.5 Conversione esplicita: operatore di cast "()"	114
3.6 Casting nelle collezioni generiche	115
3.7 Conoscere il tipo effettivo di un oggetto: operatore "is"	115
3.8 Operatore di conversione esplicita "as"	117
3.9 Ottenere il tipo di un oggetto: metodo "GetType()"	117
4 Classi astratte.....	118
4.1 Analisi del dominio del problema.....	119
4.2 Introduzione nella gerarchia di una classe astratta	120
4.3 Definizione della classe Battello.....	121
4.4 Definizione delle classi "Nave" e "Gommone"	122
4.5 Uso della nuova gerarchia di classi.....	123
4.6 Definizione formale di classi astratte e metodi astratti	123
4.7 Definizione di funzioni membro astratte.....	125
4.8 Considerazioni sulla progettazione	126
5 Varianza dei tipi	126
5.1 Invarianza	127
5.2 Covarianza	128
5.3 Controvarianza	128
6	129
Classi generiche.....	129
1 I problemi delle classi fortemente tipizzate.....	129
2 La soluzione: i generics	131
2.1 Creare una nuova classe generica	131
2.2 Limitare i tipi di parametro.....	133
7	135
Classi come collezioni di oggetti	135
1 Creare nuovi tipi Collection	135
1.1 Creare collezioni tipizzate mediante aggregazione	135
1.2 Accesso agli elementi di una collezione	136
1.3 Definizione e uso di indicizzatori	137
1.4 "set accessor" e "get accessor" negli indicizzatori.....	138
1.5 Tipo degli indici e codice ammissibile negli <i>accessor</i> di un indicizzatore	139
1.6 Iterare la collezione	139
1.7 Iteratori	141
1.8 Creare collezioni tipizzate mediante derivazione	143
8	147
Interfacce.....	147
1 Introduzione alle interfacce.....	147
2 Che cos'è un' <i>interfaccia</i>	148
2.1 Implementazione di una <i>interfaccia</i>	150
2.2 Implementazione di più <i>interfacce</i>	152
2.3 Classi che implementano la stessa <i>interfaccia</i>	154
3 Uso delle <i>interfacce</i>	155

3.1	Usò dell' <i>interfaccia</i> "IDictionary<TKey, TValue>"	155
3.2	Implementare l' <i>interfaccia</i> "IComparable<T>" per l'ordinamento di collezioni	157
3.3	Implementare l' <i>interfaccia</i> "IComparer<T>" per l'ordinamento di collezioni	158
3.4	Implementare l' <i>interfaccia</i> "IEquatable<T>" per confrontare due elementi.....	159
4	Definizione di <i>interfacce</i>	160
4.1	Il problema dell'ereditarietà multipla tra classi	161
4.2	L'ereditarietà multipla tra interfacce.....	163
4.3	Definizione delle interfacce INoleggiabile e ICaricabile	164
4.4	Implementazione della gerarchia di classi.....	165
4.5	Le interfacce come denominatore comune.....	169
4.6	Derivazione di classe e implementazione di <i>interfaccia</i> a confronto	172
5	<i>Interfacce, ereditarietà e polimorfismo</i>	172
5.1	Ereditarietà applicata alle <i>interfacce</i>	173
5.2	Implementazione di interfacce derivate	174
5.3	<i>Interfacce</i> ereditate dalla classe base.....	175
5.4	Reimplementazione di una <i>interfaccia</i> in una classe derivata	175
5.5	Implementazione di interfacce mediante funzioni virtuali.....	176
5.6	Operatori di cast e <i>interfacce</i>	177
5.7	Covarianza nelle interfacce	179
5.8	Interfacce e tipi struttura.....	179
9	180
Metodi di Estensione	180
1	Introduzione ai metodi di estensione	180
2	Riutilizzo di metodi senza l'ereditarietà	181

Panoramica generale

1 Programmazione procedurale

Per cominciare a comprendere i tratti salienti che caratterizzano la programmazione orientata agli oggetti (*Object Oriented Programming*; d'ora in avanti OOP) è necessario prima delineare cosa s'intende per "programmazione procedurale", che è poi il modello utilizzato finora.

Un programma procedurale si traduce nella definizione e nell'invocazione di procedure, o metodi, e cioè frammenti di codice identificati da un nome, che svolgono determinate elaborazioni sui dati ed eventualmente producono dei risultati. L'aspetto in particolare che qui ci interessa e che caratterizza la programmazione procedurale rispetto a quella ad oggetti può essere così sintetizzato:

esiste una netta distinzione tra i dati da elaborare e il codice – le procedure appunto – che li elabora.

Ciò si traduce:

nella mancanza di una chiara corrispondenza tra gli oggetti che caratterizzano il problema ("dominio del problema") e la rappresentazione che di essi viene fatta all'interno del programma ("dominio del programma").

Per focalizzare il concetto consideriamo il seguente esempio.

Si vuole realizzare un programma che gestisca le ore di straordinario mensile dei dipendenti di un'azienda. Di ogni dipendente è necessario memorizzare il nominativo (nome + cognome), un codice che identifica la mansione svolta in seno all'azienda e le ore di straordinario accumulate.

Per memorizzare i dati si possono utilizzare tre collezioni, ad esempio tre `List<>`, una per ogni categoria d'informazione. Le elaborazioni richieste vengono realizzate mediante metodi appositi, di cui ne prenderemo in considerazione due: un metodo che aggiunge un nuovo dipendente alla lista ed uno che visualizza l'intera lista.

Segue una rappresentazione schematica dei dati ed il codice dei due metodi:

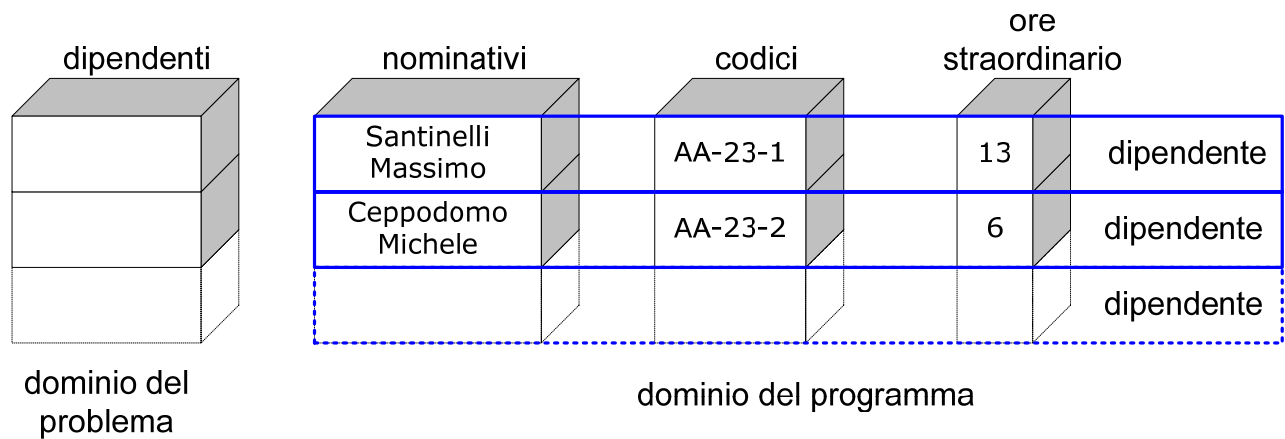


Figura 1-1 Schema del dominio del problema e della sua rappresentazione nel programma.

```
//...
List<string> nominativi = new List<string>;
List<string> codici = new List<string>
List<int> ore = new List<int>();
...
static void AggiungeDipendente(string nome, string codice, int ore)
{
    nominativi.Add(nome);
    codici.Add(codice);
    ore.Add(ore);
}
...
static void VisualizzaDipendenti()
{
    Console.WriteLine("Elenco dipendenti che hanno accumulato ore di straordinario:");
    for (int indDip = 0; indDip < nominativi.Count; indDip++)
    {
        Console.WriteLine("{0}\t{1}\t{2}", nominativi[indDip],
                                codici[indDip],
                                ore[indDip]);
    }
}
```

Il problema è caratterizzato dall'esistenza di un elenco di soggetti – i dipendenti che hanno accumulato ore di straordinario – i quali sono definiti attraverso degli attributi: nominativo, codice di mansione e ore di straordinario accumulate. Nel dominio del problema ogni dipendente rappresenta dunque un'entità distinta, caratterizzata da precise informazioni, ma nel tradurre in concreto questo scenario, nel rappresentare cioè i dipendenti attraverso le strutture dati del programma, i confini di queste entità si perdono.

All'interno del programma non esistono oggetti che possono essere messi in corrispondenza con quelli del problema, né, soprattutto, esiste alcuna istruzione che coinvolga un dipendente in quanto tale. Esistono tre liste di dati all'interno delle quali sono “disperse” le informazioni sui dipendenti, e tre metodi che operano su di esse. E' soltanto la logica del programma a rendere le due rappresentazioni compatibili.

1.1 Un primo passo verso la programmazione orientata ad oggetti

Il modo in cui è stata implementata la soluzione del problema proposto rappresenta, non a caso, un esempio estremo d'applicazione del paradigma procedurale. Praticamente ogni linguaggio di programmazione procedurale rende possibile un approccio diverso e senz'altro migliore nella rappresentazione dei dati, che entro certi limiti consente di mantenere una corrispondenza tra gli elementi del dominio del problema e la loro rappresentazione nel programma. Ciò è possibile attraverso l'uso degli aggregati, che il linguaggio C# implementa attraverso i tipi struttura, designati dalla parola chiave `struct`. L'uso di aggregati permette di definire nuovi tipi di dati, i quali rendono possibile una rappresentazione più accurata degli oggetti del problema. Ritornando all'esempio precedente, ecco come definire un nuovo tipo di dato che rappresenti un dipendente:

```
struct Dipendente
{
    public string Nominativo;
    public string Codice;
    public int Ore;
}
```

Il tipo struttura definisce tre variabili – “campi” – ognuna delle quali corrisponde ad un attributo del dipendente. A questo punto, l'elenco dei dipendenti non è più memorizzato attraverso tre liste dei relativi attributi, ma mediante una sola lista contenente oggetti del tipo `Dipendente`.

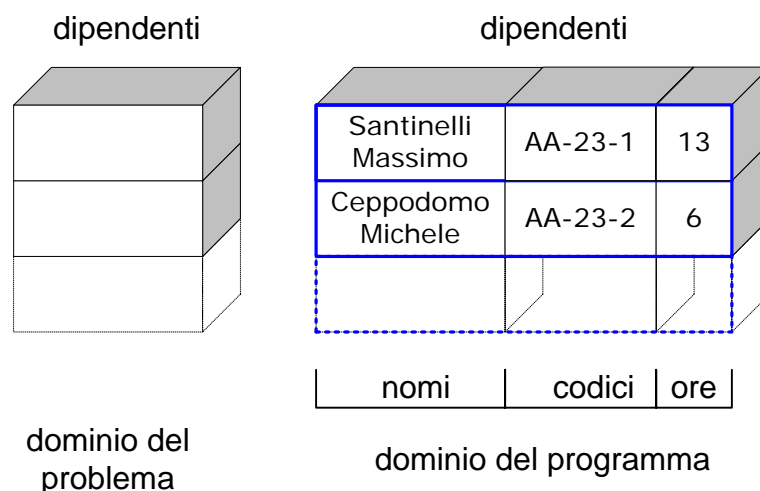


Figura 1-2 Schema del dominio del problema e della sua rappresentazione nel programma.

Con l'introduzione degli aggregati la rappresentazione del dominio del problema è più accurata, ma i connotati che caratterizzano il modello di programmazione procedurale restano gli stessi: esistono dei metodi – delle procedure – completamente indipendenti dagli oggetti – i dati – e che svolgono delle operazioni su di essi.

Questo stato di cose può essere genericamente schematizzato nel seguente modo:

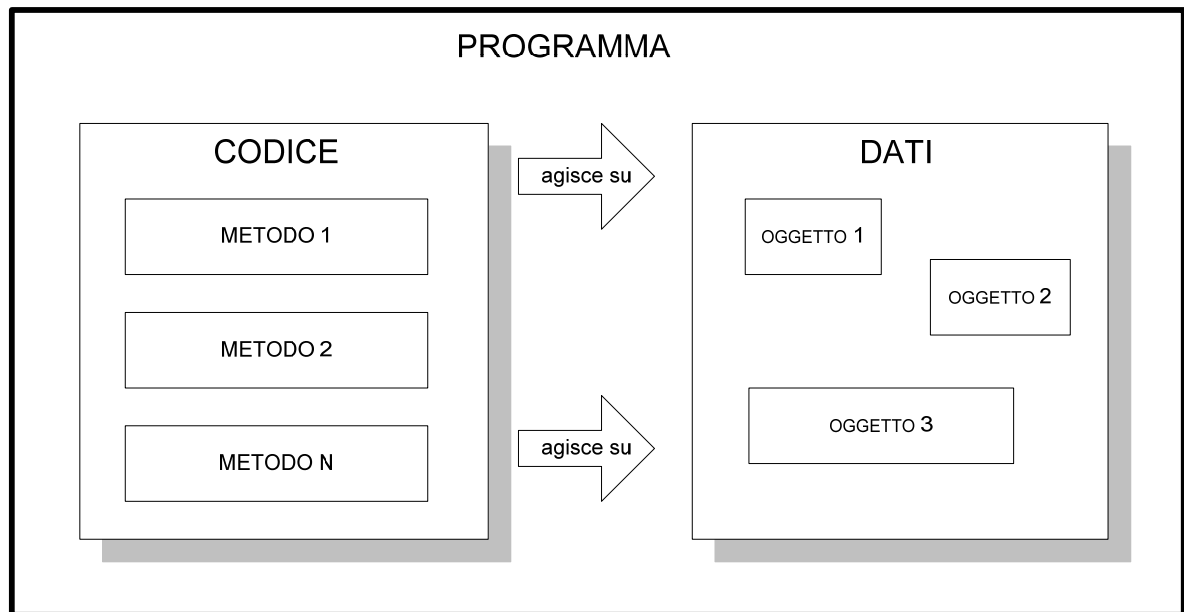


Figura 1-3 Schema generale di un programma di natura procedurale.

2 Programmazione orientata agli oggetti

La OOP nasce da un presupposto fondamentale che la distingue dalla programmazione procedurale:

i dati non sono separati dal codice che li elabora.

L'idea di oggetto assume qui un significato più stringente; un oggetto infatti:

fornisce sia una rappresentazione del dato/i sia l'insieme delle operazioni che possono essere eseguite su di esso/i.

Un oggetto unisce i dati ai metodi che operano su di essi in un'unica entità. L'insieme dei dati – dei valori che essi assumono – rappresenta lo “stato” dell'oggetto. L'insieme dei metodi rappresenta le operazioni ammissibili sull'oggetto, operazioni che ne possono modificare lo stato.

Secondo questa definizione, la struttura di un programma orientato agli oggetti assume un schema diverso da quello del modello procedurale.

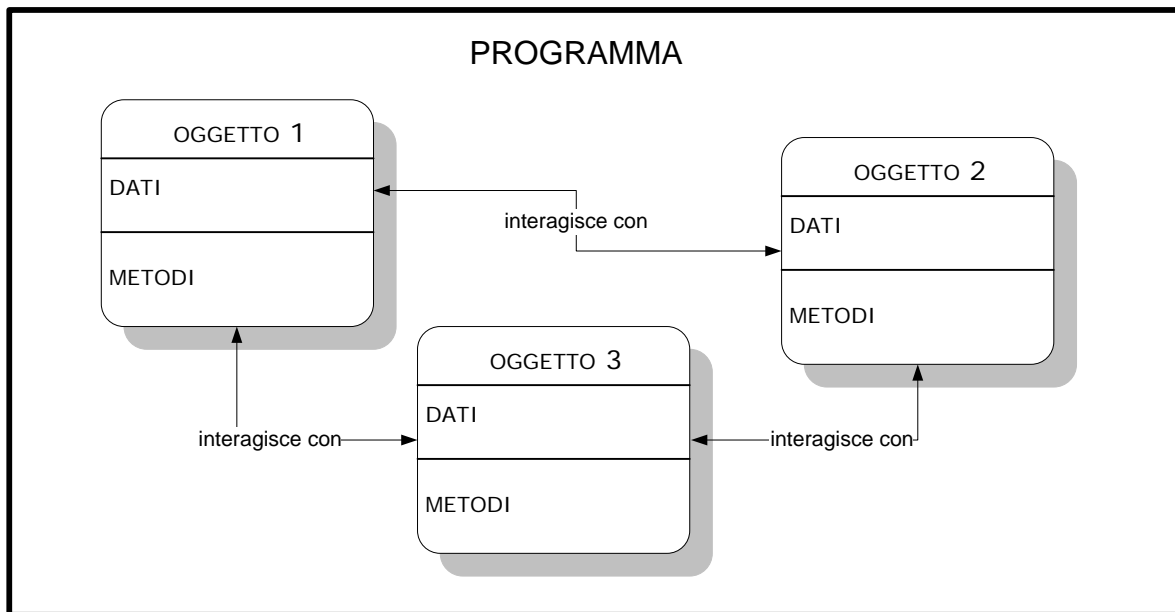


Figura 1-4 Schema generale di un programma orientato agli oggetti.

2.1 Esempio di una soluzione ad oggetti di un problema

Viene qui proposta una soluzione object oriented del problema della gestione delle ore di straordinario dei dipendenti di un'azienda. Per il momento è necessario sorvolare sugli elementi sintattici legati al nuovo paradigma e concentrarsi sugli aspetti che lo caratterizzano rispetto al modello di programmazione procedurale.

//...

```

class Dipendente
{
    private string _nominativo;
    private string _codice;
    private int _ore;

    public Dipendente(string nominativo, string codice, int ore)
    {
        _nominativo = nominativo;
        _codice = codice;
        _ore = ore;
    }

    public void Visualizza()
    {
        Console.WriteLine("{0}\t{1}\t{2}", _nominativo, _codice, _ore);
    }
}

class MainClass
{

    List<Dipendente> dipendenti = new List<Dipendente>;

```

```
static void AggiungeDipendente(string nome, string codice, int ore)
{
    Dipendente dip = new Dipendente(nome, codice, ore);
    dipendenti.Add(dip);
}

static void VisualizzaDipendenti()
{
    Console.WriteLine("Elenco dipendenti che hanno accumulato ore di
                        straordinario:");
    for (int indDip = 0; indDip < nominativi.Count; indDip++)
    {
        Dipendente dip = dipendenti[indDip];
        dip.Visualizza();
    }
}
```

Il codice definisce un nuovo tipo di dato, `Dipendente`, identificato dalla parola chiave `class`, termine che sta per “classe”. Definire una classe equivale appunto ad introdurre un nuovo tipo, descritto attraverso:

- uno o più attributi – campi – che definiscono l’insieme delle informazioni memorizzate dagli oggetti del tipo in questione;
- uno o più metodi che definiscono l’insieme delle operazioni ammissibili su tali oggetti.

La struttura assunta da una classe e il legame tra oggetti e classi saranno approfonditi nei capitoli successivi; per il momento è degno di nota esaminare l’uso che nel codice viene fatto dell’oggetto `dip`. All’interno dei due metodi è stato eliminato ogni riferimento diretto alla rappresentazione interna dell’oggetto, e cioè alle informazioni sul nominativo, sul codice e sulle ore di straordinario. L’accesso ad un oggetto avviene soltanto attraverso i metodi definiti dalla classe d’appartenenza. Tra questi, un metodo molto speciale è il “costruttore” – `Dipendente()` – il cui nome coincide con il nome della classe. Il costruttore è sempre il primo metodo che viene invocato su un oggetto, poiché è appunto il metodo che lo crea (lo costruisce appunto).

In conclusione, dalla prospettiva del codice che usa un oggetto (vedi riquadro), questo è visto come un’entità monolitica, la cui effettiva configurazione interna è inaccessibile. E’ la classe alla quale l’oggetto appartiene a definire tale configurazione, oltre all’insieme delle operazioni eseguibili sull’oggetto stesso.

Per convenzione, il codice che agisce su un oggetto ma non appartiene alla classe dell’oggetto medesimo viene definito: codice *consumer* (“consumatore”). Il nome nasce dal fatto che il codice utilizza l’oggetto, i suoi servizi, come un cliente (un consumatore appunto) utilizza i servizi di un qualche ente che li eroga.

Un frammento di codice è *consumer* sempre in relazione a una o più classi. Ciò significa che un metodo può appartenere ad una classe (e dunque non essere codice *consumer* per oggetti di *quella* classe) e utilizzare oggetti di un’altra classe, per i quali è da considerare *consumer*. In molti testi il codice *consumer* viene anche chiamato “codice utente”, intendendo con il termine “utente” designare il programmatore che scrive il codice in questione e non l’utilizzatore del programma.

3 Aspetti principali della programmazione orientata agli oggetti

La OOP è intimamente connessa a quattro concetti fondamentali:

- ❑ “**incapsulamento**” e “**astrazione**”;
- ❑ “**ereditarietà**” o “**derivazione**” o “**generalizzazione**”;
- ❑ “**polimorfismo**”.

3.1 Incapsulamento (astrazione e interfaccia pubblica)

L’idea di incapsulamento non nasce certo con la OOP, ma vi trova comunque la sua applicazione più naturale. Il termine incapsulamento (che in inglese viene anche tradotto in *information hiding*: “nascondere l’informazione”)¹ si riferisce all’abilità di:

rendere inaccessibile – e dunque nascosta – la rappresentazione interna dei dati memorizzati in un oggetto.

L’incapsulamento implica che il codice che usa l’oggetto (il codice *consumer*) non è in grado di leggere e/o modificare direttamente i dati memorizzati in esso. A questo scopo la classe d’appartenenza dell’oggetto definisce un insieme di metodi, “l’interfaccia pubblica”, i quali forniscono di esso una rappresentazione astratta, che può corrispondere in modo più o meno accurato all’effettiva implementazione interna.

Incapsulamento, astrazione, interfaccia pubblica e implementazione sono dunque concetti strettamente collegati. La classe d’appartenenza di un oggetto definisce infatti:

- ❑ la rappresentazione interna dei dati – nome e tipo delle variabili – memorizzati nell’oggetto;
- ❑ dei metodi appositi – l’interfaccia pubblica – per elaborare l’oggetto.

Il codice *consumer* ignora l’effettiva implementazione interna dell’oggetto; attraverso l’interfaccia pubblica ne ottiene una rappresentazione astratta che può differire anche in misura notevole. Detto ciò, l’incapsulamento dell’informazione rappresenta un aspetto fondamentale della OOP, poiché:

fintantoché l’interfaccia pubblica di un oggetto non varia, qualsiasi modifica alla sua implementazione non influisce sul funzionamento del codice *consumer* che lo utilizza.

A titolo di esempio viene fornita una nuova versione della classe `Dipendente`, con una diversa rappresentazione interna, ma un’identica interfaccia pubblica.

```
//...
class Dipendente
{
    private string _nominativo;
    private string _codice;
    private string _ore;           // implementazione modificata!

    public Dipendente(string nominativo, string codice, int ore)
    {
        _nominativo = nominativo;
        _codice = codice;
```

```
        _ore = ore.ToString();           // implementazione modificata!
    }

    public void Visualizza()
    {
        Console.WriteLine("{0}\t{1}\t{2}", _nominativo, _codice, _ore);
    }
}

class MainClass
{
    List<Dipendente> dipendenti = List<Dipendente>;
    static void AggiungeDipendente(string nome, string codice, int ore)
    {
        Dipendente dip = new Dipendente(nome, codice, ore);
        dipendenti.Add(dip);
    }

    static void VisualizzaDipendenti ()
    {
        Console.WriteLine("Elenco dipendenti che hanno accumulato ore di
                           straordinario:");
        for (int indDip = 0; indDip < nominativi.Count; indDip++)
        {
            Dipendente dip = (Dipendente) dipendenti[indDip];
            dip.Visualizza();
        }
    }
}
```

Nella nuova implementazione, il campo `ore` è adesso di tipo `string`. Ciò non influisce minimamente sul resto del programma, il quale risulta assolutamente invariato. Questo non sarebbe possibile se il codice *consumer* della classe, e cioè i metodi `AggiungeDipendente()` e `VisualizzaDipendenti()`, avessero accesso alla sua rappresentazione interna. In questo caso, qualsiasi modifica di tale rappresentazione si tradurrebbe in una necessaria modifica del codice che usa oggetti della classe.

3.2 Ereditarietà

Il concetto di ereditarietà, diversamente da quelli di incapsulamento e astrazione, validi anche nell'ambito della programmazione procedurale, è strettamente connesso alla OOP. Questo termine denota un significato in qualche modo simile a quello assunto in biologia, laddove viene impiegato per indicare le relazioni di parentela tra un soggetto (un animale, una pianta) e i soggetti che lo hanno generato, oppure tra una specie e le altre specie dalle quali deriva.

Nella OOP l'ereditarietà è rappresentata da una relazione di parentela esistente tra classi; essa consente di definire – derivare – una nuova classe sulla base di un'altra classe, già definita in

¹ I puristi della programmazione object oriented non sarebbero probabilmente d'accordo con questa affermazione, riservando ai termini incapsulamento e «information hiding» significati leggermente diversi.

precedenza. La nuova classe eredita le caratteristiche della classe da cui deriva, denominata “classe base”, normalmente aggiungendone di nuove.

La relazione tra classe base e classe derivata può essere schematizzata nel seguente modo:

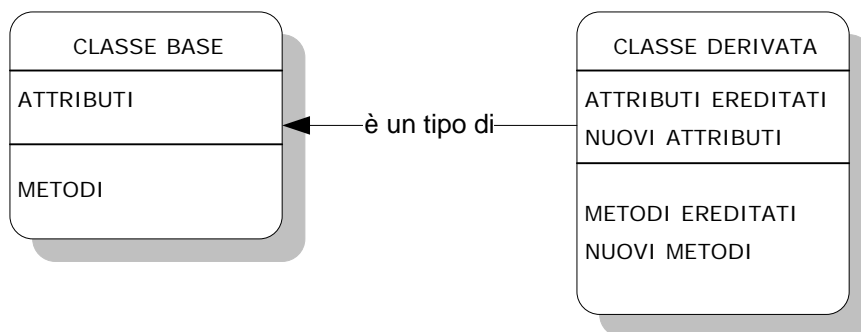


Figura 1-5 Schematizzazione di una relazione di derivazione tra classi.

La relazione di parentela “un tipo di” denota il fatto che la classe derivata eredita gli attributi e i metodi definiti dalla classe base; in un certo senso, la classe derivata è “un tipo di” classe base. Ciò non significa affatto che le due classi siano uguali; infatti la classe derivata può:

- ❑ definire ulteriori attributi e metodi oltre a quelli ereditati;
- ❑ fornire un’implementazione diversa dei metodi ereditati, che di fatto modifica il comportamento degli oggetti appartenenti a tale classe.

L’ereditarietà è centrale nella OOP, poiché sta alla base del meccanismo di riutilizzo del codice. La sua applicazione consente inoltre di costruire una gerarchia di classi arbitrariamente complessa e ramificata, nella quale ogni classe ha relazioni di parentela più o meno dirette con altre classi. (La stessa gerarchia di classi di .NET ne rappresenta un esempio.)

Nei programmi realizzati finora abbiamo già incontrato esempi di derivazione di una classe da un’altra. Ciò avviene ad esempio in tutte le “Applicazioni Windows”, infatti la loro struttura di base è:

```
using System;
using System.Windows.Forms;
using System.Drawing;
class MainForm: Form
{
    ...
    public MainForm()
    {
        ...
    }

    static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Nella riga di programma:

```
class MainForm: Form
```

il simbolo due-punti seguito dal nome della classe `Form` implica che la classe `MainForm` deriva da questa e dunque ne eredita tutte le funzionalità.

3.3 Polimorfismo

Nella OOP il polimorfismo (polimorfico: che assume più forme) è reso possibile e rappresenta una naturale conseguenza dell'ereditarietà, anche se i due concetti non devono essere confusi. Infatti, l'ereditarietà riguarda le relazioni di parentela tra classi, mentre il polimorfismo è un aspetto connesso agli oggetti.

Fondamentalmente, il polimorfismo implica che:

una variabile appartenente alla classe-A può referenziare un oggetto di classe-B, purché classe-B derivi, direttamente o indirettamente, da classe-A.

In altre parole: un oggetto di tipo classe-A può “comportarsi” come un oggetto di tipo classe-B.

Qualsiasi programma poco più che banale fa un uso intenso del polimorfismo, nella fattispecie tutte le volte che è necessario impiegare collezioni basate sul tipo `object`.

Ad esempio, nel seguente codice:

```
object[] oggetti = new object[2] { "Dante", "Petrarca" };
```

viene creato un array di `object` nel quale ogni elemento referencia una stringa. Dunque, il tipo degli elementi è `object`, mentre il tipo degli oggetti effettivamente memorizzati è `string`. Di fatto, un elemento `object` referencia un oggetto di tipo diverso. Ciò è reso possibile dal fatto che la classe `string` deriva dalla classe `object`.

Il polimorfismo è un aspetto fondamentale della OOP poiché risponde all'esigenza, frequentissima, di memorizzare ed elaborare oggetti di tipo diverso mediante la stessa struttura dati e lo stesso procedimento.

4 Il principio di entità “Aperta/Chiusa”

Nella programmazione ad oggetti il principio “**Aperta/Chiusa**” definisce che una entità software (sia essa una classe, un metodo, od una funzione) dovrebbe essere “**Aperta**” ad eventuali estensioni, ma “**Chiusa**” ad ogni tipo di modifica. In altre parole un'entità può si modificare il proprio comportamento, ma senza alterare il codice sorgente.

Appare ovvio come questa “filosofia” sia importantissima negli ambienti di produzione reali dove migliaia (se non milioni) di righe di codice dipendono da altre migliaia (se non milioni) di righe di codice.

La modifica del codice sorgente causerebbe sicuramente notevoli perdite di tempo per ispezionare del codice, compiere test e fare altre procedure che assicurino che tutto funzioni prima dell'effettuazione delle modifiche. Il codice che segue questo principio, invece, non necessita di effettuare nessun sforzo da questo punto di vista.

Il nome di questo principio viene comunemente utilizzato in due modi, che sfruttano entrambi l'ereditarietà, ma con obiettivi, tecniche e risultati differenti.

4.1 La versione di Mayer

La definizione di questo principio “**Aperta/Chiusa**” viene comunemente attribuita al Dott. Bertrand Mayer ed apparve intorno alla fine degli anni 80.

In questa versione del principio una classe, una volta completata non dovrebbe essere modificata (se non per correggere errori) e se ci fosse la necessità di nuove o differenti caratteristiche si dovrebbe creare una nuova classe.

In questa definizione si parla quindi di *ereditarietà di implementazione* in quanto l'interfaccia della nuova classe può essere (oppure no) la stessa della classe base (**APERTA**) mentre la nuova classe deve riutilizzare il codice della classe originale (**CHIUSA**)

4.2 La versione polimorfica

Durante gli anni 90, però, il principio “**Aperta/Chiusa**” assunse un altro significato per fare riferimento alle interfacce astratte².

In questa versione l'implementazione di una nuova classe può essere diversa dalla classe base e più implementazioni possono essere create e sostituite polimorficamente l'una all'altra.

Contrariamente alla definizione precedente di Meyer, questa volta è l'implementazione a cambiare dalla classe base a quella ereditata (**APERTA**) mentre l'interfaccia esistente non può essere modificata (**CHIUSA**) e le nuove classi devono, come minimo, supportarla.

² Approfondiremo questi concetti più avanti nel testo

Tipi di dati definiti dal programmatore

1 Classi che definiscono “tipi”

Il concetto di classe è indipendente da quello o quell'altro linguaggio di programmazione, anche se ognuno ne fornisce una propria rappresentazione e introduce a questo scopo determinati elementi sintattici. In C#, come in altri linguaggi (Java ad esempio), una classe può assumere due vesti concettualmente distinte:

- ❑ come semplice contenitore di metodi;
- ❑ come tipo di dato.

La “classe come semplice contenitore di metodi” non ha alcun rapporto con il concetto di classe, e dunque di tipo, proprio della OOP. Essa rappresenta una prerogativa del linguaggio C# e non è generalmente utilizzata in altri linguaggi che pure possono definirsi a pieno titolo dei linguaggi orientati agli oggetti, come ad esempio C++ e VB.NET). Le regole di C# impongono che il codice appartenga comunque a una classe; per questo motivo anche un impiego unicamente procedurale del linguaggio presuppone la definizione di almeno una classe da parte del programmatore. Un tipico esempio di classe come contenitore di metodi è la classe `Math`. La sua funzione non è quella di rappresentare un tipo di dato, (non è possibile creare oggetti di tipo `Math`); essa raggruppa semplicemente tutti i metodi di natura matematica, unificandoli sotto un unico nome.

Detto ciò, il concetto di classe esaminato in questo testo è quello proprio della OOP.

1.1 Tipi definiti dal programmatore (UDT: User Defined Types)

Definire una classe significa dunque descrivere un nuovo tipo di dato. In questo senso, il nuovo tipo descritto dal programmatore viene convenzionalmente denominato *User Defined Type* (tipo definito dall'utente), laddove il termine “utente” designa il programmatore che “usa” il linguaggio per definire il nuovo tipo. Questa precisazione è necessaria, il termine “utente” designa appunto il programmatore e non ha niente a che vedere con l'utilizzatore del programma (soggetto a volte designato dal termine “utente finale”).

1.2 Rappresentare i tipi con l'UML (Unified Modelling Language)

L'UML è, fondamentalmente, un linguaggio visuale per progettare software e rappresentare modelli di programmazione, ma se si andasse un po' più in profondità, si scoprirebbe che l'UML è nato per essere un modo semplice e comune per catturare le relazioni, i comportamenti ma anche le idee ad alto livello utilizzando una notazione facile da imparare ed efficiente da scrivere.

L'UML è diventato lo standard di-fatto per la modellazione di software ed è cresciuto di popolarità anche per la modellazione di altri domini.

Nato dalla fusione di tre distinti metodi di modellazione, *Booch*, *Object Modelling Technique* ed *Objectory*, nel 1994 è stato accettato come standard dall'Object Management Group (OMG) nel 1997 e rilasciato come versione 1.1

Mediante successivi raffinamenti ed evoluzioni, ha raggiunto attualmente la versione 2.0 che è sicuramente la più corposa, anche quella più chiara e compatta.

Prima di tutto è importante capire che l'UML è un linguaggio e che in quanto tale ha sia una sintassi che una semantica. Quando si modella un concetto in UML ci sono regole riguardo come gli elementi possono essere messi assieme e cosa significano quando li si organizzano in un certo modo.

Si può applicare UML in svariati ambiti ma i più comuni sono:

- ❑ progettazione software;
- ❑ comunicazione di processi business;
- ❑ elencare i dettagli di un sistema in termini di requisiti ed analisi;
- ❑ documentare un processo, un sistema od un'organizzazione già esistente.

Di tutte queste potenzialità noi ci occuperemo solamente della parte di modellazione del software tenendo conto che un modello UML è composto da uno o più diagrammi.

1.3 I diagrammi UML

Un diagramma rappresenta graficamente delle cose e le relazioni tra queste cose. Le cose possono essere rappresentazioni di oggetti del mondo reale, puri costrutti software oppure descrizioni del comportamento di oggetti.

UML 2.0 divide i diagrammi in due categorie: diagrammi strutturali e diagrammi comportamentali. I diagrammi strutturali sono utilizzati per catturare l'organizzazione fisica delle cose all'interno di un sistema e come gli oggetti si relazionano gli uni con gli altri. I diagrammi definiti dallo standard sono:

- ❑ diagrammi di classe;
- ❑ diagrammi di oggetto;
- ❑ diagrammi di componente;
- ❑ diagrammi di struttura composta;
- ❑ diagrammi di deployment;
- ❑ diagrammi di package.

I diagrammi comportamentali si focalizzano sul comportamento degli elementi all'interno di un sistema, come requisiti, operazioni, e cambiamenti interni di stato. Sono diagrammi comportamentali:

- ❑ diagrammi di attività;
- ❑ diagrammi di comunicazione;
- ❑ diagrammi di interazione;
- ❑ diagrammi di sequenza;
- ❑ diagrammi di stato;
- ❑ diagrammi di temporizzazione.

A fronte di questa grande potenzialità ed espressività, in questo volume ci occuperemo solamente dell'UML come strumento per la progettazione di classi ed oggetti, trascurando tutto il resto.

Quando si disegna un diagramma di classe in UML, la loro rappresentazione dipende dalla fase del processo di sviluppo in cui ci si trova e dal livello di dettaglio desiderato. Nella fase di analisi, infatti, ci si concentrerà sulle classi visibili nel dominio del problema, mentre man mano che si entra nella fase di progettazione verranno introdotte classi e relazioni che riflettono più da vicino il modello della soluzione fino a che non si arriverà ad una versione “implementabile” del modello.

E' bene comunque sottolineare che i diagrammi delle classi non mostrano come interagiscono tra loro i componenti di un modello: di questo si occupano i modelli di interazione, di sequenza o comunicazione.

Dal punto di vista grafico la notazione UML è molto semplice e verrà illustrata mano a mano che si procederà nella spiegazione degli elementi che definiscono una classe.

2 Definizione di una classe

Ogni classe è descritta attraverso:

- ❑ una “intestazione”, rappresentata dalla parola chiave **class** seguita dal nome della classe;
- ❑ un “corpo”, e cioè da un blocco (coppia di parentesi graffe) che ne definisce il contenuto.

Ogni elemento definito nel corpo è chiamato “membro” della classe e appartiene alla categoria³:

- ❑ degli “attributi” se è una variabile;
- ❑ delle “operazioni” se contiene del codice, come ad esempio un metodo;

Ciò detto, la definizione di una classe, nella sua forma semplificata, assume la seguente sintassi:

```
modificatoriopz class NomeClasse
{
    modificatoriopz tipo attributo 1;
    ...
    modificatoriopz tipo attributo n;
    modificatoriopz tipodiritorno operazione 1;
    ...
    modificatoriopz tipodiritorno operazione n;
}
```

dove l'ordine di dichiarazione degli attributi e delle operazioni è irrilevante.

2.1 Rappresentazione di una classe in UML

Dal punto di vista dell'UML la rappresentazione di una classe avviene con una determinata simbologia, mostrata in figura a pagina successiva. AL riguardo è importante aggiungere che in UML tutto è opzionale, nel senso che una classe può contenere:

- ❑ il solo nome;
- ❑ il nome e l'elenco degli attributi;

³ Esiste un ulteriore tipo di membro, e cioè è «l'evento», che però non sarà preso in considerazione.

- ❑ il nome e l'elenco delle operazioni;
- ❑ il nome e l'elenco degli attributi e l'elenco delle operazioni.

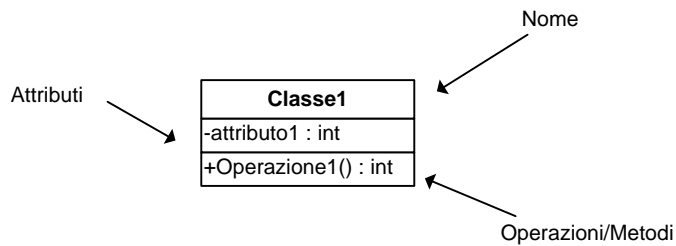


Figura 2-1 Rappresentazione di una classe in UML.

La lista degli attributi e delle operazioni rispecchia un formato ben definito, che vede i singoli elementi elencati una riga per volta in una sezione apposita.

Il nome di ogni attributo dovrebbe iniziare con una lettera minuscola mentre quello di una operazione con una lettera maiuscola⁴.

Sia davanti al nome dell'attributo che a nome di una operazione è possibile indicare un simbolo che rappresenta il modificatore. Nel diagramma UML sono previsti 4 simboli per indicare il modificatore e questi sono il “-” (privato, quello di default), il “+” (pubblico), il “#” (protetto) ed il “~” (internal)⁵

2.2 Un primo esempio di classe

Segue l'esempio di una classe che definisce il tipo “calciatore”:

```

class Calciatore
{
    // attributi
    string nome;
    string squadra;
    string ruolo;
    int golSegnati;
    // costruttore (operazione)
    public Calciatore(string nome, string squadra, string ruolo)
    {
        this.nome = nome;
        this.squadra = squadra;
        this.ruolo = ruolo;
        golSegnati = 0;
    }

    // metodo (operazione)
    public void AggiornaGolSegnati(int gol)
    {

```

⁴ In realtà l'esperienza ed i moderni editor presenti all'interno degli ambienti di sviluppo suggeriscono delle “modifiche” per quanto riguarda la nomenclatura degli attributi e questa varia a seconda del modificatore dell'attributo stesso

⁵ Del significato di modificatore ce ne occuperemo più avanti nel libro


```
    }  
    ...  
    static void Main()  
    {  
        class ClasseDefinitaNelPostoSbagliato    // errore!  
        {  
            ...  
        }  
    }  
}
```

L'argomento è trattato in modo approfondito in appendice ("Livello di accesso ai tipi, tipi nidificati e *namespaces*")

2.3 Classe parziale: suddividere una classe in più file

A partire dal .NET 2.0, la definizione di una classe può essere suddivisa in più file.

Questa possibilità è una conseguenza della evoluzione dei designer dei nuovi ambienti di sviluppo, che consentono di progettare una interfaccia senza scrivere nemmeno una riga di codice.

Per fare ciò, il designer deve essere in grado di separare il codice scritto "a mano" dall'utente da quello che l'utente ha "disegnato" ed il designer ha generato.

L'introduzione della parola chiave **partial** fa sì che si possa suddividere una classe in due o più parti. In fase di compilazione sarà poi cura del compilatore raccogliere i vari pezzi della classe, formare un sorgente unico e procedere alla compilazione dello stesso..

2.4 Campi membro

I campi membro (o "dati membro", o "campi di classe", o "attributi") sono rappresentati dalle variabili. Il seguente codice definisce una classe – *Persona* – con tre campi, rispettivamente di tipo *string*, *double* e *int*:

```
class Persona  
{  
    string nome;  
    double peso;  
    int altezza;  
    ...  
}
```

Data questa descrizione, ogni oggetto di tipo *Persona* contiene esattamente tre variabili, i cui nomi e tipi sono quelli specificati nella definizione. Ad esempio, il seguente codice:

```
Persona o1 = new Persona();  
Persona o2 = new Persona();
```

produce la creazione di due oggetti, i quali possono essere così schematizzati:

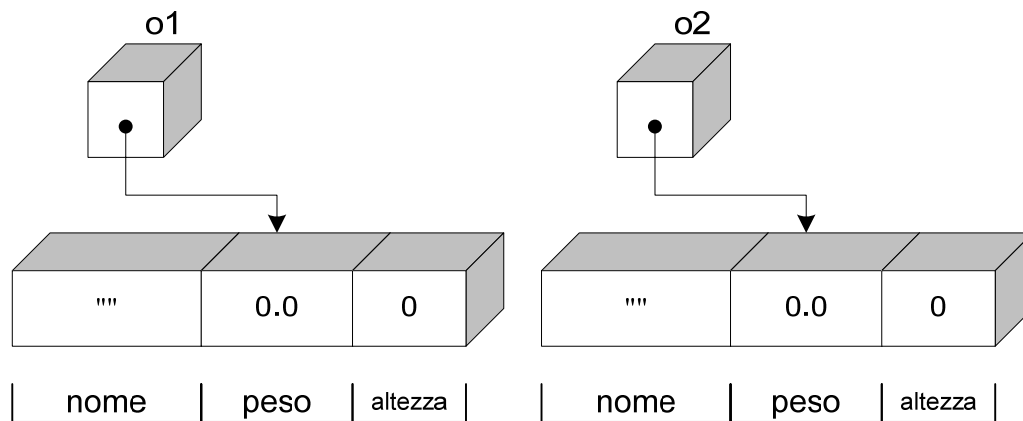


Figura 2-2 Rappresentazione di due oggetti della classe Persona.

Inizializzazione dei campi membro

Diversamente da quanto accade per le variabili locali, il valore iniziale di un campo è quello predefinito stabilito dal tipo di appartenenza. In fase di dichiarazione può essere specificato un valore diverso fornendo un iniziatore; il linguaggio garantisce che sarà questo il valore memorizzato nel campo prima che esso venga utilizzato in una qualsiasi espressione. Ad esempio:

```
class Auto
{
    string marca;
    string modello;
    string motorizzazione = "benzina";    // inizializzazione esplicita
    ...
}
```

Di norma i valori iniziali dei campi membro vengono impostati all'atto della creazione dell'oggetto (vedi paragrafo sui "costruttori") e dunque si dovrebbero specificare degli iniziatore soltanto se esiste un valore iniziale appropriato per il campo in questione.

2.5 Rappresentazione degli attributi in UML

A livello di descrizione in linguaggio UML degli attributi, essi vengono rappresentati nella sezione della classe, utilizzando la seguente notazione:

```
visibilità / nome : tipo molteplicità = default
{indicazioni di proprietà e vincoli}
```

La sintassi degli elementi è espressa dalla seguente tabella.

Tabella 1-1 Sintassi UML per gli attributi.

TIPO	DESCRIZIONE
Visibilità	Indica la visibilità dell'attributo ed utilizza i simboli +, -, # e ~ rispettivamente per pubblico, privato e protetto ed internal
/	Indica che l'attributo è derivato, calcolato, cioè a aprtire da qualcos'altro (es; netto = ricavi – costi)

TIPO	DESCRIZIONE
Nome	Nome dell'attributo che inizia tipicamente con la lettera minuscola. In fase di implementazione si può antecedere il simbolo di sottolineatura " _ "
Tipo	Il tipo dell'attributo
Molteplicità	Specifica quante istanze del tipo attributo sono referenziate da quell'attributo. Mentre l'assenza indica molteplicità 1, si possono specificare valori interi, un intervallo tra parentesi quadrate separate da ".." ed utilizzare il simbolo "*" per indicare "..più"
Default	Il valore di default dell'attributo
Proprietà	Indicate tra parentesi graffe indicano ordinamento, univocità, readonly ed altre ancora.

Riprendendo la classe `Calciatore` vista in precedenza, ecco il relativo diagramma UML:

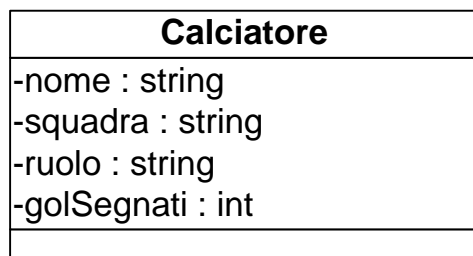


Figura 2-3 La classe `Calciatore` disegnata in UML.

Il diagramma UML, consenta anche di includere delle note che possono servire a spiegare meglio il diagramma oppure il significato di attributi ed operazioni.

Se per esempio volessimo indicare che l'attributo `golSegnati` fa riferimento alla stagione in corso, si potrebbe disegnare questo schema:

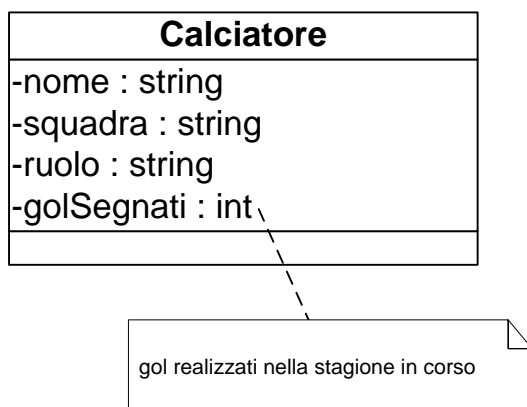


Figura 2-4 La nota relativa ai `golSegnati`.

2.6 Funzioni membro

Esistono varie categorie di funzioni membro ed in questo capitolo ne saranno esaminate due: i costruttori e i metodi.

Costruttori

Un costruttore è una particolare funzione membro che viene invocata mediante l'operatore **new** ogni qual volta viene creato un oggetto della classe; esso ha sempre il nome della classe.

Consideriamo ad esempio la seguente classe:

```
class Persona
{
    string nome;
    double peso;
    int altezza;

    public Persona(string nome, double peso, int altezza)
    {
        this.nome = nome;
        this.peso = peso;
        this.altezza = altezza;
    }
}
```

Nella creazione di un oggetto di tipo `Persona`, il costruttore è la prima funzione membro ad essere eseguita. Dunque, l'esecuzione del seguente codice:

```
Persona o1 = new Persona("Franco Rossi", 50, 172);
Persona o2 = new Persona("Angelo Bianchi", 87, 186);
```

produce la doppia invocazione del costruttore `Persona()`.

Una classe può definire più costruttori, i quali devono differenziarsi per il numero e/o il tipo dei parametri (vedi più avanti l'argomento “*overloading*”). Se non viene definito alcun costruttore, il linguaggio ne fornisce automaticamente uno senza parametri.

Nota bene: usando questa prassi occorre riporre un'estrema attenzione nello scrivere correttamente i nomi dei parametri e nel premettere sempre la parola chiave `this`, altrimenti si rischia di introdurre dei bug molto difficili da individuare.

2.7 Riferimento all'oggetto: parola chiave “this”

Nel codice presentato finora appare spesso la parola chiave `this`. Come sarà mostrato più avanti, questa parola ha molteplici significati e il suo uso non è strettamente connesso al codice contenuto nelle funzioni membro.

Nel codice di una funzione membro, la parola chiave `this` rappresenta un riferimento all'oggetto attraverso il quale la funzione viene invocata. In altre parole è un modo per riferirsi all'oggetto dall'interno del codice che appartiene all'oggetto stesso.

Usando questa prassi, però, occorre riporre un'estrema attenzione nello scrivere correttamente i nomi dei parametri e nel premettere sempre la parola chiave `this`, altrimenti si rischia di introdurre dei bug molto difficili da individuare.

Se infatti si riscrive il codice del precedente costruttore introducendo “dimenticandosi” questa avvertenza

```
public Persona(string nome, double peso, int altezza)
```

```
{
    nome = nome;
    peso = peso;
    altezza = altezza;
}
```

non otterremmo il risultato di inizializzare i campi membro dell'oggetto con i valori passati come parametro, come sarebbe auspicabile, ma assegneremmo semplicemente il valore dei parametri a loro stessi, con conseguenti bug e malfunzionamenti che verrebbero fuori in fase di esecuzione del programma.

Proprio per evitare questo problema, è doveroso parlare di una pratica comune di programmazione che suggerisce una diversa metodologia per l'assegnazione dei nomi agli attributi.

La presenza di tecnologie come l'*Intellisense* (che consentono di velocizzare la scrittura del codice "suggerendo" all'utente mentre sta scrivendo quali siano gli attributi e le operazioni definite per una certa classe) consiglia di antecedere al nome di un attributo protetto o privato (l'impostazione di default del linguaggio C#) il carattere di sottolineatura (underscore) "_"⁶, mentre si consiglia di iniziare a scrivere un attributo pubblico con una lettera maiuscola (senza aggiungere il carattere di sottolineatura)

Questa consuetudine porta molti vantaggi come:

- ❑ farà sì che lo si potrà facilmente riconoscere ed utilizzare in fase di digitazione e permette di distinguere chiaramente un attributo privato da uno pubblico
- ❑ eviterà l'uso della parola chiave **this** (il cui significato illustreremo più avanti)
- ❑ impedirà di commettere errori di assegnazione tra attributi della classe e parametri dei costruttori/metodi (nel caso ci si dimenticasse di utilizzare la parola chiave **this**)

Per quanto riguarda gli ultimi due vantaggi, è invece possibile notare i benefici riprendendo l'esempio precedente e modificandolo come segue.

```
Class Persona
{
    string _nome;
    double _peso;
    int _altezza;
    public Persona(string nome, double peso, int altezza)
    {
        _nome = nome;
        _peso = peso;
        _altezza = altezza;
    }
}
```

Anche solo da queste semplici righe è possibile notare facilmente la maggiore espressività del codice dove risulta chiara l'assegnazione del parametro all'attributo senza l'utilizzo della parola chiave **this**.

⁶ Altre scuole di pensiero suggeriscono approcci differenti, come l'aggiunta della lettera minuscola "m" davanti al carattere di sottolineatura

Metodi

I metodi descrivono le operazioni ammissibili sugli oggetti della classe. Benché, come vedremo, sia possibile consentire l'accesso diretto ai campi di un oggetto, di norma ciò avviene sempre attraverso l'invocazione di un metodo (o di altre funzioni membro che saranno esaminate più avanti nel testo).

L'invocazione di un metodo avviene attraverso un oggetto della classe⁷. Ad esempio, data la classe `Persona` precedentemente definita:

```
class Persona
{
    ...
    public bool SeSovrappeso()
    {
        return _peso > (_altezza - 100 + 10);
    }
}

...
static void Main()
{
    Persona persol = new Persona("Oliver Hardy", 107, 180);
    Persona perso2 = new Persona("Stan Laurel ", 67, 180);

    bool sovrappeso = persol.SeSovrappeso();
    if (sovrappeso == true)
        Console.WriteLine("Si consiglia di seguire una dieta!");
    if (perso2.SeSovrappeso())
        Console.WriteLine("Si consiglia di seguire una dieta!");
}
```

l'output del programma sarà:

Si consiglia di seguire una dieta!

`SeSovrappeso()` viene invocato due volte attraverso due oggetti distinti. Il metodo è sempre lo stesso, mentre le variabili su cui opera di volta in volta – peso e altezza – rappresentano copie private e distinte appartenenti ai due oggetti `persol` e `perso2`.

2.8 Rappresentazione dei metodi in UML

A livello di descrizione in linguaggio UML, non si parla di metodi ma di “operazioni”. Un'operazione, infatti, è una indicazione su come invocare un particolare comportamento. UML fa una chiara distinzione tra come invocare un comportamento (operazione) e l'implementazione attuale dello stesso (costruttore o metodo)

Le operazioni, vengono rappresentate nel terzo compartimento della classe utilizzando la seguente notazione:

visibilità nome (parametri) : tipo-di-ritorno {proprietà}

con i parametri scritti come:

direzione nome_parametro : tipo [molteplicità]
 = valore_di_default { proprietà }

⁷ Ciò è vero soltanto per i metodi non statici.

La sintassi degli elementi è espressa dalla seguente tabella

Tabella 1-2 Sintassi UML per le operazioni

TIPO	DESCRIZIONE
Visibilità	Indica la visibilità della operazione ed utilizza i simboli +, - e # rispettivamente per pubblico, privato e protetto
Nome	Nome della operazione che inizia tipicamente con la lettera Maiuscola.
Tipo-di-ritorno	Il tipo del dato che l'operazione ritornerà
Direzione	Elemento opzionale che indica la modalità del passaggio del parametro e può valere in, out ed inout (ref in C#)
Nome_parametro	Nome del parametro che inizia tipicamente con la lettera minuscola
Tipo	Il tipo del parametro
Molteplicità	Indica quante istanze del tipo parametro sono presenti. Mentre l'assenza indica molteplicità 1, si possono specificare valori interi, un intervallo tra parentesi quadrate separate da ".." ed utilizzare il simbolo "*" per indicare "..più"
Valore_di_default	Indica il valore di default del parametro
Proprietà	Indicate tra parentesi graffe indicano ordinamento, univocità, readonly ed altre ancora.

Ecco il diagramma completo in UML della classe `Persona`:

Persona
-nome : string -peso : double -altezza : int
+Persona(in nome : string, in peso : double, in altezza : int) +SeSovrappeso() : bool

Figura 2-5 Descrizione in UML della classe `Persona`.

3 Classi e oggetti

I termini classe e oggetto vengono qualche volta usati in modo indifferenziato, il che può essere accettabile purché si resti consapevoli che i due fanno riferimento a concetti distinti.

Una classe definisce un tipo di dato, descrive cioè le caratteristiche che accomunano tutti i valori appartenenti al tipo suddetto. Un oggetto rappresenta un determinato valore appartenente a una classe. La definizione di una classe non produce in sé alcuna allocazione di memoria; una classe, infatti, rappresenta soltanto una descrizione. Un oggetto, al contrario, occupa memoria secondo una precisa configurazione, definita appunto nella classe di appartenenza.

In una metafora, il rapporto tra classi e oggetti è analogo a quello esistente tra il progetto di una casa e la casa stessa. Il progetto – la classe – descrive le caratteristiche della casa, ma di per sé non è niente di tangibile. La casa – l’oggetto – rappresenta la traduzione in concreto del progetto, che avviene mediante l’operazione di costruzione. Applicando lo stesso progetto si possono costruire quante case si vuole, le quali saranno oggetti distinti, ma strutturalmente identici, poiché originati dallo stesso progetto – dalla stessa classe.

Nel linguaggio della OOP il termine oggetto viene anche tradotto nel termine “**istanza**”. Un oggetto è dunque un’istanza di una classe. L’operazione di creazione di un oggetto è anche definita “**istanziamento**” della classe ed è in pratica l’applicazione della descrizione in essa contenuta per la costruzione dell’oggetto.

Consideriamo nuovamente la classe `Calciatore`, l’espressione:

```
new Calciatore("Filippo Inzaghi", "Milan", "Attaccante");
```

determina la creazione di un’istanza di `Calciatore`, e cioè un oggetto che occupa una determinata area di memoria secondo una configurazione definita nella classe. L’istruzione completa:

```
Calciatore c = new Calciatore("Filippo Inzaghi", "Milan", "Attaccante");
```

assegna alla variabile `c` l’indirizzo dell’area di memoria occupata dall’oggetto; la variabile diventa così un “**riferimento**” all’oggetto. Ogni operazione sull’oggetto avviene mediante la variabile `c`, ma solo dopo che l’oggetto è stato creato e il suo riferimento assegnato alla variabile. Prima dell’istruzione di creazione, la variabile `c` non fa riferimento a nessun oggetto, poiché esso non esiste ancora.

Oggetto e variabile che si riferisce ad esso sono dunque due concetti distinti, anche se per semplicità viene usato il termine variabile in luogo di oggetto e viceversa.

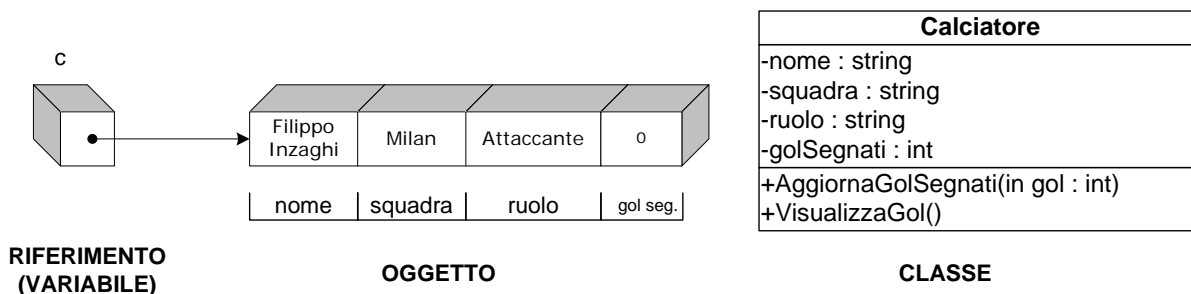


Figura 2-6 Rappresentazione schematica di classe, oggetto (istanza) e riferimento a oggetto.

3.1 “Campo di azione” di classe e “Ciclo di vita” di un oggetto

Entrambi i concetti non sono strettamente collegati alla OOP e riguardano l’intero ambito della programmazione, e precisamente le seguenti domande:

- ❑ in quali parti del programma si può fare riferimento a un nome (variabile, metodo, classe, eccetera) introdotto da una dichiarazione?
- ❑ quando si può usare un oggetto – in generale una variabile – o invocare un metodo senza temere che ciò non sia lecito in quel punto del programma?

Campo di azione di classe

Nel programma ogni nome deve fare riferimento a un’unica entità, ma ciò non significa che non possano esistere due o più entità con lo stesso nome, purché ci sia un contesto che consenta al

compilatore di decidere a quale entità si fa riferimento in quel punto del programma. Ad ogni entità è dunque associato un campo di azione (o “contesto di validità”, o “scopo”, o “spazio di visibilità”) che designa in quale parte (o parti) del programma ci si può riferire ad essa.

La definizione di una classe introduce nel programma un nuovo campo di azione, chiamato “campo di azione di classe”. Ciò fa sì che tutti i nomi utilizzati all’interno del codice della classe siano risolti a favore dei membri di classe, a meno che non si riferiscano a variabili locali, dichiarate all’interno di un metodo o di un blocco. Il seguente esempio aiuterà a chiarire il concetto:

```
class ClasseA
{
    double _x;
    public void Visualizza()
    {
        Console.WriteLine(_x);
    }
}

class ClasseB
{
    double _x;
    public double RitornaX()
    {
        return _x;
    }
}

static void Main()
{
    ClasseA o1 = new ClasseA();
    ClasseB o2 = new ClasseB();
    o1.Visualizza();
    double z = o2.RitornaX();
    ...
}
```

Entrambe le classi, `ClasseA` e `ClasseB`, definiscono un campo di nome `x`. Nel codice esistono dunque due entità che hanno lo stesso nome. Ciò rappresenta una doppia dichiarazione? E se no, in che modo il linguaggio stabilisce a quale variabile fanno riferimento le due istruzioni evidenziate in grigio?

Il concetto di campo d’azione risponde ad entrambe le domande. Si ha una doppia dichiarazione quando si dichiara due volte la stessa entità; ma le due variabili `x` appartengono a classi – e quindi a campi d’azioni – diverse, e dunque non sono la stessa entità! Per lo stesso motivo è semplice stabilire ogni volta a quale variabile faccia riferimento il nome `x`:

fa riferimento alla variabile che appartiene allo stesso campo d’azione che contiene il codice che la usa.

Il metodo `Visualizza()` appartiene alla `ClasseA` e dunque ogni uso del nome `x` all’interno di esso si riferisce alla `x` membro di `ClasseA`. Discorso analogo vale per l’uso di `x` nel metodo `RitornaX()`, definito in `ClasseB`.

Ciclo di vita

Il campo d'azione riguarda gli identificatori in generale e dipende unicamente dalle regole del linguaggio e dal luogo, nel codice sorgente, di dichiarazione di un nome. Il ciclo di vita (*lifetime*) è invece connesso all'esistenza in memoria degli oggetti ed è quindi un aspetto che riguarda la fase di esecuzione del programma.

Nel caso di oggetti appartenenti a tipi valore, come ad esempio variabili `int` o `double`, il ciclo di vita coincide in un certo senso con il loro campo d'azione. Ad esempio:

```
static void MetodoQualsiasi()
{
    double _x;
    // ...uso della variabile _x
}

...
static void Main()
{
    ...
    MetodoQualsiasi();          // inizia il ciclo di vita di _x
    // qui il ciclo di vita della variabile _x è già terminato
}
```

Il campo di azione di `x` è rappresentato dal corpo del metodo. Anche l'esistenza di `x` in memoria è legata al metodo. La variabile viene creata appena il flusso di esecuzione entra nel metodo e viene distrutta prima che il flusso di esecuzione abbandoni il metodo.

Le cose vanno diversamente per un'istanza di classe e per i membri non statici di una classe. Il loro ciclo di vita è determinato esplicitamente dal programmatore e comincia con l'istruzione di creazione dell'oggetto. Consideriamo il seguente codice:

```
class ClasseEsempio
{
    public double X;
}

static ClasseEsempio c;

static void MetodoQualsiasi()
{
    ...
    c.X = 10; // errore: c non referencia ancora alcun oggetto!
    c = new ClasseEsempio();
    c.X = 10; // ok
    ...
}
```

Esso contiene un errore legato al ciclo di vita degli oggetti. Nel metodo viene fatto riferimento al membro (la variabile `x`) di un oggetto che non è stato ancora creato. In sostanza, in quel punto il campo `x` non ha ancora iniziato il proprio ciclo di vita, nonostante il ciclo di vita della variabile `c`, in quanto statica, sia cominciato con il programma. Dunque, la variabile `c` esiste, ma non referencia alcun oggetto!

In conclusione:

il ciclo di vita di un oggetto di classe inizia con l'istruzione di creazione dell'oggetto stesso.

e:

il ciclo di vita di un campo non statico coincide con il ciclo di vita dell'oggetto a cui il campo appartiene.

3.2 Membri statici e non statici (d'istanza)

Nelle classi presentate finora abbiamo assunto che i membri, campi o funzioni, fossero definiti come non statici, e cioè privi del modificatore `static`. I membri non statici sono sempre legati a un particolare oggetto; per questo motivo sono anche chiamati “membri di istanza”, poiché esistono soltanto in relazione a una determinata istanza della classe, della quale condividono il ciclo di vita. Inoltre, se i membri in questione sono dei campi, ogni istanza ne possiede una copia e dunque esistono tante copie dello stesso membro quante sono le istanze della classe.

Le cose stanno in modo diverso se un membro è dichiarato come statico, premettendo la parola chiave `static` nella definizione:

`static` *definizione-membro*

Un membro statico non appartiene a nessun particolare oggetto; esso appartiene alla classe e il suo ciclo di vita coincide con il ciclo di vita del programma. Dunque:

un membro statico esiste a prescindere dalla creazione di una o più istanze della classe.

```
class ClasseEsempio
{
    public static int A;        // membro statico
    public double X;           // membro non statico
}
```

Nel caso dei campi membro, esiste una sola copia del membro statico, a prescindere dal numero di istanze della classe che sono state create. Il seguente codice lo dimostra:

```
static void Main()
{
    ClasseEsempio ogg1 = new ClasseEsempio()
    ClasseEsempio ogg2 = new ClasseEsempio()
    ogg1.x = 10;
    ogg2.x = 20;
    ClasseEsempio.a = 100;
    ...
}
```

La situazione prodotta dal precedente codice è schematizzata dalla figura successiva. Esistono due copie del membro d'istanza `x`, una per ogni oggetto. Esiste invece una sola variabile `a`, che appartiene alla classe.

Il fatto che un membro statico non appartenga a nessuna istanza della classe implica che:

- ❑ non esiste alcun bisogno di istanziare la classe (di creare un oggetto) per fare riferimento ad esso;

- nel codice *consumer*, per accedere al membro è necessario qualificarlo con il nome della classe e non con il nome di un oggetto appartenente ad essa.

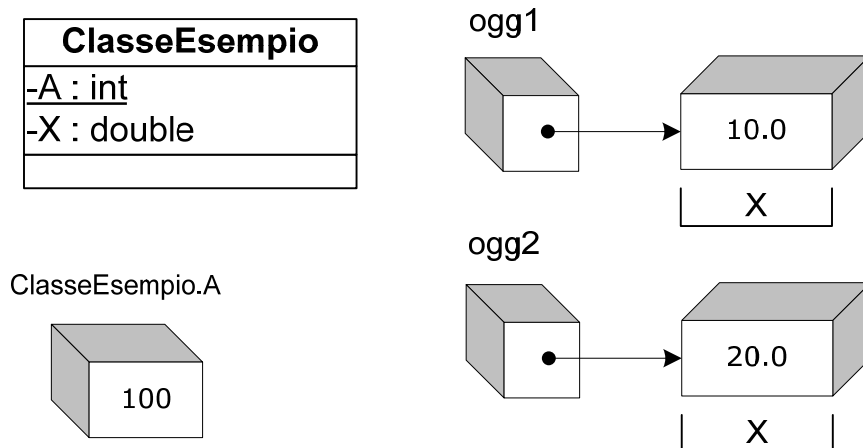


Figura 2-7 Rappresentazione in memoria di campi statici e non statici.

Una questione importante riguarda i metodi statici. Poiché un metodo statico viene invocato a prescindere da una particolare istanza della classe, al suo interno è proibito fare riferimento a un membro d'istanza, in quanto non sarebbe chiaro a quale copia (di quale istanza) farebbe riferimento. Ad esempio:

```
class ClasseEempio
{
    public static int A;          // membro statico
    public double X;             // membro non statico

    static void MetodoStaticoQualsiasi()    // metodo statico
    {
        A = 10;                   // ok: accesso a un campo statico
        X = 10;                   // errore: accesso a un campo di istanza!
    }
}

static void Main()
{
    ClasseEempio.MetodoStaticoQualsiasi();
}
```

Il codice è errato, poiché il campo non statico *x* può esistere soltanto in relazione a un determinato oggetto della classe. D'altra parte, *MetodoStaticoQualsiasi()* viene invocato attraverso il nome della classe e non di un oggetto. Al suo interno, quindi, fa riferimento a una variabile che non esiste affatto!

3.3 Rappresentazione di membri statici in UML

All'interno della simbologia UML, i membri elementi statici vengono distinti dagli altri mediante la loro sottolineatura, come mostrato di seguito:

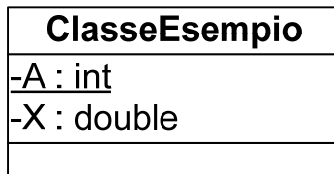


Figura 2-8 Rappresentazione in UML di un attributo statico e non statico.

In figura, A rappresenta un campo di classe statico, mentre x un campo di classe di istanza.

3.4 Classi statiche

A partire dal .Net 2.0 è possibile definire un'intera classe statica aggiungendo la parola chiave **static** nella dichiarazione della classe.

Questo tipo di classe si implementa quando la classe deve contenere metodi che non necessitano di un'istanza e non sono associati ad uno specifico oggetto.

Le caratteristiche di una classe statica sono:

- ❑ Può contenere solo membri statici
- ❑ Non può essere istanziata.
- ❑ Non può essere ereditata (sealed)
- ❑ Non può contenere dei costruttori.

In realtà era possibile costruire una classe statica anche in .NET 1.x, perché in definitiva bastava definire la classe solo con metodi statici e costruttori privati, ma il vantaggio con il .NET 2.0 sta nel compilatore che effettua un controllo sulla istanziabilità della classe.

3.5 Costruttori delle classi statiche

Anche se non è possibile istanziare una classe statica è comunque possibile fornire un costruttore statico per svolgere assegnazioni preliminari all'oggetto.

Un costruttore statico differisce da uno "normale" per le seguenti caratteristiche

- ❑ Non può avere né modificatori né accettare parametri
- ❑ È chiamato automaticamente per inizializzare la classe prima che l'istanza sia stata creata o di aver referenziato un membro statico
- ❑ Non può essere chiamato automaticamente

Oltre a queste limitazioni è bene sapere che il client non ha modo di sapere quando il costruttore viene invocato durante l'esecuzione del programma.

Quello che segue è un semplice esempio di una classe statica.

```
static class InformazioniAziendali
{
    // ....

    static public string GetNomeAzienda()
    {
        return "EduDotnet";
    }
}
```

```
static public string GetNomeSito()
{
    return "www.edudotnet.it";
}
// ....
```

4 Livello di accesso ai membri di una classe

Uno degli aspetti fondamentali della OOP è l'incapsulamento e cioè la capacità di una classe di nascondere e dunque rendere inaccessibile la propria rappresentazione interna. Ciò implica che il codice *consumer* non dovrebbe conoscere di un oggetto più di quanto sia strettamente necessario perché possa usarlo.

Nella definizione di una classe, mediante delle parole chiave chiamate “modificatori di accesso”, è possibile specificare per ogni membro a quale livello sia accessibile da parte del codice *consumer*. Esistono cinque livelli distinti, dei quali per il momento ne saranno approfonditi soltanto due.

4.1 Modificatori del livello di accesso (o protezione)

La definizione di un membro di classe (campo o metodo) può essere preceduta da uno dei seguenti modificatori del livello accesso, che stabiliscono a quali restrizioni è soggetto l'accesso al membro in questione:

Tabella 1-1. Elenco dei modificatori di accesso.

MODIFICATORE DI ACCESSO	DESCRIZIONE
private (modificatore di default)	Il membro è inaccessibile al di fuori del campo di azione della classe: soltanto nelle funzioni membro è possibile fare riferimento ad esso.
public	Il membro è accessibile ovunque.
protected	Il membro è accessibile soltanto nelle classi derivate. Per il resto il suo livello di accesso equivale a <i>private</i> .
internal	Il membro è accessibile ovunque all'interno dei file sorgenti che compongono il programma (in generale l'assembly). Nell'ambito del programma, dunque, il membro è come se fosse <i>public</i> .
protected internal	Il membro è accessibile sia nelle classi derivate che nei file sorgenti del programma (in generale dell'assembly).

L'uso dei modificatori assume la seguente sintassi:

modificatore_{opz} definizione-membro

In questa fase ci occuperemo soltanto dei primi due modificatori di accesso: `private` e `public`.

Livello di accesso "private"

Il modificatore `private` protegge un membro da qualsiasi tentativo di accesso da parte di codice *consumer*. Nel caso di campi membro il motivo principale per renderli privati è impedire che possano essere modificati da codice esterno alla classe. Ad esempio, il seguente codice definisce una classe con due campi e un metodo privati:

```
class ClasseEsempio
{
    private int _a;
    private string _s;
    private void MetodoQualsiasi()
    {
        ...
    }
}
```

Nel codice *consumer*, il tentativo di accesso ai membri in questione produce un errore di compilazione:

"ClasseEsempio.<nome membro> è inaccessibile a causa del livello di protezione"

Ad esempio:

```
static void Main()
{
    ClasseEsempio o = new ClasseEsempio();
    o._a = 10;           // errore: _a è inaccessibile!
    string nome = o._s;  // errore: _s è inaccessibile!
    o.MetodoQualsiasi(); // errore: MetodoQualsiasi() è inaccessibile!
}
```

Se la definizione di un membro non è preceduta da alcun modificatore di accesso, viene assunto il modificatore `private`. Detto ciò, scrivere:

```
class ClasseEsempio
{
    private int _a;    // private specificato in modo esplicito
}
```

equivale a scrivere:

```
class ClasseEsempio
{
    int _a;           // private assunto implicitamente
}
```

Livello di accesso "public"

Il modificatore `public`, contrariamente a `private`, non pone alcuna restrizione sull'accesso ai membri. Ad esempio, la seguente classe definisce due campi e un metodo pubblici:

```
class ClasseEsempio
```

```

{
    public int A;
    public string S;
    public void MetodoQualsiasi()
    {
        ...
    }
}

static void Main()
{
    ClasseEsempio o = new ClasseEsempio();
    o.A = 10; // ok: assegna 10 al campo A
    string nome = o.S; // ok: usa il campo S
    o.MetodoQualsiasi(); // ok: invoca il metodo: MetodoQualsiasi()
}

```

Il modificatore **public** dev'essere specificato in modo esplicito per ogni membro, o lista di membri, che si vuole rendere pubblico. Ad esempio:

```

class ClasseEsempio
{
    public int A, B; // campi pubblici
    string _s; // campo privato
    void MetodoQualsiasi() // metodo privato
    {
        ...
    }
}

```

4.2 Interfaccia pubblica e implementazione di una classe

Esistono due elementi, connessi all'incapsulamento e all'uso dei modificatori di accesso, che caratterizzano la progettazione di una classe: l' "interfaccia pubblica" e l' "implementazione" della classe. Entrambi sono concetti fondamentali della OOP e saranno presi in considerazione più volte nel testo. Il termine "interfaccia pubblica" designa:

l'insieme dei membri dichiarati **public e dunque accessibili al codice *consumer*.**

L'interfaccia pubblica è rappresentata dunque dalle funzionalità che la classe "espone" al codice esterno e che sono da esso utilizzabili. Il termine "implementazione" designa invece:

la struttura e il funzionamento effettivo della classe.

L'interfaccia pubblica si riferisce al "che cosa" la classe è in grado di fare e rappresentare, laddove l'implementazione fa riferimento al "come" la classe memorizza effettivamente i dati ed esegue effettivamente le operazioni su di essi. L'interfaccia pubblica è la facciata, ciò che appare della classe, l'implementazione è ciò che vi nasconde dietro.

Al limite, interfaccia pubblica e implementazione possono sovrapporsi, come nel seguente esempio:

```

class ClasseEsempio1
{

```

```
    public string Nome;  
    public string Cognome;  
}
```

In questo caso, la struttura della classe è completamente esposta al codice *consumer*, il quale può accedere direttamente ad essa:

```
ClasseEsempio1 persona = new ClasseEsempio1();  
persona.Nome = "Giulio";  
persona.Cognome = "Cesare";
```

Per contro, una classe può fornire di sé una rappresentazione diversa da quella effettiva. In questo caso ciò che il codice *consumer* “vede” della classe (ciò che può usare) non equivale a come la classe è effettivamente. Si consideri il seguente esempio:

```
class ClasseEsempio2  
{  
    private string _nome;  
    private string _cognome;  
    public ClasseEsempio2(string nominativo)  
    {  
        string[] s = string.Split(nominativo, ' ');  
        _nome = s[0];  
        _cognome = s[1];  
    }  
    public string ComeTiChiami()  
    {  
        return _nome + " " + _cognome;  
    }  
}
```

Come si vede, la struttura interna della classe non è esposta al codice *consumer*, il quale può interagire con gli oggetti della classe soltanto attraverso le funzioni che essa espone, il costruttore e il metodo `ComeTiChiami()`, e cioè attraverso l’interfaccia pubblica. Ad esempio:

```
ClasseEsempio2 persona = new ClasseEsempio2("Giulio Cesare");  
string nome = persona.ComeTiChiami();  
Console.WriteLine(nome);
```

Ciò semplifica l’uso di oggetti della classe, poiché il codice *consumer* fa riferimento alla sola interfaccia pubblica e dunque non viene influenzato dall’effettiva implementazione, la quale può essere modificata senza che ciò renda necessario modificare anche il codice *consumer*.

Nel progettare una classe occorre dunque operare una netta distinzione tra:

- ❑ ciò che si vuole esporre all’esterno, e cioè la sua interfaccia pubblica;
- ❑ il modo in cui si intende implementarla.

Il primo punto è senz’altro il più importante, poiché qualsiasi successiva modifica all’interfaccia pubblica della classe provoca la necessità di aggiornare tutto il codice *consumer* che la utilizza.

4.3 Stabilire il livello di accesso dell'intera classe

Il linguaggio C# consente di stabilire il livello di accesso anche relativamente all'intera classe. Nell'intestazione è infatti possibile specificare il suo livello di accesso mediante i modificatori precedentemente introdotti:

```

    modificatoreopz class
    {
        // corpo della classe
    }

```

L'uso sulle classi dei modificatori del livello di accesso sottostà a diverse regole, che dipendono dal fatto che la classe sia dichiarata a livello di file o interamente a un'altra classe. Poiché di norma è necessario intervenire sul livello di accesso di classe soltanto quando si scrivono librerie di componenti da usare in altri programmi oppure in relazione a progetti abbastanza complessi, questo argomento viene trattato in appendice. (“Livello di accesso ai tipo, tipi nidificati e *namespaces*”)

5 Metodi

I metodi implementano le operazioni che agiscono sugli oggetti della classe, consentendo al codice *consumer* di accedere ad essi ed eventualmente di modificarli. Il codice contenuto in un metodo può accedere liberamente a tutti gli altri membri dell'oggetto, indipendentemente dal loro livello di accesso⁸.

La definizione di un metodo assume la seguente forma:

```

    modificatoriopz tipo nome-metodo(lista-parametriopz)
    {
        // corpo del metodo
    }

```

Nel seguente esempio, la classe `Rettangolo` definisce due metodi, uno per il calcolo dell'area, l'altro per il calcolo del perimetro:

```

class Rettangolo
{
    double _b;
    double _h;

    public void ImpostaDimensioni(double b, double h)
    {
        _b = b;
        _h = h;
    }
    public double CalcolaArea()
    {
        return _b * _h;
    }
    public double CalcolaPerimetro()
    {

```

⁸ Esiste un'eccezione, rappresentata dai costruttori della classe, i quali non possono essere invocati esplicitamente.

```

        return (_b + _h) * 2;
    }

}

```

Segue il diagramma UML della classe:

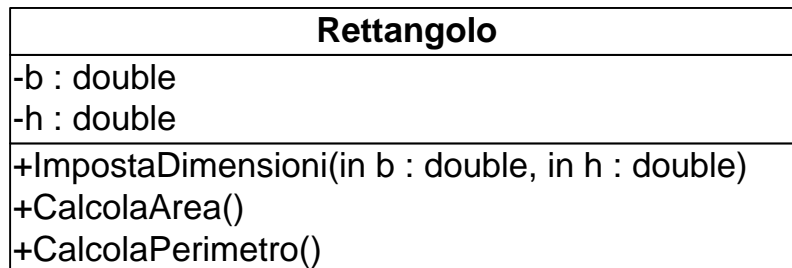


Figura 2-9 Diagramma UML della classe Rettangolo.

Ecco infine, un frammento di codice che usa la classe:

```

static void Main()
{
    Rettangolo ret = new Rettangolo();
    ret.ImpostaDimensioni(10, 20);
    double area = ret.CalcolaArea();
    double perim = ret.CalcolaPerimetro();
    ...
}

```

Non esistono restrizioni sul tipo di codice che può essere contenuto in un metodo e in teoria esso potrebbe svolgere operazioni che non possiedono alcuna relazione con l'oggetto a cui appartiene. L'unica limitazione riguarda i metodi statici, all'interno dei quali non può essere fatto riferimento ai membri non statici della classe.

5.1 Metodi di accesso

Dal punto di vista del ruolo che rivestono, i metodi dell'interfaccia pubblica possono dividersi in due categorie:

- ❑ quelli che implementano un comportamento dell'oggetto e dunque svolgono una determinata elaborazione;
- ❑ quelli che forniscono un accesso ai campi membro; questi ultimi vengono chiamati “metodi di accesso”.

L'accesso ai campi membro viene di norma gestito in modo individuale e per ognuno di essi può essere fornito uno o due metodi, in base al fatto che s'intenda fornire un accesso in sola lettura, in sola scrittura o in entrambi i modi.

Ad esempio, la seguente classe definisce tre metodi di accesso: due relativi al campo `nome`, sia in lettura che in scrittura, ed uno relativo al campo `password`, in sola scrittura:

```

class Utente
{
    string _nome;

```



```

string _password;

public Utente(string nome, string password)
{
    _nome = nome;
    _password = password;
}

public string GetNome()
{
    return _nome;
}

public void SetNome(string nome)
{
    _nome = nome;
}

public string SetPassword(string password)
{
    _password = password;
}
}

```

Segue il diagramma UML della classe:

Utente
-nome : string -password : string
+SetNome(in nome : string) +GetNome() : string +SetPassword(in password : string)

Figura 2-10 Diagramma UML della classe Utente.

Nota bene: nella denominazione dei metodi sono stati usati i suffissi “Get” e “Set” per rendere più chiaro quali metodi leggono i campi e metodi li modificano. E’ soltanto una convenzione, ma aiuta a scrivere codice più facile da comprendere.

5.2 *Overloading* di metodi

Il termine inglese *overloading* si può tradurre nell’italiano “sovraccaricare” ed indica che:

più metodi, i quali svolgono presumibilmente operazioni analoghe su tipi diversi di dati, possono condividere lo stesso nome.

Dunque, una classe può definire più versioni dello stesso metodo, purché ognuna si differenzi dalle altre per il numero e/o il tipo dei parametri.

La programmazione in C# (ed in generale in .NET) fa un uso intenso dell’*overloading*. Ad esempio, il metodo `WriteLine()` della classe `Console` è sovraccaricato 19 volte; ciò significa che

la classe `Console` definisce in realtà 19 metodi di nome `WriteLine()`, ognuno dei quali presenta un prototipo diverso dagli altri per almeno un parametro.

L'*overloading* è utile quando si desidera implementare un procedimento che è sensibile al tipo o al numero dei dati da elaborare. Come esempio, realizziamo una nuova versione della classe `Rettangolo`, la cui rappresentazione interna è questa volta fornita dalle quattro coordinate nel piano cartesiano. Come parte dell'interfaccia pubblica si vuole dare la possibilità di impostare le dimensioni del rettangolo sia specificando le quattro coordinate sia specificando le coordinate dell'angolo in alto a sinistra e una dimensione, ottenendo come risultato un quadrato. In sostanza occorrono due metodi `ImpostaDimensioni()`.

```
class Rettangolo
{
    int _x1, _y1;
    int _x2, _y2;

    public void ImpostaDimensioni(int x1, int y1, int x2, int y2)
    {
        _x1 = x1;
        _y1 = y1;
        _x2 = x2;
        _y2 = y2;
    }

    public void ImpostaDimensioni(int x1, int y1, int lato)
    {
        _x1 = x1;
        _y1 = y1;
        _x2 = x1 + lato;
        _y2 = y1 + lato;
    }
}
```

Segue il diagramma UML della classe:

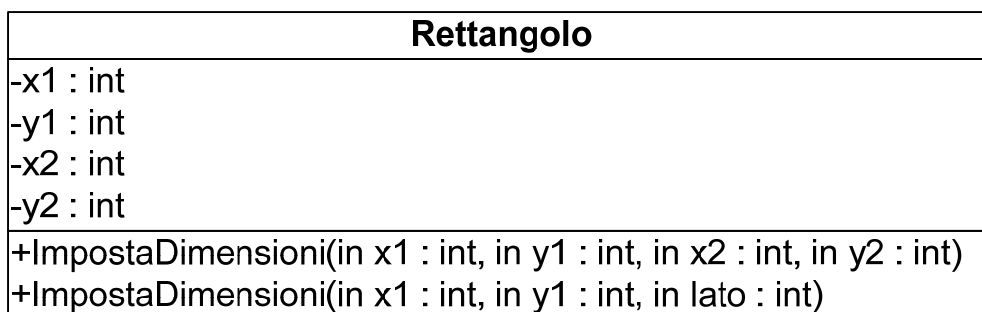


Figura 2-11 Diagramma UML della classe `Rettangolo`.

I due metodi hanno lo stesso nome, ma si differenziano per la lista dei parametri; dunque possono essere considerati come un unico metodo sovraccaricato.

Segue un frammento di codice che ne mostra l'impiego:

```
static void Main()
{
    Rettangolo retA = new Rettangolo();
```

```

    Rettangolo retB = new Rettangolo();
    retA.ImpostaDimensioni(10, 10, 15, 15);
    retB.ImpostaDimensioni(10, 10, 5);
    ...
}

```

In assenza di *overloading* saremmo stati costretti a stabilire due nomi diversi per lo stesso tipo di operazione, che è quella di stabilire le dimensioni del rettangolo.

Un aspetto importante dell'*overloading* è dato dal fatto che è possibile invocare un metodo sovraccaricato all'interno di un'altra versione dello stesso metodo, poiché i due sono a tutti gli effetti metodi distinti, i quali condividono soltanto il nome. Ad esempio, la seconda versione del metodo `ImpostaDimensioni()` rappresenta in realtà una specializzazione della prima versione:

```

class Rettangolo
{
    ...
    public void ImpostaDimensioni(int x1, int y1, int lato)
    {
        ImpostaDimensioni(x1, y1, x1+lato, y1+lato); // invoca prima versione
    }
}

```

Questo approccio, al costo di un leggero calo di performance, evita duplicazioni di codice, cosa che dovrebbe essere sempre evitata.

5.3 “Risoluzione” di un metodo sovraccaricato

Nella tradurre un'istruzione che contiene l'invocazione di un metodo, la “risoluzione” è un'operazione svolta dal compilatore e stabilisce, in presenza di *overloading*, qual è effettivamente il metodo da invocare. Essa sfrutta le differenze nei prototipi dei metodi e cerca una corrispondenza tra gli argomenti specificati nelle istruzioni di invocazione e le liste di parametri specificate nelle definizioni dei metodi.

Le regole che stabiliscono i criteri di risoluzione del nome sono abbastanza complesse e non saranno qui approfondite. Nella maggior parte dei casi è sufficiente ricordare che:

nell'operazione di risoluzione vengono messe in corrispondenza soltanto la lista di argomenti e le liste di parametri dei metodi sovraccaricati; non viene considerato il tipo di ritorno.

Se il compilatore non riesce a stabilire alcuna corrispondenza, oppure trova più corrispondenze altrettanto valide (nessuna migliore delle altre), segnala un errore formale. Il seguente esempio mostra un caso in cui due metodi condividono non soltanto il nome ma anche la lista dei parametri; ciò rappresenta un errore, anche se hanno un diverso tipo di ritorno:

```

class ClasseEsempio
{
    //...
    public void MetodoQualsiasi(int x)
    {
        ...
    }

    public int MetodoQualsiasi(int i)           // errore: metodo ridefinito!
}

```

```
{  
    return 0;  
}  
}
```

Come dimostra l'esempio, nel valutare le varie liste di parametri viene preso in considerazione il numero e il tipo, ma non il nome, che è del tutto irrilevante.

5.4 Overloading di operatori

Oltre che sovraccaricare un metodo, il .NET consente anche di fare la stessa cosa con gli operatori.

L'overloading degli operatori, permette di specificare l'implementazione di operatori dove uno più operandi sono uno tipo struttura o classe definito dal programmatore.

Per chiarire meglio il concetto, scriviamo un classico esempio di utilizzo dell'overloading degli operatori, definendo una classe per la rappresentazione di numeri complessi.

```
public class Complesso  
{  
    int _reale;  
    int _immag;  
  
    public Complesso(int reale, int immaginaria)  
    {  
        _reale = reale;  
        _immag = immaginaria;  
    }  
}
```

Fin qua nulla di nuovo, ma quando passeremmo a scrivere il codice client ci piacerebbe poter fare qualcosa di questo tipo

```
static void Main(string[] args)  
{  
    Complesso c1 = new Complesso(1, 2);  
    Complesso c2 = new Complesso(3, 7);  
    Complesso somma = c1 + c2;  
    Complesso oppos = -c2;  
    // ...  
}
```

poter utilizzare, cioè, gli operatori “+” e “-” tra numeri complessi come se si fa normalmente con gli altri tipi numerici.

Per fare ciò, si dovrà procedere ad implementare all'interno di questo particolare metodo la matematica che sta alla base della somma e della negazione tra numeri complessi.

Sovraccaricare un operatore significa quindi scrivere un metodo statico che restituisca il tipo di dato della classe definita dall'utente indicando dopo la parola chiave `operator` l'operatore da ridefinire e la lista dei parametri che varia a seconda che si stia ridefinendo un operatore unario, binario o ternario.

Nel nostro esempio, dovendo ridefinire l'operatore di somma (binario) e quello di negazione (unario), scriveremmo il seguente codice

```
public static Complesso operator +(Complesso x, Complesso y)  
{  
    return new Complesso(x._reale + y._reale, x._immag + y._immag);  
}
```

```

}

public static Complesso operator -(Complesso x)
{
    return new Complesso(x._reale, x._immag);
}

```

E' bene infine ricordare che quando si sovraccarica un operatore aritmetico binario, è automaticamente sovraccaricato anche il corrispondente operatore di assegnazione. Per esempio se sovraccarichiamo l'operatore "+" implicitamente viene sovraccaricato anche l'operatore "+=".

6 Costruttori

I costruttori sono delle particolari funzioni membro invocate nel momento in cui viene creata un'istanza della classe. Per definizione:

un costruttore è sempre la prima funzione membro che viene invocata su un oggetto della classe.

Un costruttore ha sempre il nome della classe e non dichiara un tipo di ritorno:

```

modificatoriopz nome-classe(lista-parametriopz)
{
    // corpo del costruttore
}

```

Lo scopo di un costruttore è normalmente quello di inizializzare l'oggetto e cioè di assegnare dei valori iniziali ai suoi campi membro e di svolgere altre operazioni preliminari prima che esso venga utilizzato. Quando un oggetto viene creato, e prima che venga invocato il costruttore, tutti i suoi campi membro sono inizializzati ai loro valori predefiniti, oppure ai valori specificati negli eventuali inizializzatori.

Nel codice sottostante, la classe `Dipendente` definisce un costruttore che accetta argomenti per i campi `_nome` e `_codice`; il campo `_ore` viene invece lasciato invariato al suo valore di default e cioè zero.

```

class Dipendente
{
    string _nome;
    string _codice;
    int _ore;

    public Dipendente(string nome, string codice)
    {
        _nome = nome;
        _codice = codice;
    }

    public void Visualizza()
    {
        Console.WriteLine("Nome: {0}\tCod.: {1}\tOre: {2}", _nome, _codice, _ore);
    }
}

```

```

    }
}

```

Segue il diagramma UML della classe:

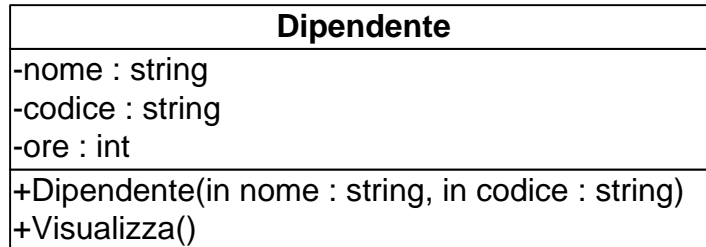


Figura 2-12 Diagramma UML della classe Dipendente.

Un codice *consumer* della classe potrebbe essere il seguente:

```

Dipendente dip = new Dipendente("Paolo Rossi", "10010");
dip.Visualizza();

```

che produce in output:

```

Nome: Paolo Rossi      Codice: 10010      Ore: 0

```

6.1 Codice ammissibile all'interno di un costruttore

In questo senso un costruttore equivale a qualsiasi altro metodo e dunque non pone restrizioni sul codice che può contenere. Nell'esempio seguente, al costruttore della classe `Dipendente` è stata aggiunta una chiamata al metodo `WriteLine()` e due chiamate al metodo `Visualizza()`, una prima e una dopo aver inizializzato i dati.

```

class Dipendente
{
    ...
    public Dipendente(string nome, string codice)
    {
        Console.WriteLine("Sono dentro il costruttore");
        Visualizza();
        this.nome = nome;
        this.codice = codice;
        Visualizza();
    }
}

```

Il codice *consumer* che segue:

```

Dipendente dip = new Dipendente("Paolo Rossi", "10010");

```

produce in output:

```

Sono dentro il costruttore
Nome:      Codice: 0      Ore: 0
Nome: Paolo Rossi      Codice: 10010      Ore: 0

```

6.2 Overloading di costruttori

Definire più costruttori è un'operazione abbastanza comune, poiché fornisce al codice *consumer* una maggiore flessibilità nella creazione dell'oggetto. Anche in questo caso si applicano le regole sulla risoluzione delle funzioni membro sovraccaricate.

Il seguente codice aggiunge due costruttori alla classe `Rettangolo`, i quali si limitano a invocare i metodi `ImpostaDimensioni()`:

```
class Rettangolo
{
    int _x1, _y1, _x2, _y2;

    public Rettangolo(int x1, int y1, int x2, int y2)
    {
        ImpostaDimensioni(x1, y1, x2, y2);
    }
    public Rettangolo(int x1, int y1, int lato)
    {
        ImpostaDimensioni(x1, y1, lato);
    }
    ...
}
```

Segue il diagramma UML della classe:

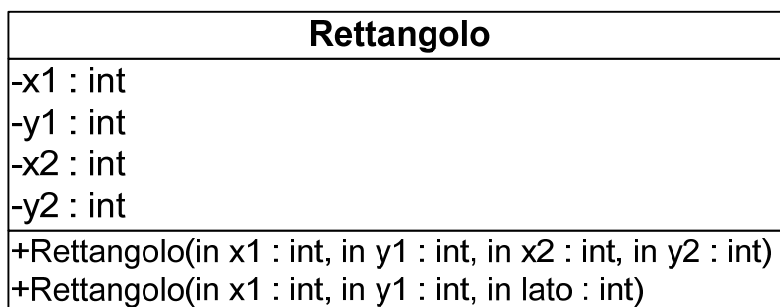


Figura 2-13 Diagramma UML della classe `Rettangolo`.

Dopo questa modifica è possibile specificare le dimensioni di un oggetto `Rettangolo` già in fase di creazione.

```
static void Main()
{
    Rettangolo recA = new Rettangolo(10, 10, 15, 15);
    Rettangolo recB = new Rettangolo(10, 10, 5);
    ...
}
```

6.3 Inizializzatori di costruttore

Nell'*overloading* dei costruttori si pone un problema che è quello dell'invocazione di un costruttore da un altro costruttore, ciò perché la chiamata esplicita ad un costruttore non è ammessa. A questo scopo, il linguaggio consente di specificare un "inizializzatore di costruttore" (*constructor-initializer*) nel prototipo del costruttore chiamante.

Un inizializzatore di costruttore può assumere due forme, delle quali in questo momento esamineremo soltanto la prima:

```

    modificatoriopz nome-classe(lista-parametriopz): this(lista-argomentiopz)
    {
        // corpo del costruttore
    }

```

L'inizializzatore segue il carattere due-punti ed è composto dalla parola chiave `this` seguita dalla lista degli argomenti da passare al costruttore invocato. In questo caso la parola `this` funge da sinonimo del costruttore chiamato. Naturalmente, perché tutto ciò possa funzionare deve esistere un costruttore la cui lista di parametri corrisponda alla lista d'argomenti specificata dopo `this`.

Nel seguente esempio, un costruttore della classe `Rettangolo` invoca l'altro, specificando un inizializzatore di costruttore nella propria intestazione:

```

class Rettangolo
{
    int _x1, _y1, _x2, _y2;

    public Rettangolo(int x1, int y1, int x2, int y2)
    {
        ImpostaDimensione(x1, y1, x2, y2);
    }

    public Rettangolo(int x1, int y1, int lato): this (x1, y1, x1+lato, y1+lato)
    {
    }
}

```

In questo caso, l'inizializzatore:

```
this(x1, y1, x1+lato, y1+lato)
```

viene risolto come una chiamata al primo costruttore, poiché gli argomenti specificati corrispondono esattamente nel numero e nel tipo con il costruttore in questione.

Nel seguente codice *consumer*:

```
Rettangolo r = new Rettangolo(10, 10, 200); // invocato il secondo costruttore
```

l'invocazione del secondo costruttore provoca le seguenti azioni:

- ❑ il controllo, attraverso l'inizializzatore di costruttore, passa immediatamente al primo costruttore;
- ❑ dopo che è terminata l'esecuzione del primo costruttore, comincia l'esecuzione del codice contenuto nel secondo costruttore (in questo caso assente).

Quando possibile, si dovrebbe sempre utilizzare questo approccio definendo un costruttore detto “**Master**” che abbia il maggior numero di parametri di inizializzazione. Seguire questa metodologia evita duplicazioni di codice, le quali rappresentano sempre delle possibili fonti di errore.

6.4 Costruttore di default

Il “costruttore di default”, o “costruttore predefinito”, è un costruttore che non accetta parametri e che dunque si presenta nella forma:


```

    modificatoriopz nome-classe(): this(lista-argomentiopz)opz
    {
        // corpo del costruttore
    }

```

L'esistenza di un costruttore di default fornisce al codice *consumer* la possibilità di creare degli oggetti i cui campi membro abbiano dei valori iniziali predefiniti. Per questo motivo esso:

è fornito automaticamente dal linguaggio nel caso in cui la classe non definisca alcun costruttore.

Ciò rende lecito un codice simile al seguente:

```

class ClasseSenzaCostruttori
{
    int _campoQualsiasi;
    ...
}

static void Main()
{
    ClasseSenzaCostruttori c = new ClasseSenzaCostruttori();
}

```

Il costruttore di default fornito dal linguaggio non svolge in pratica alcun lavoro. Ovviamente, se si desidera che i valori iniziali dei campi membro siano diversi da quelli predefiniti, è possibile definire un proprio costruttore di default (oppure usare degli inizializzatori nelle dichiarazioni dei campi). Ad esempio:

```

class ClasseConCostruttore
{
    int _campoQualsiasi;
    public ClasseConCostruttore()
    {
        _campoQualsiasi = -1; // valore iniziale
    }
}

```

Le regole del linguaggio stabiliscono che il costruttore di default viene fornito soltanto se la classe non definisce alcun costruttore, compreso un proprio costruttore di default. Ciò significa che se viene definito un costruttore che accetta dei parametri, questo sarà l'unico a poter essere invocato. Nel seguente codice, *ClasseEsempio* definisce un costruttore con parametri e dunque il linguaggio non fornisce alla classe un costruttore di default:

```

class ClasseEsempio
{
    int _campoQualsiasi;
    public ClasseEsempio(int campoQualsiasi)
    {
        _campoQualsiasi = campoQualsiasi;
    }
}

```

```
static void Main()  
{  
    ClasseEsempio c1 = new ClasseEsempio();    // errore!  
    ClasseEsempio c2 = new ClasseEsempio(100); // ok  
}
```

L'istruzione evidenziata produce un errore di compilazione, poiché `ClasseEsempio` non definisce un costruttore senza parametri e il linguaggio non gliene fornisce uno.

Questa regola ha un senso, poiché il programmatore potrebbe decidere che non esistano dei valori predefiniti appropriati per uno o più (o tutti) i campi membro e dunque non definendo un costruttore di default “costringe” il codice *consumer* a specificare per essi dei valori iniziali mediante l'invocazione del costruttore con parametri.

D'altra parte, pur fornendo almeno un costruttore con parametri, si potrebbe decidere che i valori predefiniti dei campi membro forniscano all'oggetto uno stato iniziale accettabile. In questo caso si dovrebbe dare al codice *consumer* la possibilità di istanziare la classe invocando il costruttore di default. Poiché però questo non viene fornito automaticamente dal linguaggio, occorre definirne uno, al limite basta anche un costruttore vuoto.

7 Il tipo “struttura”

Praticamente tutti i linguaggi di programmazione dispongono di un tipo “aggregato”, il quale consente di unire in un unico oggetto dati di natura diversa. Nel linguaggio C# ciò (e molto di più) può essere ottenuto attraverso le classi, ma anche con l'impiego dei tipi struttura, caratterizzati dalla parola chiave `struct`.

Diversamente da altri linguaggi, il C# fornisce ai tipi struttura molte delle potenzialità tipiche di un linguaggio orientato agli oggetti, tanto che essi condividono con le classi una sintassi e una modalità di impiego molto simili.

7.1 Tipi struttura come semplici aggregati

Nel loro impiego più semplice, i tipi struttura consentono di definire degli aggregati di dati; un esempio è stato fornito nel primo capitolo:

```
struct Dipendente  
{  
    public string Nominativo;  
    public string Codice;  
    public int Ore;  
}
```

In questo senso, con l'uso di una classe si può ottenere un risultato in apparenza molto simile:

```
class ClasseDipendente  
{  
    public string Nominativo;  
    public string Codice;  
    public int Ore;  
}
```

ma con delle fondamentali differenze. Una classe rivela la propria natura in quanto richiede espressamente di essere istanziata. In altre parole, la creazione di un oggetto richiede l'invocazione del costruttore, che in questo caso è il costruttore di default fornito dal linguaggio:

```
ClasseDipendente dipCla = new ClasseDipendente(); // istanziazione classe
dipCla.Nominativo = "Bianchi Angelo";
```

Non è necessariamente così per una variabile struttura, la cui semplice dichiarazione determina l'allocazione della memoria necessaria:

```
Dipendente dipStru;
dipStru.Nominativo = "Rossi Franco";
```

In sostanza, non esiste alcuna fase di costruzione dell'oggetto.

Una seconda differenza, strettamente connessa alla precedente, riguarda il modello di memorizzazione. Il tipo struttura è un tipo valore, esattamente come i tipi `int`, `double`, `char`, eccetera, laddove le classi sono tipi riferimento.

Riferendoci al precedente codice, la rappresentazione in memoria della variabile struttura e dell'istanza di classe è la seguente⁹:

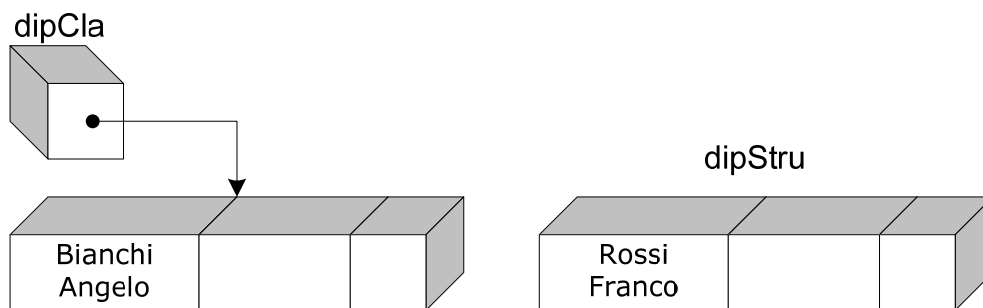


Figura 2-14 Rappresentazione in memoria di riferimento a oggetto e variabile struttura.

7.2 Tipi struttura come oggetti

Sostanzialmente, tutto ciò che è stato detto finora sulle classi (e parte di ciò che sarà detto nei capitoli successivi) può essere applicato ai tipi struttura. Essi dunque:

- ❑ possono definire dei costruttori e dei metodi;
- ❑ possono specificare il livello di accesso ai membri.

Ad esempio, ecco una versione del tipo `Rettangolo` implementato come struttura:

```
struct Rettangolo
{
    int x1, y1, x2, y2;
    public Rettangolo(int x1, int y1, int x2, int y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
}
```

⁹ Per un approfondimento sul modello di memorizzazione dei tipi riferimento e valore vedere l'appendice "Modelli di memorizzazione, Stack, Heap e Garbage Collection"

```

    public Rettangolo(int x1, int y1, int lato): this (x1, y1, x1+lato, y1+lato)
    {
    }
    void ImpostaDimensione(int x1, int x2, int y1, int y2)
    {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
}

```

L'impiego di un oggetto di tipo `Rettangolo` assume gli stessi connotati visti per gli oggetti appartenenti a classi:

```

Rettangolo rect1 = new Rettangolo(10, 10, 15, 15);
Rettangolo rect2 = new Rettangolo(10, 10, 5);
...

```

Detto questo, esistono comunque importanti differenze tra classi e strutture, sia nella definizione che nell'impiego.

Creazione di un oggetto struttura

Benché `Rettangolo` fornisca addirittura due costruttori, non è obbligatorio invocare un costruttore per creare un oggetto struttura, poiché ciò viene fatto automaticamente in fase di dichiarazione. Il seguente codice è infatti lecito:

```

Rettangolo rect;      // dichiara e crea l'oggetto
rect.ImpostaDimensione(50, 50, 100, 100);

```

Costruttore di default

Per definizione, i tipi struttura possiedono un costruttore di default (senza parametri) fornito dal linguaggio. Ebbene, diversamente da quanto avviene con le classi, non è ammesso definire un proprio costruttore di default. Dunque, il seguente codice è formalmente sbagliato:

```

struct StrutturaEsempio
{

```

```

    int a;

```

```

    public StrutturaEsempio()      // errore formale!
    {
        a = 0;
    }

```

```

}

```

Stato iniziale dei campi membro di variabili struttura locali

I valori iniziali dei campi di una variabile struttura dichiarata localmente sono indefiniti e dunque si trovano nello stato “non assegnato”. Questo stato rimane tale fintantoché non viene invocato un costruttore, il cui compito è appunto quello di fornire un valore iniziale ai campi membro. Ciò detto, dato il seguente tipo struttura:

```

struct StrutturaEsempio
{

```

```
public int a;
}
```

il codice che segue è formalmente sbagliato:

```
static void Main()
{
    StrutturaEsempio s;
    int x = s.a;          // errore: il valore del campo a è indefinito!
}
```

mentre il seguente codice è corretto:

```
static void Main()
{
    StrutturaEsempio s = new StrutturaEsempio();
    int x = s.a;          // ok: a vale zero
}
```

ed è corretto anche il seguente:

```
static void Main()
{
    StrutturaEsempio s;
    s.a = 10;             // ok: è corretto assegnare un valore a un campo indefinito
}
```

Inizializzazione dei campi membro nella dichiarazione

Diversamente da quanto accade con le classi, nei tipi struttura è proibito fornire degli inizializzatori per campi membro non statici:

```
struct StrutturaEsempio
{
    int a = 10;           // errore: proibito inizializzare campi non statici!
    static int b = 10;    // ok
}
```

Inizializzazione dei campi membro nei costruttori definiti dal programmatore

Nei tipi struttura il ruolo principale assunto dai costruttori è quello di fornire un valore iniziale ai campi membro; ebbene, il linguaggio impone che tale ruolo venga assolto fino in fondo. In altre parole:

il codice contenuto in un costruttore deve espressamente assegnare un valore a tutti i campi membro.

Ciò detto, il seguente codice è formalmente sbagliato poiché il costruttore termina senza che al campo `z` sia stato assegnato un valore:

```
struct StrutturaEsempio
{
    double x, y, z;
    public StrutturaEsempio(double x, double y)
    {
        this.x = x;
    }
}
```

```

        this.y = y;
    } // errore: z non è stato inizializzata!
}

```

Questo requisito produce delle conseguenze particolari. Infatti, all'interno di un costruttore è ammesso invocare una funzione membro non statica soltanto dopo che i campi sono stati tutti inizializzati. Ad esempio, il seguente codice:

```

struct StrutturaEsempio
{
    double x, y, z;
    public StrutturaEsempio(double x, double y)
    {
        this.x = x;
        this.y = y;
        MetodoQualsiasi(); //errore: z non è stato ancora inizializzato
        this.z = 0;
    }

    void MetodoQualsiasi()
    {
        ...
    }
}

```

produce l'errore di compilazione:

Impossibile usare l'oggetto **this** prima che tutti i suoi campi vengano assegnati

Ciò avviene a prescindere dal codice contenuto nella funzione invocata. Un funzione statica, invece, può essere invocata in ogni caso, poiché per definizione essa non può fare riferimento ai campi non statici della struttura.

8 Migliorare l'accesso agli attributi della classe

Fino ad ora abbiamo introdotto gli elementi essenziali per la definizione di nuovi tipi di dati, che possono essere implementati sia mediante classi sia mediante strutture. La possibilità di definire campi, metodi e costruttori, e i relativi livelli d'accesso, è tutto ciò che serve per tradurre in pratica due punti chiave della OOP: l'incapsulamento e l'astrazione.

In questo senso l'accesso alla rappresentazione interna della classe mediante i metodi d'accesso dell'interfaccia pubblica rappresenta un elemento centrale, ma che rende l'uso degli oggetti poco naturale.

Per chiarire il concetto consideriamo le seguenti due diverse implementazioni della stessa classe:

```

// Prima versione della classe Persona
class Persona
{
    public string Nome;
}

// Seconda versione della classe Persona

```

```

class Persona
{
    string _nome;
    public string GetNome()
    {
        return _nome;
    }
    public void SetNome(string nome)
    {
        _nome = nome;
    }
}

```

La prima versione espone pubblicamente la propria rappresentazione interna e dunque consente un codice *consumer* del tipo:

```

Persona p = new Persona();
p.Nome = "Rossi Fabio";
string nome = p.Nome;

```

il quale è chiaro e semplice, ma ha il difetto di dipendere dall'implementazione della classe. La seconda versione fornisce dei metodi di accesso e dunque richiede un codice *consumer* del tipo:

```

Persona p = new Persona();
p.SetNome("Rossi Fabio");
string nome = p.GetNome();

```

il quale è senz'altro meno naturale del codice precedente.

La questione è che `nome` rappresenta un attributo e cioè un valore memorizzato dagli oggetti della classe, e sarebbe desiderabile potervi accedere nello stesso modo in cui si accede ai valori in genere. D'altra parte non si vorrebbe rinunciare alla protezione fornita dai metodi di accesso.

Ebbene, il linguaggio C# fornisce gli strumenti sintattici per implementare funzioni membro che consentono l'accesso ai campi mediante la sintassi usata nelle espressioni e nelle assegnazioni;

8.1 “Proprietà”

Da un punto di vista astratto il termine “proprietà” assume lo stesso significato di attributo; una proprietà rappresenta una certa informazione che caratterizza la classe e che quindi determina lo stato degli oggetti appartenenti ad essa. Dal punto di vista del linguaggio una proprietà (*property*) rappresenta:

una funzione membro che implementa un meccanismo di accesso alla rappresentazione interna della classe.

Una proprietà consente di mantenere la sintassi d'uso di una variabile e fornisce contemporaneamente la stessa capacità di un metodo di garantire un accesso controllato alla rappresentazione interna della classe.

8.2 Definizione e uso di una proprietà

Anche nella definizione, le proprietà presentano una sintassi che sta a metà tra quella di una variabile e quella di un metodo. La definizione assume la seguente forma:

```

    modificatoreopz tipo nome-proprietà
    {

```

```

        modificatoreopz get {    ... }opz
        modificatoreopz set {    ... }opz
    }

```

Una proprietà è dunque definita da:

- ❑ uno o più modificatori di accesso
- ❑ un tipo;
- ❑ un nome;
- ❑ dal codice di accesso, il quale è suddiviso in due parti distinte: codice di accesso in lettura (*get accessor*) e codice di accesso in scrittura (*set accessor*), dei quali può esserne fornito uno soltanto o entrambi.

Ad esempio, la seguente classe definisce un campo privato `nome` e una proprietà che ne consente il solo accesso in lettura:

```

class Persona
{
    string _nome;
    ...
    public string Nome           // accesso in sola lettura
    {
        get
        {
            return _nome;
        }
    }
}

```

Il codice *consumer* vede della classe soltanto la proprietà, in quanto pubblica, ed è in grado di usarla come un campo membro, ma solo per leggerne il valore:

```

Persona pers = new Persona();
string nominativo = pers.Nome;           // ok
pers.Nome = "Rossi Fabio"                // errore: accesso in scrittura non ammesso!

```

Analogamente, la seguente classe definisce due campi privati, `_nome` e `_password`, per i quali fornisce l'accesso mediante due proprietà:

```

class Utente
{
    string _nome;
    string _password;
    ...
    public string Nome           // accesso in lettura/scrittura
    {
        get
        {
            return _nome;
        }
        set

```



```

    {
        _nome = value;
    }
}

```

```

public string Password    // accesso in sola scrittura
{
    set
    {
        _password = value;
    }
}
}

```

Il codice *consumer* può sia leggere che modificare il campo privato `_nome`, ma soltanto modificare il campo `_password`, come dimostra il seguente frammento di codice:

```

Utente user = new Utente();
user.Nome = "Rossi Fabio";
string nomeUtente = user.Nome;
user.Password = "rosfbo";
string parolaChiave = user.Password; // errore: lettura non ammessa!

```

8.3 Descrizione delle proprietà in UML

Il linguaggio UML non ha nessuna definizione per distinguere proprietà da attributi. Per poter evidenziare questa caratteristica però, l'UML mette a disposizione il concetto di **Stereotipo**. Uno stereotipo è un meccanismo linguistico che consente di modificare il significato di un elemento di un diagramma. Uno stereotipo di dichiara racchiudendo tra parentesi angolari il significato che si desidera associare all'elemento..

A questo punto il diagramma UML della classe sopra definita diventa il seguente

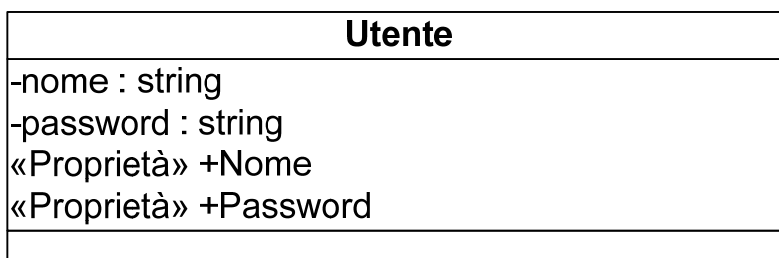


Figura 2-15 Diagramma UML della classe Utente con due proprietà.

8.4 Modificatori di accesso separati per ciascun accessor di una proprietà

A partire dalla versione 2.0 del .NET i modificatori di accesso possono essere definiti sia a livello di intera proprietà che a livello di singolo accessor. Questa caratteristica fa sì che possa avere una proprietà pubblica in lettura e privata in scrittura o viceversa, come nell'esempio che segue:

```

class Utente
{

```

```

    string _nome;
    ...
    public string Nome
    {
        get                // accesso in lettura pubblico
        {
            return _nome;
        }
        private set        // limitazione all'accesso in scrittura
        {
            _nome = value;
        }
    }
}

```

I vincoli che si devono rispettare quando si specificano modificatori differenti per i due accessor sono i seguenti:

- ❑ il vincolo dell'accessor deve essere più restrittivo di quello della proprietà;
- ❑ non si possono aggiungere modificatori ad entrambi gli accessor.

La seguente classe contiene entrambi gli errori

```

class Utente
{
    string _nome;
    string _password;
    ...
    public string Nome        // accesso in lettura/scrittura
    {
        private get          // ERRORE !!! se si definisce questo modificatore
        {
            return _nome;
        }
        protected set        // ERRORE !!! non si può definire anche questo
        {
            _nome = value;
        }
    }

    private string Password   // modificatore di accesso privato
    {
        public set            // ERRORE!!!: public è un accesso meno restrittivo
        {
            _password = value;
        }
    }
}

```

8.5 Funzionamento del “set accessor” e del “get accessor”

Per comprendere a fondo il funzionamento delle proprietà è necessario conoscere in che modo il linguaggio traduce la loro definizione all'interno della classe ed ogni riferimento ad esse presente nel codice *consumer*.

Innanzitutto, una proprietà appare al codice *consumer* come una variabile, ma non lo è affatto, rientra infatti nella categoria delle funzioni membro. Una proprietà viene tradotta in un metodo, o in una coppia di metodi se definisce entrambi gli *accessor*. Ecco in che modo il compilatore traduce il codice della classe dell'esempio precedente:

```
class Utente
{
    string _nome;
    string _password;
    ...
    public string get_Nome()          // get accessor proprietà Nome
    {
        return _nome;
    }

    public void set_Nome(string value) // set accessor proprietà Nome
    {
        _nome = value;
    }

    public void set_Password(string value) // set accessor proprietà Password
    {
        _password = value;
    }
}
```

Il compilatore, in modo trasparente al programmatore, crea un metodo per ogni *accessor* della proprietà, chiamandolo con il prefisso *get* se è un *get accessor*, *set* se è un *set accessor*. Il prototipo del metodo *get* non definisce alcun parametro e dichiara come tipo di ritorno quello della proprietà. Il metodo *set* definisce un parametro di nome *value* dello stesso tipo della proprietà, il quale è appunto l'omologo del parametro implicito *value* usato nel *set accessor*.

Nel codice *consumer*, tutti i riferimenti alla proprietà vengono tradotti in invocazioni ai metodi *accessor*, in base al fatto che la proprietà venga usata in un'espressione piuttosto che come oggetto di un'assegnazione. Ciò detto, il seguente codice:

```
user.Nome = "Rossi Fabio";
string nomeUtente = user.Nome;
user.Password = "rosfbo";
```

viene tradotto in:

```
user.Set_Nome("Rossi Fabio");           // invocazione set accessor di Nome
string nomeUtente = user.Get_Nome();    // invocazione get accessor di Nome
user.Set_Password("rosfbo");            // invocazione set accessor di Password
```

8.6 Codice ammissibile negli accessor di una proprietà

Poiché gli *accessor* di una proprietà sono tradotti in veri e propri metodi, non esiste alcuna restrizione sul codice che possono contenere, né è necessario che essi forniscano effettivamente l'accesso a un campo membro. L'unico requisito formale da rispettare riguarda il *get accessor*; poiché esso viene tradotto in un metodo che ritorna un valore dello stesso tipo della proprietà, è necessario che:

contenga effettivamente almeno un'istruzione `return` che ritorni un'espressione del tipo appropriato.

Nel seguente codice, tale requisito non viene rispettato:

```
class Persona
{
    string _nome;
    int _altezza;
    public string Nome
    {
        get
        {
            return _altezza; // errore: ritorna un valore di tipo sbagliato!
        }
    }
}
```

E' importante comprendere che quello imposto dal linguaggio è un requisito formale: dal punto di vista della correttezza sintattica non è importante cosa venga effettivamente ritornato, purché sia un'espressione compatibile con il tipo della proprietà. Il seguente codice lo dimostra:

```
class Persona
{
    string _nome;
    int _altezza;
    public string Nome
    {
        get
        {
            return "ciao come stai"; // ok: privo di senso, ma corretto
        }
    }
}
```

Per quanto riguarda il *set accessor*, non esiste invece alcun requisito da rispettare, né è obbligatorio che venga usato il parametro implicito di nome `value`. In sostanza, è perfettamente lecito, anche se di dubbia utilità, definire un *set accessor* vuoto:

```
class Persona
{
    string _nome;
    int _altezza;
    public string Nome
    {

```

```

        set          // ok: non è sbagliato definire un set accessor vuoto
        {
        }
    }
}

```

Questa flessibilità delle proprietà consente di definire un'interfaccia pubblica che espone degli attributi i quali non hanno un'effettiva corrispondenza nella rappresentazione interna della classe. Consideriamo questa nuova versione della classe Rettangolo:

```

class Rettangolo
{
    double _b;
    double _h;
    public Rettangolo(double b, double h)
    {
        _b = b;
        _h = h;
    }
    public double Area
    {
        get
        {
            return _b * _h;
        }
    }
    public double Perimetro
    {
        get
        {
            return (_b + _h) * 2;
        }
    }
}

```

Segue il diagramma UML della classe:

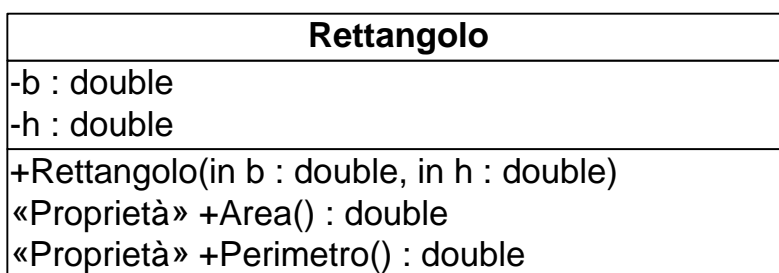


Figura 2-16 Diagramma UML della classe Utente con due proprietà.

Si noti che questa volta nel diagramma UML le proprietà sono state rappresentate all'interno dello spazio riservato alle operazioni.

Dato che non esiste una regola ben precisa per il collocamento delle proprietà nel linguaggio UML la scelta è a completa discrezione dell'utente, basta indicare con uno stereotipo il significato dell'elemento.

La classe Rettangolo definisce due proprietà, Area e Perimetro, che non forniscono l'accesso a un determinato campo membro, ma ritornano il risultato di un calcolo. Nel codice *consumer*, però, esse vengono utilizzate come un qualsiasi campo pubblico, con l'eccezione che non possono essere oggetto di assegnazione in quanto non definiscono un *set accessor*:

```
Rettangolo ret = new rettangolo(30, 10);  
Console.WriteLine("Area = {0} Perimetro = {0}", ret.Area, ret.Perimetro);
```

8.7 Proprietà “automatiche”

Spesso le classi contengono proprietà che servono solamente ad accedere agli attributi della classe senza effettuare nessun controllo o validazione.

Questo significa che si devono comunque scrivere i due *accessor set e get* che altro non hanno al loro interno che la semplice assegnazione da e verso l'attributo “mappato”.

A partire dalla versione 3.0 del C#, è stato introdotto il concetto di proprietà “automatica”, la possibilità, cioè, di dichiarare una proprietà a cui venga associata (in fase di compilazione) un attributo ed il codice per gli *accessor get e set*.

E' infatti sufficiente dichiarare la proprietà in questo modo:

```
public DateTime DataVaro  
{  
    get;  
    set;  
}
```

ed automaticamente il compilatore genererà un attributo chiamato come il nome della proprietà con aggiunto il simbolo di sottolineatura (es: DataVaro) ed i relativi codici per gli *accessor get e set*.

I limiti di questa tecnica sono che l'attributo generato automaticamente non è disponibile per i membri della classe e che si deve obbligatoriamente definire la proprietà con l'*accessor set e get*.

Nel caso non si voglia esporre pubblicamente uno dei due *accessor* sarà sufficiente anteporre il prefisso *private* come nell'esempio che segue.

```
public DateTime DataVaro  
{  
    get;  
    private set;  
}
```

8.8 Inizializzatore di oggetto

Ora che abbiamo visto sia i costruttori che le proprietà, è possibile introdurre il concetto di “Inizializzatore di Oggetto” presente a partire dalla versione 3.0 del C#.

Questa caratteristica semplifica la costruzione e la inizializzazione di un oggetto in quanto consente di specificare in un'unica istruzione entrambi i costrutti.

Per poter utilizzare questa caratteristica è sufficiente, in fase di dichiarazione dell'oggetto, specificare la sequenza di inizializzatori dei dati membro racchiusi tra parentesi graffe e separati dalla virgola, dove ogni membro da inizializzare rappresenta un campo membro o una proprietà (accessibili) seguiti dal segno uguale a da una espressione che ne rappresenta il valore.

Ipotizziamo di voler creare un oggetto e di assegnare ad esso i valori di alcune proprietà; per fare ciò scriveremmo il seguente codice:

```
Nave sestaNave = new Nave();  
sestaNave.Nome = "Estrema";  
sestaNave.Stazza = 1500;  
sestaNave.Velocita = 20;
```

Con il nuovo costrutto sarà invece possibile utilizzare una forma molto più abbreviata e leggibile:

```
Nave sestaNave = new Nave { Nome = "Estrema", Stazza = 1500, Velocita = 20 };
```

8.9 Proprietà e variabili a confronto

Come abbiamo visto, una proprietà viene utilizzata in modo del tutto analogo a una variabile, nonostante la prima rappresenti una funzione mentre la seconda un oggetto che memorizza un valore. In sintesi, ogni qual volta una proprietà compare in una espressione viene invocato il *get accessor*, il quale produce il valore corrispondente alla proprietà. Ogni qual volta compare alla sinistra di un operatore di assegnazione (semplice o composto) viene invocato il *set accessor*, il quale elabora il valore dell'espressione a destra dell'operatore.

La maggior parte degli operatori applicabili alle variabili possono essere usati anche con le proprietà, compresi (ma con alcune restrizioni) gli operatori di incremento e decremento ++ e --. D'altra parte, una proprietà non è una variabile, infatti:

- ❑ non può essere passata come argomento `out` o `ref` a un metodo, poiché alle proprietà, diversamente dalle variabili, non è associato un indirizzo di memoria;
- ❑ una proprietà a sola scrittura non può comparire in una espressione, poiché il suo valore non può essere calcolato (non esiste il *get accessor*). In sostanza, una proprietà a sola scrittura può comparire soltanto alla sinistra dell'operatore di assegnazione semplice.
- ❑ una proprietà a sola lettura non può comparire alla sinistra di un operatore di assegnazione, poiché ad essa non può essere assegnato un valore (non esiste il *set accessor*);
- ❑ gli operatori di incremento e decremento non possono essere applicati a una proprietà che sia a sola lettura o a sola scrittura, poiché tali operatori richiedono l'esecuzione di entrambi gli *accessor* (prima il *get* e poi il *set*).

8.10 Overloading di proprietà

Le proprietà sono le uniche funzioni membro che non possono essere sovraccaricate; in altre parole una classe non può definire due proprietà con lo stesso nome ma di diverso tipo. Ciò dipende dal fatto che il *get accessor* di una proprietà viene tradotto in un metodo senza parametri, e quindi per il linguaggio non esiste un modo per distinguere il *get accessor* di una proprietà da quello di un'altra con lo stesso nome. Ricordiamo, infatti, che la distinzione tra metodi sovraccaricati avviene soltanto sulla base della lista dei parametri, senza considerare il tipo di ritorno.

8.11 Linee guida nella scelta dei nomi delle proprietà

Poiché nella maggior parte dei casi una proprietà fornisce l'accesso a un campo privato, sorge il problema di come nominarla in relazione al campo in questione (o, se si preferisce, come nominare il campo in relazione alla proprietà).

Un approccio, che è quello adottato nel testo, è quello scegliere per le proprietà lo stesso nome del campo del quale forniscono l'accesso, eccetto che per il *case* della prima lettera, che dev'essere maiuscolo e togliendo il carattere di sottolineatura. Si evidenzia anche in questo caso l'utilità di

aggiungere il carattere di sottolineatura agli attributi, che rende possibile applicare questa tecnica sia utilizzando il linguaggio C# sia altri linguaggi che non facciano distinzione tra il *case* delle lettere (es VB.NET).

9 Un esempio completo: Classe Nave

Adesso che abbiamo illustrato i singoli aspetti della creazione di nuovi tipi andiamo ad analizzare la creazione di una classe completa di tutto quello visto in precedenza. Per fare ciò ci cimenteremo nella implementazione di un oggetto che ci servirà anche successivamente.

Il problema che affronteremo sarà una ipersemplificazione di un problema reale, e riguarda la gestione di una flotta navale. Allo scopo di semplificare il programma decidiamo per il momento di rappresentare soltanto un sotto insieme delle informazioni che caratterizzano la gestione di una flotta navale e ci concentriamo sulle informazioni necessarie per rappresentare una Nave, stabilendo che questa debba contenere:

- ❑ il nome della nave;
- ❑ la stazza della nave;
- ❑ la velocità di crociera;
- ❑ se sia già stata varata oppure no.

9.1 Classe “Nave”

Nel definire una classe occorre stabilire:

- ❑ gli attributi che si intende memorizzare, rappresentati dai campi membro;
- ❑ le operazioni – i metodi – che possono essere eseguite con gli oggetti della classe;
- ❑ i costruttori e dunque le modalità di costruzione di oggetti della classe che si intende mettere a disposizione del codice *consumer*.

Dev’essere inoltre stabilito cosa debba far parte dell’interfaccia pubblica e dunque essere utilizzabile dal codice *consumer*. In linea di massima, tutti gli attributi dovrebbero essere privati e dunque, per quelli che devono essere accessibili al codice *consumer*, dovranno essere forniti dei metodi o delle proprietà di accesso.

Si deve porre anche cura nel cercare di seguire le linee guida (dette anche best-practices) del .NET. Queste aiutano lo sviluppatore a costruire classi ed oggetti che siano il più possibile simili a quelle già presenti nella libreria del .NET.

Un altro aspetto da considerare con attenzione è il fatto di scegliere se e quando convenga definire degli attributi oppure se convenga fornire dei metodi che li calcolino ogni qualvolta si renda necessario.

Simili questioni si presentano abbastanza spesso nell’implementazione di un nuovo tipo di dato. La scelta dell’approccio migliore dipende dal caso specifico ed è influenzata da vari fattori. Solitamente ci si limita ad applicare una regola molto generale che afferma:

evitare la ridondanza di informazioni e dunque definire soltanto i campi membro strettamente necessari.

A questo punto siamo pronti a disegnare il diagramma della classe ed ad implementarla.

L'implementazione del tipo `Nave` è molto semplice, poiché è, almeno per il momento, semplicemente un contenitore di dati. Pertanto, la classe `Nave` deve definire soltanto quattro campi membro e le relative operazioni (costruttori/metodi di accesso).

```
public class Nave
{
    string _nome;
    double _stazza;
    int _velocita;
    bool _varata;
    . . .
}
```

9.2 Accesso agli attributi della classe “Nave”

Poiché un oggetto `Nave` rappresenta un semplice contenitore di dati, si devono creare delle proprietà che consentono l'accesso in lettura e scrittura ai campi e tutti faranno parte dell'interfaccia pubblica della classe.

```
public class Nave
{
    string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    double _stazza;
    public double Stazza
    {
        get { return _stazza; }
        set { _stazza = value; }
    }

    int _velocita;
    public int Velocita
    {
        get { return _velocita; }
        set { _velocita = value; }
    }

    bool _varata;
    public bool Varata
    {
        get { return _varata; }
        set { _varata = value; }
    }
}
```

Come si può vedere sono state aggiunte delle proprietà che consentono di esporre tutti gli attributi privati della classe, con la possibilità di accesso sia in lettura che in scrittura. Anche se apparentemente non c'è differenza tra l'aver utilizzato le proprietà ed il definire gli attributi pubblici, ciò consentirà in futuro di modificare l'implementazione della classe senza dover modificare il codice *consumer* che la usa.

9.3 Costruttori della classe “Nave”

Decidiamo a questo punto di implementare tre costruttori. Il primo che richiede i valori da assegnare ai quattro campi membro sarà il costruttore che abbiamo definito “master” (il più completo), il secondo è rappresentato dal costruttore di default, il quale non svolge alcuna operazione, mentre il terzo offre un esempio di come sfruttare il costruttore “master” onde evitare di specificare l'attributo “_varata” in fase di creazione.

```
public class Nave
{
    . . .
    public Nave(string nome, double stazza, int velocita, bool varata)
    {
        nome = nome;
        _stazza = stazza;
        _velocita = velocita;
        _varata = varata;
    }

    public Nave()
    { }

    public Nave(string nome, double stazza, int velocita)
        : this(nome, stazza, velocita, false)
    { }
}
```

La scelta di fornire un costruttore di default è coerente con la scelta di consentire al codice *consumer* di modificare i singoli campi membro attraverso i metodi dell'interfaccia pubblica della classe.

Volendo, avremmo potuto progettare una diversa interfaccia pubblica, priva del costruttore di default e dei metodi d'accesso in scrittura ai campi. In questo caso, l'unico modo per fornire i dati a un oggetto *Ordine* sarebbe stato quello di specificarli durante la creazione.

9.4 Metodi della classe “Nave”

A questo punto ci mancano solamente i metodi che facilitino la scrittura del codice *consumer*, e che svolgano delle semplici operazioni sugli attributi della classe.

Le linee da adottare nell'implementazione di una classe suggeriscono di definire un metodo che esponga in maniera testuale gli attributi della classe, il metodo `ToString()`. La presenza di questo metodo consente sia di iniziare a scrivere anche un po' di codice di test della classe sia di evitare che l'esecuzione del seguente frammento di codice

```
Console.WriteLine("Dati della nave:{0}",nave);
```

Produca il seguente risultato, poco significativo:

```
EduDotNet.FlottaNavale.Nave
```

Infatti, quando si cerca di visualizzare una variabile viene invocato il metodo `ToString()` della classe, ed in questo caso, non essendo stato ridefinito, viene restituito il nome completo del tipo dell'oggetto.

Poi possiamo scrivere due metodi che consentono di “varare” e di conoscere se una nave risulta varata. Quest'ultimo metodo sembrerebbe un'inutile duplicazione dell'attributo `Varata`, ma le linee guida suggeriscono la creazione di un metodo che inizia con il prefisso “**Is**” per recuperare lo stato di una informazione quando essa è di tipo vero/falso.

```
public override string ToString()
{
    string tmp = "";
    tmp = String.Format("{0},{1},{2},{3}", _nome, _stazza, _velocita, _stato);
    return tmp;
}
public void Vara()
{
    _varata = true;
}
public bool IsVarata()
{
    return _varata;
}
```

Segue il diagramma UML della classe finora realizzata.

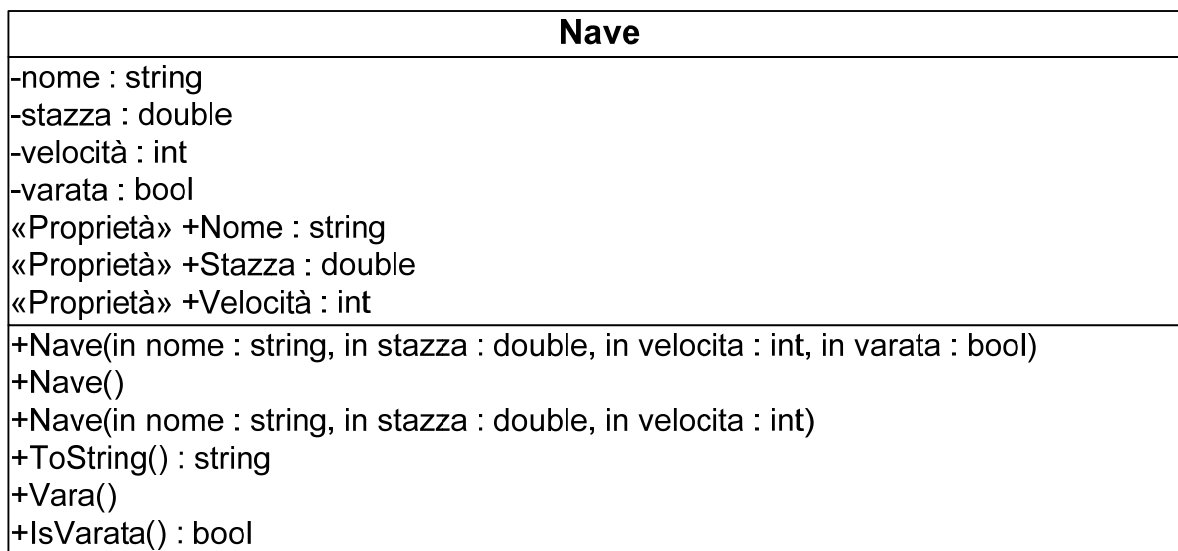


Figura 2-17 Diagramma UML della classe Nave.

A questo punto non resta che scrivere una semplice applicazione console che si limita ad eseguire alcune operazioni (evidenziate in grigio) per mettere alla prova l'implementazione della classe `Nave` e le gestione delle funzionalità relative al varo ed al cantieraggio della stessa

```
static void Main(string[] args)
{
    Nave primaNave = new Nave();
    primaNave.Nome = "Splendida";
```

```
primaNave.Stazza = 1000;
primaNave.Velocita = 10;
Console.WriteLine("Dati della prima nave: {0}", primaNave.ToString());
primaNave.Vara();
Console.WriteLine("Dati della prima nave dopo il varo: {0}",
primaNave.ToString());
Console.WriteLine();

Nave secondaNave = new Nave("Magnifica", 2000, 20, true);
Console.WriteLine("Dati della seconda nave: {0}", secondaNave.ToString());
Console.WriteLine("La nave {0} è varata ? : {1}", secondaNave.Nome,
    secondaNave.IsVarata());
Console.WriteLine();

Console.ReadKey();
}
```

Segue l'output del programma di test:

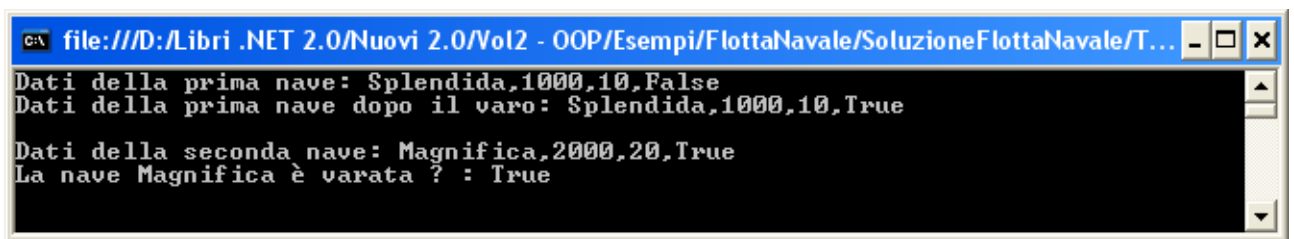


Figura 2-18 Output del programma di prova della classe Nave.

9.5 Operatori della classe “Nave”

Cercare di trovare un esempio di utilizzo dell'overloading degli operatori in questa classe potrebbe sembrare impossibile, se non tenessimo in conto un comune problema che emerge quando andiamo a confrontare tra di loro due oggetti.

A causa della natura del .NET il seguente codice

```
Nave n1 = new Nave("Magica", 400, 40);
Nave n2 = new Nave("Magica", 400, 40);

if (n1 == n2)
    Console.WriteLine("Le due navi sono uguali");
else
    Console.WriteLine("Le due navi sono diverse");
```

produce questo risultato

Le due navi sono diverse

Questo a prima vista può sembrare incredibile, ma per il .NET `n1` ed `n2`, sono effettivamente due navi differenti in quanto l'operatore di confronto effettua una verifica sul puntatore dell'oggetto e non sul suo contenuto.

Per poter risolvere questo problema è necessario sovraccaricare l'operatore di uguaglianza `==` che, ricevuti in ingresso due oggetti del tipo `Nave`, restituisca un valore booleano risultato del

confronto tra gli stessi¹⁰ scrivendo tutta la logica di programmazione necessaria a stabilire quando due oggetti sono “uguali”.

Dato che vogliamo aggiungere questo operatore, però, il .NET ci obbliga anche a definire tutti quegli operatori/metodi che lavorano in accordo con quello che abbiamo appena definito e quindi dobbiamo anche (ri)scrivere anche l’operatore di disuguaglianza !=

```
public static bool operator ==(Nave n1, Nave n2)
{
    return (n1._nome == n2._nome && n1._stazza == n2._stazza);
}

public static bool operator !=(Nave n1, Nave n2)
{
    return !(n1 == n2);
}
```

Ridefinendo i due operatori di cui sopra, però, fa sì che il .NET si aspetti che noi ridefiniamo anche il metodo Equals() ed il metodo GetHashCode(), e questo perché in realtà, il .Net al suo interno, non utilizza l’operatore di uguaglianza per confrontare due oggetti, ma usa questi due metodi

Il metodo Equals(), allo stesso modo dell’operatore ==, riceve in ingresso un oggetto di tipo Object e restituisce un valore booleano risultato del confronto l’oggetto corrente e quello passato.

Questo metodo, inoltre, non deve mai generare un’eccezione e che deve sempre restituire false se l’argomento è un riferimento ad un oggetto null o è di un tipo diverso dal tipo corrente.

```
public override bool Equals(object obj)
{
    return this == (Nave)obj;
}
```

La ridefinizione del metodo GetHashCode va fatta perché due oggetti che sono considerati uguali restituiscano anche lo stesso codice hash.

GetHashCode è un metodo che restituisce un codice hash dell’oggetto. Questo metodo viene utilizzato quando l’oggetto è una chiave di collection e tavole hash. In teoria, il codice hash dovrebbe essere univoco per una determinata istanza d’oggetto in modo che si possa controllare che due oggetti siano “uguali” confrontandone il codice hash. Tuttavia, implementare una funzione hash che fornisce valori univoci è raramente possibile, e oggetti differenti potrebbero restituire lo stesso codice hash, pertanto non si deve mai desumere che due istanze con lo stesso codice hash siano uguali, qualsiasi cosa possa significare “uguale” per quel tipo specifico

Il valore restituito del metodo GetHashCode non deve cambiare durante l’intero ciclo di vita dell’oggetto e per fare ciò, si può restituire un valore costante combinando i codici hash di due o più campi immutabili mediante l’operatore di Xor, oppure si può calcolare un valore random in fase di costruzione dell’oggetto memorizzandolo in un campo privato.

```
public override int GetHashCode()
{
    int hashcode = _nome.GetHashCode() ^ _stazza.GetHashCode();
    return hashcode;
}
```

¹⁰ In realtà, Microsoft sconsiglia di ridefinire questo operatore in quanto, essendo un operatore d’identità, dovrebbe restituire true solamente quando i due oggetti confrontati sono uguali (lo stesso oggetto).

Adesso le cose sembrano funzionare bene, anche se ci manca ancora un ultimo “tassello”.

9.6 (De)Serializzazione della classe “Nave”

Adesso le cose sembrano funzionare bene, anche se ci manca ancora un ultimo “tassello”.

Le famose “best practices” del .NET ci consigliano anche di dare la possibilità al codice *consumer* di creare un oggetto a partire da una stringa.

Per poter fare ciò ci servirà un meccanismo per “serializzare” gli attributi di una classe in una stringa e di conseguenza un sistema per “de-serializzare” la stringa negli attributi della classe.

Per quanto riguarda la serializzazione, possiamo effettuarla o sfruttando le classi del .NET, od utilizzando il formato XML, oppure, dato che l’abbiamo già fatto utilizzando un metodo ToString().

Per deserializzare e quindi costruire un oggetto a partire da una stringa, abbiamo bisogno di un altro costruttore che accetti questo parametro

```
public Nave(string unaNave)
{
    string[] parti = unaNave.Split(',');

    _nome = parti[0];
    _stazza = Convert.ToDouble(parti[1]);
    _velocita = Convert.ToInt32(parti[2]);
    _varata = Boolean.Parse(parti[3]);
}
```

Come tutte le cose facili ed estremamente comprensibili, questa soluzione risulta essere sbagliata anche se, ad una prima occhiata, sembra che sia tutto a posto

Si è infatti prima provveduto a separare la stringa nelle sue singole parti, poi si sono presi i “pezzi” risultanti e si sono inizializzati gli attributi.

Peccato che questo costruttore si comporti a dovere solamente in presenza di una stringa contenente dati completamente corretti.

Se scriviamo questo frammento di codice per provare il costruttore

```
string unaNave = "Fantastica,3000,20,True";
Nave terzaNave = new Nave(unaNave);
Console.WriteLine("Dati della terza nave: {0}", terzaNave.ToString());
```

tutto funziona perfettamente perché, come si dice, la stringa è “well-formed”.

Ma se passiamo al costruttore una stringa “sbagliata” del tipo **"Fantastica,3000,"** contenente solo due dei 4 attributi richiesti, il programma solleverà una eccezione e terminerà in maniera imprevista.

Oltre a questo c’è anche il problema che abbiamo duplicato il codice nel costruttore rendendolo quasi uguale a quello del costruttore master.

Per ovviare a questo problema la prima soluzione che viene in mente sarà sicuramente quella di chiamare da dentro questo costruttore quello master, ma come abbiamo già visto questo è impossibile,

Serve, quindi un meccanismo che contemporaneamente ci consenta di controllare che una stringa sia “well-formed” e ci eviti di duplicare il codice.

La soluzione a questo problema sta nella creazione di due metodi che possiedono quasi tutti gli oggetti semplici del .NET: il metodo **TryParse** ed il metodo **Parse**.

Questi due metodi, provvedono all’analisi della stringa ed alla successiva creazione dell’oggetto e devono essere, ovviamente, dei metodi statici di classe.

```
public static bool TryParse(string str, out Nave nuovaNave)
```

```

{
    bool creata = false;
    try
    {
        string[] parti = str.Split(',');
        nuovaNave = new Nave(parti[0],
                               Double.Parse(parti[1]),
                               Int32.Parse(parti[2]),
                               Boolean.Parse(parti[3]));

        creata = true;
    }
    catch (Exception)
    {
        nuovaNave = null;
    }
    return creata;
}

public static Nave Parse(string str)
{
    Nave nuovaNave = null;
    bool creata = Nave.TryParse(str, out nuovaNave);
    if (!creata)
        throw new FormatException("Formato Nave non corretto");
    return nuovaNave;
}

```

Come si può notare il metodo `TryParse()`, all'interno di un costrutto `try..catch` “prova” a costruire un oggetto a partire da una stringa.

In caso di successo verrà restituito il valore booleano `true` e l'oggetto costruito utilizzando il parametro passato come `out`, altrimenti verrà restituito `false` e `null`.

Il metodo `Parse()`, utilizzando il metodo precedente solleva un'eccezione se la costruzione fallisce, restituendo invece l'oggetto creato se tutto va a buon fine.

A questo punto possiamo modificare il codice client in questa maniera per testare i due nuovi metodi

```

string unaNave = "Fantastica,3000,20,True";
Nave quartaNave = Nave.Parse(unaNave);
Console.WriteLine("Dati della quarta nave: {0}", quartaNave.ToString());
Console.WriteLine();
. . .
string unaNaveSbagliata = "Fantastica,3000";
Nave quintaNave = null;
bool ok = Nave.TryParse(unaNaveSbagliata, out quintaNave);
if (!ok)
    Console.WriteLine("Dati della quinta nave errati:\ \"{0}\", unaNaveSbagliata);
else
    Console.WriteLine("Dati della quinta nave: {0}", quintaNave.ToString());

```

Che produrrà il seguente output

Dati della quarta nave: Fantastica,3000,20,True

Dati della quinta nave errati:"Fantastica,3000"

A questo punto possiamo disegnare il diagramma UML completo della classe Nave.

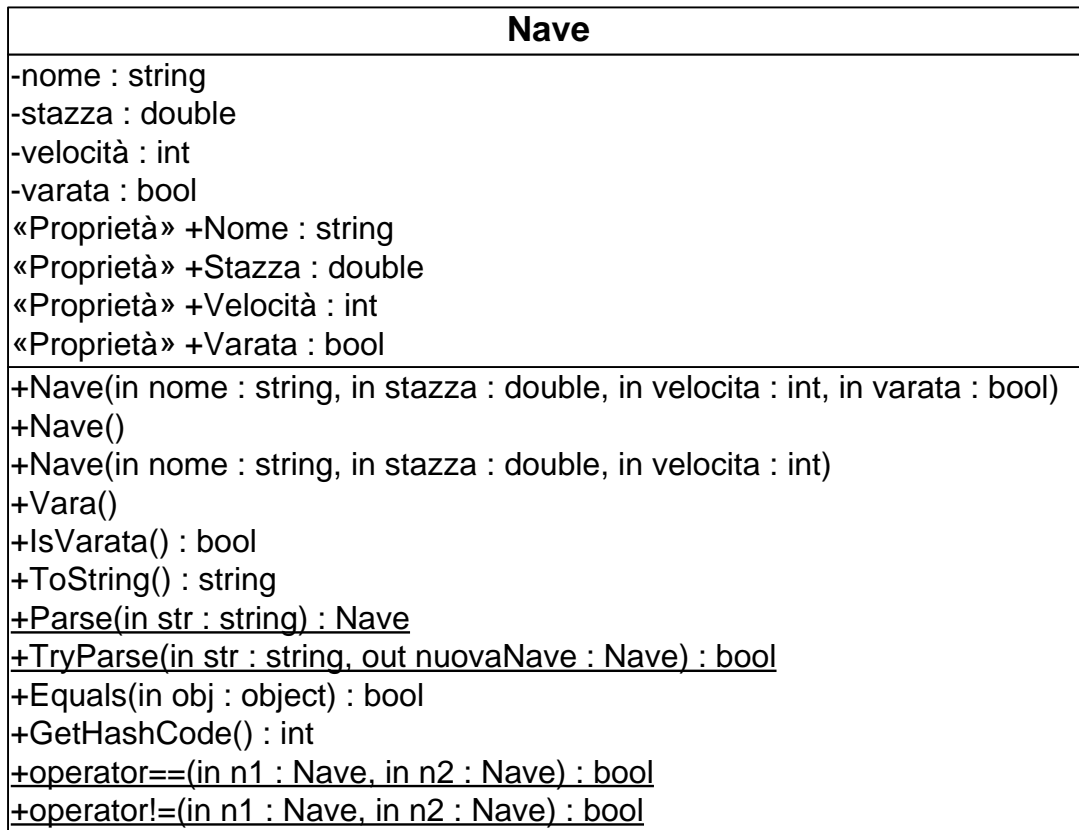


Figura 2-19 Diagramma UML definitivo della classe Nave

Relazioni tra classi

Nel capitolo precedente sono stati introdotti i concetti object oriented necessari per definire nuovi tipi di dati. Nella pratica comune, però, raramente le classi esistono come entità completamente separate le une dalle altre. In questo capitolo saranno esaminati i vari tipi di relazione che possono intercorrere tra due o più classi.

1 Tipi di relazione tra classi

Nella programmazione ad oggetti esistono almeno 5 diversi tipi di relazione tra classi e sono:

- ❑ Dipendenza
- ❑ Associazione
- ❑ Aggregazione
- ❑ Composizione
- ❑ Generalizzazione

Al pari delle metodologie di progettazione di database quelli ORM ed ER, anche l'UML fornisce molti modi per rappresentare le relazioni tra le classi e nel prosieguo del capitolo andremo ad esplorare questi cinque tipi di relazione.

A prescindere, dal loro tipo, per poter illustrare le potenzialità e le caratteristiche di queste relazioni realizzeremo un programma con lo scopo di focalizzare l'attenzione sui meccanismi di implementazione di nuovi tipi di dati e di relazioni mediante la definizione di classi.

Cercheremo, inoltre, di applicare ed approfondire tutti i concetti del capitolo precedente, dai costruttori alle proprietà, dai metodi d'istanza a quelli statici.

Per fare ciò però, andiamo ad affrontare il problema della gestione di una flotta navale.

Testo del problema

Realizzare un programma per la gestione di una flotta navale.

Definizione dei requisiti

Quando si fa progettazione object oriented, è molto importante stabilire quali siano i requisiti che deve avere il nostro modello e quindi quali siano le informazioni necessarie per svolgere le operazioni sul dominio del problema.

Esula dallo scopo del testo fornire delle metodologie di progettazione e quindi ci limiteremo all'implementazione delle classi e del codice *consumer*.

Per l'esplorazione delle varie tipologie di relazioni tra classi useremo come punto di partenza la classe `Nave`, realizzata nel precedente capitolo.

2 Dipendenza

La relazione di dipendenza è quella più debole tra i 5 tipi di relazione ed indica che una classe utilizza, oppure ha conoscenza di, un'altra classe; tipicamente, la relazione si legge come “... **usa un ...**”.

Questa relazione è per definizione di tipo transitorio intendendo con questo che la classe “dipendente” interagisce brevemente con la classe da cui dipende, e non mantiene con essa nessuna relazione per un lungo periodo di tempo.

La dipendenza è una relazione a livello di modello e non a livello di run-time e descrive la necessità di tenere conto di possibili cambiamenti del “fornitore”, dove eventuali modifiche si ripercuoteranno nel “cliente”.

Nel diagramma UML viene rappresentata come una linea tratteggiata con una freccia che va dall'elemento dipendente verso quello usato.

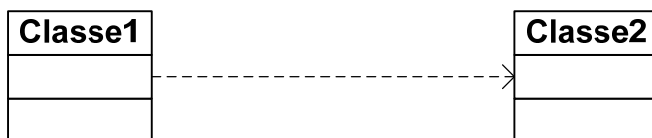


Figura 3-1 Relazione di dipendenza nel diagramma UML.

Per spiegare meglio il concetto consideriamo l'ipotesi che la nostra nave abbia un attributo che ne descrive lo stato corrente e che possa assumere uno di questi 4 valori: `Navigazione`, `Cantiere`, `Venduta`, `Affondata`.

Di primo acchito potremmo aggiungere un attributo (e la relativa proprietà) di tipo stringa in grado di contenere questi valori.

```

string _stato;
public string Stato
{
    get { return _stato; }
    set { _stato = value; }
}
  
```

Questo approccio, però, porta l'inconveniente che se è vero che una stringa può contenere le 4 parole che rappresentano i possibili stati dell'attributo è pur vero che può contenere anche altri valori che non avrebbero significato per il problema.

Meglio sarebbe creare un nuovo tipo di dato, magari di tipo enumeratore, che può essere rappresentato mediante il seguente diagramma UML:

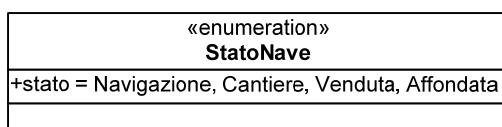


Figura 3-2 Diagramma per la reazione di un tipo di dato Enumeratore.

```
public enum StatoNave
{
    Navigazione, Cantiere, Vendita, Affondata
}
```

e far dipendere la classe nave da questo nuovo tipo di dato:

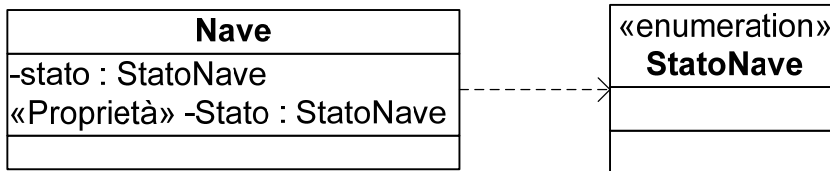


Figura 3-3 Diagramma per la rappresentazione della dipendenza

Il diagramma sta ad indicare che nella classe nave c'è un attributo stato che dipende dall'elemento (in questo caso un dato di tipo enumeratore) StatoNave.

```
StatoNave _stato;
public StatoNave Stato
{
    get { return _stato; }
    set { _stato = value; }
}
```

3 Associazione

L'associazione si colloca al secondo posto in termini di “forza” di una relazione ed indica tipicamente che una classe mantiene una relazione con un'altra classe durante un elevato periodo di tempo.

In UML l'associazione viene rappresentata come una linea continua che collega due oggetti.



Figura 3-4 Relazione di associazione nel diagramma UML

Le caratteristiche tipiche di una associazione sono il fatto che si legge come “... ha un...” e che si deve pensare ad essa come un “puntatore” ad un altro oggetto avendo la possibilità di definire la direzione dei puntatori, che può essere unidirezionale o bidirezionale.

Il ciclo di vita dei due oggetti collegati, non è detto che sia lo stesso e quindi uno dei due può essere distrutto senza necessariamente distruggere l'altro.

Tornando al problema della flotta navale, un'applicazione del concetto di associazione può essere ben illustrato dal fatto che una Nave ha un Comandante

Ipotizziamo, quindi, di avere una classe implementata seguendo le indicazioni date nei capitoli precedenti che abbia due attributi per il nome ed il cognome con il seguente diagramma UML:

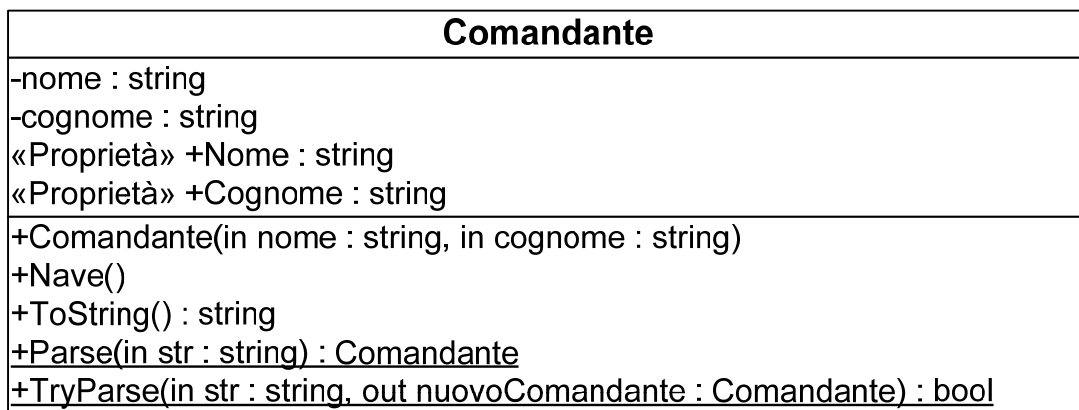


Figura 3-5 Diagramma per la reazione di un tipo di dato Comandante

E' ovvio che ci sia una relazione tra una imbarcazione e la persona che la comanda, così come è chiaro il fatto che pur essendo legati tra di loro la “demolizione” della nave non implica il “pensionamento” del comandante

Stabilire una associazione tra le due classi significa quindi aggiungere un attributo con un riferimento ad un oggetto dell'altra classe e definire in questo modo la possibilità di navigabilità tra gli oggetti..

3.1 Navigabilità

Aggiungere un attributo ad una classe con un riferimento all'altra significa rendere “Navigabile” l'associazione, si ha cioè una notazione esplicita che indica la possibilità di “navigare” da un oggetto ad un altro mediante uno o più attributi.

Se questa caratteristica è presente, viene allora esplicitata con una freccia nella direzione della classe verso cui è possibile navigare, mentre se è possibile navigare in entrambe le direzioni, allora si utilizza una doppia freccia.

Comunque questo è un falso problema in quanto raramente nel mondo reale si usano associazioni che non sono navigabili.

Ecco un diagramma UML che mostra un'associazione navigabile in entrambe le direzioni:



Figura 3-6 Diagramma per la rappresentazione della navigabilità.

I due nomi presenti a lato dei rettangoli sono gli attributi che conterranno il riferimento all'altro oggetto.

In C#, per implementare l'associazione è sufficiente aggiungere alla classe Nave un attributo di tipo Comandante e magari un metodo che ne permette l'assegnazione:

```

Comandante _capitano;
public Comandante Capitano
{
    get { return _capitano; }
    set { _capitano = value; }
}
  
```

Mentre la cosa speculare va fatta nella classe Comandante:

```
Nave _imbarcazione;
public Nave Imbarcazione
{
    get { return _imbarcazione; }
    set { _imbarcazione = value; }
}
```

A questo punto è possibile stabilire il collegamento tra le due classi, sia mediante l'assegnazione della proprietà sia utilizzando, per esempio, il costruttore, eventualità resa possibile proprio per la natura stessa della associazione, che non pone in relazione tra di loro i cicli di vita degli oggetti.

```
public Nave(string nome, double stazza, int velocita, bool varata,
            Comandante capitano)
    : this (nome, stazza, velocita, varata)
{
    _capitano = capitano;
}
```

3.2 Molteplicità

La molteplicità di un'associazione stabilisce quante istanze dell'altro oggetto sono presenti nella classe e può assumere uno dei seguenti valori:

Tabella 3-1 Molteplicità di una associazione

VALORE	DESCRIZIONE
1	Una e soltanto una
0..1	Zero o una
M..N	Da M ad N
*	Zero o più
0..*	
1..*	Una o più

La molteplicità viene indicata a fianco delle classi sotto la linea di collegamento ed anche in questo caso se non si specificano dei valori di molteplicità, si assume che essa sia 1 per entrambi i lati.

Molteplicità Singola

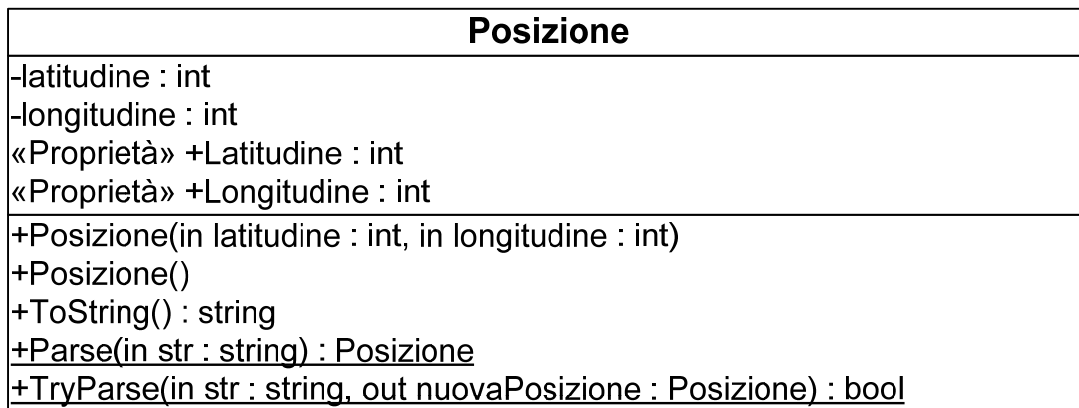
Nel caso specifico della relazione Nave-Comandante, ci sarà solo un comandante per nave (e viceversa) quindi entrambe le molteplicità saranno 1, come mostra il diagramma:



Figura 3-7 Diagramma per la rappresentazione della molteplicità singola*Molteplicità Multipla*

Esistono situazioni nelle quali ad una classe sono associate più istanze di un'altra classe; ad esempio è il caso della rotta di una nave che è composta da una serie di posizioni.

Ipotizziamo dunque di avere un oggetto di tipo `Posizione` con due attributi, `Latitudine` e `Longitudine` (che per semplicità rappresenteremo come interi) rappresentato dal seguente diagramma UML:

**Figura 3-8 Diagramma per la reazione di un tipo di dato Comandante.**

La rotta di una nave non è costituita da una semplice posizione ma da un'insieme delle stesse e pertanto il diagramma sarà il seguente:

**Figura 3-9 Diagramma per la rappresentazione della navigabilità**

Come si può notare, la freccia è rivolta verso la classe `Posizione`; ciò significa che è possibile passare (navigare) da un oggetto di tipo nave ad uno di tipo posizione, mentre non si può fare l'opposto.

Inoltre, la molteplicità della associazione nel lato della nave è 1 mentre in quello della posizione è di tipo 0 o più e questo significa che una posizione appartiene ad un'unica nave, mentre una nave può avere più posizioni.

L'implementazione di associazioni con molteplicità diversa da uno viene realizzata in .NET dichiarando un attributo che rappresenti una *collection* di oggetti che si vogliono referenziare¹¹.

```

List<Posizione> _rotta;
public List<Posizione> Rotta
{
    get { return _rotta; }
    set { _rotta = value; }
}
  
```

¹¹ In realtà, si dovrebbe creare una classe che rappresenti la collezione di oggetti, cosa che verrà spiegata più avanti nel testo.

```
}
```

3.3 Classi di Associazione

Spesso la relazione tra due elementi non è una semplice connessione strutturale, ma si ha bisogno di aggiungere un'altra classe per includere ulteriori informazioni riguardo la relazione. In questo caso si usa una classe di associazione che viene collegata all'associazione primaria.

Questa classe di associazione è rappresentata come una normale classe, con l'unica differenza che la linea di associazione tra le classi primarie si interseca con una linea tratteggiata connessa alla classe di associazione, come mostrato dalla figura seguente

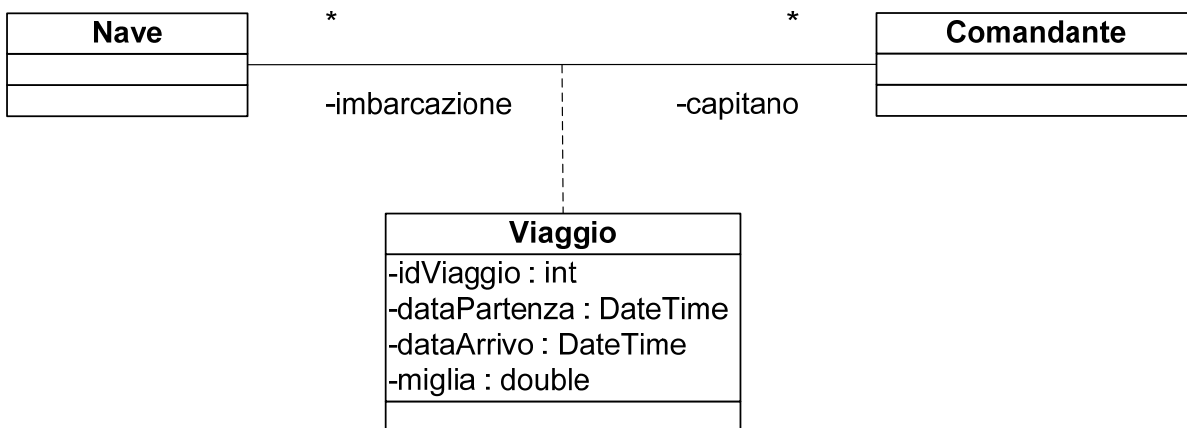


Figura 3-10 Rappresentazione di una classe di associazione.

Solitamente le classi di questo tipo identificano una relazione “multi-a-multi”, ed in fase di implementazione vengono codificate con tre classi: una ciascuna per le classi agli estremi dell'associazione (dette terminali), come visto in precedente ed una per la classe di associazione stessa come mostrato di seguito.

```

public class Viaggio
{
    Nave _imbarcazione;
    Comandante _capitano;

    int _idViaggio;
    DateTime _dataPartenza;
    DateTime _dataArrivo;
    double _miglia;
}
  
```

In questo tipo di relazione potrebbe esserci oppure no, un collegamento diretto tra le due classi terminali; nel secondo caso è ovvio che si dovrà implementare il collegamento mediante la classe di associazione.

4 Aggregazione

In alcuni casi le associazioni tra classi stanno ad indicare che gli oggetti di una classe sono composti o costituiti da oggetti di un'altra classe.

Concretamente, le notazioni di aggregazione e composizione non sono un diverso tipo di relazione, ma evidenziano piuttosto la natura della stessa che viene messa in evidenza ed espressa chiaramente all'interno dell'associazione.

Un'aggregazione viene utilizzata per indicare che, oltre ad avere attributi propri, l'istanza di una classe può comprendere istanze di altre classi; la relazione si legge tipicamente come “... **possiede un...**”.

Convenzionalmente viene definita una relazione “tutto-parti”, nel senso che un oggetto, che nella relazione rappresenta il “tutto”, è composto da più parti, designate da oggetti di tipi diversi. In questo senso, la relazione non può essere ciclica; ciò significa che una “parte” non può a sua volta contenere il “tutto”.

L'aggregazione è una versione più forte dell'associazione, ed al contrario di questa implica che il possesso può durare anche per l'intero ciclo di vita dell'oggetto, ma anche che la distruzione del “tutto” non comporta necessariamente la distruzione delle “parti”.

A questo punto è ovvio che la navigabilità e la molteplicità risiedono solo in un lato del diagramma.

In UML un'aggregazione viene rappresentata con un rombo vuoto dalla parte della classe che rappresenta il “tutto”.



Figura 3-11 Relazione di aggregazione nel diagramma UML.

Anche la classe *Nave* può avere una relazione di aggregazione. Basti pensare che essa non viene costruita come un monoblocco di metallo, ma è realizzata assemblando componenti come Motore, Scafo, Computer di Bordo, eccetera.

Alla luce di questa considerazione il diagramma UML che segue mostra la relazione tra la classe *Nave* e le parti che la compongono:

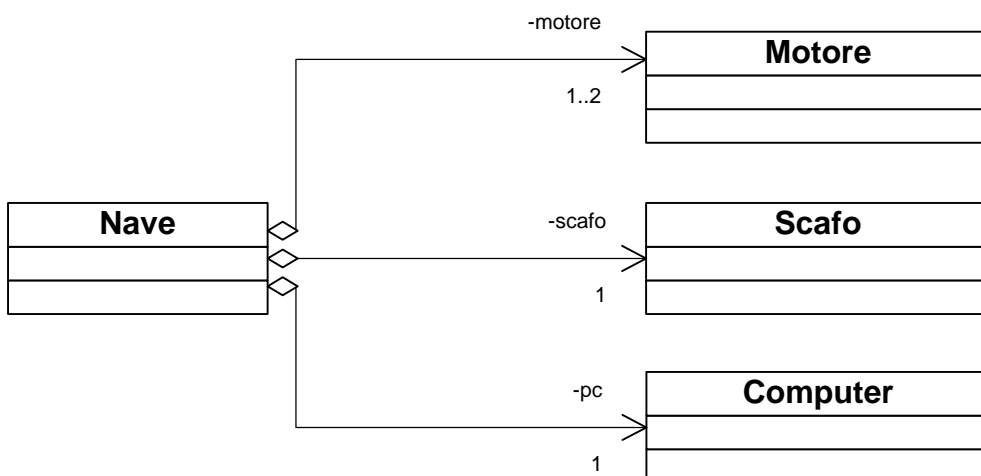


Figura 3-12 Diagramma per la rappresentazione della aggregazione di classi.

Come si può notare, le indicazioni di navigabilità e molteplicità sono tutte orientate dalla classe che rappresenta il “tutto” verso quelle che ne costituiscono la “parti”.

L'implementazione in .NET di un'aggregazione può essere quindi identica a quella di una associazione, stabilendo un collegamento ad un oggetto già esistente mediante l'utilizzo di proprietà o costruttori.

5 Composizione

La composizione, al pari dell'aggregazione, rappresenta una relazione del tipo "tutto-parti", ma contrariamente all'aggregazione implica che le istanze debbano avere il medesimo ciclo di vita. Ciò significa che quando viene creato il "tutto" vengono creati anche i suoi componenti e che quando viene cancellato il "tutto", anche i suoi componenti vengono cancellati.

Questo implica sempre una molteplicità di 1 o 0..1 in quanto non più di un oggetto per volta può essere responsabile del ciclo di vita dell'altro oggetto.

Nell'aggregazione questo non vale ed un componente può esistere senza essere parte di un assemblaggio così come l'assemblaggio può essere costituito da componenti che esistevano prima della sua costituzione. Quindi la composizione risulta una forma più stringente dell'aggregazione.

Una relazione di composizione si legge come "...è parte di...", il che significa che si deve leggere la composizione dalla "parte" verso il "tutto".

In UML la notazione per la composizione è molto simile a quella dell'aggregazione con la differenza che il rombo è colorato.



Figura 3-13 Relazione di composizione nel diagramma UML.

Tornando alla classe `Nave`, si può modificare la relazione di aggregazione vista in precedenza, decidendo di renderla più restrittiva e quindi trasformandola in una composizione.

Il diagramma si trasforma in quello seguente.

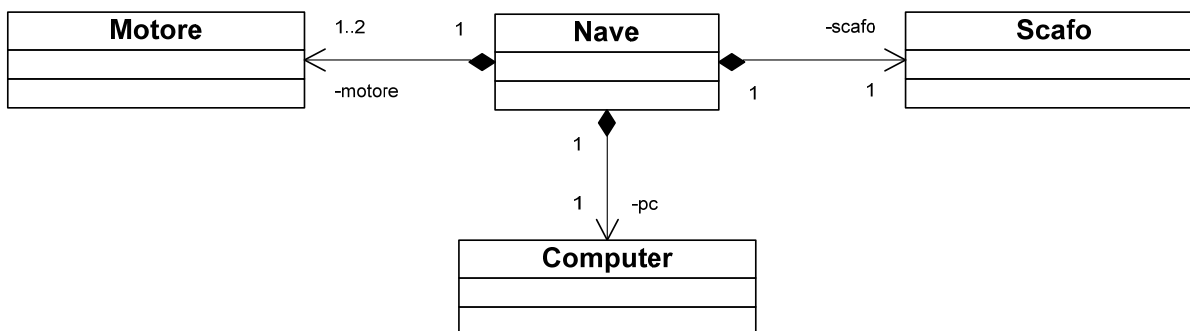


Figura 3-14 Diagramma per la rappresentazione della composizione di classi.

A questo punto ci si potrebbe chiedere a cosa serva avere sia l'aggregazione che la composizione (ed in effetti alcuni sostengono che lo standard UML 2.0 sconsigli l'utilizzo dell'aggregazione), ma in fin dei conti, ci potrebbero essere delle situazioni in cui si vuole illustrare con chiarezza questo concetto.

Dal punto di vista implementativo, il concetto di composizione (che lo ripetiamo, lega i cicli di vita degli oggetti) obbliga a procedere diversamente a seconda della molteplicità. Quando un

componente ha molteplicità con limite inferiore 1 allora lo stesso deve essere creato al momento della creazione dell'oggetto composto e quindi all'interno del costruttore.

Ipotizzando di avere un attributo che rappresenti un motore questa sarà l'implementazione:

```
Motore _motore1;
public Motore Motore1
{
    get { return _motore1; }
    set { _motore1 = value; }
}

public Nave(string nome, double stazza, int velocita, bool varata)
{
    // ... qui si inizializzano gli attributi della nave
    _motore1 = new Motore();
}
```

Se invece il componente ha molteplicità con limite inferiore 0 (0..) allora il componente può essere creato in un qualsiasi momento dopo la creazione dell'oggetto composto (ma sempre e comunque prima della cancellazione dello stesso), magari attraverso un metodo.

```
public void InstallaMotore(int potenza, int numGiri)
{
    _motore1 = new Motore(potenza, numGiri)
}
```

6 Generalizzazione

La generalizzazione è un diverso tipo di relazione tra classi e in UML viene come una freccia che parte dalle classi più specializzate (dette sottotipi) verso quella più generale (detta supertipo); la relazione può essere letta come “... un tipo di...”

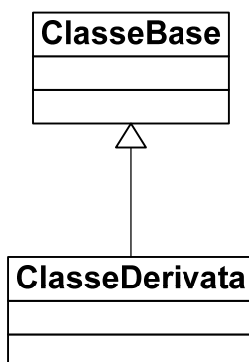


Figura 3-15 Relazione di generalizzazione nel diagramma UML

La relazione di generalizzazione è l'elemento fondante di un altro degli aspetti principali della programmazione ad oggetti, l'ereditarietà. Questo tipo di relazione è così importante da dovergli dedicare un intero capitolo, il prossimo.

4

Ereditarietà

“Ereditarietà” e “polimorfismo” rappresentano gli elementi caratterizzanti della programmazione orientata agli oggetti. I due, benché distinti, sono concetti strettamente connessi tra loro, tanto che è di fatto impraticabile trattare l’uno senza prendere in considerazione anche l’altro.

D’altra parte è l’ereditarietà a rappresentare l’elemento sul quale si fonda poi il polimorfismo. In linea di principio è infatti possibile scrivere programmi che applichino i principi dell’ereditarietà e non quelli del polimorfismo, ma non è possibile fare il contrario.

Allo scopo di semplificare l’esposizione, i due argomenti saranno inizialmente affrontati separatamente. Questo capitolo è dedicato interamente alla comprensione dei meccanismi e delle regole fondamentali relative all’ereditarietà.

1 “Tipi” e “tipi derivati”

“Tipo di dato” è un termine che ricorre spesso in un testo d’informatica. C# espone un insieme di tipi predefiniti, alcuni dei quali sono convenzionalmente chiamati tipi semplici: `int`, `double`, `bool`, eccetera. Esistono poi oggetti che non rappresentano un valore nel senso abituale che diamo a questo termine: gli array e le collezioni dinamiche come `List<>` e `Dictionary<>`, ad esempio. Vi sono inoltre i controlli: `Button`, `TextBox`, `Label`, `ListBox`, eccetera; tali oggetti hanno la caratteristica di esibire dei “comportamenti” esternamente osservabili e di interagire con l’utente. Infine, attraverso le parole chiave `class` e `struct` il programmatore può definire tipi di dati propri, stabilendone la rappresentazione interna, la modalità di creazione, le operazioni ammissibili, eccetera.

In sostanza, il concetto di tipo designa la modalità di rappresentazione degli oggetti e le operazioni che possono essere eseguite su di essi. In questo senso non esiste una sostanziale distinzione tra un vettore di interi, una variabile reale, un `TextBox` o una “nave”, poiché tutti e quattro sono degli oggetti appartenenti a determinati tipi: `int[]`, `double`, `TextBox`, `Nave`.

I tipi svolgono un ruolo fondamentale nella programmazione, permettendo di fornire una rappresentazione delle entità che caratterizzano il dominio del problema e di definire degli oggetti necessari allo svolgimento dei compiti del programma, compreso quello di interagire con l’utente. Un aspetto fondamentale degli oggetti del mondo reale, e dunque anche degli oggetti del dominio del problema, è che alcuni di essi si “somigliano”: condividono alcune caratteristiche generali, per poi diversificarsi in base ad altre.

Ad esempio, un’automobile, una moto e un camion sono senz’altro tipi di veicoli diversi tra loro, ma tutti e tre condividono perlomeno le seguenti caratteristiche: sono motorizzati e si muovono su ruote. Tutti, cioè, rientrano in una categoria che potremmo chiamare “veicolo motorizzato su ruote”, la quale definisce alcune caratteristiche generali degli oggetti ad essa appartenenti, ma non altre, come ad esempio l’uso – trasporto merci o persone – che viene fatto del veicolo.

D’altra parte esistono anche veicoli motorizzati che non si spostano su ruote (barche a motore, elicotteri ed aerei). Possiamo cioè individuare una categoria di veicoli ancora più generica della precedente, che potremmo chiamare “veicolo motorizzato”, nella quale rientrano tutti i veicoli a motore, che si muovano su ruote oppure no.

Le categorie – o tipi – appena definite hanno le seguenti relazioni tra loro:

- ❑ un “veicolo motorizzato” è un tipo di “veicolo”;
- ❑ un “veicolo motorizzato su ruote” è un tipo di “veicolo motorizzato”;
- ❑ “automobile”, “moto” e “camion” sono tipi di “veicolo motorizzato su ruote”
- ❑ “aereo”, “elicottero” e “barca a motore” sono tipi di “veicolo motorizzato”;

Di seguito è mostrato una schema che rappresenta tali relazioni. Nello schema, gli ovali identificano i tipi di veicoli, mentre i rettangoli identificano dei veicoli in particolare. Un tipo di veicolo definisce gli attributi caratteristici che accomunano tutti i veicoli di quel tipo. Ma la maggior parte dei tipi di veicolo non nasce nel vuoto, ma “deriva” da un tipo più generico, il quale definisce attributi più generali che riguardano un insieme più esteso di veicoli (quello di avere le ruote, per esempio). Questa relazione è denotata da: “un tipo di”.

Affermare che il tipo “veicolo motorizzato con ruote” deriva dal tipo “veicolo motorizzato” significare affermare che il primo eredita tutti gli attributi che caratterizzano il secondo; lo stesso vale per le altre relazioni “un tipo di”. Una “automobile”, dunque, possiede un motore e viaggia su ruote perché deriva direttamente da “veicolo motorizzato su ruote” e indirettamente da “veicolo motorizzato”, e dunque eredita gli attributi di questi. Non esiste alcun bisogno di precisarli per il tipo “automobile”, né per i tipi “camion” e “moto”, poiché entrambi sono definiti a monte.

La relazione “un tipo di” rappresenta dunque una relazione di ereditarietà; essa stabilisce che un determinato soggetto (le automobili, ad esempio) eredita le caratteristiche di un secondo soggetto (i veicoli motorizzati su ruote).

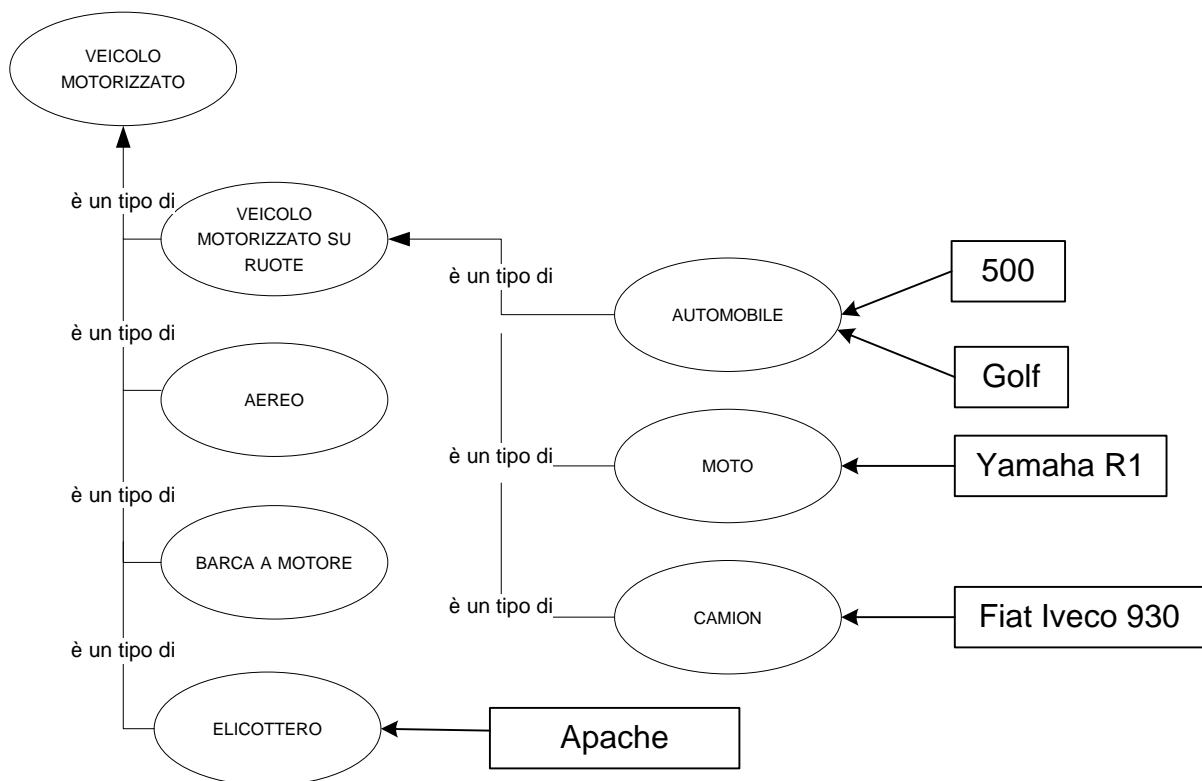


Figura 4-1 Schematizzazione della gerarchia dei veicoli motorizzati.

1.1 Esempio concreto di ereditarietà

Dopo aver introdotto in modo generale il concetto di ereditarietà, il quale collega due tipi di oggetti attraverso la relazione “un tipo di”, e prima di cominciare ad analizzarlo con maggior precisione, esamineremo la sua applicazione riprendendo in considerazione la classe `Nave`.

Anche se potrà sembrare strano, quelle che solcano i nostri mari non sono semplicemente delle `Navi`. Per meglio dire ci sono anche delle semplici imbarcazioni, ma è più probabile vederne qualcuna addetta al trasporto delle merci oppure a quello di passeggeri.

Ogni tipo di nave può essere quindi specializzata per svolgere un determinato ruolo all'interno di una flotta navale e mettere a disposizione dei servizi a seconda del proprio ruolo. Ogni nave è caratterizzata da attributi e da comportamenti comuni a tutte quante le imbarcazioni e da attributi e comportamenti che la differenziano a seconda della sua tipologia.

Nel nostro caso semplificato le caratteristiche comuni sono Nome, Stazza, Velocità, la presenza di un comandante eccetera.

Mentre, ad esempio, le navi mercantili hanno anche un carico pagante trasportabile, quelle da crociera avranno un numero di passeggeri imbarcati.

Se non volessimo utilizzare il concetto di ereditarietà e volessimo definire i due tipi di nave mercantile e passeggeri, saremmo costretti a scrivere le nostre classi nel seguente modo:

```
public class NaveMercantile
{
    string _nome;           // nome della nave
    double _stazza;         // stazza della nave
    int _velocita;          // velocità massima della nave
    . . .
    double _caricoPagante;  // carico trasportato
    . . .

    public Nave(string nome, double stazza, int velocita, bool varata)
    {
        . . .
    }
}

public class NavePasseggeri
{
    string _nome;           // nome della nave
    double _stazza;         // stazza della nave
    int _velocita;          // velocità massima della nave
    . . .
    int _numPassegeri;      // passeggeri imbarcati
    . . .

    public Nave(string nome, double stazza, int velocita, bool varata)
    {
        . . .
    }
}
```

Per ogni classe sono stati riportati soltanto tre campi membro comuni, un campo membro specifico della classe e lo scheletro del costruttore di default.

Graficamente, le due classi possono essere così rappresentate:

NaveMercantile	NavePasseggeri
-nome : string	-nome : string
-stazza : double	-stazza : double
-velocità : int	-velocità : int
-caricoPagante : double	-numPasseggeri : int
+NaveMercantile()	+NavePasseggeri()

Figura 4-2. Rappresentazione delle classi NaveMercantile e NavePasseggeri.

Questo approccio presenta notevoli problemi. Innanzitutto è necessario duplicare per ogni classe il codice relativo alla dichiarazione e all'uso dei campi comuni. Nell'esempio sono stati riportati tre campi soltanto, ma nella realtà, tra campi e funzioni membro, potrebbero esistere molte caratteristiche condivise tra due o più classi. Ma soprattutto, al di là di dover scrivere inizialmente più volte lo stesso codice, una volta che si dovesse modificare l'implementazione della parte comune, tale modifica dovrebbe essere replicata su tutte le classi che devono condividerla. Ad esempio, si supponga di volere aggiungere la funzione di stima del tempo di arrivo di una nave data la distanza da percorrere. Per far questo occorre aggiungere ad ogni classe un nuovo metodo che effettui il calcolo.

Sarebbe desiderabile un approccio diverso. In fondo ogni tipo di nave rappresenta una versione specializzata di una nave generica che definisce le caratteristiche comuni a tutte le imbarcazioni.

Possiamo dunque definire la classe generica *Nave*, come fatto in precedenza:

```
public class Nave
{
    string _nome;           // nome della nave
    double _stazza;         // stazza della nave
    int _velocita;          // velocità massima della nave
    . . .

    public Nave(string nome, double stazza, int velocita, bool varata)
    {
        . . .
    }
}
```

Essa definisce soltanto i membri comuni a tutti i tipi d'imbarcazione. A questo punto entra in scena l'ereditarietà e cioè il meccanismo che consente di "specializzare" un tipo di dato facendolo derivare da un altro, ottenendo così che il primo erediti tutte le caratteristiche del secondo:

```
public class NaveMercantile : Nave
{
    double _caricoPagante;   // carico trasportato
    . . .

    public Nave(string nome, double stazza, int velocita, bool varata)
    {
        . . .
    }
}
```

```

    }
}

public class NavePasseggeri : Nave
{
    int _numPasseggeri;        // Numero massimo passeggeri
    . . .

    public Nave(string nome, double stazza, int velocita, bool varata)
    {
        . . .
    }
}

```

Nell'intestazione di ogni classe viene indicato che essa deriva dalla classe `Nave`. Ciò equivale ad affermare che tutti i membri (campi e funzioni) definiti da `Nave` sono ereditati dalle classi in questione; a sua volta, questo significa che ogni nave possiede sia i membri definiti dalla sua classe di appartenenza sia quelli definiti dalla classe `Nave`.

Derivando le classi `NaveMercantile` e `NavePasseggeri` dalla classe `Nave` si ottiene il grande vantaggio di dover definire una sola volta gli attributi e i comportamenti comuni alle due classi, le quali si limiteranno a definire gli attributi e i comportamenti specializzati per la funzione che devono svolgere. Inoltre, qualora si decidesse di modificare una caratteristica comune esistente, o aggiungerne una nuova, come ad esempio il calcolo del tempo di arrivo, tale modifica avrebbe luogo nella sola classe `Nave`, per essere automaticamente ereditata dalle classi che derivano da essa.

Questa nuova situazione è riflessuta dal seguente schema:

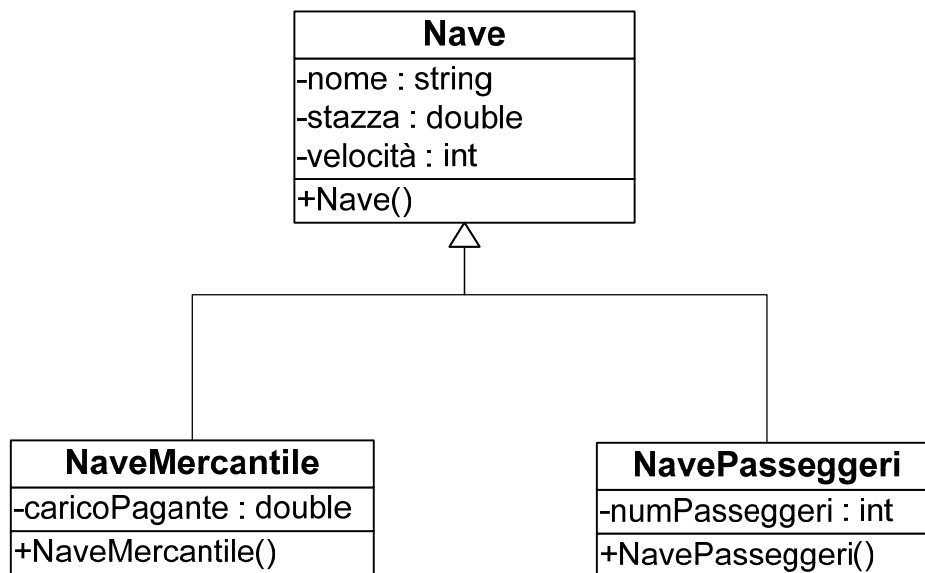


Figura 4-3 Schema della relazione di generalizzazione tra classe `Nave` e classi derivate.

2 Ereditarietà: “classi base” e “classi derivate”

L'ereditarietà rappresenta una relazione tra classi ed è una relazione di parentela tra una classe padre e una classe figlia, la quale eredita le caratteristiche della prima. Nell'ambito di tale relazione, la classe padre viene definita “classe base”, la classe figlia “classe derivata”.

Al riguardo esistono anche altri termini, sia per la classe base che per quella derivata.

- ❑ classe base: superclasse, classe ancestrale;
- ❑ classe derivata: sottoclasse, sottotipo.

E' possibile ereditare una classe da un'altra specificando il nome della seconda nell'intestazione della prima dopo il simbolo due-punti:

```
class ClasseDerivata : ClasseBase
{
    // corpo della classe derivata
}
```

Così facendo si ottiene che tutti i membri definiti da `ClasseBase` siano automaticamente definiti anche da `ClasseDerivata`. `ClasseDerivata` eredita dunque sia i campi che le funzioni membro appartenenti a `ClasseBase`, ai quali può aggiungere nuovi campi e funzioni, estendendo le capacità della classe genitrice.

La classe base viene specificata in quella che si chiama “lista base”, o “lista di derivazione”. Viene usata la parola lista benché la classe base sia una sola per due motivi. Il primo è che alcuni linguaggi di programmazione, tra cui il C++, consentono di far derivare una classe da più classi base. Il secondo motivo è dovuto al fatto che nella “lista base” oltre al nome di una classe, possono comparire uno o più nomi di *interfacce*.

Segue la definizione di due classi, delle quali la seconda deriva dalla prima:

```
class ClasseBase
{
    public int a;
    public double x;

    public void MetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}", a, x);
    }
}

class ClasseDerivata: ClasseBase
{
    public string s;
    public void AltroMetodoQualsiasi()
    {
        Console.WriteLine("s = {0}", s);
    }
}
```

Date tali definizioni, l'esecuzione del codice che segue:


```

ClasseBase B= new ClasseBase();
ClasseDerivata D = new ClasseDerivata();
B.a = -1;
B.x = 100;
D.a = 5;
D.s = "hello";

```

produce in memoria la seguente situazione:

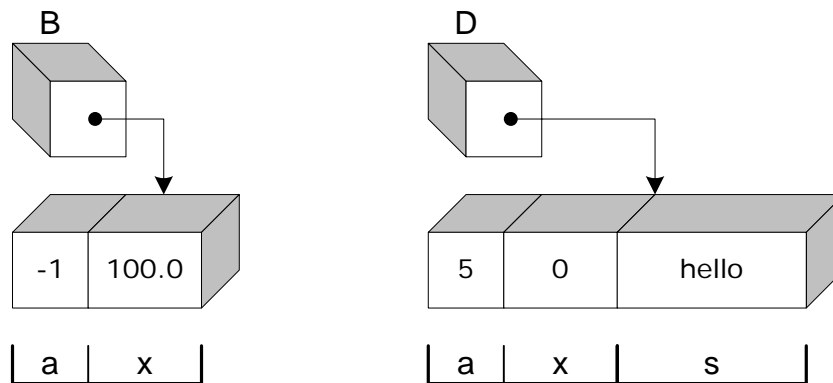


Figura 4-4 Rappresentazione in memoria degli oggetti B e D

Come si vede, l'oggetto D, oltre al campo `s` definito nella classe di appartenenza, contiene anche i campi `_a` e `_x`, i quali sono definiti in `ClasseBase`.

Naturalmente, l'ereditarietà funziona anche con le funzioni membro. `MetodoQualsiasi()`, definito in `ClasseBase`, può essere invocato anche attraverso oggetti di tipo `ClasseDerivata`. Il seguente codice è dunque assolutamente corretto:

```

B.MetodoQualsiasi();           // ok: niente di nuovo
D.MetodoQualsiasi();           // ok: invoca metodo definito nella classe base
D.AltroMetodoQualsiasi();      // ok: invoca metodo definito nella classe derivata

```

e produce in output:

```

a = -1    x = 100
a = 1000   x = 0
a = 1000   x = 0    s = hello

```

Non vale però il contrario: `AltroMetodoQualsiasi()` non può essere invocato attraverso oggetti di tipo `ClasseBase`, poiché essa non definisce affatto tale metodo. Dunque, il seguente codice è formalmente errato:

```

B.AltroMetodoQualsiasi();      // errore!

```

L'ereditarietà funziona dunque in una direzione soltanto.

L'esempio proposto ha soltanto carattere introduttivo, vi sono infatti diverse regole che intervengono nel meccanismo di derivazione di una classe, alcune delle quali saranno esaminate nel capitolo dedicato al polimorfismo.

2.1 Membri derivati e nuovi membri di una classe derivata

In precedenza abbiamo affermato che è come se i membri ereditati dalla classe base fossero implicitamente definiti nella classe derivata. In realtà la derivazione sottostà a delle regole che

stabiliscono cosa e come, della classe base, può essere utilizzato nella classe derivata. In questo senso è utile vedere la classe derivata suddivisa in due parti.

- ❑ la prima si riferisce ai membri ereditati dalla classe base;
- ❑ la seconda si riferisce ai nuovi membri (se esistono).

Tale distinzione non è soltanto convenzionale. L'esempio che segue riprende le due classi definite in precedenza, introducendo alcune modifiche, tra le quali l'impostazione a `private` del livello di protezione dei campi membro di `ClasseBase`:

```
class ClasseBase
{
    private int _a;
    private double _x;

    public ClasseBase()                // costruttore di default
    {
    }

    public ClasseBase(int a, double x)
    {
        _a = a;
        _x = x;
    }

    public void MetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}", _a, _x);
    }
}

class ClasseDerivata: ClasseBase
{
    public string s;
    public void AltroMetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}    s = {0}", _a, _x, s); // errore!
    }
}
```

Ebbene, nonostante `ClasseDerivata` non sia stata modificata rispetto all'esempio precedente, il tentativo di compilare questo codice produce due errori di tipo:

`ClasseBase.<nome membro>` è inaccessibile a causa del livello di protezione

relativi all'uso dei `_a` e `_x`. Infatti, essendo stati definiti come privati nella classe base, non sono accessibili al di fuori del loro campo d'azione, il quale è limitato alla classe che li definisce.

Il fatto che un membro della classe base sia dichiarato privato non significa che non venga ereditato dalla classe derivata, ma semplicemente che è possibile accedere ad esso soltanto attraverso codice che appartenga anch'esso alla classe base. All'interno di una classe derivata, dunque, i membri ereditati formano un sotto insieme distinto, il cui livello di accesso viene stabilito nella classe in cui sono definiti e non nella classe che li eredita.

2.2 Rendere accessibili i membri definiti nella classe base

L'esempio precedente dimostra che l'operazione di derivazione non riguarda la sola progettazione delle classe derivata; infatti il modo in cui è progettata la classe base produce a sua volta delle conseguenze. Da questo punto di vista, possiamo dire che esistono classi che sono progettate in modo da poter essere convenientemente derivate (al di là del fatto che lo siano oppure no) e classi che invece non lo sono.

La questione sta nella domanda: quanto, della rappresentazione interna di una classe, si vuole rendere accessibile alle classi che eventualmente deriveranno da essa? Una classe derivata, infatti, ha di norma bisogno di accedere alla rappresentazione interna che eredita dalla classe base.

Una soluzione ovvia è quella di dichiarare pubblici i membri della classe base che si vuole rendere accessibili nelle classi derivate. Questa non è però una soluzione valida, perché in questo modo si espone la rappresentazione interna della classe anche al codice *consumer*, cosa che viola l'importante principio dell'*information hiding*.

Per questo motivo il linguaggio ammette un ulteriore livello di protezione per i membri, che li mantiene inaccessibili al codice *consumer*, rendendoli però utilizzabili nelle classi derivate: esso è designato dalla parola chiave `protected`. Segue una nuova versione dell'esempio precedente, nella quale i campi membro vengono definiti protetti:

```
class ClasseBase
{
    protected int _a;
    protected double _x;

    public ClasseBase()
    {
    }

    public ClasseBase(int a, double x)
    {
        _a = a;
        _x = x;
    }

    public void MetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}", _a, _x);
    }
}

class ClasseDerivata: ClasseBase
{
    public string s;

    public void AltroMetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}    s = {2}", _a, _x, s); // ok
    }
}
```

Il codice è formalmente corretto, poiché i campi `_a` e `_x` sono ora accessibili anche nelle funzioni membro definite dalla classe derivata. D'altra parte, gli stessi campi restano inaccessibili al codice *consumer*.

2.3 Invocare i costruttori della classe base

I costruttori rappresentano delle funzioni membro particolari, poiché sono invocati implicitamente all'atto della creazione di un oggetto. C'è inoltre un'ulteriore caratteristica che li distingue dalle altre funzioni membro (e in questo caso anche dai campi membro):

i costruttori di una classe non vengono ereditati dalle classi che derivano da essa.

Ciò significa che una classe derivata deve comunque definire i propri costruttori, poiché non eredita quelli definiti dalla classe base. Ciò produce delle conseguenze notevoli, infatti la creazione di un oggetto che appartiene a una classe derivata deve produrre come minimo:

- ❑ l'inizializzazione dei membri ereditati e cioè l'invocazione di uno dei costruttori definiti dalla classe base (per semplicità: "costruttore base");
- ❑ l'inizializzazione dei nuovi campi membro (se esistono).

L'esecuzione di un costruttore base avviene specificando un "inizializzatore di costruttore" nell'intestazione del costruttore della classe derivata, secondo la sintassi:

```
public nome-classe-derivata(lista-parametriopz): base(lista-argomentiopz)
{
    // corpo costruttore
}
```

La parola chiave `base` funziona come sinonimo di "classe base", ed è in pratica un riferimento implicito alla parte di oggetto definita nella classe base.

Come esempio, forniamo un costruttore a `ClasseDerivata`:

```
class ClasseDerivata: ClasseBase
{
    public string s;

    public ClasseDerivata(int a, double x): base(a, x)
    {
        s = "hello";
    }
    ...
}
```

Nel caso in cui la classe base definisca più costruttori, il linguaggio decide quale invocare confrontando la lista degli argomenti con la lista dei parametri dei costruttori base, analogamente a quanto avviene per l'invocazione di un metodo sovraccaricato.

La creazione di un oggetto di tipo `ClasseDerivata` produce le seguenti azioni:

- 1) viene invocato il costruttore della classe derivata;
- 2) prima che venga eseguito il corpo del costruttore, viene invocato il costruttore base `ClasseBase(int a, double x)`;
- 3) viene eseguito il corpo del costruttore base;

- 4) dopo che l'esecuzione del costruttore base è terminata, viene eseguito il corpo del costruttore della classe derivata.

2.4 Invocazione implicita del costruttore di default della classe base

La costruzione di un oggetto appartenente ad una classe derivata avviene in due parti: viene prima costruita la parte di oggetto definita nella classe base, poi viene costruita la parte definita nella classe derivata. Ciò si traduce sempre nell'invocazione di almeno un costruttore per classe. Cosa accade, allora, se il costruttore della classe derivata non specifica la chiamata a un costruttore base? Il linguaggio fornisce automaticamente la chiamata al costruttore base di default.

Ad esempio, nella nuova versione di `ClasseDerivata`, il costruttore non specifica nessuna chiamata ad un costruttore base:

```
class ClasseDerivata: ClasseBase
{
    public string s;

    public ClasseDerivata(int a, double x)    // nessuna invocazione a base()
    {
        _a = a;    // inizializza campo ereditato
        _x = x;    // inizializza campo ereditato
        s = "hello";
    }

    public void AltroMetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}    s = {2}", _a, _x, s);    // ok!
    }
}
```

L'esecuzione del seguente codice:

```
ClasseDerivata D = new ClasseDerivata(10, 1000);
```

produce le seguenti azioni:

- 1) viene invocato il costruttore della classe derivata;
- 2) prima che venga eseguito il corpo del costruttore, viene invocato implicitamente il costruttore predefinito di `ClasseBase`;
- 3) viene eseguito il corpo del costruttore base;
- 4) dopo che l'esecuzione del costruttore base è terminata, viene eseguito il corpo del costruttore della classe derivata.

Abbiamo mostrato due approcci distinti nella definizione del costruttore della classe derivata:

- ❑ invocare il costruttore base mediante l'inizializzatore di costruttore;
- ❑ non invocare il costruttore base e inizializzare esplicitamente i campi ereditati.

I due approcci sono equivalenti, oppure uno dei due è da preferire? E' da preferire il primo. Infatti, si dovrebbe sempre evitare d'inizializzare i campi ereditati nel costruttore della classe derivata, invocando invece l'appropriato costruttore base, ciò per vari motivi:

- ❑ efficienza: con il secondo approccio esiste la concreta possibilità che i campi ereditati siano inizializzati due volte, la prima volta dal costruttore base, la seconda volta dal costruttore della classe derivata;
- ❑ compattezza del codice: perché scrivere due volte il codice di inizializzazione dei campi membro ereditati?

Esiste inoltre una terza possibilità e cioè che la classe base non definisca un costruttore di default. In questo caso il costruttore della classe derivata *deve* esplicitamente specificare un costruttore della classe base, altrimenti il compilatore segnalerà un errore del tipo:

```
Nessun overload del metodo ClasseBase accetta '0' argomenti
```

3 Ridefinire i membri della classe base

L'obiettivo della derivazione di una classe è solitamente quello di estendere o modificare le caratteristiche della classe base. Ciò si ottiene:

- ❑ aggiungendo nuovi campi e funzioni membro;
- ❑ ridefinendo le funzioni membro della classe base, facendo sì che svolgano altri compiti, oppure che svolgano gli stessi compiti ma mediante una diversa implementazione.

Sul primo punto c'è poco da dire, poiché l'ereditarietà non svolge alcun ruolo nella definizione di nuovi membri da parte di una classe derivata. Ridefinire uno o più membri della classe base rappresenta invece un'operazione centrale della derivazione; essa sottostà a svariate regole e può assumere obiettivi diversi. Il più importante di questi è strettamente connesso con il polimorfismo, che sarà esaminato nel prossimo capitolo. Per ora ci limiteremo a studiare i meccanismi che consentono di fornire una nuova versione di un membro ereditato.

3.1 Definire un campo membro con lo stesso nome di un campo ereditato

Questa operazione produce la conseguenza di “nascondere” il nome del campo ereditato, al quale, nel codice della classe, si potrà comunque accedere qualificandolo come appartenente alla classe base. Ad esempio, nel seguente codice `ClasseBase` definisce un campo `double` di nome `a`, mentre `ClasseDerivata` definisce un campo intero con lo stesso nome:

```
class ClasseBase
{
    public double a = -1;
    public void MetodoQualsiasi()
    {
        Console.WriteLine("a = {0}", a);
    }
}

class ClasseDerivata: ClasseBase
{
    public int a = 10;           // campo ridefinito
    public void AltroMetodoQualsiasi()
    {
```

```

        Console.WriteLine("a (ereditato) = {0}", base.a);
        Console.WriteLine("a (ridefinito) = {0}", a); }
    }

```

Date queste definizioni, un oggetto della classe derivata memorizza due campi di nome `a`, dei quali il campo intero nasconde quello `double`, definito nella classe base:

```

ClasseDerivata D = new ClasseDerivata();
int b = D.a;          // accede al campo int e non al campo double!

```

E' importante comprendere che gli oggetti della classe derivata contengono effettivamente due variabili di nome `a`. D'altra parte non c'è conflitto di nomi, poiché ognuna delle due variabili ha il proprio campo d'azione, stabilito dalla classe che la definisce. Per stabilire a quale variabile faccia effettivamente riferimento il nome in questione, il linguaggio applica le normali regole sulla risoluzione dei nomi:

ogni riferimento ad un nome viene risolto a favore dell'oggetto definito nello stesso campo d'azione della funzione membro.

Ciò significa che in `AltroMetodoQualsiasi()` ogni riferimento ad `a` è un riferimento alla variabile intera definita dalla classe derivata. D'altra parte, in `MetodoQualsiasi()`, il cui campo d'azione è quello di `ClasseBase`, ogni riferimento ad `a` è un riferimento alla variabile `double` definita dalla classe base. Ciò detto, l'esecuzione del seguente codice:

```

D.MetodoQualsiasi();
D.AltroMetodoQualsiasi();

```

produce in output:

```

a = -1
a (ereditato) = -1
a (ridefinito) = 10

```

Come dimostra l'esempio, anche in `AltroMetodoQualsiasi()` è possibile accedere al campo `a` ereditato, purché questo sia qualificato come campo definito nella classe base; ciò si ottiene premettendo al campo la parola chiave `base`.

Di norma non esiste alcun motivo per ridefinire un campo, poiché in questo modo un oggetto memorizza due variabili distinte con lo stesso nome. Proprio per questo motivo il compilatore emette un *warning*, segnalando che così facendo si nasconde il campo ereditato, cosa che potrebbe essere fonte di errori o comunque di ambiguità nel codice.

Nel caso in cui si desideri effettivamente nascondere il campo ereditato, è possibile sopprimere l'avvertimento del compilatore utilizzando la parola `new` nella dichiarazione:

```

class ClasseDerivata: ClasseBase
{
    new public int a = 10;
    ...
}

```

con la quale si afferma esplicitamente il desiderio di definire un campo con lo stesso nome di un campo ereditato.

3.2 Definire funzioni membro con lo stesso nome di funzioni ereditate

Se ci si limita a quanto detto finora, la definizione di funzioni membro con lo stesso nome di funzioni ereditate sottostà alle stesse regole viste per la ridefinizione di campi membro, con l'importante differenza che esiste la possibilità di sovraccaricare metodi e indicizzatori, possibilità che non è ovviamente prevista per i campi membro (né per le proprietà).

Consideriamo per il momento l'ipotesi di definire un nuovo metodo che abbia lo stesso prototipo di un metodo ereditato. In questo caso, come avviene per i campi, il metodo ereditato viene nascosto. Ad esempio, ecco una nuova implementazione di `ClasseBase` e `ClasseDerivata`; questa volta la classe derivata fornisce una nuova versione di `MetodoQualsiasi()`:

```
class ClasseBase
{
    protected int _a = -1;
    protected double _x = 10;
    ...
    public void MetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}", _a, _x);
    }
}

class ClasseDerivata: ClasseBase
{
    public string s = "hello";

    new public void MetodoQualsiasi()    // nuova versione del metodo ereditato
    {
        base.MetodoQualsiasi();          // invocazione metodo ereditato
        Console.WriteLine("a = {0}    x = {1}    s = {2}", _a, _x, s);
    }
}
```

Come si vede, anche in questo caso viene utilizzata la parola chiave `new` per sopprimere il *warning* del compilatore.

Sia nel codice *consumer* che all'interno della classe derivata, qualsiasi riferimento al metodo in questione viene risolto a favore del metodo ridefinito. D'altra parte, all'interno della classe derivata è ancora possibile invocare il metodo ereditato qualificandolo con la parola chiave `base`. Detto ciò il seguente codice:

```
ClasseDerivata D = new ClasseDerivata();
D.MetodoQualsiasi();
```

produce in output:

```
a = -1    x = 10
a = -1    x = 10    s = hello
```

Quanto mostrato per i metodi vale anche per le proprietà e gli indicizzatori; come sappiamo, infatti, entrambi vengono tradotti in una coppia di metodi *accessor*, i quali sono soggetti alle stesse regole di qualsiasi altro metodo della classe.

3.3 Sovraccaricare il metodo ereditato

Per quanto riguarda metodi e indicizzatori – e dunque non le proprietà – la situazione cambia se il nuovo metodo ha una diversa lista parametri rispetto al metodo ereditato. In questo caso, poiché il linguaggio per risolvere l'invocazione a una funzione membro non usa soltanto il nome ma anche la lista dei parametri, il nuovo metodo non nasconde il metodo ereditato ma lo sovraccarica:

```
class ClasseBase
{
    protected int _a = -1;
    protected double _x = 10;
    ...
    public void MetodoQualsiasi()
    {
        Console.WriteLine("a = {0}    x = {1}", _a, _x);
    }
}

class ClasseDerivata: ClasseBase
{
    public string s = "hello";

    public void MetodoQualsiasi(double z)
    {
        MetodoQualsiasi();           // invocazione metodo ereditato
        Console.WriteLine("a = {0}    x = {1}    s = {2}    z = {3} ", _a, _x, s,
z);
    }
}
```

Come si vede:

- ❑ non c'è più alcun bisogno di usare la parola `new` per sopprimere il *warning* del compilatore, poiché il nuovo metodo non nasconde affatto il metodo ereditato;
- ❑ all'interno del codice della classe non è più necessario qualificare il metodo ereditato mediante la parola chiave `base`;
- ❑ nel codice *consumer* si può fare riferimento a entrambi i metodi.

Detto ciò, il seguente codice:

```
ClasseDerivata D = new ClasseDerivata();
D.MetodoQualsiasi();
D.MetodoQualsiasi(1000);
```

produce in output:

```
a = -1    x = 10
a = -1    x = 10    s = hello    z = 1000
```

3.4 Conclusioni

Il concetto di ridefinizione, così come è stato applicato in questo paragrafo, è da intendersi soltanto come termine convenzionale. In realtà, negli esempi proposti non è stato ridefinito alcunché, ma soltanto fornito un nuovo membro che possiede lo stesso nome di un membro ereditato. Ciò produce la conseguenza di nascondere il membro ereditato, il quale non è più accessibile al codice *consumer* e, nel codice della classe, lo è soltanto se qualificato con l'appropriata parola chiave *base*.

Il concetto di ridefinizione vero e proprio, applicabile soltanto alle funzioni membro, sarà affrontato nel prossimo capitolo e si riferisce alla possibilità della “derivazione virtuale”, e cioè di fornire una diversa implementazione a una funzione membro che per il resto mantiene completamente invariate le sue caratteristiche.

Polimorfismo

1 Premessa al polimorfismo

Una conseguenza fondamentale dell'ereditarietà fa sì che un oggetto possa essere considerato come appartenente non solo alla classe che lo definisce, ma anche alla classe (o alle classi) base di questa. Ciò è il risultato della relazione di parentela “è un”:

dati i tipi classe-B e classe-D, dove il secondo deriva dal primo, un oggetto di tipo classe-D può essere considerato anche appartenente al tipo classe-B. In altre parole: un oggetto di tipo Classe-D è anche un oggetto di tipo Classe-B.

Un esempio chiarirà il concetto. Il codice che segue definisce due classi, delle quali la seconda deriva dalla prima:

```
class ClasseBase
{
    public string nome;
    public void VisualizzaNome()
    {
        Console.WriteLine(nome);
    }
}

class ClasseDerivata: ClasseBase
{
    public string cognome;
    public void VisualizzaNomeCognome()
    {
        Console.WriteLine(nome + " " + cognome);
    }
}
```

Tra le due esiste una relazione che si legge: ClasseDerivata è un tipo di ClasseBase. Tale relazione si estende anche agli oggetti, e cioè: un oggetto di tipo ClasseDerivata può essere considerato come se fosse di tipo ClasseBase. Ed è così perché un oggetto di tipo ClasseDerivata contiene anche i membri definiti da ClasseBase e quindi tutto ciò che caratterizza gli oggetti di tipo ClasseBase caratterizza anche gli oggetti di tipo ClasseDerivata.

Sulla base di queste affermazioni è possibile scrivere il seguente codice:

```
ClasseBase B = new ClasseDerivata();
B.Nome = "Albert";
B.VisualizzaNome();
```

Il cui risultato in output è:

Albert

In esso, alla variabile B di tipo `ClasseBase` viene assegnato il riferimento a un oggetto di tipo `ClasseDerivata`. L'istruzione evidenziata in grigio produce in memoria la seguente situazione:

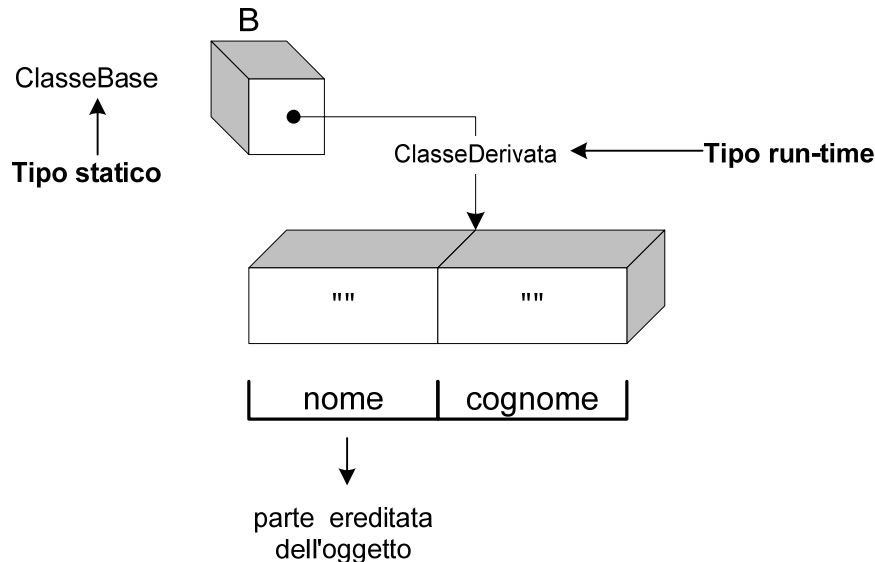


Figura 5-1 Rappresentazione in memoria dell'oggetto referenziato dalla variabile B

La figura evidenzia che l'oggetto effettivamente creato è di tipo `ClasseDerivata`, e ciò è dovuto al costruttore specificato nell'istruzione di creazione. D'altra parte, l'oggetto viene utilizzato come se fosse di tipo `ClasseBase`, poiché è proprio questo il tipo della variabile **B**. In una simile situazione, esistono dunque due tipi in gioco:

- ❑ il tipo della variabile, che è anche chiamato “tipo statico” ed è definito esplicitamente nella dichiarazione (nell'esempio precedente è `ClasseBase`);
- ❑ il tipo effettivo dell'oggetto. Questo è denominato “tipo run-time”, poiché soltanto durante l'esecuzione del programma, dopo l'invocazione del costruttore, l'oggetto esiste e quindi il suo tipo è disponibile.

In sostanza viene creato un oggetto di tipo `ClasseDerivata`, ma di questo vengono utilizzati soltanto i membri definiti in `ClasseBase`, poiché è `ClasseBase` il tipo della variabile attraverso la quale si usa l'oggetto.

Dunque, rappresenta un errore formale accedere ai membri dell'oggetto che *non* sono definiti in `ClasseBase`, come mostra il seguente codice:

```
B.Cognome = "Einstein";           // errore!
B.VisualizzaNomeCognome();       // errore!
```

In conclusione, l'ereditarietà, con la sua relazione “è un”, fa sì che una variabile di un certo tipo possa riferirsi ad oggetti di un tipo derivato da esso, purché l'uso che viene fatto della variabile sia coerente con il tipo di appartenenza (tipo statico).

1.1 Relazione “un tipo di” applicata agli oggetti

La possibilità per una variabile di un determinato tipo di referenziare oggetti di un tipo derivato è fondamentale nella OOP ed è alla base del polimorfismo, anche se da sola non è sufficiente. Mostreremo il perché di entrambe le riprendendo l'esempio sviluppato precedentemente

All'interno della flotta navale sono presenti due categorie di navi: la nave semplice e la nave mercantile: la nave semplice serve ad un uso interno della flotta, mentre quella mercantile può trasportare un certo carico pagante. Si vuole calcolare il costo di ogni viaggio.

Rappresentiamo il dominio del problema mediante due classi: *Nave* e *NaveMercantile*.

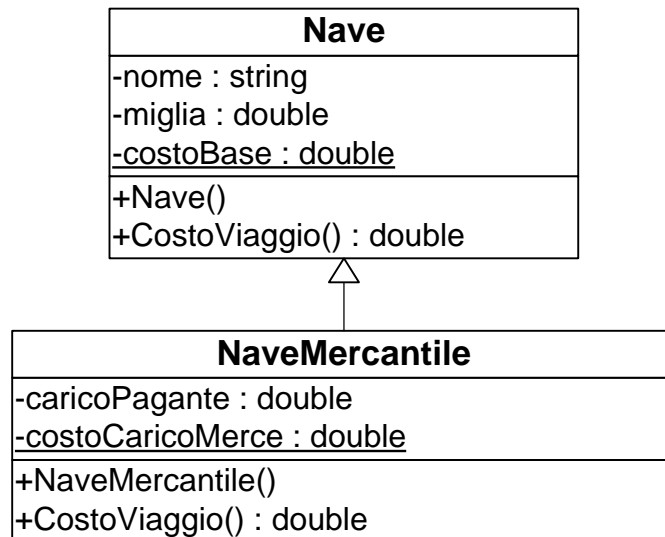


Figura 5-2 Schema delle classi *Nave* e *NavePasseggeri*

Il tipo *Nave* definisce gli attributi e le funzioni di base relative ad ogni nave della flotta, sia essa specializzata o meno:

- ❑ “nome”;
- ❑ “miglia da percorrere”;
- ❑ “costo base”, il quale è definito come campo statico poiché contiene la costante da applicare per il calcolo di ogni viaggio di una nave (valido per tutte le navi);
- ❑ un metodo che ritorna la stima del costo del viaggio di una nave.

Il tipo *NaveMercantile* eredita ovviamente gli attributi “nome”, “miglia” e “costo avviamento” e:

- ❑ definisce due nuovi attributi: “carico pagante” e “costo carico merce”;
- ❑ ridefinisce il metodo di calcolo del costo del viaggio, poiché nel calcolo devono essere inclusi i costi del carico pagante.

L’implementazione del codice delle due classi è il seguente.

```

public class Nave
{
    protected static double _costoBase = 0.10;

    protected string _nome;
    public string Nome
    {
        get { return _nome; }
    }
}
  
```

```
protected double _miglia;
public double Miglia
{
    get { return _miglia; }
}

public Nave(string nome, double miglia)
{
    _nome = nome;
    _miglia = miglia;
}

public double CostoViaggio()
{
    return _costoBase * _miglia;
}
}

public class NaveMercantile : Nave
{
    static double _costoCaricoMerce = 0.20;

    double _caricoPagante;

    public NaveMercantile(string nome, double miglia, double caricoPagante)
        : base(nome, miglia)
    {
        _caricoPagante = caricoPagante;
    }

    public new double CostoViaggio()
    {
        return _costoBase * _miglia + _caricoPagante * _costoCaricoMerce;
    }
}
```

Mettiamo entrambe le classi alla prova con un breve frammento di codice *consumer*:

```
Nave nave = new Nave("Bally",100);

NaveMercantile naveMer;
naveMer = new NaveMercantile("Trasporter", 200, 220);

Console.WriteLine("Nave:{0}, Costo del viaggio: {1}", nave.Nome,
nave.CostoViaggio());
Console.WriteLine();
Console.WriteLine("Nave mercantile:{0}, Costo del viaggio: {1}", naveMer.Nome,
naveMer.CostoViaggio());
Console.WriteLine();
```

L'output è:

Nave:Bally, Costo del viaggio: 10

Nave mercantile:Trasporter, Costo del viaggio: 64

Il tutto funziona, ma quello presentato non è certo un esempio realistico di codice *consumer*. Ciò che serve è ovviamente poter gestire una lista di oggetti *Nave* e *NaveMercantile*, ad esempio memorizzati in un vettore oppure in una collezione dinamica. In questo caso, se non avessimo applicato l'ereditarietà, saremmo di fronte ad un serio problema, infatti: quale dovrebbe essere il tipo degli elementi del vettore? Un tipo escluderebbe l'altro. In assenza di ereditarietà potremmo adottare due soluzioni:

- ❑ gestire due vettori, uno per le navi semplici e l'altro per le navi mercantili, una soluzione senz'altro insoddisfacente, poiché costringerebbe a duplicare il codice *consumer* per ogni tipo di nave;
- ❑ definire un'unica classe *Nave*, contenente anche gli attributi necessari per rappresentare le navi mercantili, più un campo il cui solo scopo è quello di stabilire se un oggetto si riferisce a una nave semplice o mercantile. Tale campo verrebbe usato nel metodo *CostoViaggio()* per discriminare il tipo di calcolo da effettuare.

Anche la seconda soluzione non è soddisfacente, poiché implica la definizione di un tipo che non ha un equivalente nel dominio del problema.

L'ereditarietà rende attuabile l'approccio migliore; infatti possiamo dichiarare un vettore di tipo *Nave* e successivamente istanziarne i vari elementi in base al tipo di nave – semplice o mercantile – da memorizzare.

Segue un esempio:

```
Nave [] flotta = new Nave[3];
```

```
flotta[0] = new Nave("Bally", 100);
```

```
flotta[1] = new NaveMercantile("Trasporter", 200, 220);
```

```
flotta[2] = new Nave("Nottera", 300);
```

Ogni elemento del vettore rappresenta un riferimento di tipo *Nave*. Ma il tipo dell'oggetto effettivamente referenziato – il tipo run-time – dipende dall'istruzione di creazione e può essere uno o l'altro tipo. Ciò è reso possibile dalla relazione “è un”, poiché una *NaveMercantile* è anche una *Nave* e dunque può essere referenziato da una variabile di questo tipo.

E' adesso possibile scrivere del codice che iteri attraverso il vettore svolgendo una qualche elaborazione, come ad esempio visualizzare il nome delle navi della flotta:

```
for (int i = 0; i < flotta.Length; i++)
{
    Console.WriteLine(flotta[i].Nome);
}
```

Il risultato in output è:

Bally

Trasporter

Nottera

Il codice funziona correttamente, poiché la proprietà *Nome* è definita dalla classe *Nave* ed è ereditata da *NaveMercantile*; è quindi utilizzabile qualunque sia l'oggetto effettivamente referenziato. Su

questa base, ci si aspetta che funzioni anche il codice seguente, che calcola il costo dei viaggi di ogni nave:

```
for (int i = 0; i < flotta.Length; i++)
{
    Console.WriteLine("Nave:{0}, Costo del viaggio: {1}", flotta[i].Nome,
                      flotta[i].CostoViaggio());
}
```

Invece l'output prodotto è inesatto:

```
Nave:Bally, Costo del viaggio: 10
Nave:Trasporter, Costo del viaggio: 20
Nave:Nottera, Costo del viaggio: 30
```

Facendo un po' di conti scopriamo infatti che il costo del viaggio della nave Transporter dovrebbe essere di 64,00 Euro. Sull'oggetto di tipo `NaveMercantile` qualcosa non ha funzionato nell'invocazione del metodo `CostoViaggio()`.

E' accaduto questo: `flotta[1]` contiene un riferimento ad un oggetto di tipo `NaveMercantile`, ma anche per esso è stato invocato la versione del metodo `CostoViaggio()` definito nella classe `Nave`, invece che la versione definita nella classe `NaveMercantile`, come sarebbe stato necessario. Per capire come mai, abbandoniamo temporaneamente il vettore `flotta` e consideriamo una singola variabile di tipo `Nave`:

```
NaveMercantile cargo = new NaveMercantile("Tramog", 200, 220);
double costo = cargo.CostoViaggio();
```

Durante la traduzione del codice nel programma eseguibile, il compilatore traduce l'istruzione:

```
double costo = cargo.CostoViaggio();
```

nell'invocazione del metodo definito dalla classe d'appartenenza del variabile `cargo`, e tale classe è appunto `Nave`!. E' del tutto irrilevante che la variabile in questione referenzi in realtà un oggetto di tipo `NaveMercantile`, poiché:

il tipo dell'oggetto referenziato da una variabile è determinato durante l'esecuzione del programma, mentre la decisione di quale metodo deve essere invocato è prodotta durante la compilazione e dipende unicamente dal tipo della variabile (tipo statico).

Il corretto calcolo del costo del viaggio implica che la scelta del metodo da invocare si basi sul tipo dell'oggetto e non sul tipo della variabile che lo referencia. Ma ciò è possibile soltanto se l'invocazione del metodo avviene mediante un meccanismo diverso da quello suddetto, in modo che la versione del metodo da invocare sia determinata durante l'esecuzione del programma e non prima.

2 Funzioni membro virtuale e "invocazione ritardata"

Facciamo un breve riepilogo. Attraverso variabili di tipo `Nave` siamo in grado di referenziare oggetti di tipo `NaveMercantile` e in generale di qualsiasi tipo che derivi da `Nave`. D'altra parte, quando invochiamo il metodo `CostoViaggio()` attraverso una variabile di tipo `Nave` desideriamo che il metodo effettivamente invocato sia quello definito dalla classe d'appartenenza dell'oggetto e non della variabile, sia cioè quello definito dal tipo run-time e non dal tipo statico. La tecnica usata finora, e cioè quella di fornire una nuova versione del metodo, in questo caso non funziona; essa è

utilizzabile soltanto se l'invocazione avviene attraverso variabili il cui tipo è lo stesso dell'oggetto che referenziano.

Invece di definire un nuovo metodo che nasconde quello della classe base, ciò che serve è fornire una diversa implementazione dello *stesso* metodo. Ma perché ciò sia possibile, il metodo definito dalla classe base dev'essere caratterizzato dalla parola chiave `virtual`.

2.1 Definizione e ridefinizione (*override*) di una funzione membro virtuale

E' possibile definire una funzione membro come virtuale semplicemente utilizzando la parola chiave `virtual`:

```
class Nave
{
    ...
    public virtual double CostoViaggio()
    {
        return _costoBase * _miglia;
    }
}
```

L'ordine in cui compaiono le parole `public` e `virtual` non è importante. E' invece importante che la funzione in questione non sia privata, in caso contrario otterremmo un errore di compilazione. Infatti una funzione viene dichiarata virtuale perché possa essere ridefinita nelle classi derivate, se il suo livello d'accesso è `private`, ciò non è possibile.

Semplicemente definire un metodo come virtuale non cambia le cose; ciò avviene quando questo viene ridefinito nelle classi derivate, risultato che si ottiene fornendo una nuova implementazione e specificando la parola chiave `override`:

```
class NaveMercantile: Nave
{
    ...
    public override double CostoViaggio ()
    {
        return _costoBase * _miglia + _caricoPagante * _costoCaricoMerce;
    }
}
```

Così facendo operiamo una cosiddetta “derivazione virtuale” di una funzione membro. Il termine `override` sta per “sovrapporre”, “sovrascrivere”, ed è ciò che realmente avviene. Infatti la funzione così dichiarata si sovrappone a quella definita nella classe base. Di fatto, è come se esistesse un solo metodo, del quale vengono fornite due implementazioni diverse. Ciò si desume anche dal fatto che:

- ❑ è proibito qualificare una funzione con `override` senza che esista una funzione virtuale omologa definita nella classe base;
- ❑ il prototipo della funzione ridefinita deve coincidere perfettamente con quello della funzione base;
- ❑ il modificatore di accesso della funzione ridefinita dev'essere uguale a quello – `protected` o `public` – specificato nella funzione base.

2.2 “Invocazione ritardata” (collegamento ritardato) di una funzione

Dopo questa modifica alle due classi, il codice dell'esempio precedente i risultati desiderati:

Nave:Bally, Costo del viaggio: 10

Nave:Trasporter, Costo del viaggio: 64

Nave:Nottera, Costo del viaggio: 30

Scopriamo per quale motivo, ancora una volta prendendo in considerazione singole variabili, come nel seguente codice:

```
Nave rimorchiatore = new Nave("Bally", 100);
NaveMercantile cargo = new NaveMercantile("Tramog", 200, 220);
double costoRimorchiatore = rimorchiatore.CostoViaggio();
double costoCargo = cargo.CostoViaggio();
```

Il compilatore traduce l'invocazione di un metodo nell'istruzione in linguaggio oggetto `CALL`, che ha come argomento l'indirizzo di memoria del metodo da invocare. In pratica viene prodotto qualcosa del tipo:

`CALL indirizzo-metodo`

Nel tradurre l'invocazione di un metodo virtuale non avviene esattamente la stessa cosa. Infatti, come argomento dell'istruzione `CALL` non compare l'indirizzo di un metodo ma un indice da utilizzare per accedere a una tabella dei metodi virtuali (*method virtual table*) definita dalla classe dell'oggetto referenziato. Viene dunque prodotto qualcosa del tipo:

`CALL tabella-virtuale-classe [indice-metodo]`

Dunque, le istruzioni:

```
double costo rimorchiatore = rimorchiatore.CostoViaggio();
double costoCargo = cargo.CostoViaggio();
```

vengono tradotte approssimativamente in:

`CALL tabella-virtuale Nave[0]`

`CALL tabella-virtuale NaveMercantile[0]`

Gli elementi di indice zero delle due tabelle virtuali contengono l'indirizzo dei metodi da invocare, e tali metodi sono diversi per i due oggetti, poiché ogni tabella contiene gli indirizzi dei metodi virtuali definiti dalla classe di appartenenza dell'oggetto.

Lo schema mostrato di seguito riassume la situazione:

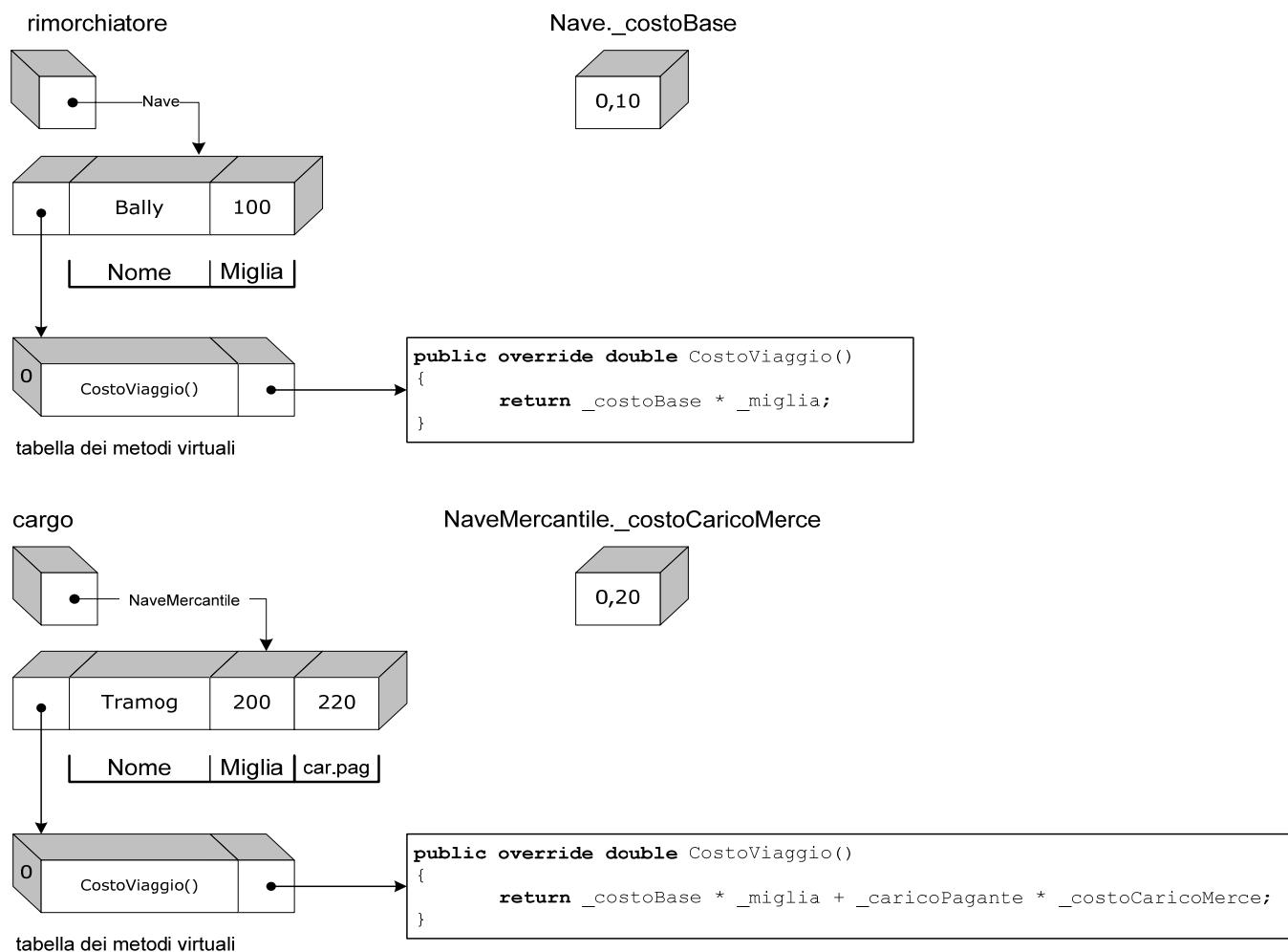


Figura 5-3 Rappresentazione degli oggetti rimorchiatore e cargo e delle loro tabelle virtuali.

Usando la derivazione virtuale dei metodi, il metodo effettivamente invocato dipende dunque dal tipo dell'oggetto referenziato e non dal tipo della variabile. L'indirizzo di tale metodo viene determinato durante l'esecuzione del programma e non durante la compilazione; avviene dunque ciò che si chiama "invocazione ritardata" o "collegamento ritardato" (*late binding*).

La parola "ritardata/o" sta a significare che il compilatore non traduce immediatamente il nome del metodo nel suo indirizzo di memoria, ma in un indice relativo ad una tabella di metodi. In fase di esecuzione è appunto attraverso quest'indice che l'indirizzo del metodo viene ricavato e quindi il metodo eseguito.

2.3 Invocare la funzione virtuale della classe base

All'interno di una funzione virtuale ridefinita è possibile invocare la funzione virtuale base – come qualsiasi altra funzione che sia accessibile – mediante la parola chiave `base`. In realtà questo non è affatto un evento raro, poiché molto spesso nel ridefinire una funzione virtuale l'obiettivo è quello di estenderne il funzionamento e non di modificarlo per intero.

Ritornando alle classi `Nave` e `NaveMercantile`, ad esempio, si nota che il metodo `CostoViaggio()` della classe derivata estende la modalità di calcolo del viaggio, aggiungendo gli importi specifici del carico a quelli di base validi per tutte le navi. Il metodo può dunque essere modificato nel seguente modo:

```

class NaveMercantile: Nave
{
    ...
    public override double CostoViaggio ()
    {
        return base.CostoViaggio() + _caricoPagante * _costoCaricoMerce;
    }
}

```

Il vantaggio di questo approccio sta nell'applicazione dei principi di incapsulamento e riutilizzo del codice. Utilizzando infatti il metodo base si rende il metodo ridefinito indipendente da qualsiasi modifica nella modalità di calcolo costo di viaggio valido per tutte le navi. In altre parole, un cambiamento nella modalità di calcolo del costo di viaggio base richiederebbe la modifica del metodo `CostoViaggio()` della classe `Nave`, perché è in questo metodo che avviene tale calcolo.

2.4 Ridefinizione di funzioni virtuali sovraccaricate

Nel fornire una nuova implementazione a una funzione virtuale della classe base attraverso la parola chiave `override` si crea una corrispondenza uno a uno tra le due funzioni, rappresentata dal fatto che esse hanno un prototipo identico. In questo, il fatto che la classe base, la classe derivata o entrambe sovraccarichino le due funzioni in questione non produce nessuna conseguenza.

Ad esempio, nel seguente codice, `ClasseBase` definisce due versioni di `Metodo()`, entrambe virtuali. `ClasseDerivata` si limita a sovrascrivere la versione senza parametri:

```

class ClasseBase
{
    public virtual void Metodo()
    {
        Console.WriteLine("ClasseBase:Metodo()");
    }
    public virtual void Metodo(int a)
    {
        Console.WriteLine("ClasseBase:Metodo(int)");
    }
}

class ClasseDerivata: ClasseBase
{
    public override void Metodo()
    {
        Console.WriteLine("ClasseDerivata:Metodo()");
    }
}

```

Il risultato è che gli oggetti di tipo `ClasseDerivata` ereditano la versione base con un parametro mentre sovrascrivono quella senza parametri. Il seguente codice lo dimostra:

```

ClasseBase ogg = new ClasseDerivata();
ogg.Metodo(10);           // invocazione metodo ereditato
ogg.Metodo();             // invocazione metodo ridefinito

```

Il risultato in output è:

```

ClasseBase:Metodo(int)

```

`ClasseDerivata:Metodo()`

Detto ciò, benché non sia richiesto dal linguaggio, se una classe sovraccarica una funzione virtuale, nelle classi derivate tutte le versioni della funzione dovranno probabilmente essere ridefinite.

3 Polimorfismo

Il polimorfismo è una conseguenza della derivazione virtuale delle funzioni membro. Tale termine implica che una variabile può produrre comportamenti diversi in base al tipo effettivo dell'oggetto referenziato. In questo senso, dunque, sono sempre due i tipi in gioco:

- ❑ il tipo della variabile (del riferimento): tipo statico;
- ❑ il tipo dell'oggetto referenziato, che dev'essere uguale o derivare dal primo: tipo run-time;

Nell'invocazione di una funzione membro virtuale è sempre il tipo run-time a determinare qual è la funzione effettivamente invocata. Se la funzione in questione non è virtuale, ciò viene invece determinato dal tipo statico.

Il polimorfismo è fondamentale nella OOP, poiché la sua applicazione rende possibile la realizzazione di collezioni e algoritmi che elaborano oggetti il cui tipo – e il cui comportamento – può essere uno qualsiasi all'interno di una determinata gerarchia.

3.1 Funzioni virtuali e funzioni non virtuali a confronto

All'interno di una certa linea di discendenza, il polimorfismo è possibile soltanto se la classe base definisce una o più funzioni membro virtuali. Una classe che non rispetta questo requisito potrà essere ereditata, ma non usata in modo polimorfico. Ciò significa che il polimorfismo dev'essere progettato in partenza. In altre parole, quando si progetta una classe che potrebbe essere oggetto di derivazione, occorre stabilire per quali funzioni membro le classi derivate potrebbero dover fornire una diversa implementazione: tali funzioni devono essere definite virtuali.

In alcuni casi questa decisione può essere relativamente arbitraria. Non è certo il caso della piccola gerarchia dei precedenti esempi. La classe `Nave` definisce due funzioni, la proprietà `Nome` e il metodo `CostoViaggio()`. Benché nulla impedisca di dichiarare `Nome` virtuale, non esiste alcun ragionevole motivo per farlo, poiché non si vede per quale motivo la classe `NaveMercantile` (ed altre classi derivate) ne debba modificare l'implementazione. Diverso è il caso del metodo `CostoViaggio()`, poiché il modo in cui viene calcolato il costo del viaggio dipende dal tipo di nave. Vale dunque la regola che:

una funzione membro dovrebbe essere definita virtuale se la sua implementazione dipende dalla classe.

Questa affermazione implica anche che, in caso contrario, la funzione non dovrebbe essere definita virtuale. Il perché di questo è legato alle prestazioni. Definire una funzione virtuale implica infatti:

- ❑ minor velocità nell'invocazione della funzione. Come abbiamo visto, l'invocazione viene tradotta in una chiamata indiretta alla funzione (attraverso un indice si accede all'indirizzo del metodo da invocare), senz'altro meno efficiente di una chiamata diretta;
- ❑ un aumento della dimensione della tabella delle funzioni virtuali;

Ciò detto, nulla impedisce di dichiarare virtuali tutte le funzioni membro di una classe, in modo che le classi derivate ne possano fornire una diversa implementazione, ma ciò risulterebbe inefficiente sia in termini di memoria occupata che di prestazioni.

3.2 Polimorfismo in .NET

.NET espone una vastissima gerarchia di classi, molte delle quali appositamente progettate per favorire l'applicazione del polimorfismo. Dato il numero elevatissimo anche una loro semplice elencazione sarebbe inattuabile. Alcune di esse in particolare, comunque, sono estremamente importanti, poiché rappresentano la base sulla quale sono costruite intere linee di discendenza. Di queste ne sarà presa in considerazione una soltanto, con lo scopo di approfondire i principi e i vantaggi del polimorfismo.

3.3 Classe "Object"

La classe `Object` implementa il tipo omonimo `object` ed è la classe più semplice e più importante; essa sta infatti alla base di qualsiasi linea di discendenza, definita in .NET o implementata dal programmatore. Per questo motivo, una variabile di tipo `object` può referenziare oggetti di qualsiasi tipo, poiché ogni oggetto appartiene a una classe che direttamente o indirettamente deriva da `Object`. In sostanza: ogni oggetto è un `object`.

Definire una nuova classe senza specificare una classe base significa dunque definire una classe che deriva direttamente da `Object`. Quindi, scrivere:

```
class MiaClasse
{
    ...
}
```

equivale a scrivere:

```
class MiaClasse: Object
{
    ...
}
```

E' importante comprendere che la classe `Object` implementa in realtà un tipo di oggetto astratto, il quale non memorizza informazioni, né è in grado di eseguire particolare operazioni. Lo scopo di tale classe è infatti quello di fornire un denominatore comune a tutte le altre classi. Per questo motivo, `Object` definisce alcuni metodi virtuali, i quali possono dunque essere ridefiniti nella classi derivate. Di questi prenderemo in considerazione il metodo `ToString()`, il cui scopo è quello di fornire una rappresentazione testuale dell'oggetto. La sua implementazione fornita dalla classe `Object` equivale alla seguente:

```
class Object
{
    public Object()
    {
    }
    ...
    public virtual string ToString()
    {
        return "System.Object";
    }
}
```

In realtà il metodo `ToString()` non è implementato in questo modo, ma per semplicità possiamo fingere che lo sia.

Poiché le istanze di tipo `object` sono semplicemente degli oggetti astratti, il metodo `ToString()` si limita a ritornare il nome della classe, in quanto non esiste una rappresentazione testuale significativa dell'oggetto. Tutti i tipi per i quali esiste una rappresentazione testuale significativa dovrebbero ridefinire tale metodo. Per questo motivo ne forniamo una implementazione nelle classi `Nave` e `NaveMercantile`:

```
class Nave
{
    ...
    public override string ToString()
    {
        return String.Format("{0} | {1}", _nome, _miglia.ToString());
    }
}

class NaveMercantile: Nave
{
    ...
    public override string ToString()
    {
        return base.ToString(); + " | " + _caricoPagante.ToString();
    }
}
```

Da notare come l'implementazione fornita nella classe `NaveMercantile` faccia riferimento alla versione definita nella classe base. A questo punto è possibile scrivere codice come il seguente:

```
Nave rimorchiatore = new Nave("Bally", 100);
NaveMercantile cargo = new NaveMercantile("Tramog", 200, 220);
Console.WriteLine(rimorchiatore);
Console.WriteLine(cargo);
```

il quale produce il seguente output:

```
Bally | 100
Tramog | 200 | 220
```

Come ha potuto funzionare? La spiegazione sta nei principi del polimorfismo. Tra le varie versioni del metodo sovraccaricato `WriteLine()` ne esiste una implementata più o meno così:

```
static void WriteLine(object o)
{
    string s = o.ToString();
    // visualizza la stringa nello schermo
}
```

che è poi la versione invocata nel codice precedente.

Il metodo definisce un parametro di tipo `object`, del quale invoca `ToString()` per ottenerne la rappresentazione testuale. Senza la derivazione virtuale delle funzioni membro, e dunque senza il polimorfismo, la stringa visualizzata sarebbe sempre: `"System.Object"`¹². Poiché però

¹² Non è esatto, ma la sostanza delle cose non cambia.

`ToString()` è virtuale, la versione del metodo invocata è quella fornita dal tipo dell'oggetto referenziato. Dunque, nell'istruzione:

```
Console.WriteLine(rimorchiatore);
```

viene invocato `ToString()` definito dalla classe `Nave`, mentre nell'istruzione:

```
Console.WriteLine(cargo);
```

viene invocato `ToString()` definito dalla classe `NaveMercantile`.

3.4 Conversione implicita

L'argomento sulle conversioni di tipo e sull'uso dell'operatore di cast è già stato affrontato nel primo volume. In questa sede lo prenderemo nuovamente in considerazione:

- ❑ analizzando il significato delle operazioni di conversione, implicita ed esplicita, tra classi che hanno un rapporto di derivazione;
- ❑ considerando alcuni esempi tipici di programmazione in cui l'uso dell'operatore di cast si rende necessario;

Si ha una conversione implicita quando:

il riferimento ad un oggetto di Tipo-D viene assegnato a una variabile di Tipo-B, nel caso in cui Tipo-D deriva direttamente o indirettamente da Tipo-B.

Una conversione è definita implicita quando è sicura, e cioè quando non produrrà mai una perdita di informazione o un utilizzo non valido di un oggetto. Nei paragrafi precedenti abbiamo già visto in azione questo tipo di conversione, ad esempio nell'istruzione:

```
Nave cargo = new NaveMercantile("Trasporter", 200, 220);
```

Essa produce infatti quanto segue:

- ❑ l'operatore `new` determina la creazione di un oggetto di tipo `NaveMercantile` e produce come risultato un riferimento all'oggetto in questione;
- ❑ tale riferimento viene assegnato alla variabile `rimorchiatore`, di tipo `Nave`;

In pratica, l'istruzione viene tradotta dal linguaggio in:

```
Nave cargo;
```

```
NaveMercantile tmp = new NaveMercantile("Trasporter", 200, 220);
```

```
cargo = tmp;
```

Nell'ultima istruzione si ha un'assegnazione di una variabile di tipo `NaveMercantile` ad una variabile di tipo `Nave`: avviene dunque una conversione, che comunque si traduce nella semplice assegnazione a `cargo` dell'indirizzo contenuto in `tmp`.

Perché una simile assegnazione è sicura e dunque rientra nella categoria delle conversioni implicite? Perché usare una variabile di tipo `Nave` per accedere a un oggetto di tipo `NaveMercantile` non potrà mai provocare un utilizzo non valido della variabile. Ed è così perché tutti i membri definiti da `Nave` e dunque accessibili attraverso una variabile di questo tipo sono ereditati dalla classe `NaveMercantile`. Ciò detto, anche il seguente codice è lecito:

```
object o = new NaveMercantile("Trasporter", 200, 220);
```

```
Console.WriteLine(o);
```

e produce in output:

```
Tramog | 200 | 220
```


poiché `NaveMercantile`, come qualsiasi altro tipo, deriva da `object`.

Non è invece lecito il codice che segue:

```
object ogg = new NaveMercantile("Trasporter", 200, 220);
NaveMercantile cargo = ogg; // errore formale!
Console.WriteLine(cargo);
```

perché la classe `Object` non deriva dalla classe `NaveMercantile` e dunque in fase di compilazione non è possibile stabilire quale tipo di oggetto la variabile `ogg` referenzierà durante l'esecuzione del programma.

3.5 Conversione esplicita: operatore di cast “()”

Partiamo dall'ultimo frammento di codice. Dopo l'istruzione di creazione, la variabile `ogg`, il cui tipo statico è `object`, referencia un oggetto di tipo `NaveMercantile`; perché dunque è impossibile assegnarla a una variabile di tipo `NaveMercantile`?

La risposta è: non è affatto impossibile, ma è necessario rendere esplicito che `ogg` referencia effettivamente un oggetto di tipo `NaveMercantile` e che dunque tale assegnazione è valida. La questione sta in questi termini. Quando il compilatore traduce il codice sorgente nel programma eseguibile non è in grado di sapere il tipo effettivo dell'oggetto referenziato da una variabile (tipo run-time), perché tale informazione è disponibile soltanto durante l'esecuzione del programma. Dunque, nel valutare la correttezza di un'istruzione, il compilatore si affida ai tipi statici e non ai tipi run-time, ed è sulla loro base che giudica la correttezza di una conversione. In questo caso, assegnare una variabile di tipo `object` a una variabile di tipo `NaveMercantile` viola le regole del linguaggio, perché `object` *non* deriva da `NaveMercantile`¹³.

D'altra parte, il programmatore sa qual è il tipo effettivo dell'oggetto referenziato e dunque sa che in realtà l'assegnazione è valida, e dunque può indicarlo in modo esplicito al compilatore mediante l'operatore di cast, effettuando cioè una conversione esplicita:

```
NaveMercantile cargo = (NaveMercantile) ogg;
```

In sostanza, specificando un tipo accanto a una variabile (in generale a una espressione) si induce il compilatore a effettuare una conversione dal tipo della variabile (tipo sorgente) al tipo del cast (tipo destinazione) della conversione. Come risultato, il compilatore accetta innanzitutto l'istruzione e genera inoltre delle istruzioni che in fase di esecuzione verificano che il tipo run-time sia effettivamente uguale o compatibile (derivati da) con quello specificato nel cast; se così non è, durante l'esecuzione sarà sollevata l'eccezione `InvalidCastException`.

Quello che invece non si può fare è il cast da un oggetto base ad uno ereditato.

Se infatti proviamo a scrivere il seguente frammento di codice:

```
Nave unaNave = new Nave("Nemo", 150);
NaveMercantile cargOne = (NaveMercantile)unaNave;
```

Anche se tutto sembra che funzioni in fase di compilazione, in fase di runtime otterremo anche in questo caso di sollevare l'eccezione di `InvalidCastException`.

E' facile capire il perché di questo problema in quanto tenendo a mente il fatto che la classe ereditata dovrebbe sempre estendere la classe base (aggiungendo magari degli attributi) il .NET non è in grado autonomamente di aggiungere questi attributi all'oggetto del tipo della classe base.

Per risolvere questo problema sarà sufficiente scrivere un metodo statico alla classe base (`Nave` in questo caso) che effettui una conversione “Toxxx” verso la classe ereditata come in questo esempio.

¹³ Verrà approfondito più avanti nel testo il concetto di “varianza” dei tipi di dato

```

public static NaveMercantile ToNaveMercantile(Nave unaNave)
{
    NaveMercantile mercantile =
        new NaveMercantile(unaNave._nome, unaNave._miglia, 0.20);
    return mercantile;
}

```

Potendo quindi scrivere il seguente codice

```
NaveMercantile cargOne = Nave.ToNaveMercantile(unaNave);
```

Uso delle conversioni esplicite

L'aspetto centrale del polimorfismo è quello di rendere possibili procedimenti e strutture dati che sono entro certi limiti indipendenti dal tipo degli oggetti. D'altra parte, in alcune circostanze è necessario che il programmatore specifichi in modo esplicito il tipo effettivo dell'oggetto facendo uso dell'operatore di cast. Tuttavia, con l'introduzione delle collezioni tipizzate nel .NET 2.0, questa esigenza si è molto ridotta.

3.6 Casting nelle collezioni generiche

Nelle collezioni generiche introdotte nel .Net 2.0 come `List<T>` ed altre gli oggetti presenti nella collezione sono, proprio perché "generiche", del tipo utilizzato nella dichiarazione della variabile.

Se scriviamo questo frammento di codice

```

List<Nave> flottiglia = new List<Nave>();

flottiglia.Add(new Nave("Bally", 100));
flottiglia.Add(new NaveMercantile("Trasporter", 200, 220));
flottiglia.Add(new Nave("Nottera", 300));

```

e poi proviamo a visualizzare l'elenco delle navi presenti nella flottiglia con un semplice ciclo for in questo modo

```

for (int i = 0; i < flottiglia.Count; i++)
{
    Console.WriteLine(flottiglia[i].CostoViaggio());
}

```

Otterremo il seguente output

```

Bally | 100
Trasporter | 200 | 220
Nottera | 300

```

Il che significa che avendo dichiarato la lista di tipo `Nave`, anche per il secondo battello, pur se di tipo `NaveMercantile` è stato invocato il corretto metodo `CostoViaggio()` senza aver bisogno di effettuare nessun cast esplicito.

3.7 Conoscere il tipo effettivo di un oggetto: operatore "is"

Nell'usare l'operatore di cast `()` il programmatore si prende la responsabilità di un eventuale fallimento della conversione.

Negli esempio precedente il problema non si pone, in quanto tutti gli oggetti memorizzati nelle collezioni sono di tipo `Nave` od `NaveMercantile` e non esiste la necessità di distinguere tra i due tipi, in quanto l'unica operazione richiesta sugli oggetti è l'invocazione del metodo virtuale

`CostoViaggio()`, implementato da entrambi i tipi. Ma se fosse necessario elaborare gli oggetti di tipo `Nave` in modo diverso da quelli di tipo `Navemercantile`?

Chiariamo con un esempio, e a questo scopo modifichiamo innanzitutto l'interfaccia pubblica della classe `Navemercantile` aggiungendo una proprietà per l'accesso al campo `caricoPagante`:

```
class Navemercantile
{
    double _caricoPagante;
    ...
    public double CaricoPagante
    {
        get { return _caricoPagante; }
    }
}
```

L'obiettivo è adesso quello realizzare un metodo che visualizzi un prospetto per ogni nave, il quale deve riportare:

- ❑ categoria: nave semplice o mercantile;
- ❑ nome e miglia da percorrere
- ❑ carico pagante, nel caso di navi mercantili.

Ipotizziamo adesso di elaborare la lista `convoglio`; il problema è che dev'essere in grado di conoscere il tipo effettivo di ogni elemento della collezione, in base al quale dovrà produrre il prospetto. Ciò è possibile attraverso l'operatore `is`, il quale consente di stabilire se un oggetto appartiene o meno a un determinato tipo.

La sintassi d'uso è:

espressione is tipo

e il risultato è il valore booleano `true` se il tipo run-time di *espressione* è uguale o deriva da quello specificato, `false` altrimenti.

Detto ciò, ecco come può essere realizzato il metodo:

```
static void VisualizzaFlotta(List<Nave> flotta)
{
    for (int i = 0; i < flotta.Count; i++)
    {
        Console.WriteLine();
        if (flotta[i] is NaveMercantile)
        {
            NaveMercantile nave = (NaveMercantile) flotta[i];
            Console.WriteLine("Tipo: Nave mercantile");
            Console.WriteLine("Nome: " + nave.Nome);
            Console.WriteLine("Costo viaggio: {0}", nave.CostoViaggio());
            Console.WriteLine("Carico Pagante: {0}", nave.CaricoPagante);
        }
        else
        {
            Nave nave = flotta[i];
```

```

        Console.WriteLine("Tipo: Nave");
        Console.WriteLine("Nome: " + nave.Nome);
        Console.WriteLine("Costo viaggio: {0}", nave.CostoViaggio());
    }
}

```

Come si vede, l'operatore `is` consente di stabilire se l'esimo elemento di `flotta` è di tipo `NaveMercantile` oppure no. In base questa condizione si ottiene mediante l'operatore di cast un riferimento del tipo appropriato e si procede di conseguenza.

Nell'uso dell'operatore `is` è importante comprendere che non è richiesta l'uguaglianza stretta tra il tipo dell'espressione e il tipo specificato, ma semplicemente che il primo sia uguale o derivi dal secondo. Comprendere questo è fondamentale, infatti se avessimo scritto:

```

if (flotta[i] is Nave)
    ...

```

invertendo conseguentemente la parte `if()` con la parte `else`, il metodo avrebbe visualizzato alcuni prospetti errati, poiché tutti gli elementi di `flotta` sono di tipo `Nave`, e quindi la condizione sarebbe sempre risultata `true`.

3.8 Operatore di conversione esplicita “as”

Oltre all'operatore `()`, il linguaggio mette a disposizione un secondo operatore di cast, `as`, che presenta la seguente sintassi:

espressione as tipo

Analogamente all'operatore `()`, anche `as` esso converte il tipo dell'espressione nel tipo specificato, ma con la differenza che non solleva un'eccezione nel caso in cui la conversione non possa essere eseguita; in questo produce il valore `null`.

In alcune situazioni, l'operatore `as` può sostituire l'operatore `is`, producendo un codice più efficiente, poiché richiede l'esecuzione di una sola operazione di cast. Ecco come potrebbe essere reimplementato il metodo precedente:

```

static void VisualizzaFlotta(List<Nave> flotta)
{
    for (int i = 0; i < flotta.Count; i++)
    {
        Console.WriteLine();
        NaveMercantile naveMe = flotta[i] as NaveMercantile;
        if (naveMe != null)
            ...
    }
}

```

Come si vede, cambia la natura della condizione usata nella `if()` `else`. In questo caso consiste nella verifica che il riferimento memorizzato in `naveMe` sia diverso da `null`; il codice eseguibile corrispondente è più efficiente di quello risultante dall'applicazione dell'operatore `is`.

3.9 Ottenere il tipo di un oggetto: metodo “GetType()”

Di ogni tipo, il linguaggio memorizza tutte le informazioni necessarie per caratterizzarlo: nome, elenco dei campi e delle funzioni membro, *namespace* all'interno dei quali il tipo è stato definito,

eccetera. Alcune di queste informazioni sono utilizzate dal linguaggio nell'applicazione degli operatori di cast.

A questo punto ci si può chiedere: è possibile da programma avere l'accesso alle informazioni che caratterizzano il tipo run-time di un oggetto? La risposta è sì, ottenendo un'istanza della classe `Type` mediante l'invocazione del metodo `GetType()` sull'oggetto.

Il seguente codice ottiene due oggetti `Type` che descrivono il tipo run-time delle variabili `rimorchiatore` e `cargo`:

```
Nave rimorchiatore = new Nave("Bally", 100);
NaveMercantile cargo = new NaveMercantile("Tramog", 200, 220);
Type tNave = rimorchiatore.GetType();
Type tNaveMercantile = cargo.GetType();
Console.WriteLine("Il tipo run-time di rimorchiatore è: {0}", tNave.Name);
Console.WriteLine("Il tipo run-time di cargo è: {0}", tNaveMercantile.Name);
```

La sua esecuzione produce in output:

```
Il tipo run-time di ope è: Nave
Il tipo run-time di opeSpe è: NaveMercantile
```

La classe `Type` definisce un vasto insieme di proprietà e funzioni che consentono di accedere a tutte le informazioni che caratterizzano un tipo. Nell'esempio è stata utilizzata la sola proprietà `Name`, che ritorna il nome del tipo.

4 Classi astratte

Elemento essenziale nella OOP è una corretta progettazione delle classi che devono rappresentare gli oggetti del dominio del programma. L'obiettivo è quello di stabilire una corrispondenza quanto più accurata possibile tra queste e gli oggetti che rappresentano:

- ❑ negli attributi;
- ❑ nelle operazioni da rendere disponibili al codice *consumer* (interfaccia pubblica);
- ❑ nelle relazioni di parentela e nell'insieme di funzioni che dipendono dal tipo (e che dunque devono essere dichiarate virtuali).

In tutti gli esempi mostrati finora i principi dell'ereditarietà e del polimorfismo sono stati applicati a due classi soltanto, in un semplice rapporto "classe base – classe derivata", ma la maggior parte dei problemi di un certo livello richiede la progettazione di una gerarchia di classi, piccola o grande che sia. Per mostrare un esempio del tipo di analisi richiesta nella progettazione di una gerarchia di classi prenderemo nuovamente in considerazione il problema della gestione della flotta navale, aggiungendo un nuovo oggetto al dominio del problema.

Occorre gestire la presenza all'interno della flotta anche di mezzi marini differenti da navi come, per esempio i gommoni, utili per trasportare i piloti da e verso le grandi navi prima di entrare in porto.

4.1 Analisi del dominio del problema

Ripartiamo dal lavoro di analisi già svolto in precedenza, da quale erano emersi i tipi *Nave* e *NaveMercantile*, che mantengono stessi attributi e funzioni. Definiamo dunque il tipo “gommone” caratterizzato da:

- “nome”;
- “miglia”;
- “costobase”;
- “tipologia”, che può essere: “diporto” o “alto mare”.

A questo punto occorre definire in modo formale la classe *Gommone*, valutando eventualmente la possibilità di ridisegnare le classi *Nave* e *NaveMercantile* allo scopo di ottenere una rappresentazione accurata del dominio del problema. Intuitivamente si possono subito individuare due possibili disegni:

- 1) Mantenere invariate le classi *Nave* e *NaveMercantile* e definire a parte la classe *Gommone*, senza alcun legame con le altre due.

Questo approccio è estremamente insoddisfacente, poiché non sfrutta il meccanismo di riutilizzo del codice offerto dall’ereditarietà, né la possibilità, caratteristica del polimorfismo, di scrivere codice *consumer* generico per l’elaborazione degli oggetti appartenenti alle tre classi.

- 2) Far derivare la classe *Gommone* dalla classe *Nave*.

Questo secondo approccio è senz’altro migliore del precedente, poiché consente di condividere gli attributi comuni alle tre classi e di scrivere codice che sfrutti i vantaggi del polimorfismo. In base a questa scelta otterremmo la seguente gerarchia:

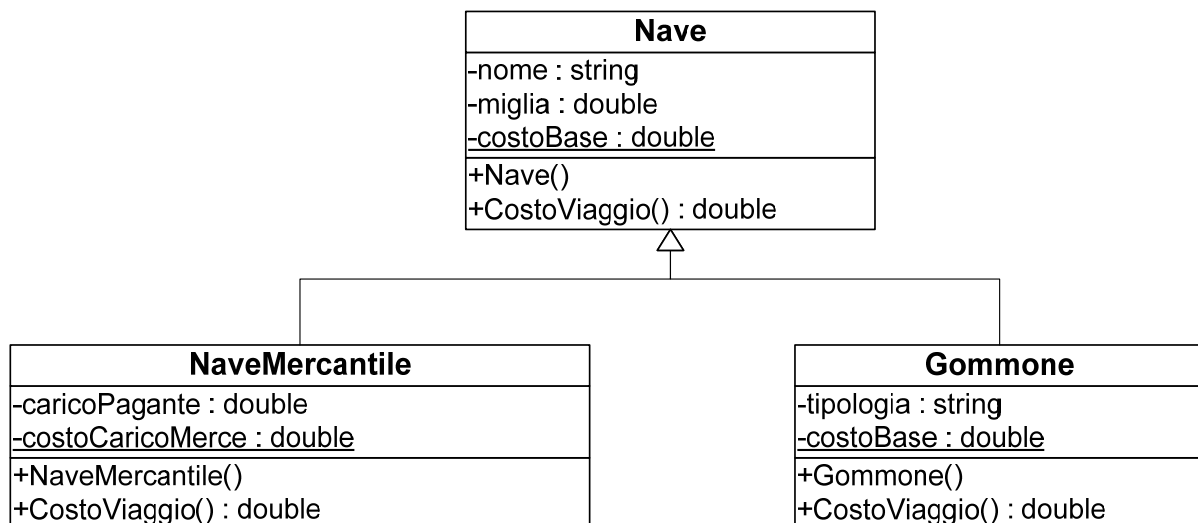


Figura 5-4 Schema della gerarchia di classi nell’ipotesi di fare derivare *Gommone* da *Nave*

Un simile progetto è valido dal punto di vista pratico, ma soffre egualmente di una grave pecca: non rappresenta un modello accurato degli oggetti del dominio del problema. Il perché è presto detto: un “gommone” non è un tipo di “nave”. Un gommone, cioè, condivide alcune informazioni comuni con una nave, ma non per questo rappresenta una specializzazione di questa categoria.

La questione non riguarda semplicemente i nomi che identificano le classi. Infatti, successive modifiche ai requisiti del problema potrebbero rendere necessario modificare la classe *Nave*, ad

esempio con l'aggiunta di attributi, senza che per questo sia richiesto di alterare anche la classe *Gommone*. Ma ciò non è ovviamente possibile se *Gommone* deriva da *Nave*.

La questione sta dunque in questi termini: la semplice definizione delle tre classi non è sufficiente per ottenere un disegno che modelli adeguatamente il dominio del problema. Ciò lo si può ottenere introducendo una quarta classe, che fa da denominatore comune alle altre. Infatti, navi semplici, navi mercantili e gommoni sono tutti dei “battelli” della flotta, e in quanto tali:

- sono identificati attraverso un “nome”;
- hanno un costo per l'azienda in base alla quantità di “miglia” percorse e dal “costo” specifico di ogni mezzo.

4.2 Introduzione nella gerarchia di una classe astratta

Introduciamo quindi la classe *Battello*, la quale implementa le caratteristiche comuni a tutti i mezzi della flotta. Da essa derivano direttamente le classi *Nave* e *Gommone*. la classe *NaveMercantile* mantiene la propria relazione con la classe *Nave*.

Battello viene definita “classe astratta”, poiché essa non rappresenta nessun oggetto reale del dominio del problema. Nessun mezzo della flotta è semplicemente un “battello”. Nondimeno, emerge la figura del generico battello, e cioè colui che, a prescindere dalla tipologia di imbarcazione, fa parte della flotta ed ha un costo in base alle miglia percorse ed al costo di base.

Una classe astratta facilita dunque il disegno di gerarchie di classi, consentendo di condividere attributi e funzioni comuni, fungendo da tipo base per classi che altrimenti sarebbe complicato, o comunque poco appropriato, mettere in relazione di parentela tra loro.

Di seguito è mostrato il nuovo disegno della gerarchia di classi ed il fatto che una di esse sia astratta viene indicato in UML dalla notazione corsiva nel nome della classe.

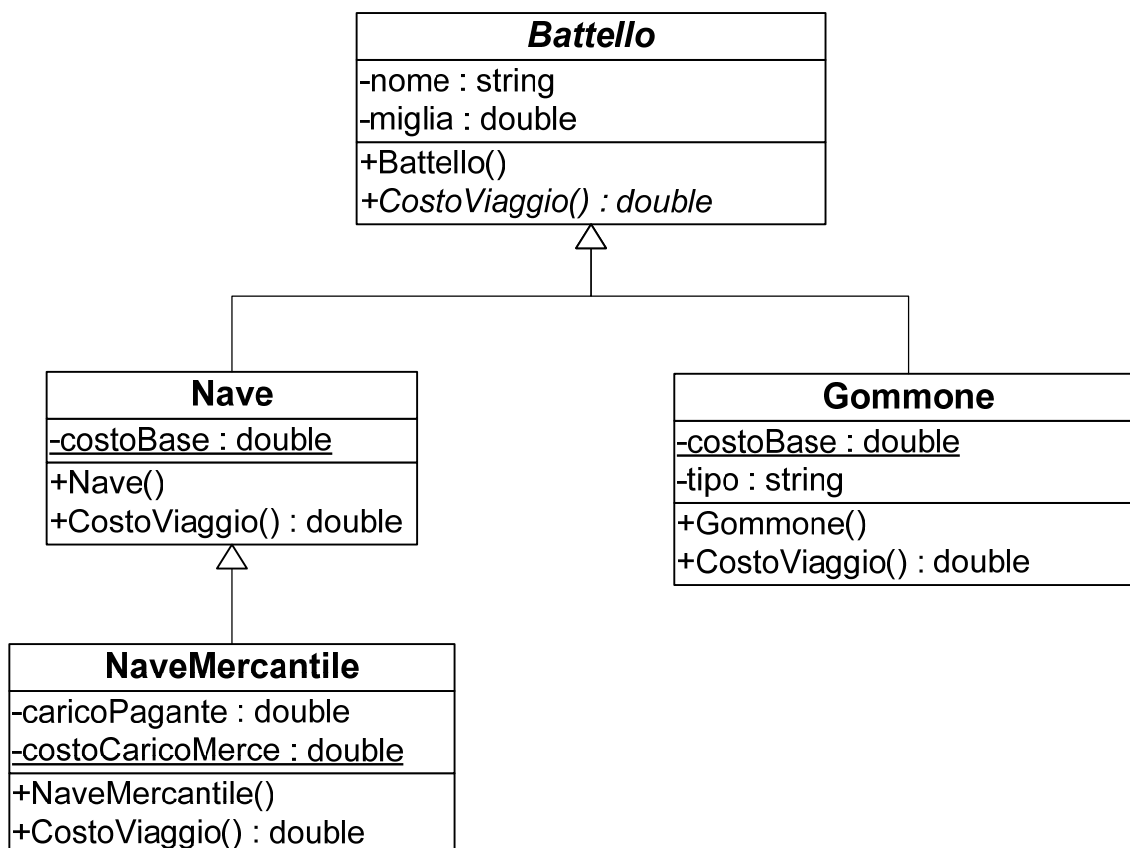


Figura 5-5 Schema della gerarchia di classi nell'ipotesi di avere la classe astratta Battello..

4.3 Definizione della classe Battello

Ecco il codice completo della classe:

```
public abstract class Battello
{
    protected string _nome;
    public string Nome
    {
        get { return _nome; }
    }

    protected double _miglia;
    public double Miglia
    {
        get { return _miglia; }
    }

    public Battello(string nome, double miglia)
    {
        _nome = nome;
        _miglia = miglia;
    }

    public virtual double CostoViaggio()
    {
        return 0;
    }

    public override string ToString()
    {
        return String.Format("{0} | {1}", _nome, _miglia.ToString());
    }
}
```

Degna di nota è l'implementazione del metodo virtuale `CostoViaggio()`; come si vede esso non svolge in realtà nessuna operazione significativa. Ed ovvio che sia così, poiché in realtà non esistono oggetti di tipo `Battello` per i quali calcolare il costo del viaggio, non a caso la classe non definisce nemmeno un campo che memorizzi il costo base.

Dunque, `CostoViaggio()` è in realtà un “metodo astratto”, non svolge alcun compito, ed esiste soltanto per essere ridefinito dalle classi derivate. Ciò consentirà di scrivere codice *consumer* che elabori in modo generico una collezione di battelli, invocando per ognuno il metodo `CostoViaggio()` implementato dall'effettiva classe di appartenenza.

Un'altra considerazione riguarda la decisione di implementare il metodo `ToString()`, nonostante durante l'esecuzione del programma non vengano mai creati oggetti di tipo `Battello` dei quali ottenere una rappresentazione testuale. Tale scelta ha lo scopo di favorire il riutilizzo del codice e obbedisce alla regola che:

all'interno di una gerarchia di classi le funzioni virtuali dovrebbero essere implementate il più in alto possibile nella gerarchia, in modo che le classi derivate possano ereditarle e non siano obbligate a fornire una propria implementazione, o possano comunque utilizzare l'implementazione base.

Ovviamente ciò è fattibile se la classe astratta è in grado di fornire un'implementazione significativa della funzione.

4.4 Definizione delle classi “Nave” e “Gommone”

Mentre la classe `NaveMercantile` non viene modificata dal nuovo disegno della gerarchia, le classi `Nave` e `Gommone` sono adesso definite sulla base della loro derivazione dalla classe `Battello`:

```
public class Nave : Battello
```

```
{
    protected static double _costoBase = 0.10;
```

```
    public Nave(string nome, double miglia)
        : base (nome, miglia)
    { }
```

```
    public override double CostoViaggio()
    {
        return _costoBase * _miglia;
    }
}
```

```
public class Gommone : Battello
```

```
{
    protected static double _costoBase = 0.05;
```

```
    private string _tipo;
    protected string Tipo
    {
        get { return _tipo; }
        set { _tipo = value; }
    }
}
```

```
    public Gommone(string nome, double miglia, string tipo)
        : base (nome, miglia)
    {
        _tipo = tipo;
    }
}
```

```
    public override double CostoViaggio()
    {
        return _costoBase * _miglia;
    }
}
```

```
    public override string ToString()
```

```

    {
        return base.ToString() + " | " + _tipo;
    }
}

```

Degne di nota sono le implementazioni dei costruttori di entrambe le classi, che si affidano al costruttore di `Battello` per inizializzare i campi ereditati. Come si vede, la classe `Nave` non ha più bisogno di fornire la propria implementazione del metodo `ToString()`, perché la versione definita dalla classe `Battello` è perfettamente adeguata allo scopo.

4.5 Uso della nuova gerarchia di classi

Data la nuova gerarchia di classi, la memorizzazione dei dati può avvenire attraverso una collezione di oggetti `Battello`, ad esempio un vettore:

```

Battello[] flotta = new Battello[3];
flotta[0] = new Nave("Bally", 100);
flotta[1] = new NaveMercantile("Trasporter", 200, 220);
flotta[2] = new Gommone("Gommy", 300, "Diporto");

```

Il seguente codice *consumer* visualizza i dati sui battelli e calcola il costo del viaggio. La visualizzazione avviene sulla base del metodo `ToString()` definito da ogni classe, il quale è invocato automaticamente dal metodo `WriteLine()`:

```

double totale = 0;
foreach (Battello bat in flotta)
{
    Console.WriteLine("Per il battello {0}, il costo è {1}", bat,
                      bat.CostoViaggio());
    totale+=bat.CostoViaggio();
}
Console.WriteLine("Il totale del costo è:{0}", totale);
Console.WriteLine("\nTotale ammontare retributivo: {0:#,#}", totale);

```

L'output è:

```

Per il battello Bally | 100, il costo è 10

Per il battello Trasporter | 200 | 220, il costo è 64

Per il battello Gommy | 300 | Diporto, il costo è 15

Il totale del costo è:89

```

4.6 Definizione formale di classi astratte e metodi astratti

Nei precedenti paragrafi è stato introdotto il concetto di “classe astratta”, intesa come classe che non riflette oggetti realmente esistenti nel dominio del problema. La rappresentazione che diamo al mondo reale è piena di entità simili. Il termine “mammifero”, ad esempio, definisce una generica classe di animali che possiedono determinate caratteristiche biologiche. In realtà, nessun animale è semplicemente un mammifero, poiché ognuno appartiene ad una determinata specie. In questo senso, “mammifero” assume il ruolo di classe astratta, laddove, ad esempio, “lupo” assume il ruolo di classe non astratta, per la quale esistono cioè esemplari concreti di animali.

Anche nella gerarchia di classi precedentemente definita, `Battello` assume il ruolo di classe astratta; il suo scopo è infatti quello di rappresentare le caratteristiche di una generica imbarcazione

della flotta e non quello di essere istanziata per memorizzare i dati di un particolare battello. Ciò detto, niente nel modo in cui è definita la classe impedisce di farlo. Il seguente codice fa esattamente questo:

```
Battello bat = new Battello("Bally", 100);
Console.WriteLine(bat.CostoViaggio());
```

Il codice è corretto da punto di vista formale, ma è concettualmente errato, poiché non esiste nessuna imbarcazione nella flotta che non sia una nave, un mercantile oppure un gommone. In altre parole, il codice usa la gerarchia di classi in modo non appropriato. Ciò dovrebbe essere impedito, ed esistono due modi per farlo.

Rendere inaccessibili i costruttori della classe astratta

Quello più immediato è usare il modificatore di accesso `protected` per il costruttore o i costruttori della classe astratta. Ciò rende di fatto impossibile istanziarla, in quanto una funzione membro protetta non può essere invocata dal codice *consumer*. Ad esempio:

```
public class Battello
{
    protected string _nome;
    protected double _miglia;

    protected Battello(string nome, double miglia)
    {
        _nome = nome;
        _miglia = miglia;
    }
    . . .
}
```

Dopo questa modifica, il seguente codice risulta formalmente scorretto:

```
Battello bat = new Battello("Bally", 100); // errore formale!
```

Naturalmente ciò non impedisce ai costruttori delle classi `Nave` e `Battello` di continuare a far riferimento al costruttore così modificato.

Parola chiave "abstract": definizione di "classe formalmente astratta"

Il precedente approccio è efficace, ma rappresenta in un certo senso una scappatoia, poiché non introduce alcun meccanismo che renda la classe `Battello` effettivamente diversa dalle altre classi della gerarchia, qualificandola come classe realmente astratta.

Per ottenere questo scopo, il linguaggio mette a disposizione la parola chiave `abstract`, che nell'intestazione della classe deve precedere la parola chiave `class`:

```
abstract class Battello
{
    . . .
}
```

Dopo questa modifica, e indipendentemente dal fatto che la classe definisca o meno uno o più costruttori pubblici, è formalmente impossibile istanziarla. Il seguente codice è dunque errato:

```
Battello bat = new Battello("Bally", 100); // errore formale!
```

4.7 Definizione di funzioni membro astratte

Definire una classe come formalmente astratta mediante la parola chiave `abstract` garantisce che essa non possa essere usata dal codice *consumer*, ma di per sé non modifica affatto la sua implementazione, anche quando ciò sarebbe desiderabile.

A questo proposito consideriamo nuovamente il metodo virtuale `CostoViaggio()` così com'è implementato nella classe `Battello`:

```
public virtual double CostoViaggio()
{
    return 0;
}
```

Il metodo non svolge alcun calcolo e non verrà mai invocato attraverso oggetti della classe `Battello`; la sua ragion d'essere è soltanto quella di consentire un uso polimorfico di oggetti per i quali sarà possibile invocare l'implementazione del metodo in questione. D'altra parte, il linguaggio richiede che ad ogni metodo venga fornito un corpo (anche vuoto, eventualmente), anche se questo non verrà mai eseguito ed è quindi sostanzialmente inutile. E proprio a questo serve l'istruzione `"return 0;"`.

Ma esiste un approccio migliore, infatti all'interno di una classe astratta è possibile, sempre mediante la parola chiave `abstract`, qualificare una funzione come astratta, la quale non contiene cioè alcuna istruzione ed è definita soltanto attraverso il prototipo:

```
public abstract double CostoViaggio();
```

Nel diagramma UML, la rappresentazione di un metodo astratto, viene fatta scrivendo il metodo in corsivo (similmente al nome della classe).

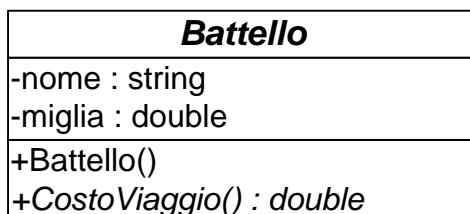


Figura 5-6 Rappresentazione UML di una classe astratta con un metodo astratto

Un metodo astratto è implicitamente virtuale ed esiste solo per essere ridefinito nelle classi derivate; dunque non può essere invocato! Soltanto una classe astratta, caratterizzata dalla parola chiave `abstract` nell'intestazione, può definire una o più funzioni astratte, o ereditare funzioni astratte da una classe base (senza dunque essere obbligata a fornirne una implementazione).

Definire un metodo come astratto non solo evita la necessità di fornire un'implementazione inutile, ma produce un secondo e più importante vantaggio: costringe formalmente le classi derivate a ridefinirlo.

Ad esempio, qualora la classe `Gommone` non fornisse la propria implementazione del metodo `CostoViaggio()`, il compilatore segnalerebbe un errore formale, informando il programmatore che `Gommone` è obbligata a implementare il metodo.

Tutto ciò evita che per una dimenticanza del programmatore una classe si limiti a ereditare un metodo invece di ridefinirlo. Ritornando alla classe `Gommone`, se questa ereditasse `CostoViaggio()` invece di fornire la propria implementazione, si otterrebbe come risultato un costo pari a zero per tutti i viaggi fatti con un gommone.

Ebbene, si pensi a una classe astratta che definisce una decina di funzioni astratte senza usare l'apposita parola chiave, fornendo invece una implementazione vuota. Si ipotizzi inoltre che una

classe derivata non ridefinisca una o più di tali funzioni: il risultato potrebbe essere un bug estremamente difficile da scovare, poiché dipendente da un errore di progettazione.

4.8 Considerazioni sulla progettazione

Nella progettazione di gerarchie di classi la definizione di classi astratte è connessa a due aspetti, uno di natura sostanziale, l'altro di natura formale.

La loro funzione principale, come classi che stanno alla base di una linea di discendenza, è quella di consentire una rappresentazione accurata del dominio del problema, evitando la necessità di stabilire relazioni arbitrarie tra classi al solo scopo di favorire il riutilizzo del codice. Come naturale conseguenza, sempre che tali classi siano implementate in modo appropriato, viene favorita la realizzazione di codice *consumer* chiaro e compatto, che sfrutta appieno i principi del polimorfismo.

Dal lato formale, il linguaggio C# mette a disposizione la parola chiave `abstract` con lo scopo di garantire un uso appropriato delle classi e dei metodi astratti; infatti, sia le prime che i secondi esistono per essere derivati e non per essere utilizzati da codice *consumer*.

5 Varianza dei tipi

All'interno di un sistema dei tipi rappresentato da una gerarchia di classi, assume una notevole importanza il concetto di “varianza”, termine che investe gli stessi significati che si trovano anche all'interno delle definizioni degli spazi vettoriali.

La prima cosa da capire per iniziare a comprendere i concetti di varianza è il fatto che per due tipi di dato T ed U solamente una delle seguenti definizioni è vera:

- ❑ T è più grande di U
- ❑ T è più piccolo di U
- ❑ T è uguale ad U
- ❑ T non è in relazione con U

Riprendiamo quindi la gerarchia di classi sviluppata in precedenza ed andiamo ad analizzarla sotto questo aspetto.

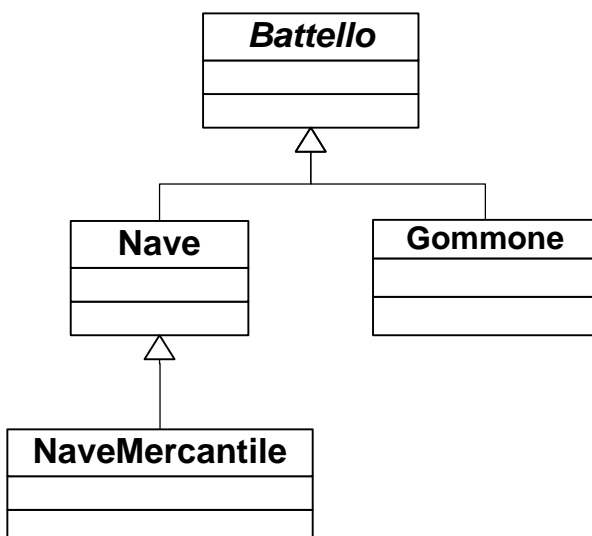


Figura 5-7 Rappresentazione UML della gerarchia di classi di esempio

In questo caso risulta essere che il tipo di dato `Nave` è più grande del tipo `NaveMercantile`, e più piccolo del tipo `Battello`, non ha nessuna relazione con il tipo `Gommone`, ed è ovviamente uguale al tipo `Nave`.

Queste considerazioni diventano importanti quando si devono memorizzare delle variabili in C# in quanto, rispetto ad una locazione di un certo tipo, viene consentito di memorizzare un oggetto che abbia un tipo uguale o più piccolo.

Pertanto se abbiamo una variabile di tipo `Nave`, essa potrà contenere od un altro oggetto di tipo `Nave` oppure un oggetto di tipo `NaveMercantile`, ma mai un `Battello` oppure un `Gommone`.

L'idea di memorizzare il tipo di dato in una locazione di memorizzazione è un esempio specifico di un principio più generale chiamato "principio di sostituzione".

A questo punto possiamo iniziare a definire i concetti di varianza, che si applicano agli operatori che manipolano i tipi (come gli operatori di conversione o di assegnamento).

Un operatore si definisce quindi:

- ❑ **Covariante**: se preserva l'ordinamento \leq dei tipi (da quelli più specifici fino a quelli più generici);
- ❑ **Controvariante**: se inverte questo ordine (dai tipi più generici ai tipi più specifici);
- ❑ **Invariante**: se non applica nessuna delle due relazioni.

Numerosi sviluppatori non ne avranno mai sentito parlare (anche se li avranno sicuramente utilizzati) ma è comunque indispensabile conoscere questi concetti perché gli operatori di conversione del C# sono spesso covarianti, spesso controvarianti e spesso invarianti.

5.1 Invarianza

All'interno del linguaggio C#, il concetto di invarianza viene applicato a tutti i vettori di oggetti.

Questo sta a significare che, ad esempio, ad una variabile di tipo `Nave[]` è possibile assegnare solo un riferimento ad un oggetto dello stesso tipo.

Quindi considerando la gerarchia di classi sviluppata in precedenza è possibile scrivere solamente il seguente codice

```
Nave[] vetNave = new Nave[3];
```

Mentre, in linea teorica, qualsiasi altra assegnazione alla variabile `vetNave` di un riferimento di un altro tipo dovrebbe produrre un errore. Ma se prendessimo in considerazione il codice seguente:

```
NaveMercantile[] vetNaveMerc = new NaveMercantile[3];  
vetNave = vetNaveMerc;
```

ci si aspetterebbe di ottenere un errore in fase di compilazione.

A prima vista questa è una grossa limitazione ed infatti nel linguaggio C# viene consentito a tutti i vettori di oggetti di tipo riferimento di essere covarianti.

Ora l'assegnazione precedente diventa lecita a patto, però, che tutti gli elementi del vettore siano, a questo punto, del tipo di dato `NaveMercantile`.

Se infatti provassimo a scrivere questo codice:

```
vetNave[0] = new NaveMercantile("Trasporter", 200, 220);  
vetNave[1] = new Nave("Bally", 100); // errore in fase di esecuzione
```

entrambe le righe verrebbero compilate correttamente, ma in fase di esecuzione otterremo il sollevamento dell'eccezione `ArrayTypeMismatchException` quando cercheremmo di assegnare un oggetto di tipo `Nave` ad un elemento del vettore

5.2 Covarianza

In unione al concetto di invarianza, quelli di covarianza e controvarianza si applicano ai valori di ritorno ed agli argomenti dei metodi.

All'interno del linguaggio C# il valore di ritorno di ogni metodo è covariante, mentre tutti gli argomenti sono controvarianti.

Per capire meglio questo concetto facciamo ancora un esempio, ipotizzando di avere un metodo che accetta come argomento un oggetto di tipo `Nave` e restituisce un oggetto dello stesso tipo:

```
static Nave Metodo(Nave n)
{
    return new Nave();
}
```

Il fatto che il valore di ritorno del metodo sia covariante implica che esso può essere assegnato solo ad oggetti che, gerarchicamente parlando, sono di tipo maggiore od uguale a quello ritornato.

Quindi queste righe di codice sono lecite:

```
Nave unaNave = new Nave();

Battello b = Metodo(unaNave);
Nave n = Metodo(unaNave);
```

Mentre il valore di ritorno non potrà essere assegnato ad un oggetto di tipo `NaveMercantile` in quanto più specializzato (quindi minore) rispetto al tipo `Nave`

Il seguente codice produrrà quindi un errore di compilazione

```
NaveMercantile nMerc = Metodo(unaNave); // errore in compilazione
```

5.3 Controvarianza

Utilizzando sempre lo stesso codice scritto un precedenza, spieghiamo il concetto per cui tutti gli argomenti di un metodo sono controvarianti (ma non covarianti).

Questo significa che si possono passare oggetti che appartengono allo stesso tipo dell'argomento (ovviamente), ma anche a tipi derivati da esso (quindi minori).

```
Metodo(unaNave);
Metodo(unaNaveMercantile);
```

Al contrario, non sarà possibile utilizzare argomenti di tipo "maggiore" rispetto al tipo dell'argomento come in questa riga di codice.

```
Metodo(unBattello); // errore in compilazione
```

che produrrà, quindi un errore in fase di compilazione.

6

Classi generiche

Nei capitoli precedenti abbiamo visto come poter rappresentare degli oggetti astratti, oggetti, cioè che non hanno nessuna corrispondenza con degli oggetti appartenenti al dominio del problema. Durante la trattazione dell'argomento abbiamo spesso descritto sia le classi base sia quelle astratte come classi generiche in grado, quindi, di poter rappresentare oggetti appartenenti ad esse o classi derivate.

1 I problemi delle classi fortemente tipizzate

Con l'avvento del .NET 2.0, però, la parola “generico” ha assunto un altro significato, quello cioè di fornire al programmatore un modo per definire dei “segnaposti” (chiamati formalmente parametri di tipo) sia per gli argomenti di metodi sia per definizione di tipi che sono specificati in fase di invocazione del metodo generico o nella creazione del tipo generico.

Per chiarire meglio questo concetto, riprendiamo la classe Posizione utilizzata precedentemente.

Posizione
-latitudine : int -longitudine : int «Proprietà» +Latitudine : int «Proprietà» +Longitudine : int
+Posizione(in latitudine : int, in longitudine : int) +Posizione() +ToString() : string <u>+Parse(in str : string) : Posizione</u> <u>+TryParse(in str : string, out nuovaPosizione : Posizione) : bool</u>

Figura 6-1 Diagramma della classe Posizione.

Della quale, giusto per semplicità riportiamo il codice più significativo

```
public class Posizione
{
    int _latitudine;
    public int Latitudine
    {
        get { return _latitudine; }
        set { _latitudine = value; }
    }
}
```



```
int _longitudine;
public int Longitudine
{
    get { return _longitudine; }
    set { _longitudine = value; }
}

public Posizione(int latitudine, int longitudine)
{
    _latitudine = latitudine;
    _longitudine = longitudine;
}

public Posizione()
{
}
}
```

L'utilizzo della classe `Posizione` è molto semplice, come evidenziato da questo frammento di codice consumer

```
Posizione primaPosizione = new Posizione(10, 10);
Console.WriteLine("La prima posizione è :{0}", primaPosizione.ToString());
Console.WriteLine();
```

che produce il seguente output:

La prima posizione è :10#10

Così come implementata, però, questa classe ha il problema di costringere l'utilizzatore ad usare un valore intero per esprimere le coordinate di latitudine e longitudine.

Se si volesse concedere la possibilità di usare valori `double`, o ancor meglio creare un nuovo tipo “coordinata” ed usare quello, saremmo costretti a creare delle altre classi, che abbiano gli attributi, i costruttori e quanto serve, in grado di accettare questi tipi di dato, come nell'esempio che segue.

```
public class PosizioneDouble
{
    double _latitudine;
    public double Latitudine
    {
        get { return _latitudine; }
        set { _latitudine = value; }
    }

    double _longitudine;
    public double Longitudine
    {
        get { return _longitudine; }
        set { _longitudine = value; }
    }

    public Posizione(double latitudine, double longitudine)
```

```

{
    _latitudine = latitudine;
    _longitudine = longitudine;
}

public Posizione()
{ }
. . .
}

```

L'ideale, quindi, sarebbe la possibilità di costruire un nuovo tipo di dato in grado di “contenere” attributi di più tipi diversi. Questa possibilità è offerta dai generics!

2 La soluzione: i generics

Per capire che cosa sono i generics, si pensi a loro come ad una classe che non ci obbliga a specificare un determinato tipo di dato, ma ci concede la possibilità di specificare questo tipo come fosse un parametro.

In UML ciò si rappresenta con il nome del parametro `Tipo` in alto a destra all'interno di un rettangolo tratteggiato.

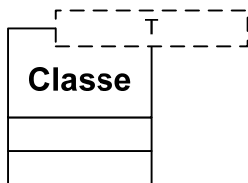


Figura 6-2 Rappresentazione di una classe generica.

I generics forniscono il controllo sui tipi ed è a livello di compilatore che ci si assicura che i tipi utilizzati siano consistenti esercitando un controllo molto rigoroso, senza quindi, nessuna perdita di prestazioni od incremento nelle dimensioni del codice generato.

2.1 Creare una nuova classe generica

Per creare una classe in grado di supportare i generics, è sufficiente sostituire il tipo (che nel caso di `Posizione` è `int`) con un tipo generico (per esempio `T`¹⁴)

Quindi:

```
int _latitudine;
```

diventerà

```
TPosizione _latitudine;
```

Il tipo utilizzato per dichiarare l'attributo, essendo “generico” verrà definito quando si creerà la classe, e questo lo si potrà fare dichiarando il tipo “vero” all'interno di due parentesi angolari.

La classe `Posizione` diventerà la seguente:

```
public class Posizione<TPosizione>
```

¹⁴ E' una consuetudine utilizzare `T` per indicare un “Tipo”, ma le più recenti linee guida consigliano di dare nomi più descrittivi, come `TPosizione`.

```

{
    private TPosizione _latitudine;
    public TPosizione Latitudine
    {
        get { return _latitudine; }
        set { _latitudine = value; }
    }

    TPosizione _longitudine;
    public TPosizione Longitudine
    {
        get { return _longitudine; }
        set { _longitudine = value; }
    }

    public Posizione(TPosizione latitudine, TPosizione longitudine)
    {
        _latitudine = latitudine;
        _longitudine = longitudine;
    }

    public Posizione()
    { }
    . . .
}

```

Segue il diagramma UML aggiornato della classe:

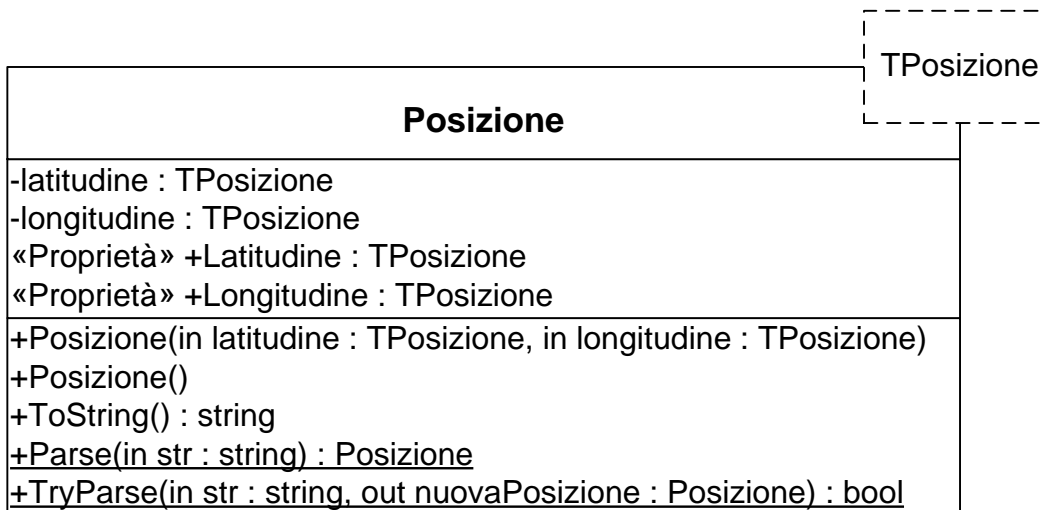


Figura 6-3 Diagramma della nuova classe Posizione.

A questo punto il gioco è fatto e per poter utilizzare la classe `Posizione` con differenti tipi di dato, sarà sufficiente specificare il “parametro” del tipo all’interno delle parentesi angolari, come in questo esempio:

```

Posizione<int> primaPosizione = new Posizione<int>(10, 10);
Posizione<double> secondaPosizione = new Posizione<double>(20.12, 20.13);

```

```

Posizione<Coordinata> terzaPosizione = new Posizione<Coordinata>(
new Coordinata(1,2,3), new Coordinata(4,5,6));
Console.WriteLine("La prima posizione è :{0}", primaPosizione.ToString());
Console.WriteLine("La seconda posizione è :{0}", secondaPosizione.ToString());
Console.WriteLine("La terza posizione è :{0}", terzaPosizione.ToString());

```

che produrrà il seguente output

```
La prima posizione è :10#10
```

```
La seconda posizione è :20#20
```

```
La terza posizione è :1,2,3#4,5,6
```

2.2 Limitare i tipi di parametro

La possibilità di avere il tipo della classe come fosse un parametro, però, non ci vieta di creare una istanza della classe `Posizione` utilizzando tipi inadatti a rappresentare delle coordinate, come ad esempio la classe `string` o addirittura una classe `Comandante`! Ad esempio:

```
Posizione<string> posizioneStringa = new Posizione< string >("20", "20");
```

Per risolvere questo problema, i generics offrono la possibilità di limitare il tipo dei parametri con il quale verrà costruita la classe utilizzando la clausola **where** in fase di dichiarazione, il cui significato può assumere i seguenti valori.

Tabella 6-1 Limitazione del tipo di parametro in una classe generica.

VINCOLO	DESCRIZIONE
where T : struct	L'argomento di tipo deve essere un tipo di valore. È possibile specificare qualsiasi tipo di valore tranne Nullable
where T : class	L'argomento di tipo deve essere un tipo di riferimento, incluso qualsiasi tipo di classe, interfaccia, delegato o matrice.
where T : new()	L'argomento di tipo deve disporre di un costruttore pubblico senza parametri. Se utilizzato insieme ad altri vincoli, il vincolo <code>new()</code> deve essere specificato per ultimo.
where T : NomeDiClasseBase	L'argomento di tipo deve corrispondere alla classe base specificata o derivare da tale classe.
where T : NomeDiInterfaccia	L'argomento di tipo deve corrispondere all'interfaccia specificata o implementare tale interfaccia. È possibile specificare più vincoli di interfaccia. L'interfaccia vincolante può anche essere generica.
where T : U	L'argomento di tipo fornito per T deve corrispondere all'argomento fornito per U o derivare da tale argomento. In questo caso si tratta di un vincolo di tipo naked.

I vincoli possono anche essere concatenati tra di loro. Ad esempio, ipotizziamo di modificare l'intestazione della classe aggiungendo i vincoli **class** e `new()`:

```
public class Posizione<T> where T : class, new()
```

Dopo questa modifica, soltanto la terza delle istruzioni che seguono è formalmente valida:

```

Posizione<int> p1 = new Posizione<int>(10, 10);           // errore
Posizione<double> p2 = new Posizione<double>(20.12, 20.13); // errore
Posizione<Coordinata> p3 =
    new Posizione<Coordinata>(new Coordinata(1,2,3),

```

```
new Coordinata(4,5,6);
```

Verificare il tipo effettivo del parametro di tipo

La precedente modifica vincola la natura dei tipi utilizzabili con `Posizione`, ma non i tipi stessi. Ad esempio, anche con la nuova versione nulla impedisce di creare oggetti di tipo `Posizione` che usino come coordinate oggetti di tipo `Comandante`!

Per risolvere anche questo problema è possibile usare l'operatore `typeof()` che ci permette di ricavare il tipo passato come parametro alla classe generica.

L'operatore `typeof()` restituisce un oggetto di tipo `Type`, il quale mette a disposizione numerosi metodi e proprietà (come la proprietà `Name`) che consentono di stabilire il tipo effettivo utilizzato in fase di costruzione di un oggetto.

Ad esempio, il seguente codice limita a `Coordinata`, `Int32` e `Double` i tipi utilizzabili come coordinate in un oggetto `Posizione`:

```
public Posizione(T latitudine, T longitudine)
{
    Type t = typeof(T);
    if (t.Name == "Coordinata" || t.Name == "Int32" || t.Name == "Double")
    {
        _latitudine = latitudine;
        _longitudine = longitudine;
    }
    else
        throw new ArgumentException("Tipo della coordinata non valido");
}
.
```

7

Classi come collezioni di oggetti

Nei capitoli precedenti abbiamo visto come poter rappresentare gli oggetti del dominio del problema mediante la creazione di nuovi tipi. Abbiamo visto inoltre che in molti casi gli oggetti non esistono come entità isolate ma sono raggruppati in collezioni. Ad esempio, la classe `Nave` utilizza una collezione di tipo `List<Posizione>` per memorizzare la rotta di percorrenza.

Nonostante .NET fornisca molte classi che implementano una collezione (e `List<T>` ne è un esempio), spesso è buona norma progettare le proprie collezioni, implementando una classe ad hoc. In questo modo si può stabilire a priori il tipo degli elementi, definire i metodi esporre al codice consumer e in generale porre dei vincoli sull'uso che si può fare della collezione.

Convenzionalmente, si parla in questo caso di collezioni tipizzate. Una collezione tipizzata può chiamarsi con un nome proprio se il suo utilizzo è legato strettamente ad una specifica funzionalità (ad esempio `Rotta` per collezioni di oggetti di tipo `Posizione`), oppure aggiungendo la parola "Collection" al nome del tipo degli elementi se è stata creata per un utilizzo più generale (ad esempio `PosizioneCollection`).

1 Creare nuovi tipi Collection

1.1 Creare collezioni tipizzate mediante aggregazione

Uno dei metodi con cui si possono creare delle collezioni tipizzate risiede nello sfruttare il concetto di aggregazione visto in precedenza, definendo, cioè, un membro della classe in grado di contenere 0 o più oggetti del tipo desiderato utilizzando, magari una collezione esistente nel framework come la classe `List<>`.

Quello che segue è lo scheletro minimo di una classe grado di rappresentare la rotta di una nave ed il diagramma UML corrispondente.

```
public class Rotta
{
    List<Posizione> _posizioni;

    public Rotta()
    {
        _posizioni = new List<Posizione>();
    }
    // . . .
}
```

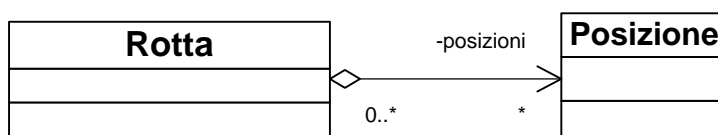


Figura 7-1 Diagramma della classe Rotta

1.2 Accesso agli elementi di una collezione

Traendo sempre spunto dal nostro esempio della flotta navale, aggiungiamo alla classe appena creata tutto ciò che serve per essere in grado di rappresentare la rotta di una nave, compresi anche un paio di metodi per consentire l'accesso agli elementi

```
public class Rotta
{
    List<Posizione> _posizioni;

    public Rotta()
    {
        _posizioni = new List<Posizione>();
    }

    public void Add(Posizione pos)
    {
        _posizioni.Add(pos);
    }

    public Posizione GetPosizione(int indice)
    {
        return _posizioni[indice];
    }

    public void SetPosizione(int indice, Posizione pos)
    {
        _posizioni[indice] = pos;
    }
}
```

La classe appena realizzata è sufficiente per scrivere il seguente codice consumer

```
Rotta navigazione = new Rotta();
navigazione.Add(new Posizione(10, 10));
Posizione pos = new Posizione(30, 30);
```

```
Console.WriteLine("Posizione :{0}", navigazione.GetPosizione(0).ToString());
navigazione.SetPosizione(0, pos);
```

Appare subito evidente, però, che l'implementazione fornita non è di facile utilizzo per l'accesso ai singoli elementi della collezione. Infatti per poter recuperare o impostare una posizione si deve ricorrere ai metodi di accesso Get e Set, mentre sarebbe desiderabile poter accedere alla collezione di oggetti `Posizione` nello stesso modo in cui è possibile accedere agli elementi di qualsiasi collezione, e cioè usando un indice. In sostanza sarebbe desiderabile poter scrivere qualcosa del tipo:

```
Rotta elencoPos = new Rotta();
elencoPos.Add(new Posizione(10, 10));
```

```
Posizione punto = elencoPos[0];
elencoPos[0] = new Posizione(30, 30);
```

Il linguaggio C# ci fornisce questa possibilità attraverso gli indicizzatori

1.3 Definizione e uso di indicizzatori

Un indicizzatore funziona in modo del tutto analogo a una proprietà, con la differenza che consente un accesso di natura indicizzata, nel quale una variabile (o più variabili) viene usata come indice all'interno degli *accessor* per stabilire a quale elemento si intende accedere.

La definizione di un indicizzatore assume la seguente forma:

```
modificatoreopz tipo this[lista-indici]
{
    modificatoreopz get { ... }opz
    modificatoreopz set { ... }opz
}
```

Un indicizzatore è dunque definito da:

- ❑ un tipo;
- ❑ uno o più variabili indice;
- ❑ dal codice di accesso, suddiviso in *get accessor* e *set accessor*.

Diversamente dalle proprietà, gli indicizzatori non hanno un nome, poiché essi vengono applicati direttamente agli oggetti della classe; per questo motivo ogni indicizzatore è caratterizzato dalla parola chiave *this*.

Ritornando all'esempio precedente un indicizzatore rappresenta un'alternativa naturale alla coppia di metodi di accesso `GetPosizione()` e `SetPosizione()`:

```
public class Rotta
{
```

```
    ...
```

```
    public Posizione this[int indice]
    {
        get
        {
            return _posizioni[indice];
        }
        set
        {
            _posizioni[indice] = value;
        }
    }
}
```

```
    ...
```

```
}
```

Confrontando questo codice con quello precedente si vede che:

- ❑ l'indicizzatore contiene le stesse istruzioni dei due metodi di accesso, che adesso sono sostituiti dal *get* e dal *set accessor*.

- ❑ esattamente come i due metodi di accesso, anche l'indicizzatore definisce una variabile indice che viene utilizzata negli *accessor* per accedere alla lista `posizioni`;
- ❑ nel *set accessor* il parametro implicito `value` rappresenta l'equivalente del parametro `nome` definito in `SetPosizione()`.

Adesso, nel codice *consumer* gli oggetti della classe `Rotta` possono essere utilizzati come collezioni. Ad esempio:

```
Rotta elencoPos = new Rotta();
elencoPos.Add(new Posizione(10, 10));
elencoPos.Add(new Posizione(20, 20));
```

```
Posizione punto = elencoPos[1];
elencoPos[0] = new Posizione(30, 30);
```

1.4 “set accessor” e “get accessor” negli indicizzatori

Per meglio comprendere il funzionamento degli indicizzatori vediamo in che modo il linguaggio traduce la definizione di un indicizzatore ed ogni riferimento ad esso nel codice *consumer*.

Il codice del predente esempio viene tradotto in:

```
public class Posizione
{
    ...
    public string get_Item (int indice)
    {
        return _posizioni[indice];
    }

    public void set_Item(int indice, string value)
    {
        _posizioni[indice] = value;
    }
    ...
}
```

Il linguaggio crea un metodo per ogni *accessor*, chiamandolo con il prefisso `get` se è un *get accessor*, `set` se è un *set accessor*. Il prototipo del metodo *get* definisce il solo parametro indice e dichiara come tipo di ritorno quello dell'indicizzatore. Il metodo *set* definisce due parametri, l'indice e il parametro `value`, dello stesso tipo dell'indicizzatore.

Nel codice *consumer* tutti i riferimenti all'indicizzatore vengono tradotti in invocazioni ai metodi *accessor*, in base al fatto che esso sia usato in una espressione o sia oggetto di assegnazione. Dunque, il seguente codice:

```
Rotta elencoPos = new Rotta();
elencoPos.Add(new Posizione(10, 10));
elencoPos.Add(new Posizione(20, 20));
```

```
Posizione punto = elencoPos[1];
elencoPos[0] = new Posizione(30, 30);
```

viene tradotto dal linguaggio in:

```

Rotta elencoPos = new Rotta();
elencoPos.Add(new Posizione(10, 10));
elencoPos.Add(new Posizione(20, 20));

Posizione punto = elencoPos.get_Item(1);
elencoPos.set_Item(0) = new Posizione(30, 30);

```

1.5 Tipo degli indici e codice ammissibile negli accessor di un indicizzatore

L'uso degli indicizzatori rende gli oggetti di una classe *collection* molto simili a un array, in quanto consente al codice di *consumer* di accedere ad essi in modo indicizzato; esiste però una sostanziale differenza nella modalità d'accesso:

nell'accesso agli elementi di un array è obbligatorio specificare un indice (o indici) di tipo intero; un indicizzatore può invece definire indici di tipo qualsiasi.

Il perché di questo è ovvio se si considera in che modo il linguaggio traduce gli indicizzatori, e cioè in una coppia di metodi di accesso, per i quali non esiste alcuna restrizione sul tipo del parametro usato come indice.

Un indicizzatore rappresenta soltanto un meccanismo di accesso e dunque, come avviene anche per le proprietà, non esiste alcuna restrizione nemmeno sul codice, purché il *get accessor* ritorni un valore compatibile con il tipo dell'indicizzatore. Tutto questo consente al programmatore di scrivere codice alquanto stravagante (e privo di senso), ma dal punto di vista formale perfettamente lecito:

```

class ClasseEsempio
{
    public int this[string indice]           // indice di tipo stringa!
    {
        get
        {
            return 0;    // privo di senso ma corretto
        }
        set              // accessor vuoto
        {
        }
    }
}

```

Come si vede, la classe definisce un indicizzatore che non fornisce l'accesso ad alcunché, ma che pure rispetta le regole formali del linguaggio e può essere utilizzato nel codice *consumer*:

```

ClasseEsempio oggi = new ClasseEsempio();
oggi["ciao come stai"] = 100;           // questa istruzione non produce niente!
int a = oggi["non molto bene"];         // ad a viene assegnato 0

```

1.6 Iterare la collezione

Nonostante l'aggiunta di un indicizzatore, *Rotta* non può ancora definirsi una collezione; infatti, è opportuno fornire al codice *consumer* la possibilità di iterare sugli elementi della lista mediante un ciclo *for()* o un *foreach()*.

Proprietà Count

Per rendere la collezione iterabile mediante un ciclo `for()` è sufficiente che questa esponga al codice *consumer* il numero degli elementi, ad esempio implementando la proprietà `Count`:

```
public int Count
{
    get
    {
        return _posizioni.Count;
    }
}
```

Ciò consente di scrivere il seguente codice:

```
Rotta elencoPos = new Rotta();
elencoPos.Add(new Posizione(10, 10));
...
for (int i = 0; i < elencoPos.Count; i++)
{
    Posizione posi = (Posizione) elencoPos[i];
    Console.WriteLine(posi.ToString());
}
```

Interfaccia IEnumerable

Benché non sia strettamente necessario, per rendere `Rotta` una vera collezione è necessario che sia iterabile anche con un ciclo `foreach()`. Quest'ultimo richiede che la classe implementi l'interfaccia `IEnumerable`¹⁵.

L'interfaccia `IEnumerable` espone un solo metodo, `GetEnumerator()`, il quale deve ritornare un oggetto di tipo `IEnumerator`. Realizzare tale metodo è molto semplice, poiché basta invocare il metodo omologo appartenente alla lista che contiene gli elementi:

```
public class Rotta : IEnumerable
{
    ...
    public IEnumerator GetEnumerator()
    {
        return _posizioni.GetEnumerator();
    }
}
```

Adesso la classe è in grado di essere utilizzata dal seguente codice

```
Rotta elencoPos = new Rotta();
elencoPos.Add(new Posizione(10, 10));
...
foreach (Posizione posi in elencoPos)
{
    Console.WriteLine(posi.ToString());
}
```

¹⁵ Ci occuperemo approfonditamente del concetto d'interfaccia nel prossimo capitolo

1.7 Iteratori

Il ciclo `foreach()` sfrutta l'enumeratore ritornato dal metodo `GetEnumerator()` per accedere sequenzialmente ad ogni elemento di una collezione, partendo dal primo fino ad arrivare all'ultimo. Sebbene molto utile, questo costrutto ha dei limiti, poiché non permette di stabilire né l'ordine di scansione né i criteri dello stesso.

Se volessimo “filtrare” solo una parte della collezione oppure scorrerla in maniera inversa dovremmo ricorrere al più classico ciclo `for()` con tutto quello che ne consegue. Oppure, potremmo implementare uno o più iteratori.

Gli iteratori sono una nuova funzionalità del C# 2.0 e rappresentano una sezione di codice che restituisce una determinata sequenza di valori dello stesso tipo. Strettamente legati al ciclo `foreach()`, gli iteratori vengono utilizzati per “personalizzare” il meccanismo di scansione della sequenza.

Un iteratore utilizza l'istruzione **`yield return`** per restituire di volta in volta i singoli elementi e l'istruzione **`yield break`** per terminare l'iterazione.

Il tipo restituito di un iteratore deve essere uno tra i seguenti: `IEnumerable`, `IEnumerator`, `IEnumerable<T>` o `IEnumerator<T>`.

Facciamo un esempio ed ipotizziamo di voler scandire gli elementi della collezione `Rotta` partendo dall'ultimo fino al primo. Se usassimo un ciclo `for()` dovremmo scrivere il seguente codice:

```
for (int i = elencoPos.Count-1; i >= 0 ; i--)
{
    Posizione posi = elencoPos[i];
    Console.WriteLine(posi.ToString());
}
```

E' possibile implementare un iteratore che ritorni gli elementi nell'ordine appropriato e che sia utilizzabile da un ciclo `foreach()`:

```
public IEnumerable Inversa
{
    get
    {
        for (int i = Count - 1; i >= 0; i--)
        {
            yield return _posizioni[i];
        }
    }
}
```

La parola chiave **`yield`** viene utilizzata per specificare il valore o i valori restituiti. Quando viene raggiunta l'istruzione **`yield return`**, la posizione corrente viene archiviata ed alla successiva chiamata dell'iteratore, l'esecuzione verrà riavviata a partire da questa posizione.

In questo caso l'iteratore è stato implementato mediante una proprietà, e come tale può essere utilizzato nel codice *consumer*:

```
foreach (Posizione posi in elencoPos.Inversa)
{
    Console.WriteLine(posi.ToString());
}
```

Filtrare gli elementi della collezione

Una classe può definire più iteratori, i quali possono essere implementati mediante proprietà, metodi e operatori. L'implementazione mediante un metodo fa degli iteratori uno strumento estremamente flessibile, poiché consente al codice consumer di intervenire sul meccanismo di scansione degli elementi.

Come esempio ipotizziamo di voler creare un iteratore che filtri gli elementi della collezione, restituendo solamente quelli che corrispondono ad un certo criterio, in questo caso una determinata latitudine:

```
public IEnumerable LatitudineUgualeA(int latitudine)
{
    foreach (Posizione p in _posizioni)
    {
        if (p.Latitudine == latitudine)
            yield return p;
    }
}
```

L'iteratore è implementato mediante un metodo che riceve come argomento la latitudine di riferimento. Ecco come utilizzarlo nel codice consumer:

```
foreach (Posizione posi in elencoPos.LatitudineUgualeA(20))
{
    Console.WriteLine(posi.ToString());
}
```

Interrompere la scansione degli elementi

Il costrutto **yield break** consente di interrompere la scansione della collezione e facilita la realizzazione di iteratori che ritornano un sottoinsieme degli elementi.

Ad esempio, il seguente iteratore restituisce gli elementi della collezione a partire dal primo fino a quello di posizione specificata:

```
public IEnumerable FinoA (int indice)
{
    for (int i = 0; i < Count; i++)
    {
        if (i < indice)
            yield return _posizioni[i];
        else
            yield break;
    }
}
```

Ecco come utilizzare l'iteratore per ottenere i primo 10 elementi della collezione::

```
foreach (Posizione posi in elencoPos.FinoA(10))
{
    Console.WriteLine(posi.ToString());
}
```

1.8 Creare collezioni tipizzate mediante derivazione

I paragrafi precedenti hanno introdotto gli elementi principali che caratterizzano una collezione e dimostrano che la realizzazione di una collezione tipizzata richiede un certo sforzo di programmazione.

Ebbene, in realtà il codice da scrivere può essere ridotto al minimo semplicemente derivando la collezione tipizzata da una collezione esistente. In questo modo, la nuova classe eredita già tutte le caratteristiche tipiche di una collezione.

Implementazione della collezione PosizioneCollection

Ecco come creare una collezione tipizzata che memorizzi una lista di oggetto `Posizione`:

```
public class PosizioneCollection : List<Posizione>
{ }
```

A questo punto, metodi, indicizzatori, possibilità di iterare con cicli `for` e `foreach` sono già inclusi “di serie” in quanto comportamenti predefiniti dalla classe generica del .NET `List<>`.

Il seguente codice dimostra che la nuova classe `PosizioneCollection` è una collezione a tutti gli effetti:

```
PosizioneCollection collPos = new PosizioneCollection();
collPos.Add(new Posizione(10, 10));
collPos.Add(new Posizione(20, 20));
collPos[0] = new Posizione(30, 3);
```

```
for (int i = 0; i < elencoPos.Count; i++)
{
    Console.WriteLine(collPos[i].ToString());
}
```

```
foreach (Posizione pos in elencoPos)
{
    Console.WriteLine(pos.ToString());
}
```

Naturalmente, è possibile aggiungere alla nuova classe operazioni e proprietà specifiche, che ne estendano le funzionalità rispetto ad una collezione qualsiasi.

Per implementare nuove funzionalità è di norma necessario accedere alla lista degli elementi, cosa che si ottiene mediante la parola chiave `this`.

Il seguente codice definisce un metodo che ritorna l’oggetto `Posizione` corrispondente ad coppia latitudine, longitudine specificate:

```
public class PosizioneCollection : List<Posizione>
{
    . . .
    public int IndexOf(int latitudine, int longitudine)
    {
        int pos = -1;
        for (int i = 0; i < this.Count; i++)
        {
            Posizione posizione = this[i];
            if (posizione.Latitudine == latitudine &&
                Posizione.Longitudine == longitudine)
```

```

        pos = i;
    }
    return pos;
}
}
. . .

```

Segue il diagramma UML che mostra le relazioni tra le classi `PosizioneCollection`, `Posizione` e `List<Posizione>`:

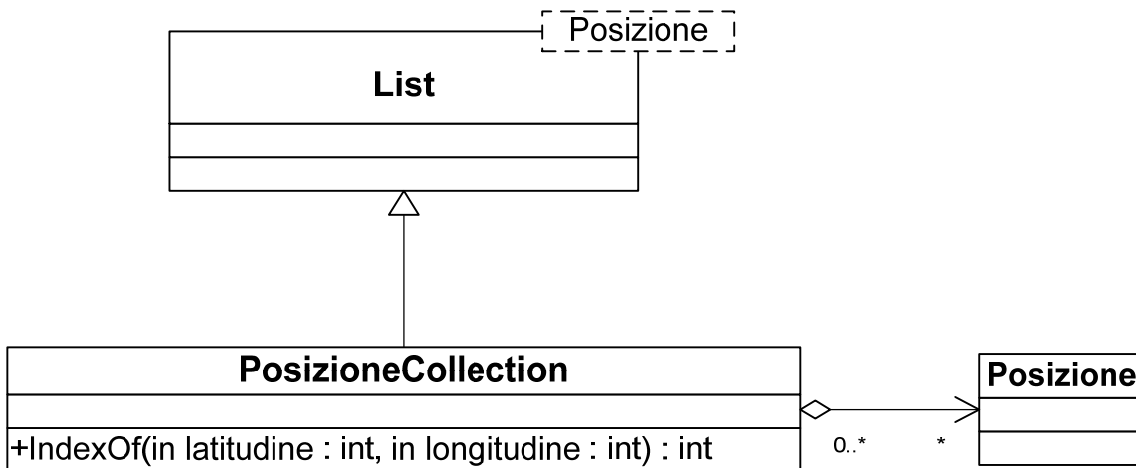


Figura 7-2 Diagramma della classe `PosizioneCollection`.

Implementazione della collezione `NaveCollection`

Così come abbiamo creato la collezione di oggetti `Posizione`, possiamo fare altrettanto per gli oggetti di tipo `Nave`, costruendo finalmente la nostra flotta di navi.

Anche in questo caso possiamo scegliere il nome tra `Flotta` oppure `NaveCollection`, ma l'implementazione sarà sempre la stessa.

```

public class NaveCollection : List<Nave>
{
    public int IndexOf(string nome)
    {
        int pos = -1;
        for (int i = 0; i < this.Count; i++)
        {
            if (this[i].Nome == nome)
                pos = i;
        }
        return pos;
    }

    public void Remove(string nome)
    {
        int pos = IndexOf(nome);
        if (pos != -1)
            this.RemoveAt(pos);
    }
}

```

```

    }

    public NaveCollection GetListaNavi(StatoNave stato)
    {
        NaveCollection elenco = new NaveCollection();
        foreach (Nave barca in this)
        {
            if (barca.Stato == stato)
                elenco.Add(barca);
        }
        return elenco;
    }
}

```

Segue il diagramma UML della classe:

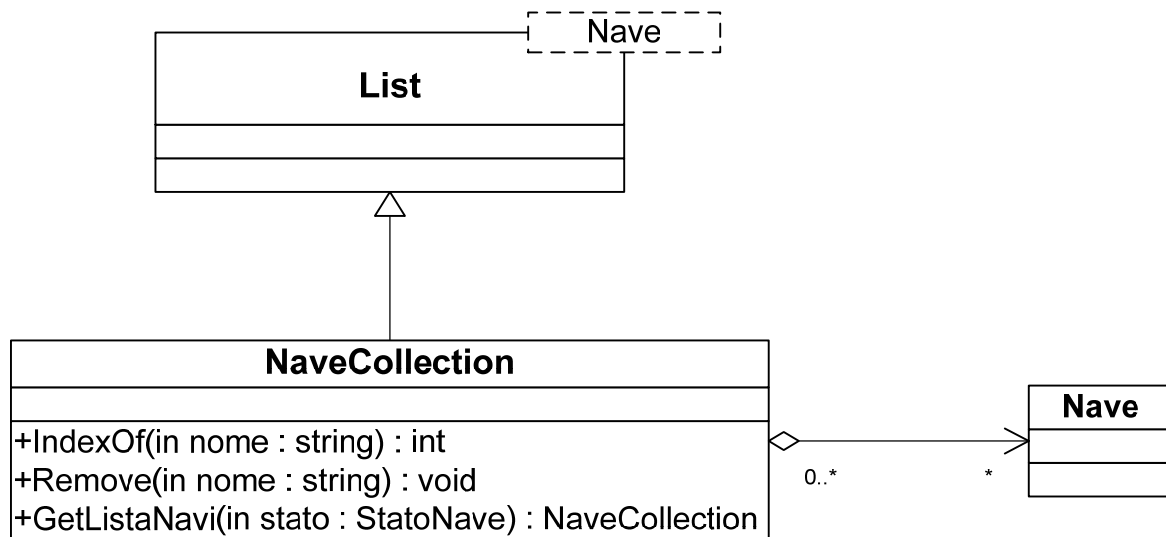


Figura 7-3 Diagramma della classe `NaveCollection`.

Sono necessari alcuni commenti.

Implementando solo tre metodi abbiamo realizzato una classe che ci consente di gestire delle navi, cercarle, rimuoverle od ottenerne una selezione.

La classe definisce il metodo `GetListaNavi` che, seguendo le best-practices del .NET, ritorna una nuova collezione contenente le navi che corrispondono al criterio specificato.

A questo punto sfruttando il fatto di far ereditare la nostra collezione dalla classe `List<>` del .NET ci potrebbe venire in mente di utilizzarne alcuni metodi molto utili, per svolgere, ad esempio l'ordinamento nella collezione (`Sort`), oppure fare ricerche (`Find`).

Ipotizziamo, quindi, di voler ordinare la nostra collezione di posizioni in senso crescente. Sapendo che la classe `List<>`, mette a disposizione il metodo `Sort`, proviamo ad applicarlo alla nostra `PosizioneCollection` e vediamo cosa succede.

Se proviamo ad eseguire il seguente frammento di codice però:

```
collPos.Sort();
```

otteniamo questo errore in fase di esecuzione

```
System.InvalidOperationException non è stata gestita
```



```
Message="Impossibile confrontare due elementi nella matrice."
```

Questo accade perché la classe `List<>`, pretende che per eseguire l'ordinamento in una collezione di oggetti, sia fornito un metodo che indichi come svolgere questo ordinamento, esige, cioè, l'implementazione di una certa interfaccia.

Per scoprire come come fare, passiamo a leggere il capitolo successivo.

8

Interfacce

1 Introduzione alle interfacce

Il termine “interfaccia” è particolarmente frequente nell’ambito della programmazione e in senso generale designa un “mezzo di comunicazione”, tra due entità. Ad esempio:

- ❑ interfaccia grafica o a caratteri: collega le attività dell’utente con le operazioni svolte dal programma: consente al primo di interagire con il secondo;
- ❑ interfaccia pubblica di una classe: collega il codice *consumer* agli oggetti della classe: consente al primo di interagire – eseguire operazioni su – con i secondi.

Nell’ambito della OOP è ovviamente il secondo significato quello che ci interessa; il concetto di interfaccia pubblica è infatti fondamentale nei programmi orientati agli oggetti, poiché molto del funzionamento di un programma si riconduce all’interazione tra codice *consumer* e le funzioni pubbliche di un qualche oggetto. Per questo motivo la progettazione di una classe parte dalla definizione di “cosa” la classe dev’essere in grado di fare – l’interfaccia pubblica – e soltanto in second’ordine di “come” potrebbe farlo – l’implementazione.

In sostanza, dal punto di vista del codice *consumer* un classe si riduce in pratica alla sua interfaccia pubblica, indipendentemente dal numero di membri privati o protetti che essa definisce o eredita da altre classi. Ciò fa sì che due oggetti appartenenti a due classi non imparentate tra loro ma che espongono la stessa interfaccia pubblica possono essere utilizzati indifferentemente dallo stesso codice *consumer*¹⁶.

Ad esempio, i seguenti due frammenti di codice svolgono le stesse operazioni su due dizionari:

```
Dictionary<string, string> dizionario = new Dictionary<string, string>();
dizionario.Add("Albert Einstein", "Fisico");
dizionario.Add("Enrico Fermi", "Fisico");
dizionario.Add("Dante Alighieri", "Poeta");
string nomeFamoso = Console.ReadLine();
string professione;
bool trovato = dizionario.TryGetValue(nomeFamoso, out professione);
if (trovato)
    Console.WriteLine("{0} è contenuto nell'elenco ed era un: {1}",
                      nomeFamoso, professione);
```

```
SortedList<string, string> dizionario = new SortedList<string, string>();
dizionario.Add("Albert Einstein", "Fisico");
dizionario.Add("Enrico Fermi", "Fisico");
```

```
dizionario.Add("Dante Alighieri", "Poeta");

string nomeFamoso = Console.ReadLine();
string professione;
bool trovato = dizionario.TryGetValue(nomeFamoso, out professione);
if (trovato)
    Console.WriteLine("{0} è contenuto nell'elenco ed era un: {1}",
        nomeFamoso, professione);
```

Il primo dizionario è rappresentato da una collezione di tipo `Dictionary<, >`, il secondo da una collezione di tipo `SortedList<, >`. L'istruzione di creazione della collezione – evidenziata in grigio in entrambi gli esempi – è l'unica che differenzia i due frammenti di codice, ed è così perché le classi `Dictionary<, >` e `SortedList<, >`, pur non essendo imparentate, espongono almeno parzialmente la stessa interfaccia pubblica, nonostante la loro implementazione sia completamente diversa.

L'interfaccia pubblica di una classe mette dunque in pratica il concetto di incapsulamento, il quale implica che il codice *consumer* non debba affatto conoscere come sia effettivamente implementata una classe per poterla usare. Questo aspetto della OOP è così importante è stato formalizzato mediante l'introduzione di uno specifico elemento del linguaggio che traduce in pratica il concetto stesso di interfaccia pubblica. Esso è appunto denominato *interfaccia* ed è designato dalla parola chiave `interface`.

D'ora in avanti il termine "interfaccia" inteso come elemento del linguaggio sarà scritto in corsivo, mentre la parola "interfaccia" intesa nel significato generale di interfaccia pubblica sarà scritta normalmente.

La comprensione e l'uso delle *interfacce* sono connessi a tre aspetti, che più avanti prenderemo in considerazione separatamente:

- ❑ **l'uso** di una *interfaccia*. Con ciò si intende l'utilizzo di una variabile *interfaccia* nello stesso modo in cui si può usare un riferimento un oggetto;
- ❑ **l'implementazione** di una *interfaccia*. Ciò si riferisce alla possibilità, da parte di una classe, di implementare una o più *interfacce*, ad esempio tra quelle fornite da .NET;
- ❑ **la definizione** di una *interfaccia*. Con ciò si intende la progettazione vera e propria della *interfaccia* e cioè la dichiarazione delle funzioni membro che essa espone.

Naturalmente, prima di considerare questi tre aspetti è necessario fornire un'introduzione formale di cosa realmente sia una *interfaccia* e di quale sia la sua relazione con le classi e i tipi struttura.

2 Che cos'è un'interfaccia

Un'*interfaccia* rientra nella categoria dei tipi e definisce un elenco di prototipi di operazioni.

Un'*interfaccia* si limita dunque a dichiarare una o più operazioni, senza definirne il corpo. Nella sua forma più semplice, la definizione di una *interfaccia* segue la sintassi:

¹⁶ Questa affermazione non dev'essere presa alla lettera, poiché in molti casi il codice *consumer* non dipende soltanto dalle operazioni che svolge sugli oggetti, ma anche dal loro tipo.

```

interface nome-interfaccia
{
    operazione1;
    operazione2;
    ...
    Operazionen;
}

```

dove le operazioni possono essere proprietà, metodi e indicizzatori, ma non costruttori¹⁷.

Nel diagramma UML *l'interfaccia* si può rappresentare in due modi, a seconda di quello che si vuole mostrare.

La prima rappresentazione è la classica nozione per le classi, con l'aggiunta dello stereotipo “interfaccia”:

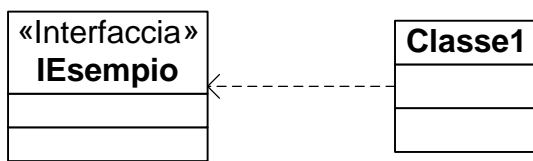


Figura 8-1 Diagramma di rappresentazione di un'interfaccia.

La seconda, invece, viene definita come notazione “palla-canestro”. Questa rappresentazione mostra meno dettagli per *l'interfaccia* ma è più conveniente da usare per mostrare relazioni tra classi.

Le classi che implementano *l'interfaccia* vengono rappresentate collegando una “palla” con il nome dell'*interfaccia* scritto sotto. Le classi che invece necessitano di elementi che implementino una determinata *interfaccia* vengono disegnate collegando un “canestro” a cui corrisponde *l'interfaccia*.



Figura 8-2 Diagramma alternativo di rappresentazione di una interfaccia.

Nelle figura, la *Classe1* implementa *l'interfaccia* *IEsempio*, mentre la *Classe2*, ha bisogno di elementi che la implementino.

Comunque in entrambe le rappresentazioni, le operazioni esposte dell'*interfaccia* vengono scritte in corsivo.

Il codice che segue definisce *l'interfaccia* *IEsempio*, la quale dichiara un metodo e una proprietà, ma non definisce il loro corpo, il quale è sostituito da un punto-e-virgola. Essa, inoltre, non fa uso di modificatori, come *public*, *private*, *static*, *virtual*, i quali non sono ammessi:

```

public interface IEsempio
{
    void Visualizza();           // solo prototipo
    string Nominativo           // solo prototipo
    {

```

¹⁷ Tra i membri di un'interfaccia, come d'altra parte di una classe, possono figurare anche gli «eventi», i quali però non sono stati trattati e dunque non saranno presi in considerazione.

```

        get;
    }
}

```

A cui corrisponde il seguente diagramma UML:

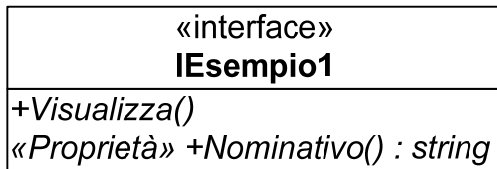


Figura 8-3 Diagramma UML della interfaccia IEsempio.

L'uso del suffisso "I" nel nome delle *interfacce* è un'utile convenzione che qualifica il tipo come una *interfaccia*, ma non rappresenta un requisito del linguaggio. E' comunque questo lo stile suggerito nelle linee-guida ed è quindi adottato nel testo.

2.1 Implementazione di una *interfaccia*

Di per sé la definizione di una *interfaccia* non produce alcuna conseguenza. Ogni *interfaccia* viene infatti definita per essere successivamente implementata da una o più classi, o tipi struttura.

L'implementazione di una *interfaccia* implica due aspetti, uno di natura formale, l'altro di natura sostanziale. Dal punto di vista formale, una classe implementa una *interfaccia*:

- ❑ dichiarandola nella "lista base";
- ❑ fornendo un'implementazione per tutte le funzioni dichiarate dall'*interfaccia*.

Segue un'ipotetica classe che implementa l'*interfaccia* IEsempio:

```

class ClasseEsempio: IEsempio // dichiarazione di IEsempio
{
    string _nome;
    string _codFisc;
    public ClasseEsempio(string nome, string codFisc)
    {
        _nome = nome;
        _codFisc = codFisc;
    }
    public string Nominativo //implementazione proprietà definita in IEsempio
    {
        get { return _nome; }
    }

    public string CodiceFiscale
    {
        get { return _codFisc; }
    }

    public void Visualizza() //implementazione metodo definito in IEsempio
    {
        Console.WriteLine("Nome: {0} Codice fiscale: {1}", _nome, _codFisc);
    }
}

```

```

    }
}

```

Di seguito è mostrato il diagramma UML che rappresenta sia la classe che l'*interfaccia*:

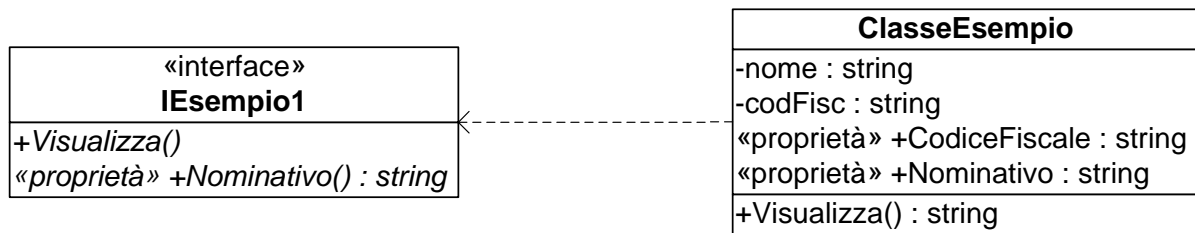


Figura 8-4 Diagramma di una classe che implementa l'interfaccia IEsempio.

Si possono fare subito alcune osservazioni:

- ❑ la classe definisce tutte le funzioni dichiarate dall'*interfaccia* IEsempio. In caso contrario il compilatore produrrebbe un errore, segnalando che una o più funzioni non sono state implementate;
- ❑ il prototipo di tali funzioni deve coincidere con quello dichiarato nell'*interfaccia*. In altre parole, una funzione dichiarata nell'*interfaccia* si intende implementata dalla classe soltanto se essa fornisce una implementazione con un prototipo identico;
- ❑ le funzioni della classe che implementano quelle dichiarate dall'*interfaccia* devono essere dichiarate pubbliche. Dunque, i modificatori `protected`, `private` e `internal` non sono ammessi. Non sono inoltre ammesse funzioni statiche;
- ❑ la classe può definire funzioni che non sono dichiarate dall'*interfaccia* e che dunque non hanno nessun legame con essa. Inoltre nulla vieta che tali funzioni sovraccarichino quelle definite dall'*interfaccia*.

Da un punto di vista sostanziale, l'implementazione di una *interfaccia* stabilisce un legame tra l'*interfaccia* stessa e la classe che la implementa. Tale legame si traduce:

nella possibilità di utilizzare una variabile interfaccia per riferirsi a oggetti della classe.

Il seguente codice lo dimostra:

```

ClasseEsempio ce = new ClasseEsempio("Donald Duck", "DNLDCCK20H32U131E");
ce.Visualizza(); // ok: niente di nuovo
string cf = ce.CodiceFiscale; // ok: niente di nuovo

```

```

IEsempio ie = ce; // ok: assegna a ie un riferimento all'oggetto creato!
ie.Visualizza(); // ok: invoca il metodo attraverso la variabile interfaccia!
cf = ie.CodiceFiscale; // errore: CodiceFiscale non è dichiarata da IEsempio!

```

Esaminiamo una per una le ultime tre istruzioni.

```
IEsempio ie = ce;
```

Questa istruzione fa sì che alla variabile *interfaccia* `ie` venga assegnato un riferimento ad un oggetto di tipo `ClasseEsempio`. Di fatto, in memoria viene prodotta la seguente situazione:

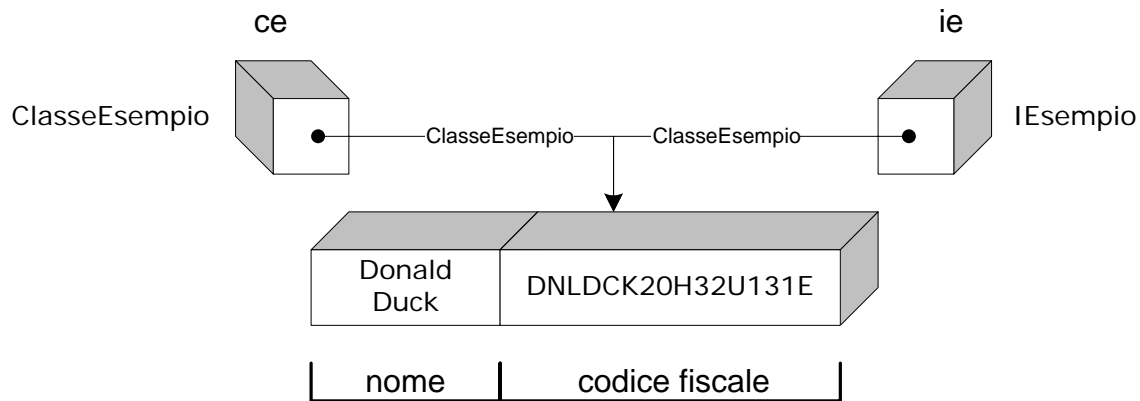


Figura 8-5 Rappresentazione in memoria dell'oggetto *ce* e della variabile interfaccia *ie*.

L'istruzione:

```
ie.Visualizza();
```

determina l'invocazione del metodo `Visualizza()` attraverso la variabile *ie*. Questa forma di invocazione non è diversa da quella prodotta attraverso la variabile *ce*. Infine, l'istruzione:

```
cf = ie.CodiceFiscale;
```

è formalmente scorretta e infatti provoca il seguente errore di compilazione:

'IEsempio' non contiene la definizione per 'CodiceFiscale'

Ciò è dovuto al fatto che attraverso la variabile *ie* non è possibile invocare i metodi definiti da *ClasseEsempio* che non siano a loro volta dichiarati da *IEsempio*.

In sostanza, dunque, mediante l'implementazione di una *interfaccia* è possibile esporre al codice *consumer* un sotto insieme – una “vista” – dell'interfaccia pubblica di una classe. Attraverso una variabile *interfaccia* è infatti possibile accedere soltanto a quelle funzioni della classe che sono definite dall'*interfaccia* in questione.

Questo stato di cose può essere schematizzato nel seguente modo:

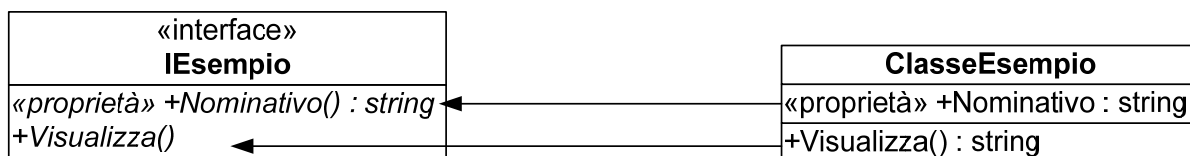


Figura 8-6 Implementazione di *IEsempio* da parte di *ClasseEsempio*.

Le funzioni definite dall'*interfaccia* vengono cioè “mappate” nella classe che la implementa. In questo modo, l'*interfaccia* – come suggerisce appunto la parola – funge da collegamento tra il codice *consumer* e un oggetto della classe, ma solo per quelle funzioni definite all'interno dell'*interfaccia* stessa.

2.2 Implementazione di più *interfacce*

Nulla impedisce ad una classe di implementare più *interfacce*, in modo che attraverso di esse possa esporre al codice *consumer* più “viste” diverse della propria interfaccia pubblica. Come esempio definiamo l'*interfaccia* *ICodiceFiscale*:

```
interface ICodiceFiscale
{
    string CodiceFiscale { get; }
}
```

e modifichiamo `ClasseEsempio` in modo che implementi anche la nuova *interfaccia*. Poiché essa definisce già la proprietà pubblica `CodiceFiscale` è sufficiente specificare il nome della nuova *interfaccia* nella lista base:

```
class ClasseEsempio: IEsempio, ICodiceFiscale
{
    ...
}
```

Segue il diagramma UML:

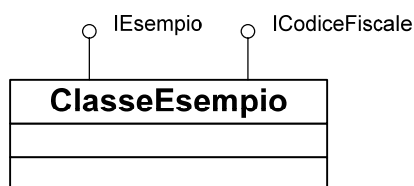


Figura 8-7 Diagramma UML di `ClasseEsempio`.

Adesso, attraverso `IEsempio` e `ICodiceFiscale` la classe espone al codice *consumer* due viste diverse della propria interfaccia pubblica. Il nuovo stato di cose può essere così schematizzato:

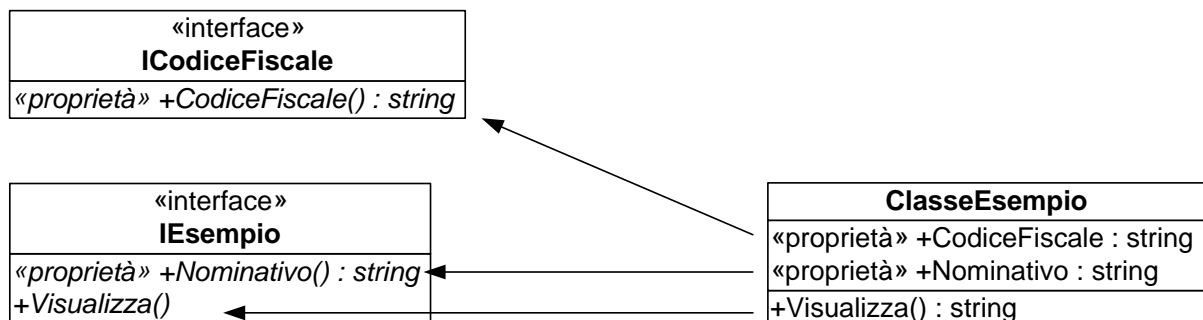


Figura 8-8 Implementazione di `IEsempio` e `ICodiceFiscale` da parte di `ClasseEsempio`.

Il seguente codice mostra come sia possibile accedere a un oggetto di tipo `ClasseEsempio` attraverso una variabile *interfaccia* di tipo `ICodiceFiscale`.

```
ICodiceFiscale icf = new ClasseEsempio("Donald Duck", "DNL DCK20H32U131E");
string cf = icf.CodiceFiscale; // ok: CodiceFiscale è accessibile attraverso icf
icf.Visualizza();             // errore: Visualizza() non è dichiarato da ICodiceFiscale
```

La prima istruzione crea un oggetto di tipo `ClasseEsempio` e lo assegna alla variabile interfaccia `icf`. Attraverso questa variabile è possibile accedere alla sola proprietà `CodiceFiscale`, poiché è l'unica definita dalla interfaccia `ICodiceFiscale`. Per questo motivo l'ultima istruzione è scorretta, poiché il metodo `Visualizza()` non è dichiarato da `ICodiceFiscale` e quindi non può essere invocato attraverso una variabile di questo tipo.

2.3 Classi che implementano la stessa *interfaccia*

La vera utilità delle *interfacce* – come sarà mostrato anche nei successivi paragrafi – consiste nella possibilità per più classi di implementare la stessa *interfaccia*. In questo modo è possibile attraverso una variabile *interfaccia* referenziare oggetti di classi diverse, che pure non hanno alcuna relazione di parentela tra loro. Ciò si traduce in pratica nella possibilità di scrivere codice *consumer* polimorfico pur senza applicare i principi dell'ereditarietà e del polimorfismo.

Per comprendere il concetto definiamo una nuova classe, *Studente*, che implementa l'*interfaccia* *IEsempio*:

```
class Studente: IEsempio
{
    string _nome;
    string _classe;
    public Studente(string nome, string classe)
    {
        _nome = nome;
        _classe = classe;
    }

    public string Nominativo //implementazione proprietà definita in IEsempio
    {
        get { return _nome; }
    }

    public string Classe
    {
        get { return _classe; }
    }

    public void Visualizza() //implementazione metodo definito in IEsempio
    {
        Console.WriteLine("Nome: {0} classe: {1}", _nome, _classe);
    }
}
```

Data questa nuova definizione è perfettamente lecito scrivere codice come il seguente:

```
IEsempio is = new Studente("Santinelli Massimo", "III A");
string nome = is.Nominativo;
is.Visualizza();

IEsempio ic = new ClasseEsempio("Donald Duck", "DNLDCK20H32U131E");
nome = ic.Nominativo;
ic.Visualizza();
```

Come si vede, attraverso variabili *interfaccia* dello stesso tipo è possibile referenziare oggetti di tipo diverso, che non condividono niente se non appunto il fatto di implementare la stessa *interfaccia*. Ciò consente di scrivere codice *consumer* che esegua procedimenti generici senza mettere in campo il polimorfismo.

Ad esempio, il seguente codice elabora un vettore di tipo `IEsempio`. Poiché gli elementi del vettore sono di tipo `Iesempio`, possono referenziare qualsiasi oggetto la cui classe di appartenenza implementi l'interfaccia in questione:

```
static void Main()
{
    IEsempio[] elenco = new IEsempio[4];
    elenco[0] = new ClasseEsempio("Donald Duck", "DNLDCCK20H32U131E");
    elenco[1] = new Studente("Santinelli Massimo ", "III A");
    elenco[2] = new Studente("Ceppodomo Michele", "IV B");
    elenco[3] = new ClasseEsempio("Mickey Mouse", "MCKMSS12H31U121E");
    ...
    for(int i = 0; i < elenco.Length; i++)
        elenco[i].Visualizza();
}
```

Per ogni iterazione del ciclo `foreach()`, il metodo `Visualizza()` effettivamente invocato è quello implementato dall'oggetto referenziato da `elenco[i]`, il quale può essere un oggetto di tipo `ClasseEsempio` oppure di tipo `Studente`. L'output prodotto dal programma è:

Nome: Donald Duck Codice fiscale: DNLDCCK20H32U131E

Nome: Santinelli Massimo classe: III A

Nome: Ceppodomo Michele classe: IV B

Nome: Mickey Mouse Codice fiscale: MCKMSS12H31U121E

3 Uso delle *interfacce*

Nei paragrafi precedenti è stata fornita un'introduzione generale alle *interfacce* restando all'interno di una cornice puramente dimostrativa. Ma è molto più facile comprenderne la reale utilità e potenza espressiva in applicazioni concrete. In questo senso, com'è già stato detto nel paragrafo introduttivo del capitolo, è possibile individuare tre ambiti distinti:

- ❑ il semplice uso di *interfacce* preesistenti e già implementate da determinate classi;
- ❑ l'implementazione di *interfacce* preesistenti in classi scritte dal programmatore;
- ❑ la progettazione di nuove *interfacce* da parte del programmatore.

Forniremo immediatamente un esempio che ricade nel primo ambito, probabilmente il più semplice e il più importante, poiché nella maggior parte dei casi l'obiettivo del programmatore è quello di scrivere codice *consumer* che utilizzi *interfacce* già scritte e implementate altrove, nella fattispecie implementate all'interno di .NET.

3.1 Uso dell'*interfaccia* “IDictionary<TKey, TValue>”

.NET espone un notevole numero di classi il cui scopo è quello di implementare una collezione dinamica di oggetti. Tra queste ve ne sono alcune che implementano un dizionario, chiamato anche “array associativo”. Brevemente, un dizionario è rappresentato da una lista di coppie “chiave-valore”. L'operazione più comune eseguita su un dizionario è quella di ottenere il valore associato a una determinata chiave, ma è in genere possibile eseguire altre operazioni, quali ottenere l'elenco delle coppie “chiave-valore”, oppure ottenere elenchi delle sole chiavi o dei soli valori, eccetera.

L'implementazione di un dizionario deve venire incontro a diversi compromessi, in relazione alla necessità di privilegiare la velocità di ricerca, piuttosto che quella di inserimento, l'efficienza nell'occupazione di memoria, l'ordinamento delle chiavi, eccetera. Per questo motivo .NET espone varie implementazioni, incontrano esigenze diverse, pur mantenendo tutte l'interfaccia pubblica che caratterizza un dizionario. Tra queste vi sono: `Dictionary<,>`, `SortedList<,>`, `SortedDictionary<,>`.

Consideriamo adesso la possibilità di realizzare un metodo d'uso generico il cui compito è quello di memorizzare su un file di testo il contenuto di un dizionario, secondo il seguente formato:

`"chiave = valore"`

Eccone una prima versione, che accetta come argomenti il nome del file e un riferimento a un dizionario di tipo `Dictionary<,>`:

```
static void ScriviDizionario(string nomeFile,
                             Dictionary<string, string> dizionario)
{
    StreamWriter sw = new StreamWriter(nomeFile);
    foreach (KeyValuePair<string, string> kv in dizionario)
        sw.WriteLine("{0} = {1}", kv.Key, kv.Value);
    sw.Close();
}
```

`KeyValuePair<,>` è un tipo che definisce i riferimenti, `Key` e `Value`, ad una coppia “chiave-valore”. Per ogni iterazione del ciclo `foreach()` la variabile `kv` assume il contenuto dell'iesima coppia “chiave-valore”, la quale viene successivamente scritta su file secondo il formato deciso in precedenza.

Ecco un esempio di codice *consumer* che usa il metodo:

```
static void Main()
{
    Dictionary<string, string> di = new Dictionary<string, string>();
    di.Add("Marco Materazzi", "Squadra: Inter; Gol: 14; P.G: 18");
    di.Add("Filippo Inzaghi", "Squadra: Milan; Gol: 11; P.G: 21");
    di.Add("Alex Del Piero", "Squadra: Juventus; Gol: 13; P.G: 19");
    ScriviDizionario("dizionario.txt", di);
}
```

Il tutto funziona, ma soltanto se l'argomento passato al metodo `ScriviDizionario()` è di tipo `Dictionary`. D'altra parte, il problema di partenza era quello di realizzare un metodo generico in grado di scrivere su file qualsiasi tipo di dizionario, a prescindere dall'implementazione specifica.

Sappiamo già che è possibile realizzare dei procedimenti generici applicando i meccanismi dell'ereditarietà e del polimorfismo, ma in questo caso essi non sono utilizzabili, poiché i tre tipi di dizionari citati a inizio paragrafo non derivano da una classe base comune. Non esiste dunque una classe astratta che funga da tipo base e che dunque possa essere utilizzata nel metodo `ScriviDizionario()`. Tutte e tre le classi, d'altra parte, implementano l'*interfaccia* `IDictionary`, la quale dichiara le funzioni che caratterizzano l'interfaccia pubblica di un dizionario.

Segue l'implementazione generica del metodo che fa uso di tale *interfaccia*:

```
static void ScriviDizionario(string nomeFile,
                             IDictionary<string, string> dizionario)
{
    StreamWriter sw = new StreamWriter(nomeFile);
    foreach (KeyValuePair<string, string> de in dizionario)
```

```

        sw.WriteLine("{0} = {1}", de.Key, de.Value);
    sw.Close();
}

```

Rispetto al precedente esempio è cambiato soltanto il tipo del parametro dizionario, che è adesso `IDictionary<string, string>`. Utilizzando una variabile *interfaccia* di questo tipo è infatti possibile accedere alla parte di interfaccia pubblica condivisa da tutte le classi che implementano un dizionario. Dopo questa modifica è possibile invocare il metodo per qualsiasi tipo di dizionario:

```

static void Main()
{
    Dictionary<string, string> di = new Dictionary<string, string>();
    di.Add("Marco Materazzi", "Squadra: Inter; Gol: 14; P.G: 18");
    di.Add("Filippo Inzaghi", "Squadra: Milan; Gol: 11; P.G: 21");
    di.Add("Alex Del Piero", "Squadra: Juventus; Gol: 13; P.G: 19");
    ScriviDizionario("dizionario.txt", di);

    SortedList<string, string> sl = new SortedList<string, string>();
    sl.Add("Francesco Totti", "Squadra: Roma; Gol: 10; P.G: 12");
    sl.Add("Alex Del Piero", "Squadra: Juventus; Gol: 13; P.G: 19");
    ScriviDizionario("dizionario2.txt", sl);

    SortedDictionary<string, string> sd = new SortedDictionary<string, string>();
    sd.Add("Simone Inzaghi", "Squadra: Lazio; Gol: 10; P.G: 16");
    sd.Add("Roberto Carlos", "Squadra: Real Madrid; Gol: 9; P.G: 14");
    ScriviDizionario("dizionario3.txt", sd);
}

```

Dal punto di vista del codice *consumer* non è rilevante quale sia il tipo effettivo dell'oggetto referenziato da `dizionario`, purché questo implementi l'interfaccia `IDizionario`.

3.2 Implementare l'interfaccia "IComparable<T>" per l'ordinamento di collezioni

.NET espone un notevole numero di classi il cui scopo è quello di implementare una collezione dinamica di oggetti. Molte di queste collezioni, possono essere ordinate mediante il metodo `Sort()`, che utilizza un veloce algoritmo di QuickSort per ordinare le istanze degli oggetti presenti nella collezione.

Abbiamo visto, però, che se noi invochiamo questo metodo nella nostra classe `NaveCollection` (lo stesso discorso vale per la classe `PosizioneCollection`), otteniamo un errore in fase di esecuzione.

Questo perché l'algoritmo del QuickSort, quando va ed effettuare i confronti, ha bisogno di sapere come stabilire l'ordine di due oggetti (chi viene prima e chi viene dopo insomma) e ci si potrebbe chiedere come faccia la collezione ad applicare il criterio appropriato nel confrontare due elementi tra loro. All'interno del .NET le collezioni invocano il metodo `CompareTo()`, definito dall'interfaccia `IComparable`, per ogni coppia di elementi. Ecco la struttura dell'interfaccia:

Quindi, per fare sì che gli elementi possano essere ordinati questi devono appartenere a un tipo di dato che implementi tale *interfaccia*; in caso contrario, il tentativo di ordinamento scatenerà l'eccezione `InvalidOperationException`.

In sostanza, se è nostra intenzione definire un tipo di dato i cui oggetti possano essere ordinati è necessario che questo implementi l'interfaccia `IComparable<T>`.

Quella che segue è la rappresentazione ed il codice C# necessario all'implementazione di `IComparable<T>` nella classe `Nave`.

```
public class Nave : IComparable<Nave>
{
    . . .

    public int CompareTo(Nave naveDaConfrontare)
    {
        return String.Compare(this._nome, naveDaConfrontare._nome);
    }

    . . .
}
```

Segue il diagramma UML che mostra l'implementazione dell'*interfaccia*:

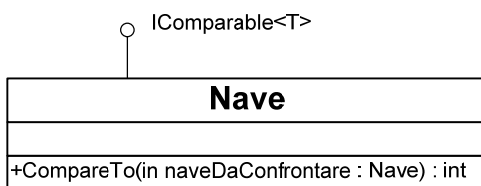


Figura 8-9 Diagramma UML dell'implementazione di `IComparable<Nave>`.

Che ci permetterà di invocare il metodo `Sort` come mostrato dal seguente codice consumer

```
Nave primaNave = new Nave("Splendida", 1000);
Nave secondaNave = new Nave("Magnifica", 2000);

NaveCollection flotta = new NaveCollection();
flotta.Add(primaNave);
flotta.Add(secondaNave);
```

```
flotta.Sort();
foreach (Nave n in flotta)
{
    Console.WriteLine(n);
}
```

con il seguente risultato:

Magnifica | 2000

Splendida | 1000

3.3 Implementare l'*interfaccia* “`IComparer<T>`” per l'ordinamento di collezioni

Abbiamo appena visto come implementando una determinata interfaccia sia possibile utilizzare il metodo `Sort`. Spesso, però c'è l'esigenza di ordinare una collezione non solamente in base ad un certo criterio, ma in base a più criteri.

Ipotizziamo quindi di voler ordinare la flotta non solo in base al valore dell'attributo `nome`, ma, in alternativa, in base all'attributo `miglia percorse`.

Per fare questo esistono due possibilità e la prima prevede di passare come argomento a `Sort()` un metodo che definisca il criterio di ordinamento. Quest'ultimo, ricevuti in ingresso due oggetti di

tipo `Nave`, dovrà restituire -1, 0 oppure 1 a seconda che il primo sia inferiore, uguale o superiore al secondo.

La seconda alternativa è quella di implementare una classe “helper”, così chiamata poiché “aiuta” la classe `Nave` nello svolgimento della funzione di ordinamento.

La classe helper dovrà implementare l'*interfaccia* `IComparer<T>` che definisce un solo metodo, di nome `Compare()`, il cui funzionamento è identico al metodo `SortByMiglia` visto in precedenza.

Dovremo quindi scrivere questo codice:

```
public class SortNaveByMigliaHelper : IComparer<Nave>
{
    public int Compare(Nave n1, Nave n2)
    {
        int val = -1;
        if (n1.Miglia > n2.Miglia)
            val = 1;
        if (n1.Miglia == n2.Miglia)
            val = 0;
        return val;
    }
}
```

Segue il diagramma UML della classe:

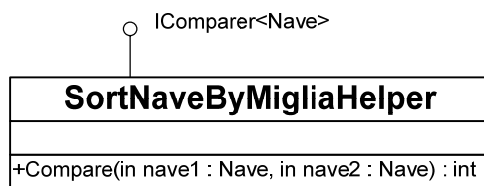


Figura 8-10 La classe helper ed il diagramma UML corrispondente.

Ecco come usare la classe helper per ordinate la flotta:

```
flotta.Sort(new SortNaveByMigliaHelper());
foreach (Nave n in flotta)
{
    Console.WriteLine(n);
}
```

Sarà compito del metodo `Sort()` utilizzare l'*interfaccia* `IComparer<T>` implementata dalla classe helper.

3.4 Implementare l'*interfaccia* “`IEquatable<T>`” per confrontare due elementi

Perché gli oggetti di tipo `Nave` siano convenientemente utilizzabili come elementi di una collezione è necessario implementare ancora un'*interfaccia*: `IEquatable<T>`. Il motivo risiede nel problema della ricerca di un elemento mediante il metodo `IndexOf()`.

Consideriamo il seguente codice, che effettua una ricerca nella flotta mediante `IndexOf()`:

```
Nave unaNave = new Nave("Magica", 3000);
flotta.Add(unaNave);
Nave unaNaveUguale = new Nave("Magica", 3000);
int pos = flotta.IndexOf(unaNaveUguale);
```

```
if (pos != -1)
    Console.WriteLine("Trovata la nave in posizione {0}", pos);
```

Dato che l'oggetto `unaNaveUguale` equivale al primo e unico elemento della flotta, ci si attende che il valore di `pos` sia diverso da -1. Ma non è così: infatti, due oggetti di tipo riferimento vengono sempre giudicati come elementi diversi, a prescindere dal loro contenuto. Questo a meno che la classe d'appartenenza degli oggetti non implementi l'interfaccia `IEquatable<T>`, la quale consente di stabilire il criterio di uguaglianza da applicare nel confrontare due oggetti.

Implementare `IEquatable<T>` significa definire il metodo `Equals()`, similmente a quanto fatto in precedenza durante la costruzione della classe `Nave`:

```
public class Nave : IEquatable<Nave>
{
    ...
    bool IEquatable<Nave>.Equals(Nave n)
    {
        return (_nome == n._nome && _miglia == n._miglia);
    }
}
```

Segue il diagramma UML che mostra l'implementazione dell'*interfaccia*:

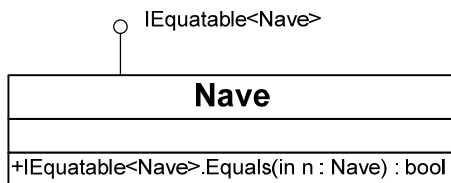


Figura 8-11 Diagramma UML dell'implementazione di `IEquatable<Nave>`.

Così facendo il codice di esempio produce questo risultato:

```
Trovata la nave in posizione 2
```

4 Definizione di *interfacce*

L'implementazione di *interfacce* consente agli oggetti di una classe di essere utilizzati da codice *consumer* che ne ignora l'effettivo tipo di appartenenza.

Vediamo adesso l'utilità di definire interfacce estendendo il nostro esempio della flotta navale.

Testo del problema

La nota compagnia di navigazione **Shipping & Delivery Fleet** vuole espandersi dal mercato navale a quello terrestre ed aereo e pertanto incrementerà la flotta da essa utilizzata per la spedizione, la consegna di oggetti ed il noleggio a clienti sia per il trasporto che per il semplice spostamento.

I mezzi della SDF saranno composti dai seguenti tipi (Aereo, Nave, Furgone) che condivideranno l'attributo *Nome* oltre ad avere delle caratteristiche peculiari che sono:

Aereo: *Autonomia*, Nave: *Pescaggio*, Furgone: *posti*

Aereo e Furgone sono mezzi noleggiabili per un certo numero di passeggeri, mentre Nave e Furgone sono caricabili con una certa quantità di materiale.

Analizzando il problema, si notano sicuramente delle relazioni tra i mezzi da rappresentare in un ipotetico diagramma delle classi.

La presenza di attributi comuni e di attributi specifici ci suggerisce di utilizzare il concetto di generalizzazione mediante l'implementazione dell'ereditarietà tra i mezzi.

Un possibile diagramma UML potrebbe essere il seguente:¹⁸

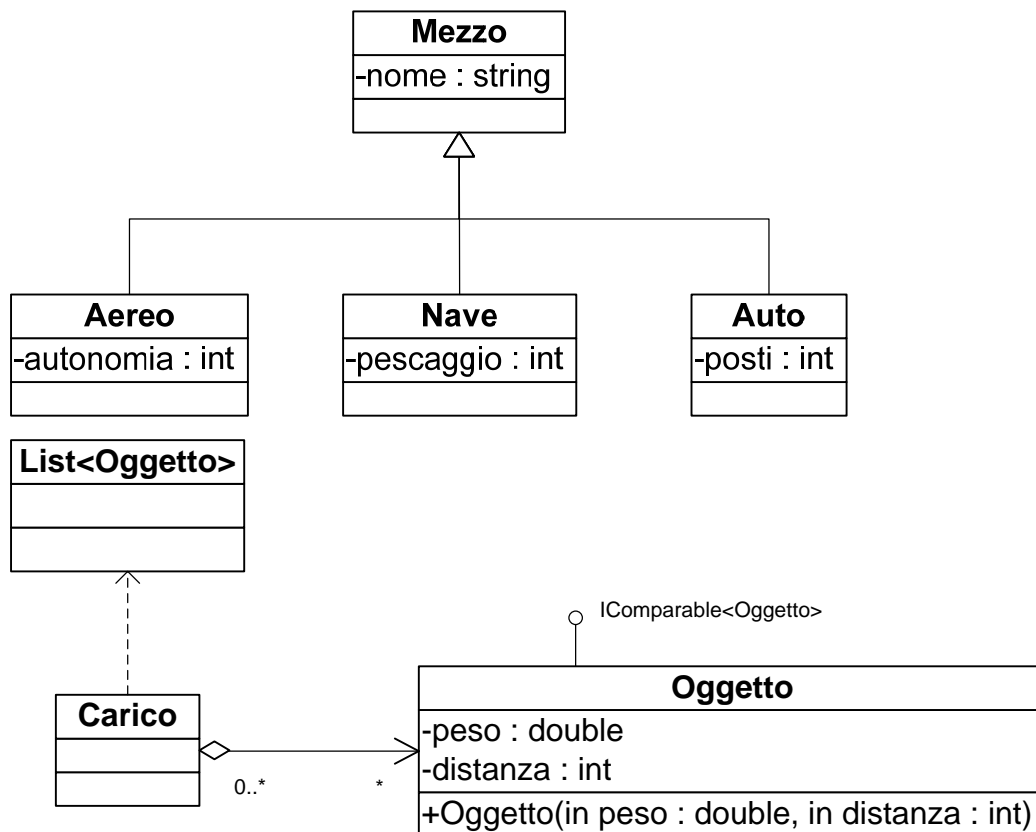


Figura 8-12 Diagramma UML delle classi del dominio del problema

4.1 Il problema dell'ereditarietà multipla tra classi

Fin qua tutto bene, ma proseguendo nell'analisi ci imbattiamo nei concetti di noleggiabilità e caricabilità dei mezzi.

Infatti un mezzo noleggiabile deve avere la possibilità di *Restituire* o *Noleggiare* un veicolo ad un certo numero di *Passeggeri*, oltre a calcolarne il costo di noleggio, mentre un mezzo caricabile deve essere *Caricato* e *Scaricato* di una certa *Quantità* di merce, che presuppone quindi un costo di trasporto.

Se tutti i mezzi fossero noleggiabili e caricabili, potremmo risolvere il problema aggiungendo alla classe base **Mezzo** gli attributi e le operazioni necessarie a svolgere i compiti di noleggio e di carico.

Il diagramma assumerebbe la seguente struttura.

¹⁸ Tralasciamo volutamente l'indicazione di attributi ed operazioni non significativi per la esposizione del problema.

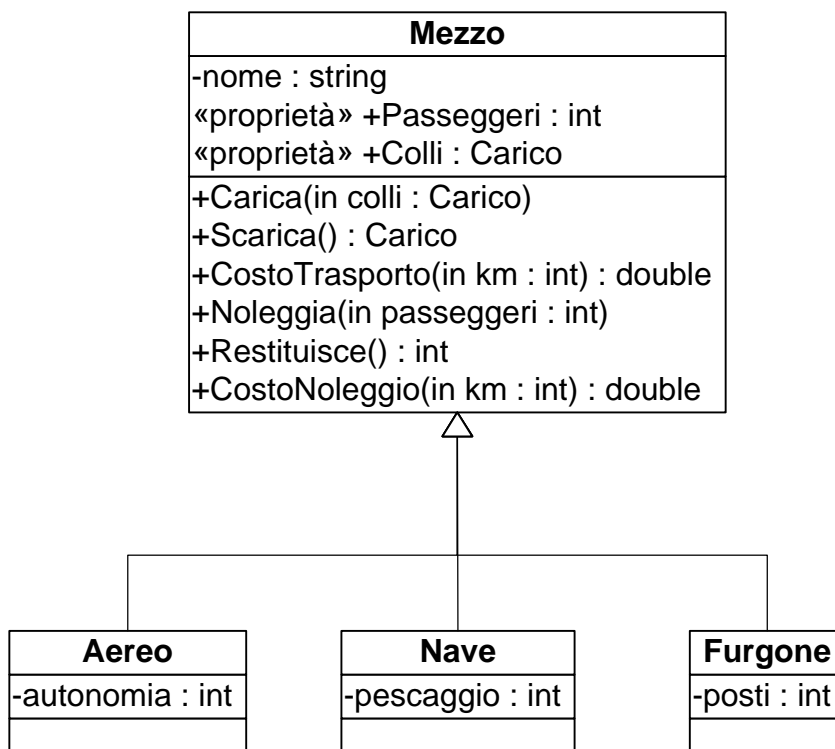


Figura 8-13 Diagramma UML delle classi del dominio del problema.

Il problema, però, è che la nave è caricabile, l'aereo è noleggiabile mentre il furgone è entrambe le cose.

Usare questa gerarchia di classi, fa sì che le classi che specializzano il `Mezzo`, ereditano attributi, proprietà e metodi che poi non potranno essere utilizzati e questo va contro il paradigma dell'ereditarietà che impone alla classe ereditata di estendere quella base, non di limitarne le funzionalità.

Come alternativa, si potrebbe pensare di progettare due classi di mezzi, la prima che rappresenta un `MezzoNoleggiabile`, ed un'altra che rappresenta un `MezzoCaricabile` e disegnare il seguente schema.

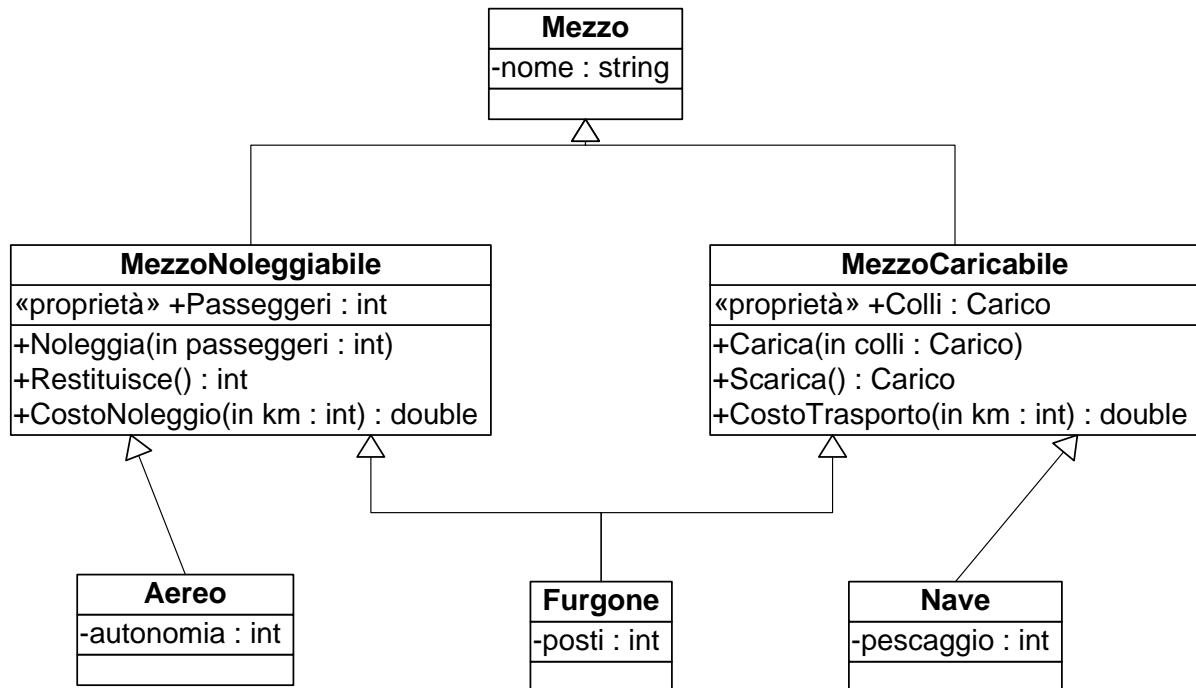


Figura 8-14 Diagramma UML delle classi del dominio del problema

Così, facendo, però, ci troveremo di fronte ad un altro problema, questa volta insormontabile in fase di implementazione, poiché il .NET non permette ad una classe di ereditare da due classi base, non potremmo mai scrivere il codice C# della classe *Furgone*.

4.2 L'ereditarietà multipla tra interfacce

A questo punto ci vengono in aiuto le *interfacce*, in quanto mentre non possiamo ereditare da due classi, possiamo invece implementare due *interfacce*.

Non ci resta quindi che trasformare le due classi *MezzoNoleggiabile* e *MezzoCaricabile* in due *interfacce*, chiamandole *INoleggiabile* ed *ICaricabile*.

A questo punto i mezzi non dovranno più ereditare dalle classi *MezzoXXX* ma dovranno semplicemente implementare le *interfacce* necessarie.

Lo schema diventerà il seguente.

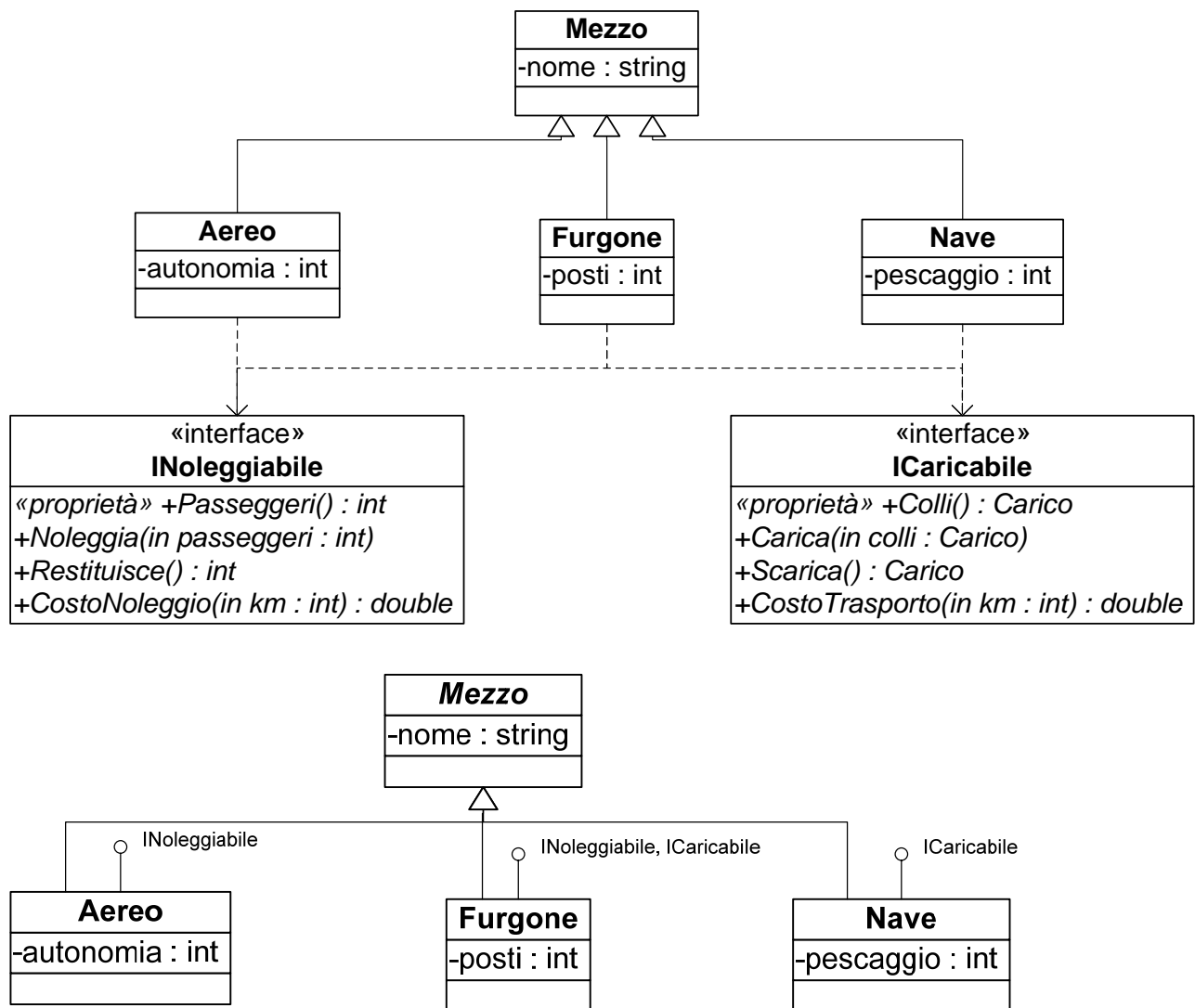


Figura 8-15 Diagramma UML delle classi del dominio del problema.

4.3 Definizione delle interfacce INoleggiabile e ICaricabile

Di seguito sono definite le due *interfacce*:

```

public interface INoleggiabile
{
    int Passeggeri { get; }
    int Restituisci();
    void Noleggia(int passeggeri);
    double CostoNoleggio(int km);
}
  
```

```

public interface ICaricabile
{
    Carico Colli { get; }
    Carico Scarica();
    void Carica(Carico colli);
    double CostoTrasporto(int km);
}
  
```

```
}
```

4.4 Implementazione della gerarchia di classi

Ovviamente si implementa per prima la classe che sta alla base della gerarchia:

```
public class Mezzo
{
    protected string _nome;
    public string Nome
    {
        get { return _nome; }
    }

    public Mezzo(string nome)
    {
        _nome = nome;
    }

    public override string ToString()
    {
        return _nome;
    }
}
```

Per quanto riguarda la classe `Aereo`, essa deve derivare da `Mezzo` ed implementare l'*interfaccia* `INoleggiabile`.

```
public class Aereo : Mezzo, INoleggiabile
{
    . . .
    double _autonomia;
    public double Autonomia
    {
        get { return _autonomia; }
    }

    public Aereo(string nome, double autonomia)
        : base(nome)
    {
        _autonomia = autonomia;
    }

    public override string ToString()
    {
        return base.ToString() + " | " + _autonomia.ToString();
    }
}
```

```
// Implementazione di INoleggiabile
int _passeggeri;
int INoleggiabile.Passeggeri
{
    get { return _passeggeri; }
}

int INoleggiabile.Restituisci()
{
    int tmp = _passeggeri;
    _passeggeri = 0;
    return tmp;
}

void INoleggiabile.Noleggia(int passeggeri)
{
    _passeggeri = passeggeri; ;
}

double INoleggiabile.CostoNoleggio(int km)
{
    return _passeggeri * km * 100.0;
}
}
```

Possiamo notare come l'implementazione interna sia lasciata alla volontà del programmatore, come in questo caso in cui la proprietà `Passeggeri`, viene mappata ad un attributo chiamato `_passeggeri` mentre il calcolo del costo del noleggio, moltiplica la distanza per un fattore 1000.

La classe `Furgone`, invece, deve specializzare la classe `Mezzo` ed implementare sia la *interfaccia* `INoleggiabile` che quella `ICaricabile`.

```
public class Furgone : Mezzo, INoleggiabile , ICaricabile
{
    protected double _posti;
    public double Posti
    {
        get { return _posti; }
    }

    public Furgone(string nome, double posti) : base(nome)
    {
        _posti = posti;
    }

    public override string ToString()
    {
        return base.ToString() + " | " + _posti.ToString();
    }
}
```

```
// Implementazione di INoleggiabile
int _passeggeri;
public int Passeggeri
{
    get { return _passeggeri; }
}

public int Restituisci()
{
    int tmp = _passeggeri;
    _passeggeri = 0;
    return tmp;
}

public void Noleggia(int passeggeri)
{
    _passeggeri = passeggeri; ;
}

public double CostoNoleggio(int km)
{
    return _passeggeri * km * 10.0;
}
```

```
// Implementazione di ICaricabile
Carico _colli;
public Carico Colli
{
    get { return _colli; }
}

public Carico Scarica()
{
    Carico tmp = _colli;
    _colli = new Carico();
    return tmp;
}

public void Carica(Carico colli)
{
    _colli = colli;
}

public double CostoTrasporto(int km)
{
    double totale = 0;
    foreach (Oggetto pacco in _colli)
    {
```

```

        totale = totale + pacco.Distanza * pacco.Peso;
    }
    return totale * 50.0;
}
}

```

La classe Nave, infine, implementa solo ICaricabile:

```

public class Nave : Mezzo, ICaricabile
{
    double _pescaggio;
    public double Pescaggio
    {
        get { return _pescaggio; }
    }

    public Nave(string nome, double pescaggio) : base(nome)
    {
        _pescaggio = pescaggio;
    }

    public override string ToString()
    {
        return base.ToString() + " | " + _pescaggio.ToString();
    }
}

```

```

// Implementazione di ICaricabile
Carico _colli;
public Carico Colli
{
    get { return _colli; }
}

public Carico Scarica()
{
    Carico tmp = _colli;
    _colli = new Carico();
    return tmp;
}

public void Carica(Carico colli)
{
    _colli = colli;
}

public double CostoTrasporto(int km)
{
    double totale = 0;
    foreach (Oggetto pacco in _colli)

```

```
        {
            totale = totale + pacco.Distanza * pacco.Peso / 2;
        }
        return totale * 200.0;
    }
}
```

A questo punto possiamo scrivere il seguente codice consumer

```
Oggetto pacco1 = new Oggetto(100, 10);
Oggetto pacco2 = new Oggetto(200, 20);
Oggetto pacco3 = new Oggetto(300, 30);
```

```
Carico carico1 = new Carico();
carico1.Add(pacco1);
carico1.Add(pacco2);
```

```
Carico carico2 = new Carico();
carico2.Add(pacco3);
```

```
nave.Carica(carico1);
furgone.Carica(carico2);
aereo.Carica(carico2); // errore
aereo.Noleggia(10);
furgone.Noleggia(2);
nave.Noleggia(3);      // errore
```

Le due righe di codice evidenziate, inserite a titolo dimostrativo, provocano un errore in fase di **compilazione**, in quanto utilizzano metodi che non sono implementati da quel particolare oggetto.

4.5 Le interfacce come denominatore comune

Le interfacce `INoleggiabile` e `ICaricabile` possono essere usate come denominatore comune nel caso volessimo creare una collezione di mezzi caricabili o noleggiabili. Infatti, anche se non possono essere istanziate direttamente, le interfacce possono essere utilizzate per creare delle collezioni, come in questo esempio:

```
List<INoleggiabile> mezziNoleggiabili = new List<INoleggiabile>();
mezziNoleggiabili.Add(furgone);
mezziNoleggiabili.Add(aereo);
Console.WriteLine("Lista dei mezzi noleggiabili");
foreach (INoleggiabile mezzoNol in mezziNoleggiabili)
{
    Console.WriteLine("{0} pass:{1}", mezzoNol.ToString(),
                      mezzoNol.Passeggeri.ToString());
}
```

Ovviamente, se ad esempio provassimo ad aggiungere una nave alla collezione:

```
mezziNoleggiabili.Add(nave); //errore
```

otterremo un errore di compilazione.

La stessa cosa si può fare con i mezzi caricabili:

```
List<ICaricabile> mezziCaricabili = new List<ICaricabile>();
mezziCaricabili.Add(nave);
mezziCaricabili.Add(furgone);
Console.WriteLine("Lista dei mezzi noleggiabili");
double tot = 0;
foreach (ICaricabile mezzoCar in mezziCaricabili)
{
    Console.WriteLine(mezzoCar.ToString());
    tot = tot + mezzoCar.CostoTrasporto(10);
}
Console.WriteLine("Totale del carico:{0}", tot);
```

Il vantaggio delle interfacce diventa ancora più evidente quando introduciamo una classe *Magazzino*, che nulla a che fare con furgoni ed aerei, ma deve solamente caricare/scaricare merce da un mezzo “*caricabile*”.

Decidiamo, quindi che anche questa classe debba implementare l’interfaccia *ICaricabile*.

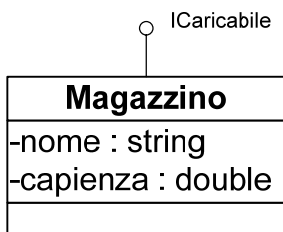


Figura 8-16 Diagramma UML della classe *Magazzino*.

Ecco l’implementazione della classe:

```
public class Magazzino: ICaricabile
{
    string _nome;
    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    double _capienza;
    public double Capienza
    {
        get { return _capienza; }
        set { _capienza = value; }
    }

    public Magazzino(string nome, double capienza)
    {
        _nome = nome;
        _capienza = capienza;
    }
}
```

```
public override string ToString()
{
    return String.Format("{0} - {1}", _nome, _capienza);
}
```

```
...
```

```
// Implementazione di ICaricabile
Carico _pacchi;
public Carico Colli
{
    get { return _pacchi; }
}

public Carico Scarica()
{
    Carico tmp = _pacchi;
    _pacchi = new Carico();
    _capienza = 0;
    return tmp;
}

public void Carica(Carico colli)
{
    double pesoCarico = 0;
    foreach (Oggetto ogg in colli)
    {
        pesoCarico = pesoCarico + ogg.Peso;
    }
    if (_capienza - pesoCarico >= 0)
    {
        if (_pacchi == null)
            _pacchi = new Carico();
        _pacchi.AddRange(colli);
        _capienza = _capienza - pesoCarico;
    }
    else
        throw new Exception("Il magazzino è troppo pieno");
}

public double CostoTrasporto(int km)
{
    return 0;
}
}
```

Possiamo notare come questa classe implementi in maniera differente dalle altre i metodi `Carica` e `Scarica` esposti dalla *interfaccia*,

L'implementazione dell'interfaccia sia da parte della classe `Magazzino` che da parte della classe `Nave`, rende le due classi **tipo-compatibili**.

Questo vuol dire che oggetti completamente differenti tra di loro possono essere trattati alla stessa maniera.

Il seguente codice consumer "carica" un magazzino con i colli di una nave e, il fatto che i due tipi siano diventati compatibili, consente di calcolare il valore della giacenza sia del magazzino che della nave utilizzando identiche modalità

```
Magazzino lax = new Magazzino("Los Angeles Exchange", 600);
lax.Carica(nave.Colli);
```

```
List<ICaricabile> giacenza = new List<ICaricabile>();
giacenza.Add(furgone);
giacenza.Add(lax);
```

```
double totGiacenza = 0;
for (int i = 0; i < mezzi.Length; i++)
{
    ICaricabile pacco = giacenza[i];
    Console.WriteLine("il Carico {0} ha un costo di {1}", pacco,
        pacco.CostoTrasporto(10));
    totGiacenza = totGiacenza + pacco.CostoTrasporto(20);
}
Console.WriteLine("Totale del carico:{0}", totGiacenza);
```

4.6 Derivazione di classe e implementazione di *interfaccia* a confronto

Nel precedente paragrafo è stato mostrato l'uso delle *interfacce* come sostituto dell'ereditarietà e ciò significa che i due approcci sono sostanzialmente equivalenti? Assolutamente no!

L'implementazione della stessa *interfaccia* da parte di due classi implica che esse definiscono una o più funzioni membro con lo stesso prototipo, dichiarate nell'*interfaccia* stessa. Ciò non produce alcuna relazione tra le due classi in questione, che restano completamente indipendenti una dall'altra. Una *interfaccia*, in sostanza, è un "contratto" che riguarda la classe e il codice *consumer*, e che indica in che modo possono essere utilizzati gli oggetti della classe suddetta; in questo senso, le eventuali relazioni che tale classe ha (o non ha) con le altre sono del tutto irrilevanti.

L'ereditarietà, diversamente, stabilisce una relazione tra due classi, e indirettamente tra più classi di una gerarchia. Tale relazione produce delle conseguenze sulla struttura delle classi, poiché una classe che deriva da un'altra eredita da questa tutti i membri, campi o funzioni che siano.

L'ereditarietà rende dunque possibile il riutilizzo del codice comune (sia dichiarativo che esecutivo), laddove l'implementazione di una *interfaccia* non lo consente e questo aspetto è di fondamentale importanza.

5 *Interfacce, ereditarietà e polimorfismo*

Esistono alcuni aspetti che connettono le *interfacce* all'ereditarietà e al polimorfismo e che è necessario conoscere per sfruttare al meglio questo strumento offerto dal linguaggio. La loro comprensione implica la risposta alle seguenti domande:

- ❑ è possibile applicare l’ereditarietà anche alle *interfacce*?
- ❑ una classe D che deriva da una classe B, eredita anche le *interfacce* da questa implementate?
- ❑ è ammesso definire virtuali le funzioni membro definite nell’*interfaccia* implementata da una classe? E conseguentemente: è possibile ridefinire tali funzioni nelle classi derivate e invocarle attraverso una variabile *interfaccia*?
- ❑ è possibile conoscere se la classe di appartenenza di un oggetto implementa una determinata *interfaccia*? O in generale: è possibile usare gli operatori di cast con le *interfacce*?

5.1 Ereditarietà applicata alle *interfacce*

Analogamente alle classi, è possibile realizzare una *interfaccia* che deriva da un’altra e che dunque eredita le funzioni membro da questa dichiarate. Ciò, esattamente come per le classi, avviene specificando l’*interfaccia* base (o le *interfacce* base) nella “lista base” dell’*interfaccia* derivata.

Ad esempio, il seguente codice:

```
interface IBase
{
    string Nominativo { get ; }
    void Visualizza();
}

interface IDerivata: IBase
{
    string CodiceFiscale { get; }
}
```

definisce due *interfacce*, delle quali la seconda deriva dalla prima e cioè eredita i prototipi dichiarati in IBase.

Nelle *interfacce*, la derivazione si traduce in una aggregazione dei prototipi dichiarati dalle *interfacce* base con quelli definiti nella *interfaccia* derivata. Nell’esempio precedente, il risultato è che IDerivata definisce tre funzioni.

Come abbiamo detto è possibile specificare più *interfacce* base. Ad esempio:

```
interface IBase
{
    string Nominativo { get ; }
    void Visualizza();
}

interface IEsempio
{
    double CalcolaStipendio();
    void Visualizza();
}

interface IDerivata: IBase, IEsempio
{
    string CodiceFiscale { get; }
}
```

In questo caso, `IDerivata` aggiunge le proprie funzioni a quelle di `IBase` e `IEsempio`, definendo in totale 4 funzioni. Da notare che sia `IEsempio` che `IBase` definiscono il metodo `Visualizza()`; il fatto che `IDerivata` erediti la stessa versione del metodo da due *interfacce* diverse non produce nessuna particolare conseguenza: essa definisce uno e un solo metodo `Visualizza()`.

La situazione cambia se una funzione viene definita con una diversa lista di parametri:

```
interface IBase
{
    string Nominativo { get ; }
    void Visualizza();           // versione senza parametri
}

interface IEsempio
{
    double CalcolaStipendio();
    void Visualizza(string msg); // versione con un parametro
}

interface IDerivata: IBase, IEsempio
{
    string CodiceFiscale { get; }
}
```

In questo caso `Visualizza()` viene sovraccaricato e `IDerivata` eredita entrambe le versioni, definendo in totale 5 funzioni membro.

Per inciso esiste l'ulteriore possibilità che due interfacce definiscano due funzioni che abbiano ugual nome e lista di parametri, ma diverso tipo di ritorno. Il linguaggio ammette questa possibilità e fornisce uno strumento, "l'implementazione esplicita", che consente a una classe di implementare entrambe le versioni della funzione, in apparente contrasto con la regola dell'*overloading* che proibisce la definizione di due funzioni che differiscono per il solo tipo di ritorno. Dato il carattere introduttivo del testo, questo argomento, come altri connessi alla derivazione di *interfacce*, non sarà trattato.

5.2 Implementazione di interfacce derivate

Una classe che implementa una *interfaccia* deve fornire un'implementazione di tutte le funzioni che questa definisce. Nel caso delle *interfacce* derivate ciò vale anche per le funzioni che queste ereditano dalle *interfacce* base. Dunque, una classe che implementasse l'*interfaccia* `IDerivata` dovrebbe definire perlomeno quattro funzioni membro. Ad esempio:

```
class ClasseEsempio: IDerivata
{
    public string Nominativo { ... }
    public string CodiceFiscale { ... }
    public double CalcolaStipendio() { ... }
    public void Visualizza() { ... }
    // altri campi e funzioni della classe
}
```

5.3 Interfacce ereditate dalla classe base

Questo paragrafo e quelli successivi sono strettamente collegati. Cominciamo con il dire che una classe derivata eredita dalla classe base non solo i membri ma anche le eventuali interfacce che questa implementa. Ad esempio:

```
interface IEsempio
{
    void Visualizza();
}

class ClasseBase: IEsempio
{
    public void Visualizza()
    {
        Console.WriteLine("Classe base");
    }
}

class ClasseDerivata: ClasseBase
{
    new public void Visualizza()
    {
        Console.WriteLine("Classe derivata");
    }
}
```

ClasseDerivata eredita l'interfaccia implementata da ClasseBase, tanto che è perfettamente lecito scrivere il seguente codice:

```
IEsempio ie = new ClasseDerivata();
ie.Visualizza();
```

nel quale un oggetto di tipo ClasseDerivata è referenziato da una variabile *interfaccia* IEsempio.

Nel caso specifico, però, questo non produce il risultato atteso. Infatti, in output viene prodotto “Classe base” e non “Classe derivata” come ci si aspetterebbe. Il risultato ottenuto non è diverso da quello prodotto dal seguente codice, perfettamente analogo:

```
ClasseBase cb = new ClasseDerivata();
cb.Visualizza();
```

In entrambi i casi, essendo Visualizza() definito come non virtuale, il metodo effettivamente invocato dipende dal tipo statico (tipo della variabile) e non dal tipo run-time (tipo dell'oggetto referenziato). Ritornando all'*interfaccia* IEsempio, questa viene implementata da ClasseBase e dunque il metodo Visualizza() da essa dichiarato viene “mappato” sul metodo Visualizza() definito in ClasseBase, a prescindere da un'eventuale nuova versione definita nelle classi derivate.

Come vedremo nel prossimo paragrafo, l'invocazione del metodo giusto, determinato cioè dal tipo run-time dell'oggetto, può essere ottenuta definendolo virtuale e sovrascrivendolo nella classe derivata. Esiste però un altro approccio, che non fa uso della derivazione virtuale ed è quello della “reimplementazione” dell'*interfaccia* nelle classe derivata.

5.4 Reimplementazione di una interfaccia in una classe derivata

Una classe derivata può esplicitamente implementare un'*interfaccia*, specificandola nella propria lista base, anche se questa è già stata implementata dalla classe base. Ad esempio, la seguente

versione di `ClasseDerivata` dichiara esplicitamente l'interfaccia `IEsempio`, nonostante questa sia già stata implementata da `ClasseBase`:

```
class ClasseDerivata: ClasseBase, IEsempio
{
    new public void Visualizza()
    {
        Console.WriteLine("Classe derivata");
    }
}
```

Dopo questa modifica, una variabile di tipo `IEsempio` può essere usata per invocare il metodo `Visualizza()` definito da `ClasseDerivata`. Ciò è dimostrato dal fatto che il codice dell'esempio precedente:

```
IEsempio ie = new ClasseDerivata();
ie.Visualizza();
```

produce adesso “Classe derivata”

Reimplementando esplicitamente una *interfaccia* in una classe derivata si ha dunque la possibilità di scrivere codice *consumer* polimorfico pur senza definire virtuali le funzioni dichiarate nell'*interfaccia* stessa.

5.5 Implementazione di interfacce mediante funzioni virtuali

Una interfaccia si limita a definire un elenco di prototipi di funzioni, senza aggiungere ad esse alcun modificatore (`public`, `private`, `virtual`, `override`, eccetera). Dal punto di vista formale, dunque, è del tutto irrilevante che una funzione dichiarata in una *interfaccia* e implementata da una classe sia definita virtuale o rappresenti la ridefinizione (`override`) di una funzione virtuale definita nella classe base. D'altra parte, ciò influisce sul funzionamento del codice *consumer* che utilizza gli oggetti della classe attraverso delle variabili *interfaccia*.

Per comprendere la questione prendiamo ancora una volta in esame le classi del paragrafo precedente.

```
interface IEsempio
{
    void Visualizza();
}
```

```
class ClasseBase: IEsempio
{
    public virtual void Visualizza()
    {
        Console.WriteLine("Classe base");
    }
}
```

```
class ClasseDerivata: ClasseBase
{
    public override void Visualizza()
    {
        Console.WriteLine("Classe derivata");
    }
}
```

```

    }
}

```

Definendo virtuale il metodo `Visualizza()` di `ClasseBase` e sovrascrivendolo in `ClasseDerivata` è possibile accedere in modo polimorfico agli oggetti delle due classi utilizzando una variabile di tipo `IEsempio`.

Ad esempio, il seguente codice:

```

IEsempio ieb = new ClasseBase();
IEsempio ied = new ClasseDerivata();
ieb.Visualizza();    // invoca Visualizza() definito in ClasseBase
ied.Visualizza();    // invoca Visualizza() definito in ClasseDerivata

```

produce in output:

Classe base

Classe derivata

5.6 Operatori di cast e *interfacce*

Nel capitolo dedicato al polimorfismo abbiamo affrontato una problematica che si presenta in modo frequente nella OOP e cioè quella di ottenere il tipo effettivo (run-time) di un oggetto. Questa problematica resta attuale anche quando in gioco entrano le *interfacce*. Gli operatori di cast `is`, `as` e `()` possono essere usati con le *interfacce* (sia intese come tipi che come variabili) nello stesso in modo in cui vengono impiegati con le classi.

Interfacce e operatori di cast "()" e "as"

In molti casi può rivelarsi necessario utilizzare l'operatore di cast per convertire un riferimento al tipo *interfaccia* appropriato. Riprendiamo l'esempio della flotta di cui sopra e consideriamo il seguente frammento di codice, nel quale un vettore di tipo `Mezzo` viene popolato con una nave e un furgone:

```

// ... crea oggetti furgone e nave
Mezzo[] mezzi = new Mezzo[2];
mezzi[0] = furgone;
mezzi[1] = nave;
double totl = 0;
for (int i = 0; i < mezzi.Length; i++)
{
    ICaricabile caricata = (ICaricabile)mezzi[i];
    totl = totl + caricata.CostoTrasporto(20);
}
Console.WriteLine("Totale del carico:{0}", totl);

```

Mediante un `for()` viene calcolato il costo totale di carico, invocando il metodo `CostoTrasporto()` dei due oggetti del vettore. Ma il tipo statico degli elementi del vettore è `Mezzo` e non `ICaricabile`: è necessario applicare l'operatore di cast.

Ovviamente questo codice funziona perché tutti gli oggetti memorizzati nel vettore `mezzi` implementano l'interfaccia `ICaricabile`; qualora così non fosse sarebbe sollevata un'eccezione di tipo `InvalidCastException`.

Nello stesso codice può essere usato l'operatore di cast `as`, che è del tutto analogo all'operatore `()` con la differenza che non scatena un'eccezione nel caso in cui il tipo run-time dell'oggetto non

corrisponda al tipo del cast; in questo caso si limita infatti a produrre come risultato un riferimento nullo.

Uso degli operatori "is" e "as" per verificare se una classe implementa una interfaccia

In alcuni procedimenti può accadere di dover eseguire delle istruzioni invece di altre in base al fatto che un oggetto implementi o meno una determinata *interfaccia*. Per fare ciò si esegue ciò che si dice “interrogazione dell’oggetto”, che si traduce nell’uso dell’operatore `is` per verificare se il suo tipo run-time è compatibile (implementa) una certa *interfaccia*:

```
Mezzo[] flottaMezzi = new Mezzo[3];
flottaMezzi[0] = furgone;
flottaMezzi[1] = nave;
flottaMezzi[2] = aereo;
for (int i = 0; i < flottaMezzi.Length; i++)
{
    Mezzo m = flottaMezzi[i];
    if (m is ICaricabile)
        Console.WriteLine("Il mezzo {0} è caricabile", m);
}
```

Lo stesso risultato può essere ottenuto con maggiore efficienza mediante l’operatore `as`:

```
for (int i = 0; i < flottaMezzi.Length; i++)
{
    INoleggiabile m = flottaMezzi[i] as INoleggiabile;
    if (m != null)
        Console.WriteLine("Il mezzo {0} è noleggiabile", m);
}
```

Conoscere il tipo run-time referenziato da una variabile interfaccia

In precedenza sono stati usati gli operatori di cast `is` e `as` per verificare se un oggetto implementa o meno una determinata *interfaccia*. E’ anche possibile ottenere l’informazione opposta, e cioè sapere se una variabile *interfaccia* fa riferimento a un oggetto di un determinato tipo (o un tipo derivato da esso). Nel seguente frammento viene verificato se la variabile `a` referencia un oggetto di tipo `Aereo`:

```
INoleggiabile a = new Aereo("Spirit", 1000);
// . . .
if (a is Aereo)
    Console.WriteLine("Il mezzo {0} è un aereo", a);
else
    Console.WriteLine("Il mezzo {0} non è un aereo", a);
```

Avremmo potuto ottenere lo stesso risultato usando il metodo `GetType()`, il quale, eseguito attraverso una variabile *interfaccia*, non ritorna il tipo dell’interfaccia ma il tipo dell’oggetto referenziato dalla variabile:

```
INoleggiabile al = new Aereo("Air 1", 1000);
Type t = al.GetType();
if (t.Name == "Aereo")
    Console.WriteLine("Il mezzo {0} è un aereo", al);
else
```

```
Console.WriteLine("Il mezzo {0} non è un aereo", al);
```

5.7 Covarianza nelle interfacce

Abbiamo visto in precedenza i concetti di varianza dei tipi. È giunto il momento di vederli anche applicati alle collezioni generiche ed alle interfacce.

Fino al .NET 4.0 le collezioni e le interfacce si sono comportate come i vettori, essendo cioè invarianti.

Ipotizzando di avere una classe `NaveMercantile` che deriva da `Nave` nel seguente codice

```
List<NaveMercantile> listaDiNaveMercantile = new List<NaveMercantile>();
List<Nave> listaDiNavi = new List<Nave>();
listaDiNavi = listaDiNaveMercantile;
listaDiNavi = listaDiNaveMercantile.ToList<Nave>();
IEnumerable<Nave> iEnumNave = new List<NaveMercantile>();
listaDiNavi.AddRange(listaDiNaveMercantile);
```

le righe evidenziate danno tutte errore in fase di compilazione.

Questo è un comportamento che a sembra decisamente strano in quanto non si capisce perché si può scrivere una istruzione di questo tipo

```
NaveMercantile mercantile = new NaveMercantile();
Nave n = mercantile;
```

mentre non è possibile fare la stessa cosa con le collezioni di oggetti come in questo modo

```
List<NaveMercantile> listaDiNaveMercantile = new List<NaveMercantile>();
List<Nave> listaDiNavi = listaDiNaveMercantile; // errore in compilazione
```

Per ovviare a questo problema a partire dalla versione 4.0 del .NET è stata aggiunto anche alle interfacce che utilizzano dei tipi generici il concetto di covarianza.

Tornando al nostro esempio, dato che la classe `List<T>` implementa l'interfaccia `IEnumerable<T>` allora vuol dire che è possibile assegnare ad un oggetto di tipo `IEnumerable<Nave>` un oggetto di tipo `IEnumerable<NaveMercantile>`.

Quindi delle 4 istruzioni di assegnamento viste in precedenza che davano errore solamente la prima continuerà ad avere “problemi”, mentre le altre saranno perfettamente accettate in quanto tutte e tre utilizzano l'interfaccia `IEnumerable<T>`.

```
listaDiNavi = listaDiNaveMercantile; // da ancora errore in compilazione
listaDiNavi = listaDiNaveMercantile.ToList<Nave>(); // ok
IEnumerable<Nave> iEnumNave = new List<NaveMercantile>(); // ok
listaDiNavi.AddRange(listaDiNaveMercantile); // ok
```

5.8 Interfacce e tipi struttura

Anche i tipi struttura possono implementare le *interfacce* ed è dunque possibile usare variabili *interfaccia* per riferirsi a valori di tipo struttura, purché essi implementino l'*interfaccia* in questione.

Non esiste sostanziale differenza tra l'implementazione e l'uso di *interfacce* con le classi piuttosto che con le strutture, se non che le seconde non supportano l'ereditarietà e la derivazione virtuale e dunque non è possibile usare le variabili *interfaccia* per referenziare oggetti struttura derivati.

Metodi di Estensione

1 Introduzione ai metodi di estensione

Uno dei capisaldi della programmazione ad oggetti è la possibilità di estendere tipi base oppure creati dall'utente mediante l'ereditarietà; ci sono però dei casi in cui non si può (oppure non è possibile) utilizzare questo paradigma.

Tipico è il caso della classe `string` che essendo dichiarata di tipo `sealed` (bloccata) non è possibile derivarla per poterla estendere.

Ipotizziamo a questo punto di voler controllare se il contenuto di una variabile stringa abbia al suo interno degli spazi bianchi.

Prima dell'introduzione dei metodi di estensione l'unico modo era quello di creare una classe statica che avesse al suo interno un metodo che effettuasse il controllo.

```
static class Estensione
{
    static public bool HaSpazi(string s)
    {
        bool ris = false;
        if (s.IndexOf(' ') != -1)
            ris = true;
        return ris;
    }
}
```

A questo punto potremmo invocare il metodo statico in questo modo

```
string cognome = "Del Furia";
bool cnt = Estensione.HaSpazi(cognome);
```

Certamente questa forma non è molto leggibile ma soprattutto l'utente non ha modo di sapere che esista un metodo `HaSpazi` che consente di effettuare questo controllo su di una stringa; con l'utilizzo dei metodi di estensione tutto risulta molto più facile ed intuitivo.

La semplicità sta nel fatto che per trasformare un metodo statico in un metodo di estensione è sufficiente aggiungere la parola chiave `this` prima del primo parametro.

Il codice risultante sarà quindi il seguente:

```
static public bool HaSpaziExt(this string s)
{
    bool ris = false;
    if (s.IndexOf(' ') != -1)
        ris = true;
    return ris;
}
```

```
}
```

Il codice *consumer* potrà utilizzare questo metodo come qualsiasi altro metodo “nativo” della classe *string* in questa maniera:

```
string cognome = "Del Furia";  
bool cnt = cognome.HaSpaziExt();
```

rendendolo molto più comprensibile, senza dimenticare il fatto che l’intellisense dell’editor sarà in grado di proporre anche questo ulteriore metodo in fase di scrittura del codice.

Facciamo un altro esempio estendendo la classe *List<T>* in maniera tale da disporre di due metodi *Pop* e *Push* mancanti nella versione originale.

Scrivendo due semplici metodi di estensione come questi:

```
public static void Push<T>(this List<T> lista, T valore)  
{  
    lista.Add(valore);  
}
```

```
public static T Pop<T>(this List<T> lista)  
{  
    if (lista.Count == 0)  
        throw new Exception("Niente da estrarre.");  
    int ultimo = lista.Count - 1;  
    T valore = lista[ultimo];  
    lista.RemoveAt(ultimo);  
    return valore;  
}
```

sarà possibile utilizzarli successivamente in questa maniera:

```
List<int> Stack = new List<int>();
```

```
for (int i = 0; i <= 10; i++)  
    Stack.Push(i);
```

```
while (Stack.Count > 0)  
    Console.WriteLine(Stack.Pop());
```

come si può notare nei metodi di estensione abbiamo utilizzato la classe *List<T>* come tipo a cui applicare il metodo, ma nulla avrebbe vietato di utilizzare la interfaccia *IEnumerable<T>* e di poterli quindi utilizzare in ogni oggetto che implementi detta interfaccia.

2 Riutilizzo di metodi senza l’ereditarietà

Fin qua abbiamo visto i metodi di estensione come un qualcosa che estende e semplifica l’utilizzo di tipi, ma analizziamo adesso anche un aspetto molto interessante.

Spesso capita di dover utilizzare uno stesso metodo in classi che non condividono una stessa classe base ed in molti casi questo obiettivo comporta un grande lavoro di refactoring del codice (e forse risulterà impossibile senza modificare la gerarchia delle classi).

Un altro approccio comporta la definizione di interfacce con la necessità di implementazione del codice in ogni classe.

Utilizzando i metodi di estensione, invece, sarà possibile aggiungere una funzionalità ad una serie di classi senza troppa fatica.

Riprendiamo l'esempio fatto in precedenza nel quale la gerarchia di classi della flotta navale prevede che alcune classi d'imitazione implementino già una particolare interfaccia chiamata `ICaricabile` definita come segue

```
public interface ICaricabile
{
    Carico Colli { get; }
    Carico Scarica();
    void Carica(Carico colli);
    double CostoTrasporto(int km);
}
```

Si ipotizzi adesso che si voglia calcolare il peso trasportato dal mezzo, sommando i singoli pesi del carico. Implementare questa funzionalità significa dover riscrivere la definizione dell'interfaccia in questa maniera:

```
public interface ICaricabile
{
    Carico Colli { get; }
    Carico Scarica();
    void Carica(Carico colli);
    double CostoTrasporto(int km);
    double PesoCarico();
}
```

Abbiamo però il problema che in un'interfaccia non è possibile stabilire un comportamento predefinito e quindi saremo obbligati a scrivere in tutte le classi che la implementano il metodo `PesoCarico()`, uguale per tutti, del tipo:

```
public double PesoCarico()
{
    double totale = 0;
    foreach (Oggetto pacco in _carico.Colli)
    {
        totale = totale + pacco.Peso;
    }
    return totale;
}
```

Utilizzando i metodi di estensione, invece, sarà sufficiente dichiarare una classe statica con al suo interno il metodo statico `PesoCarico` che abbia come primo parametro un oggetto di tipo `ICaricabile`.

Il codice sarà il seguente:

```
static public class Estensione
{
    static public double PesoCarico(this ICaricabile carico)
    {
        double totale = 0;
        foreach (Oggetto pacco in carico.Colli)
```

```
        {  
            totale = totale + pacco.Peso;  
        }  
        return totale;  
    }  
}
```

A questo punto, ogni oggetto che sia di un tipo che implementa l'interfaccia `ICaricabile`, potrà utilizzare questo metodo in maniera molto semplice e comprensibile.

```
Nave nave = new Nave("Splendida", 1000);  
Furgone furgone = new Furgone("Mater", 4);  
  
double caricoNave = nave.PesoCarico();  
double caricoFurgone = furgone.PesoCarico();
```