TimeRange

Case study sull'implementazione di un tipo omologo a TimeSpan

Anno 2018/2019

Indice generale

1 Introduzione

In questo tutorial utilizzo un semplice problema di programmazione come mezzo per affrontare le problematiche connesse all'implementazione di un nuovo tipo di dato. L'obiettivo è quello di realizzare un tipo analogo a `TimeSpan`, il quale incapsula un intervallo di tempo.

Dato un problema, introdurrò dapprima un approccio completamente procedurale, mostrandole i limiti. Quindi fornirò una soluzione che usa `TimeSpan`, evidenziando i vantaggi offerti dall'uso di un *oggetto*. Infine, ponendo come presupposto che `TimeSpan` non esista, realizzerò una classe che ne emuli le caratteristiche. Procederò gradualmente, realizzando dapprima un progetto "grossolano", per poi rifinirlo adottando i principi di progettazione OO.

1.1 Problema: maratona olimpica

Un file di testo in formato CSV memorizza i tempi dei partecipanti dalla gara della maratona olimpica:

```
Tadesse Abraham, Svizzera, 2:11:42
Ghirmay Ghebreslassie, Eritrea, 2:11:4
Feyisa Lilesa, Etiopia, 2:09:54
Eliud Kipchoge, Kenya, 2:8:44
Munyo Mutai, Uganda, 2:11:49
Galen Rupp, USA, 2:10:05
Alphonce Simbu, Tanzania, 2:11:15
Jared Ward, USA, 2:11:30
...
```

Si chiede di visualizzare l'ordine d'arrivo della maratona, compreso il distacco dal primo classificato.

2 Uso del modello procedurale

Davanti a un problema la prima questione è: come rappresentare i dati. Il modello scelto deve avere la funzione di favorire l'implementazione delle richieste del problema: visualizzazione delle informazioni sugli atleti ed elaborazioni sulla base dei loro tempi.

Di seguito introduco una semplice soluzione procedurale.

2.1 Opzione 1: uso di un *record*

Dall'analisi del problema emerge chiaramente la necessità di rappresentare i singoli maratoneti. È possibile farlo mediante un *record* i cui campi memorizzano le informazioni necessarie:

```
class Atleta
{
    public string Nominativo;
    public string Nazione;
    public string Tempo;
}
```

Questa soluzione semplifica la gestione della lista dei maratoneti e il loro caricamento dal file:

```
static List<Atleta> CaricaAtleti()
{
    string[] righe = File.ReadAllLines(@"Maratona.txt");
    var atleti = new List<Atleta>();
    for (int i = 0; i < righe.Length; i++)
    {
        string[] campi = righe[i].Split(',');
        var a = new Atleta
        {
            Nominativo = campi[0].Trim(),
            Nazione = campi[1].Trim(),
            Tempo = campi[2].Trim()
        };
        atleti.Add(a);
    }
}
```

Il tipo `Atleta` incapsula il concetto di *maratoneta* e consente di accedere facilmente ai suoi dati: `Nominativo`, `Nazione` e `Tempo`. Quest'ultimo, però, essendo di tipo `string`; non consente di eseguire operazioni sui tempi. Ad esempio, il tempo di:

`"Tadesse Abraham, Svizzera, 2:11:42"`

è maggiore del tempo di:

`"Eliud Kipchoge, Kenya, 2:8:44"`

ma così non risulterebbe da un confronto tra stringhe, poiché la prima è minore della seconda.

Stesso discorso vale per il calcolo dei distacchi dal primo arrivato.

Rappresentare i dati mediante tipi adeguati è fondamentale, poiché sono i tipi a stabilire le operazioni ammissibili su di essi. I tipi primitivi – `int`, `char`, `string`, `double`, etc – non sempre sono sufficienti; in alcuni casi è necessario definire nuovi tipi:

- Perché non esiste un tipo in grado di rappresentare un determinato concetto (vedi, ad esempio, il maratoneta).
- Perché i tipi utilizzabili non forniscono le operazione richieste.

Il secondo punto riguarda la rappresentazione dei tempi, la necessità di poterli confrontare e di poter calcolare i loro distacchi.

2.2 Rappresentare il tempo degli atleti: intervallo di tempo

Elaborare il tempo degli atleti significa poter conoscere ore, minuti e secondi, confrontare due tempi, ottenere l'intervallo che intercorre tra due tempi. Un approccio è quello di rappresentare un singolo tempo mediante un *record*: [1]

```
class TimeRange
{
    public int Hours;
    public int Minutes;
    public int Seconds;
}
...
class Atleta
{
    public string Nominativo;
    public string Nazione;
    public TimeRange Tempo;
}
```

2.2.1 Implementare le operazioni sui tempi

`TimeRange` si limita a memorizzare i dati; al codice applicativo resta la responsabilità di implementare le operazioni necessarie:

- Confrontare due tempi tra loro.
- Calcolare la differenza tra due tempi.
- Creare un tempo a partire dalla sua rappresentazione in stringa.
- Ottenere la rappresentazione stringa di un tempo.

Non si tratta di problemi che riguardano questa specifica applicazione, ma hanno una natura generale, poiché sono molti gli scenari nei quali occorre elaborare dei tempi. Per questo motivo è estremamente utile poter incapsulare la loro soluzione in un componente, in modo da poterla facilmente riutilizzare ovunque sia necessario.

1 Uso la nomenclatura inglese per omologare il codice alla versione successiva del tipo **TimeRange**, il cui scopo è emulare **TimeSpan**.

2.3 Uso di TimeSpan

`TimeSpan` implementa il concetto di *intervallo di tempo*, quindi si presta perfettamente ad essere utilizzato per rappresentare i tempi degli atleti:

```
class Atleta
{
    public string Nominativo;
    public string Nazione;
    public TimeSpan Tempo;
}
```

Adottando questa soluzione, scrivere il codice applicativo diventa estremamente semplice:

```
class Program
{
    static List<Atleta> atleti;
    static void Main(string[] args)
    {
        CaricaAtleti();
        Ordina();
        Visualizza();
    }

    static void CaricaAtleti()
    {
        string[] righe = File.ReadAllLines("Maratona.txt");
        atleti = new List<Atleta>();
        for (int i = 0; i < righe.Length; i++)
        {
            string[] campi = righe[i].Split(',');
            var a = new Atleta
            {
                Nominativo = campi[0].Trim(),
                Nazione = campi[1].Trim(),
                Tempo = TimeSpan.Parse(campi[2])  //crea un tempo da una stringa
            };
            atleti.Add(a);
        }
    }

    private static void Visualizza()
    {
        Console.WriteLine($"{"Nominativo",-25}{"Nazione",-15}{"Tempo",-12}{"Distacco"}");
        var primo = atleti[0];
        foreach (var a in atleti)
        {
            TimeSpan d = a.Tempo - primo.Tempo;  //calcola la differenza tra due tempi
            Console.WriteLine($"{a.Nominativo,-25}{a.Nazione,-15}{a.Tempo,-12:t}{d:t}");
        }
        Console.WriteLine();
    }
```

```
    static void Ordina()
    {
        for (int i = 0; i < atleti.Count - 1; i++)
        {
            for (int j = i + 1; j < atleti.Count; j++)
            {
                if (atleti[i].Tempo > atleti[j].Tempo) //confronta due tempi
                {
                    var a = atleti[i];
                    atleti[i] = atleti[j];
                    atleti[j] = a;
                }
            }
        }
    }
}
```

Nel codice ho evidenziato le operazione condotte sui tempi. `TimeSpan` consente di:

- Creare un nuovo tempo a partire dalla sua rappresentazione stringa:

  ```
  Tempo = TimeSpan.Parse(campi[2]);
  ```

  Nota bene: adotta lo stesso metodo usato dai tipi `int`, `double`, etc.

- Confrontare due tempi:

  ```
  if (atleti[i].Tempo > atleti[j].Tempo) ...
  ```

- Calcolare la differenzi tra due tempi:

  ```
  TimeSpan d = a.Tempo - primo.Tempo;
  ```

- Visualizzare i tempi (ottenere una rappresentazione stringa):

  ```
  Console.WriteLine($"{a.Nominativo,-25}{a.Nazione,-15}{a.Tempo,-12:t}{d:t}");
  ```

In sostanza, `TimeSpan` consente di rappresentare i tempi in modo quantitativo, analogamente ai tipi numerici; e ciò permette di manipolare i tempi nello stesso modo in cui si manipola i numeri, confrontandoli, sottraendoli, etc.

Di seguito, partendo dal presupposto che `TimeSpan` non esista, implementerò un tipo che esibisca lo stesso funzionamento.

3 Oggetto TimeRange

La programmazione *object oriented* presuppone che i tipi incorporino sia dati che le operazioni da eseguire su di essi. L'obiettivo è realizzare un componente (l'*oggetto*) che definisca tutto ciò che serve a implementare il concetto di intervallo di tempo.

3.1 *Oggetto* TimeRange: dati + operazioni

Se ci si limita al "mantra" della OOP, *combinare dati e operazioni nello stesso contenitore*, la trasformazione di `TimeRange` da *record* a *oggetto* è semplice: basta incorporare i metodi di elaborazione dei campi che rappresentano un intervallo di tempo.

Segue una prima implementazione dell'oggetto `TimeRange`:

```
public class TimeRange
{
    public int Hours;
    public int Minutes;
    public int Seconds;

    public void Parse(string value)
    {
        string[] campi = value.Split(':');
        Hours = int.Parse(campi[0]);
        Minutes = int.Parse(campi[1]);
        Seconds = int.Parse(campi[2]);
    }

    //viene usato in Subtract() per creare un tempo a partire dai secondi totali
    private TimeRange(int totalSeconds)
    {
        Hours = totalSeconds / 3600;
        Minutes = totalSeconds / 60 % 60;
        Seconds = totalSeconds % 60;
    }

    //-> this == t -> 0;  this < t -> -1;  this > t-> 1
    public int CompareTo(TimeRange t)
    {
        return TotalSeconds(this).CompareTo(TotalSeconds(t));
    }

    //-> this - t
    public TimeRange Subtract(TimeRange t)
    {
        int offset = TotalSeconds(this) - TotalSeconds(t);
        return new TimeRange(offset);
    }

    private int TotalSeconds(TimeRange t)
    {
        return t.Hours * 3600 + t.Minutes * 60 + t.Seconds;
    }
}
```

`TimeRange` memorizza l'intervallo di tempo nei componenti *ore*, *minuti* e *secondi*. Il metodo `TotalSeconds()` restituisce il numero di secondi totali corrispondenti; ciò semplifica le operazioni di sottrazione e confronto.

Nota bene: ho definito un costruttore privato, e cioè ad uso interno, che consente di creare un nuovo `TimeRange` sulla base del numero totale di secondi.

La classe presenta molte differenze rispetto a `TimeSpan`, alcune delle quali, però, sono apparenti. Il metodo `Subtract()` svolge le della sottrazione, mentre `CompareTo()` implementa le operazioni di confronto `==`, `<`, `>`. Anche `TimeSpan` definisce questi metodi, i quali hanno la medesima funzione.

Segue la nuova versione del codice applicativo, che adesso usa `TimeRange`:

```
class Atleta
{
    public string Nominativo;
    public string Nazione;
    public TimeSpan Tempo;
    public TimeRange Tempo;
}
...

static void Main(string[] args)
{
    CaricaAtleti();
    Ordina(atleti);
    VisualizzaClassifica(atleti);
}

private static void Visualizza()
{
    Console.WriteLine($"{"Nominativo",-25}{"Nazione",-15}{"Tempo",-12}{"Distacco"}");
    var primo = atleti[0];
    foreach (var a in atleti)
    {
        TimeSpan d = a.Tempo - primo.Tempo;
        TimeRange d = a.Tempo.Subtract(primo.Tempo);

        string tempoStr = TRToString(a.Tempo);
        string dStr = TRToString(d);
        Console.WriteLine($"{a.Nominativo,-25}{a.Nazione,-15}{tempoStr,-12}
                                                              {dStr}");
    }
    Console.WriteLine();
}

static string TRToString(TimeRange t)
{
    return $"{t.Hours:00}:{t.Minutes:00}:{t.Seconds:00}";
}
```

```
static void Ordina(Atleta[] atleti)
{
    for (int i = 0; i < atleti.Length-1; i++)
    {
        for (int j = i+1; j < atleti.Length; j++)
        {
            if (Compare(atleti[i].Tempo > atleti[j].Tempo))
            if (atleti[i].Tempo.CompareTo(atleti[j].Tempo) > 0)
            {
                var temp = atleti[i];
                atleti[i] = atleti[j];
                atleti[j] = temp;
            }
        }
    }
}

static void CaricaAtleti()
{
    string[] righe = File.ReadAllLines(@"Maratona.txt");
    atleti = new List<Atleta>();
    for (int i = 0; i < righe.Length; i++)
    {
        string[] campi = righe[i].Split(',');
        var tr = new TimeRange();
        tr.Parse(campi[2]);
        var a = new Atleta
        {
            Nominativo = campi[0].Trim(),
            Nazione = campi[1].Trim(),

            atleti[i].Tempo = tr;
            Tempo = TimeSpan.Parse(campi[2])
            Tempo = tr
        };
        atleti.Add(a);
    }
}
```

Ho evidenziato le parti nuove e mantenuto (in grigio) quelle vecchie. Il codice risulta leggermente più complicato rispetto a prima, per i seguenti motivi:

- `TimeRange` non definisce gli operatori `>` e `-` ; al loro posto esistono i metodi `Subtract()` e `CompareTo()`.
- Il *parsing* da stringa (`Parse()`) è stato implementato mediante un metodo di istanza.
- `TimeRange` non definisce un metodo che restituisce la rappresentazione stringa del tempo; attualmente questa è gestita in `Program` mediante `TRToString()`.

Non sono le uniche questioni da affrontare, ne esiste una più importante: `TimeRange` mantiene ancora la caratteristica di *record*, poiché il codice esterno può manipolare direttamente i suoi campi.

Ciò viola il principio di incapsulamento; infatti, soltanto i metodi di un *oggetto* possono manipolare il suo *stato* (i suoi campi).

Una soluzione è quella di trasformare i campi in proprietà *get-only*:

```
public class TimeRange
{
    public int Hours { get; private set; }
    public int Minutes { get; private set; }
    public int Seconds { get; private set; }
    ...
}
```

(Tutto ciò non riguarda semplicemente la possibilità o meno di modificare ore, minuti o secondi; c'è una differenza sostanziale tra l'uso di proprietà e di variabili, come vedremo più avanti.)

3.2 Implementare il parsing mediante un costruttore

L'implementazione del *parsing* mediante un metodo di istanza non è una scelta ottimale: costringe prima a creare un oggetto e poi a modificarlo in base alla stringa specificata. Ma perché eseguire due operazioni, quando lo scopo è comunque quello di ottenere un nuovo oggetto a partire da una stringa? È opportuno implementare l'operazione in un costruttore:

```
public class TimeRange
{
    ...
    public TimeRange(string value)
    {
        string[] campi = value.Split(':');
        Hours = int.Parse(campi[0]);
        Minutes = int.Parse(campi[1]);
        Seconds = int.Parse(campi[2]);
    }

    public void Parse(string value) { ... }
    ...
}
```

Ciò consente di semplificare il codice di creazione degli atleti:

```
static void CaricaAtleti()
{
    ...
        var tr = new TimeRange();
        tr.Parse(campi[2]);
        var a = new Atleta
        {
            Nominativo = campi[0].Trim(),
            Nazione = campi[1].Trim(),
            Tempo = tr
            Tempo = new TimeRange(campi[2]);
```

```
        };
        atleti.Add(a);
    }
}
```

3.3 Immutabilità di TimeRange

Dopo quest'ultima modifica, `TimeRange` è diventato un tipo ***immutabile***; cioè un tipo il cui stato non può essere modificato dopo la creazione. In generale, si tratta di un risultato desiderabile, poiché semplifica l'implementazione (è semplice implementare l'*invariante di classe*, ad esempio), ma anche il suo uso, dato che non esistono metodi che possono modificare gli oggetti.

Infine, facilita la gestione di scenari *multi-thread*, poiché più *thread* possono condividere lo stesso oggetto senza il rischio di produrre effetti collaterali indesiderati.

3.4 Produrre la rappresentazione stringa dell'oggetto: ToString()

Di norma, tutto ciò che riguarda un tipo dovrebbe essere implementato al suo interno. Dato che `TimeRange` è in grado di eseguire il *parsing* di un tempo rappresentato come stringa, dovrebbe essere in grado di eseguire anche l'operazione opposta:

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

Questa modifica semplifica ulteriormente il codice applicativo (oltre a migliorare la rappresentazione stringa, gestendo anche i tempi negativi):

```
private static void Visualizza()
{
    Console.WriteLine($"{"Nominativo",-25}{"Nazione",-15}{"Tempo",-12}{"Distacco"}");
    var primo = atleti[0];
    foreach (var a in atleti)
    {
        TimeRange d = a.Tempo.Subtract(primo.Tempo);
        string tempoStr = TRToString(a.Tempo);
        string dStr = TRToString(d);
        Console.WriteLine($"{a.Nominativo,-25}{a.Nazione,-15}{a.Tempo.ToString(),-12}
                                                             {d.ToString()}");
    }
    Console.WriteLine();
}
```

In generale, tutti i tipi per i quali esiste una rappresentazione in stringa dovrebbero implementare il metodo `ToString()`.[2]

3.5 Aggiungere un costruttore che accetti ore, minuti, secondi

L'attuale applicazione non ha bisogno di un costruttore simile, ma è opportuno implementarlo. IN generale, deve essere possibile creare un nuovo oggetto specificandone esplicitamente il valore, senza dover eseguire il *parsing* di una stringa.

```
public class TimeRange
{
    ...
    public TimeRange(int hours, int minutes, int seconds)
    {
        Hours = hours;
        Minutes = minutes;
        Seconds = seconds;
    }
    ...
}
```

3.6 Rendere statico il metodo TotalSeconds()

Questo metodo presenta una piccola incoerenza: non accede ai campi della classe.

```
private int TotalSeconds(TimeRange t)
{
    return t.Hours * 3600 + t.Minutes * 60 + t.Seconds;
}
```

Si tratta di un metodo che viene usato internamente per ottenere i secondi totali di un oggetto. Ebbene, se un metodo non elabora i campi, è inutile (e non ha senso) che sia implementato come metodo di istanza, *poiché non agisce sull'istanza!* È opportuno che sia dichiarato statico:

```
private static int TotalSeconds(TimeRange t)
{
    return t.Hours * 3600 + t.Minutes * 60 + t.Seconds;
}
```

Per quanto riguarda il suo uso, in `Subtract()` e `CompareTo()`, non cambia assolutamente niente.

2 C'è una questione importante che riguarda **ToString()**, ma sulla quale qui intendo sorvolare.

4 Migliorare l'implementazione di TimeRange

Il tipo `TimeRange` è funzionale ai requisiti dell'applicazione nella quale è stato realizzato, ma se vogliamo renderlo utilizzabile in un ampia gamma di scenari dobbiamo adottare criteri di progettazione più stringenti. Nella programmazione OO ci sono diverse aree da considerare:

- **Validità**: occorre garantire che gli oggetti abbiano sempre dei valori coerenti e che le loro operazioni restituiscano sempre dei valori validi.
- **Incapsulamento**: separare la funzione dell'*oggetto* dal modo in cui è implementata.
- **Efficienza**: le *perfomance*, occupazione di memoria e velocità di esecuzione delle operazioni, sono importanti.
- **Usabilità**: rendere l'uso delle variabili il più semplice e naturale possibile.

Di seguito modificherò progressivamente il progetto di `TimeRange` in modo che aderisca completamente a questi criteri.

4.1 Migliorare le performance: velocità

`TimeRange` memorizza il tempo in ore, minuti e secondi, ma lo elabora calcolando ogni volta il tempo totale corrispondente (metodo `TotalSeconds()`). Tanto vale calcolare subito questo dato e memorizzarlo in un campo della classe:

```
class TimeRange
{
    public int Hours { get; private set; }
    public int Minutes { get; private set; }
    public int Seconds { get; private set; }
    public int TotalTime {get; private set; }

    public TimeRange(string value)
    {
        //... esegue il parsing della stringa
        TotalTime = Hours * 3600 + Minutes * 60 + Seconds;
    }

    public TimeRange(int hours, int minutes, int seconds)
    {
        //... assegna a Hours, Minutes e Seconds
        TotalTime = Hours * 3600 + Minutes * 60 + Seconds;
    }

    private TimeRange(int totalTime)
    {
        TotalTime =  totalTime
    }
```

```
    public int CompareTo(TimeRange t)
    {
        return TotalTime.CompareTo(t.TotalTime);
    }

    public TimeRange Subtract(TimeRange t)
    {
        return new TimeRange(TotalTime - t.TotalTime);
    }

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

    public static int TotalSeconds(TimeRange t)
    {
        return t.Ore * 3600 + t.Minuti * 60 + t.Secondi;
    }
}
```

L'introduzione del nuovo campo semplifica il codice della classe e velocizza notevolmente le operazioni di sottrazione e confronto.

4.2 Migliorare le performance (e l'implementazione): memoria

L'introduzione di `TotalTime` ha reso l'implementazione della classe ridondante: adesso, infatti, il tempo è memorizzato in due modi distinti. `TotalTime` viene usato per i calcoli; `Hours`, `Minutes`, `Seconds` sono usate per ottenere la rappresentazione stringa.

Anche se `TimeRange` funziona correttamente, questa incoerenza deve essere risolta, anche perché implica un'occupazione di memoria ingiustificabile.

La soluzione è trasformare `Hours`, `Minutes` e `Seconds` in proprietà derivate, che ottengono il loro valore da `TotaleTime`:

```
class TimeRange
{
    public int Hours {get { return TotalTime / 3600 % 3600; }}

    public int Minutes {get { return TotalTime / 60 % 60; }}

    public int Seconds {get { return TotalTime % 60; }}
    ...
}
```

Adesso, un oggetto `TimeRange` occupa solo 4 bytes e il suo comportamento dipende dal solo campo `TotalTime`.

4.3 Incapsulare completamente TimeRange

Incapsulare un tempo significa fornire un insieme di operazioni utilizzabili indipendentemente dall'implementazione. È ciò che fa `TimeRange`, come è stato dimostrato in 4.1 e 4.2: la modalità di memorizzazione del tempo è stata modificata senza produrre un impatto sul codice esterno.

Il termine *incapsulamento* assume dunque un significato stringente: creare una "barriera" tra la funzione di un *oggetto* e la sua implementazione; le prima è utilizzabile senza che sia necessario dipendere dalla seconda.

Ebbene, in realtà `TimeRange` non è completamente incapsulato, poiché `TotalTime` è accessibile, anche se non modificabile, dall'esterno.

4.3.1 "Proteggere" lo stato dell'oggetto

Esiste una concezione diffusa sull'incapsulamento, che lo vede come strumento per nascondere l'implementazione allo scopo di proteggerla dalle modifiche. Ad esempio, la classe `List<>` definisce il campo privato `_size`, che memorizza il numero degli elementi presenti nella lista. Se `_size` fosse pubblico, il codice esterno potrebbe modificarne il valore, con il rischio di compromettere il funzionamento della lista.

Incapsulare `_size` mediante la proprietà `Count` appare una decisione scontata, ma che dire di `TimeRange` e di `TotalTime`? È gestito mediante una proprietà automatica *get-only*; ma anche se non lo fosse, la sua modifica non comprometterebbe il funzionamento dell'oggetto, poiché qualsiasi valore assegnato a `TotalTime` è un valore valido.

In conclusione, si potrebbe pensare che definire `TotalTime` pubblico, o addirittura modificabile dall'esterno, non violi affatto il principio di incapsulamento. Ma non è così.

4.3.2 Incapsulamento: non dipendere dall'implementazione

Ipotizza un'applicazione che debba ottenere il numero totale di secondi che intercorrono tra due tempi. Una soluzione è la seguente:

```
TimeRange start = StartClock();
...
TimeRange stop = StopClock();
TimeRange interval = start.Subtract(stop);
 int elapsedSeconds = interval.Hours * 3600 + interval.Minutes * 60 + interval.Seconds;
...
```

Ma, conoscendo l'implementazione di `TimeRange`, conviene optare per un'alternativa più semplice ed efficiente:

```
TimeRange start = StartClock();
...
TimeRange stop = StopClock();
TimeRange interval = start.Subtract(stop);
 int elapsedSeconds = interval.TotalTime;
...
```

Però, mentre la prima opzione è valida per definizione, la seconda si basa sull'assunzione che `TotalTime` memorizzi il numero totale di secondi. Si tratta di assunzione che dipende dall'implementazione e potrebbe non essere più valida se questa venisse cambiata, ad esempio aggiungendo la gestione dei millisecondi. In quest'ultimo caso, `TotalTime` aumenterebbe di un fattore 1000 e tutto il codice applicativo che si basa su quell'assunzione smetterebbe di funzionare. Rendendo `TotalTime` inaccessibile si evita che il codice esterno dipenda da esso:

```
class TimeRange
{
    ...
    public int TotalTime {get; private set; }
    private int totalTime;
    ...
}
```

4.3.3 Fornire l'accesso ai secondi totali

Supponi, adesso, di voler comunque fornire l'accesso al numero totale di secondi, poiché si tratta di un'informazione utile. Ebbene, è sufficiente definire una proprietà derivata:

```
class TimeRange
{
    ...
    public int TotalSeconds {get { return TotalTime; }}
    private int totalTime;
    ...
}
```

A questo punto è legittimo chiedersi che differenza ci sia tra l'attuale proprietà `TotalsSeconds` e la precedente proprietà `TotalTime`, considerato che entrambe restituiscono lo stesso valore.

La differenza è che la "vecchia" `TotalTime` memorizzava il valore del tempo totale, mentre `TotalSeconds` lo ottiene da un campo privato. Con la versione attuale è possibile modificare la natura del valore memorizzato nel campo privato preservando il corretto funzionamento della proprietà `TotalSeconds`.

Ad esempio, se si decide di gestire anche il millisecondi, `TotalSeconds` sarebbe modificata in:

```
class TimeRange
{
    ...
    public int TotalSeconds {get { return TotalTime * 1000; }}
    private int totalTime;
    ...
}
```

Il codice esterno non sarebbe minimamente influenzato da questo cambiamento.

5 “Standardizzare” TimeRange

Dopo le precedenti modifiche, il tipo `TimeRange` è efficiente e ben incapsulato; resta il fatto che è nato per soddisfare i requisiti di una specifica applicazione. Se vogliamo renderlo utilizzabile ovunque sia necessario elaborare il concetto di tempo, è opportuno aumentare le sue funzionalità e allineare la sua interfaccia pubblica a quella degli altri tipi predefiniti.

5.1 Definire gli operatori standard

Quando si progetta un nuovo tipo è utile implementare tutti gli operatori coerenti con il concetto rappresentato. Per quanto riguarda `TimeRange`, sono senz'altro significativi i seguenti operatori: `==`, `!=`, `>`, `<`, `>=`, `<=`, `+`, `-`.[3]

```
public class TimeRange
{
    private int totalTime;
    ...
    public static bool operator == (TimeRange left, TimeRange right)
    {
        return left.totalTime == right.totalTime ;
    }

    public static bool operator != (TimeRange left, TimeRange right)
    {
        return left.totalTime != right.totalTime;
    }

    public static bool operator > (TimeRange left, TimeRange right)
    {
        return left.totalTime > right.totalTime;
    }

    public static bool operator < (TimeRange left, TimeRange right)
    {
        return left.totalTime < right.totalTime;
    }

    public static bool operator >= (TimeRange left, TimeRange right)
    {
        return left.totalTime > right.totalTime;
    }

    public static bool operator <= (TimeRange left, TimeRange right)
    {
        return left.totalTime <= right.totalTime;
    }
```

3 Ho semplificato l’implementazione degli operatori non tenendo conto del fatto che **left** o **right** potrebbero essere **null**.

```
    public static TimeRange operator + (TimeRange left, TimeRange right)
    {
        return new TimeRange(left.totalTime + right.totalTime);
    }

    public static TimeRange operator - (TimeRange left, TimeRange right)
    {
        return new TimeRange(left.totalTime - right.totalTime);
    }
}
```

Nota bene: coerentemente con i tipi primitivi, gli operatori `+` e `-` producono un nuovo valore, che equivale al risultato dell'operazione.

Dopo questa modifica diventa possibile scrivere il seguente codice:

```
var t1 = new TimeRange(1, 2, 3); // -> 1:2:3
var t2 = new TimeRange(2, 2, 3); // -> 2:2:3
var b1 = t1 == t2;  // -> false
var b2 = t1 < t2;   // -> true
var b3 = t1 >= t2;  // false
var t3 = t1 + t2;   // -> 3:4:6
var t4 = t1 - t2;   // -> -1:0:0
```

Nel codice applicativo è possibile modificare il metodo di ordinamento:

```
static void Ordina(Atleta[] atleti)
{
    for (int i = 0; i < atleti.Length-1; i++)
    {
        for (int j = i+1; j < atleti.Length; j++)
        {
            if (atleti[i].Tempo > atleti[j].Tempo)
            {
                var temp = atleti[i];
                atleti[i] = atleti[j];
                atleti[j] = temp;
            }
        }
    }
}
```

5.2 Implementare il parsing da stringa

I tipi predefiniti sono allineati a uno standard per quanto riguarda la conversione da e per stringa. Definiscono un metodo statico `Parse()` che restituisce un nuovo valore a partire da una stringa e un metodo di istanza `ToString()` che produce una rappresentazione stringa del valore. Inoltre, il metodo `Parse()` solleva l'eccezione `FormatException` se l'argomento stringa non memorizza un valore valido.

```
public class TimeRange
{
    ...
    public TimeRange(string value)
    {
        string[] campi = value.Split(':');
        Hours = int.Parse(campi[0]);
        Minutes = int.Parse(campi[1]);
        Seconds = int.Parse(campi[2]);
    }

    public static TimeRange Parse(string value)
    {
        try
        {
            if (value .Contains(' '))
                throw new Exception();

            string[] fields = value.Split(':');
            if (fields.Length > 3)
                throw new Exception();

            int seconds = 0;
            int minutes = 0;
            int hours = int.Parse(fields[0]);
            if (fields.Length > 0)
                minutes = int.Parse(fields[1]);
            if (fields.Length > 1)
                seconds = int.Parse(fields[2]);
            return new TimeRange(hours, minutes, seconds);
        }
        catch
        {
            throw new FormatException("Valore non valido: " + value);
        }
    }
    ...
}
```

5.3 Informazioni sul valore

`TimeRange` consente di ottenere ore, minuti e secondi di un tempo; ma esistono altre informazioni rilevanti che si possono ricavare: le ore, i minuti e i secondi totali. Questi sono valori reali, perché possono memorizzare anche frazioni di tempo. Prima di implementare queste proprietà, definisco delle costanti simboliche che memorizzano dei rapporti utili nei vari calcoli:

```
public class TimeRange
{
    const int TIME_PER_HOURS = 3600;
    const int TIME_PER_MINUTES = 60;
```

```
    private int totalTime;
    ...
    public double TotalSeconds
    {
        get { return totalTime; }
    }

    public double TotalMinutes
    {
        get { return (double) totalTime / TIME_PER_MINUTES; }
    }

    public double TotalsHours
    {
        get { return (double) totalTime / TIME_PER_HOURS; }
    }
    ...
}
```

Nota bene: non sarebbe necessario definire `double` anche `TotalSeconds`, ma facendolo evito di legare il codice all'implementazione.[4]

5.3.1 Informazioni sul tipo

Come gli altri tipi numerici, è utile che `TimeRange` definisca i valori massimo e minimo:

```
public class TimeRange
{
    ...
    public static readonly TimeRange MaxValue = new TimeRange(int.MaxValue);
    public static readonly TimeRange MinValue = new TimeRange(int.MinValue);
    ...
}
```

I campi devono essere *readonly*, altrimenti nulla impedirebbe al codice esterno di sostituire il valore originale con un altro.

5.4 Creazione di tempi sulla base di valori totali

I seguenti metodi statici realizzano la funzione inversa delle proprietà che restituiscono il tempo totale in ore, minuti o secondi:

```
public class TimeRange
{
    ...
    public static TimeRange FromHours(double hours)
    {
        return new TimeRange((int) ( hours * TIME_PER_HOURS));
    }
```

4 In **TimeSpan** la proprietà analoga si chiama **Ticks** e restituisce un valore integrale corrispondente al tempo totale.

```
    public static TimeRange FromMinutes(double minutes)
    {
        return new TimeRange((int)(minutes * TIME_PER_MINUTES));
    }

    public static TimeRange FromSeconds(double seconds)
    {
        return new TimeRange((int)seconds);
    }
    ...
}
```

Nota bene: prima viene eseguito il calcolo e soltanto dopo viene applicata la conversione a intero; in caso contrario sarebbe persa l'eventuale parte frazionaria del valore specificato.

Adesso è possibile scrivere il seguente codice:

```
var t1 = TimeRange.FromHours(1.5);          // -> 1:30:0
var t2 = TimeRange.FromMinutes(90);         // -> 1:30:0
var b1 = t1 == t2;                          // -> true
var b2 = t1 < t2;                           // -> false
double ore = t1.TotalHours;                 // -> 1.5
double secondi = t1.TotalSeconds;           // -> 5400
```