TAP
Tecniche Avanzate di Programmazione

Samuele Crea

September - December 2021

1

Contents

1 DOTNET e CSharp

1.1 Introducing DOTnet e CSharp

Guardare le slide "Introducing DOTnet e Csharp.

2 Versioning System - Git

2.1 Introduzione

Quando lavoriamo ad un progetto chiaramente creiamo più versioni di quest'ultimo a seconda dei progressi fatti.
E' necessario quindi dare nomi significativi alle cartelle man mano che vengono create.
In più se stiamo creando un grande software che andrà sul mercato sarà necessario migliorare anche versioni precedenti all'ultima.
Un altro problema sta nel fatto che i software sono sempre sviluppati in gruppo quindi bisogna fare in modo da fornire agli altri membri del team sempre codice funzionante.
Per far fronte a queste problematicce sono nati sistemi software di versioning system tra cui Git.

2.2 Git

In git una repository è vista come un DAG (grafo aciclico) dove ogni nodo è un commit del progetto e gli archi vanno dal figlio al padre uniti ad una serie di puntatori.
Ogni commit corrisponde ad uno snapshot del progetto in quel momento unito ad un puntatore al commit padre e a dei metadati.
Uno snapshot è composto da una serie di file.
Un commit è accessibile attraverso il suo hash identificativo.
Nel processo di git le entità coinvolte sono:

- La repository dove si trovano tutti i dati(ogni membro del team la deve avere in locale)
- La working directory ovvero la cartella dove si sta lavorando
- HEAD ovvero un puntatore all'ultimo commit fatto.
- la staging area ovvero l'area inizializzata con il puntatore HEAD dei file.

2.3 Repository Creation

Il comando per creare una repository di lavoro è:

> git init

In questo modo si va a creare una nuova repo vuota dove potremo iniziare a lavorare.
con > git status possiamo avere informazioni sulla repository appena creata.

2.4 Add Files

I file possono essere Tracked(fanno parte del repo) o Untracked(non fanno parte del repo ma sono nella working directory).

> git add < $filename$ >
> git add . adds all file

Modificare il file .gitignore per specificare i file che non devono essere aggiunti.
I file che aggiungo faranno parte del prossimo commit.

2.5 Commit

Un file in un commit può anche rimanere uguale al commit precedente.
Un commit si esegue con:

> git commit

2.6 Linear Workflow

E' il modo di lavorare con git più semplice:
- Si crea la repo
- si fa il primo commit
e così via lavorando su tracked files.

2.7 Branching

Con branch si intende uno sviluppo in parallelo di un progetto.
Con il branching si viene a creare un nuovo puntatore ad un commit che andrà quindi a creare una nuova versione indipendente di un progetto che prenderà unastrada diversa da quello originale.
Per creare un nuovo branch:

> git branch < $NewBranchName$ >

Per cambiare da un branch ad un altro:

> git chekout < $destination$ >

Questo comando cambia la posizione di HEAD.
Per creare una nuova branch e andarci direttamente a lavorare:

> git checkout -b < $NewBranchName$ >

Per fare una lista dei branch esistenti:

> git branch con opzioni –no-merged e –merged

Per cancellare un branch:

> git branch -d < $NomeBranch$ >

2.8 Merging

In alcuni casi vogliamo unire lavori paralleli in un unico branch.

> git merge b0 //mentre siamo in b1

Prende b0 e b1 cercando il loro predecessore comune A e crea una nuova versione δ basata su A con b1 e b0 facendo poi commit.
Se lo stesso file è stato modificato sia in b0 sia in b1 allora c'è un conflitto.
Git non può risolverlo ed è qindi compito dell'utente trovare una soluzione.

2.9 rebasing

> git rebase b0

Prende b1 e lo unisce a b0 non creando un nuovo commit ma mantenendo b0.
Tutti i commit precedenti a b1 nel suo branch diventano inaccessibili e saranno quindi eliminati.

2.10 Amending Commits

L'ultimo commit fatto può essere modificato se ci siamo accorti che mancava qualcosa.

> git commit –amend $[-m < NewMessage >]$

In questo modo modifico il commit e posso anche aggiungere un messaggio.

2.11 Reset

> git reset $< Commit >$

Cambia la branch corrente con commit.
Possibile perdita di file.
Possibile anche specificando un path.

2.12 Cloning a repo

Necessario se vogliamo una cpoia locale del progetto (ci serve per lavorare sul nostro computer in locale).

> git clone < Url >

Prende la copia del commit puntato da Head e crea anche un Head locale che lo punta.

2.13 Fetching a repo

Serve per aggiornare la copia locale dei file alla versione corrente.

> git fetch [$Remote$]

2.14 Pushing a repo

Quando vuoi inserire il tuo lavoro locale sul repo online è il momento di usare push.

> git push [< $remote$ >][< $branch$ >]

2.15 Implementazione Git - Github

E' una piattaforma che utilizza git alla base ma ha molte altre funzionalità. E' presente ad esempio la possibilità di fare fork dei progetto ovvero clonare la repo sul proprio account github e modificarla in modo indipendente rispetto al creatore originale.

3 CSharp - Sintassi

3.1 Type Var

Il tipo Var è molto utile perchè permette di omettere nella dichiarazione di una variabile il tipo di quest'ultima che viene invece dedotto dal compilatore. Il suo utilizzo non è solo "comodo" per risparmiare linee di codice ma in alcuni casi è addirittura necessario.

```
List<string> lls = new List<List<string>>();
lls = new List<List<string>>();
```

3.2 GET/SET

I metodi get e set possono essere specificati e utilizzati trammite la sintassi di accesso ad un campo.
Sembrano accessi a campi normali ma sono invece dei getter o setter.
I due metodi possono essere dichiarati in un modo molto semplice

public int I { get; set; } = 37;

In questo modo vengono auto-creati dal compilatore.
Ovviamente possiamo creare i nostri metodi senza lasciarlo fare al compilatore.
Possiamo anche non specificarli o specificarne uno solo.
Altro modo di definirli:

```
class Person
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

3.3 Overload degli operatori

Un tipo definito dall'utente può eseguire l'overload di un operatore CSharp definito in precedenza (+,-,* ecc.ecc).
Si usa la parola chiave **operator** e devono sempre essere definiti come *public static*
Alcuni operatori sono "collegati" quindi è buona norma fare l'overloading di entrambi.
Ad esempio se eseguiamo l'overloading di == dobbiamo fare la stessa cosa per equals() e GetHashCode()
La sintassi è quella normale per l'override unita all'operatore operator

public static C operator + (C x, int y)
return null;

Il tipo sorgente o destinazione deve corrispondere al tipo che contiene la conversione

3.4 Tipo Dynamic

Corrisponde al tipo staticamente dinamico ovvero che richiede il binding dinamico.
Alla fine diventa **Object** ma è comunque molto elegante.

E' totalmente diverso da var

E' un tipo statico ma ignora il controllo di tipo statico.
Nella maggior parte dei casi funziona come Object ma non sempre.

Se, ad esempio, il metodo di istanza exampleMethod1 nel codice seguente dispone di un solo parametro, il compilatore riconosce che la prima chiamata al metodo, ec.exampleMethod1(10, 4), non è valida perché contiene due argomenti. La chiamata genera errori di compilazione. La seconda chiamata al metodo, dynamic-ec.exampleMethod1(10, 4), non viene controllata dal compilatore poiché il tipo di dynamic-ec è dynamic. Non viene, pertanto, segnalato alcun errore del compilatore. L'errore, tuttavia, non viene ignorato indefinitamente.

```
ExampleClass ec = new ExampleClass();

dynamic dynamic-ec = new ExampleClass();
```

4 Delegate, Lambda e Eventi

4.1 Delegate

Un tipo delegate ha solo un costruttore e metodi di invocazione.
E' quindi un tipo che incapsula in modo sicuro un metodo.
Può essere visto come un puntatore ad una funzione in C o C++ ma in questo caso sono orientati ad oggetti.
Iniziamo con la parola chiave "delegate" che è essenzialmente quello che si userà per lavorare con i delegate.
Il codice che il compilatore genera quando si usa la parola chiave delegate eseguirà il mapping alle chiamate ai metodi che richiamano i membri delle classi Delegate.
Per definire un tipo delegato si usa una sintassi simile alla definizione di

una firma di metodo. È sufficiente aggiungere la parola chiave delegate alla definizione.

public delegate int Comparison< inT >(T left, T right);

Il compilatore genera una classe, derivata da System.Delegate che corrisponde alla firma usata (in questo caso, un metodo che restituisce un valore integer e ha due argomenti).
Il tipo di quel delegate è Comparison.

Dopo aver definito il delegate è possibile creare un'istanza di quel tipo.

public Comparison< T > comparator;

ESEMPIO DELEGATE:

```
public static void HelloWorld(string message){
Console.writeline(message); }
Del Handler = HelloWorld;
Handler("ciao");      OutPut = "ciao"
public static void aux(int a, int b, del callback)
      callback("ciao" + (a+b).tostring());
aux(1,2,handler);      OutPut = "ciao 3"
```

La dichiarazione di un metodo delegate estende la classe di sistema delegate. ogni delegato con cui si lavora è derivato da MulticastDelegate. Un delegato multicast significa che si possono richiamare più destinazioni di metodo quando la chiamata è effettuata attraverso un delegato.

4.2 Espressioni Lambda

Le espressioni lambda servono per creare funzioni anonime.
Ogni espressione anonima lambda può essere convertita in delegate.
Il tipo delegato in cui è possibile convertire un'espressione lambda è definito dai tipi dei relativi parametri e del valore restituito. Se un'espressione lambda non restituisce alcun valore, può essere convertita in uno dei tipi delegati Action, altrimenti può essere convertita in uno dei tipi delegati Func.
Una espressione lambda può essere x → x * x e il suo tipo delegate
Func <int, int> square = x → x * x ;

Le espressioni labda sono definizioni di funzioni inline molto simili alla sintassi di ocaml.

(int x, int y) => x + y
rappresenta la funzione che somma x e y

4.3 Multicast Delegate

E' un'estensione dei delegate.
Può incapsulare più metodi delegate e la chiamata del multicast delegate corrisponde alla chiamata simultanea di tutti i metodi incapsulati.
Ogni oggetto multicast delegate ha una invocation list associata.
I metodi vengono eseguiti uno dopo l'altro seguendo la lista e se uno lancia un'eccezione quelli dopo non vengono eseguiti.

5 Metadati e Reflection

5.1 Introduzione

I metadati sono semplici annotazioni che vengono associate al codice.
Servono per generare del codice che influenza il modo in cui il nostro programma viene eseguito.
Sono affini come idea ai modificatori public e private (modificano il modo in cui il codice viene generato ma non la struttura del codice stesso).
Cambia solo il comportamento del compilatore su quel determinato codice.
In java si usa la chiocciola (@override).
In CSharp si usano le quadre e i metadati sono già predefiniti nella classe System.Attribute.
Se voglio si può ampliare con cose create da me.
Se vogliamo definire una classe attributo che estende Attribute dobbiamo chiamarlo con un nome che finisce con Attribute.
Alcuni attributi influenzano il comportamento del compilatore (ad esempio conditional).
(non su come esegue ma proprio come crea il codice eseguibile)
Tutte le classi utilizzate come metadati sono salvate dentro l'assembly e sono quindi recuperabili a runtime tramite reflection.

```
[AttributeUsage(AttributeTargets.Method|
AttributeTargets.Class,AllowMultiple = true,
Inherited = false)]
public class AuthorAttribute : Attribute {
private readonly string AuthorName;
public AuthorAttribute(string authorName) {
this.AuthorName = authorName;}
public string AuthorEmail { get; set; }
public string GetAuthorName() {
return this.AuthorName; } }
[Author("arthur")]
[Author("ford")]
public class Program {
[Author("marvin", AuthorEmail =
"marvin@lifesucks.org")]
public static void Main(string[] args) { }
}
```

Un esempio di attributo giocattolo.
L'attributo estende la classe Attribute e poi può essere usato come specifica nelle classi.
Ad esempio in questo caso la classe Program viene arrichita con l'attributo che specifica chi è stato a crearla.

5.2 Reflection and Introspection

Si intende la capacità di un programma si osservare se stesso e modificarsi di conseguenza nella struttura e comportamento.
È molto comune nei linguaggi dinamici.
La reflection di java e CSharp è una reflection in sola lettura quindi ha più senso chiamarla Introspection.
Possiamo solo leggere le informazioni sul codice ma non può cambiare dinamicamente il comportamento.
Come funziona la reflection?
All'interno di dotnet la compilazione è molto simile alla compilazione normale.
Le unità di esecuzione sono assembly.
Gli assemby possono essere composti da uno o più moduli (solitamente in visual studio uno).
Per utilizzare la reflection ovviamente il codice assembly deve essere in memo-

ria.
A livello di system abbiamo il tipo type che contiene le informazioni minimali dell'assembly.

Type type = typeof(program);

Crea una variabile di tipo type che contiene il tipo della classe program.

Object[] attributes = type.GetCustomAtrributes(false).

In questo modo posso ottenere tutti i custom attributes definiti nella classe program.
La reflection ha senso quando vogliamo utilizzare delle classi che mentre eseguiamo non hanno staticamente il loro tipo noto.
Bisogna qundi caricare un .dll che ci permette di specificare i tipi dei custom attribute.

6 Unit Test base

6.1 Introduzione

Uno unit test è un pezzo di codice che serve per vedere se un altro pezzo di codice fa quello che dovrebbe fare.
Controlla quindi se un metodo funziona correttamente.
Se il metodo funziona correttamente lo unit test ha successo e fallisce altrimenti.
Esistono diversi livelli di Test dal più generico al più generale.
Non è da sempre che si usano gli unit test.
Eseguire il codice per vedere se funziona è chiaramente un metodo sbagliato e impreciso di testing.
Con lo unit test l'esecuzione è automatica.
Framework: aiuta a costruire i test
Test runner: esegue i test e restituisce i risultati
Ogni singolo test deve essere totalmente indipendente dagli altri così da, se fallisce, capire dov'è il e in modo tale da poter essere eseguiti in parallelo tra loro.
Sono test semplici e veloci da creare e se sono scritti bene sono anche una sorta di documentazione in forma di codice.

6.2 Test Driven Developement

Metto prima i test e poi scrivo il codice.
Viene scritto un test e poi implementato il metodo che il test sta controllando.
In questo modo si ottiene codice molto più sicuro e ordinato.
Quando tutti i test passano il codice è pronto.
Se non li passa si va a correggere.
Il refactoring è la modifica di un frammento di codice che non ne modifica le funzionalità ma lo rende più pulito e preciso.
Il refactory nella programmazione è essenziale per favorire la leggibilità del codice.
È importante aggiornare i test nel caso in cui ci sia refactory.

6.3 Testing Framework

Ci sono moltissimi testing framework.
Noi Usiamo NUnit.
Nunit è utilizzato in molti linguaggi diversi ed è supportato da Resharper.
Ci sono convenzioni di struttura dei test.
Per ogni class library vogliamo avere almeno un project che testa quella libreria.
Per ogni classe che ho nella libreria voglio una classe che la testa nel progetto dei test.
Infine per ogni metodo ci devono essere molti metodi di testing.
Il nome del test deve essere qualcosa di significativo per capire cosa sto andando a testare.
In NUnit abbiamo dei custom attribute che ci permettono di utilizzare il testing.
[TestFixture] serve a specificare che in quella determinata classe possono potenzialmente esserci dei metodi che sono test.
[Test] specifica che quel determinato metodo è un Test e deve essere eseguito come tale.
Resharper e visual studio riescono a riconoscere i test grazie all'attribute e li evidenziano con delle visualizzazioni grafiche come sfere o insiemi di tre sfere che possono essere cliccate per eseguire il codice.
Per usare un test runner le classi e metodi di test devono essere public e avere un costruttore senza parametri.
In più ovviamente i test devono avere valore di ritorno void perché non ha senso che abbia il valore di ritorno.

6.4 Struttura unit test

SETUP: creare tutti gli oggetti coinvolti e inizializzarli perché altrimenti i test non possono fisicamente funzionare.
CALL UNDER TEST: invoco quello che il test deve controllare.
ASSERT: devo controllare che il metodo chiamato restituisce il risultato voluto.
TEAR DOWN: (opzionale) se abbiamo preso risorse rilasciarle È opzionale perché solitamente non è necessario ma se ad esempio utilizziamo una base di dati si.
Il test runner di resharper solitamente se ne accorge e chiede di eliminarle ma in ogni caso bisogna rilasciarla autonomamente.

6.5 Assert

Ci sono diversi stili per utilizzare l'assert .
Quello classico è invocare un metodo statico della classe Assert per ogni asserzione.
Non ha senso perché dovrebbero esserci troppi metodi in assert.
Quello più nuovo dice che l'unico metodo statico di assert che ha senso utilizzare è .That() che prende due parametri: il risultato vhe mi serve e la chiamata che mi deve restituire quel risultato.

```
Assert.That(returnValue, Is.EqualTo(42));
```

Il costraint (secondo parametro) è molto leggibile perché ha la forma di una frase in inglese.
Tecnicamente è un istanza di una classe che implementa l'interfaccia di IResolveCostraint<>.
Posso anche ampliare l'interfaccia ma non è mai il caso di farlo.
Si è ottenuto questa sintassi molto leggibile grazie all'introduzione di classi helper che ci permette di collegare i valori dell'asserzione con verbi che sono le classi.

ES: is e i loro metodi che vanno proprio a creare delle frasi quasi di senso compiuto.

Quante asserzioni ci possono essere in un test? Quante ne vogliamo.

Se mettiamo due asserzioni in un test e la prima fallisce la seconda non viene neanche eseguita.
Sotto questo punto di vista allora servirebbe un'unica assert per test.
In questo caso avrebbe senso utilizzare un test parametrico ovvero scrivere un unico test e utilizzare un custom atribute [TestCase()] che ci permette di creare test con gli argomenti di testcase basati su un test già scritto e compatibile con quei argomenti.
Con questo attributo i test vengono creati a load-time da NUnit.

```
[TestCase("42", 42)]
[TestCase("0", 0)]
[TestCase("1", 1)]
public void ParserParseValidArg(string a, int r){
int res = parser.Parse(a);
Assert.That(res, Is.EqualTo(r));}
```

L'attributo test case quindi definisce un metodo passando i parametri.
Ci sono altri metodi come l'utilizzo di sequential:

```
$[TestCase, sequential]$ \\
public void ParserParseValidArg ($[Values("42", "0", "1")]
string a, [Values(42, 0, 1)]int r)
{Assert.That(parser.Parse(a), Is.EqualTo(r));}
```

//equivalente alla versione sulla slide precedente }
Ci sono tanti altri casi come l'utilizzo di range o valori randomici.
Usare le cose random sono molto utili per trovare errori inattesi di cui non ci eravamo resi conto ma non è un modo ragionevole per fare tutto il debugging.
In ogni caso possiamo avere avere più assert in un test attraverso il costrutto

Assert.Multiple(() => Assert.That(...); Assert.That(...);)

Prende come argomento un delegate senza nessun argomento(perché deve essere eseguito dal test runner) e come corpo della lambda la nostra serie di assert.
Questo è l'unico modo sensato di fare più asserzioni in un unico metodo di test.
Dal punto di vista tecnico in questo modo risolve il problema ma dal punto di vista logico è sempre molto poco preciso e difficile da leggere.

Ci sono anche particolari casi in cui ci stiamo aspettando un eccezione.
Possiamo testare che un metodo sollevi un eccezione del tipo giusto in un determinato momento.
Questo è possibile attraverso la classe statica di costraint Throws con i metodi .TypeOf<> e .IstanceOf<>. Il primo è specifico su un tipo di eccezione mentre il secondo tollera tutte la famiglia di quella eccezione.
Per chiamare un metodo che genera una eccezione è necessario usare una lamba.

Assert.That(
()=>new Parser().Parse(null), Throws.TypeOf<ArgumentNullException>);

Ogni test deve allocare e rilasciare le proprie risorse.
Molto spesso test simili hanno bisogno delle stesse risorse.
La cosa ovvia è creare metodi ausiliari che creano e poi distruggono quelle risorse.
Esistono particolari custom attribute per questo che generano un meccanismo automatico al posto di chiamare metodi in ogni test.
I metodi annotati con $[SetUp]$ e $[TearDown]$ vengono aggiunti dal framework in ogni test di quella classe di test.
Se una classe di test eredita da un'altra? Vengono ereditati anche i setup e i teardown ma ovviamente funzionano tutte le regole dell'erdità per le quali se nella classe figlio viene creato un nuovo setup più specifico ovviamente viene usato quello.
$[OneTimeSetUp]$ e $[OneTimeTearDown]$ vengono eseguiti solo prima dell'esecuzione del primo test.
$[Ignore]$ permette di ignorare un determinato test.
Ha senso per il momento come se commentassimo una classe per poi affrontarla più avanti.
$[Ignore("Incomplete - test")][Test]$ public void Foo(){};
È poi anche possibile associare categorie ai test con $[Category]$ e si può anche eseguire solo determinate categorie di test.
Può essere utile per ragruppare test molto lenti (come quelli del bd) che non voglio eseguire sempre ma solo in determinate situazioni.
Le classi test sono in un assembly diverso rispetto a quello delle classi testate.
La prima soluzione è aggiungere i test al codice che sto testando così che siano nello stesso assembly.
Pessima idea perché quando disrìtribuisco il codice do anche il test.
L'altro metodo è usare un attributo che mi permette di rendere visibili le cose di un progetto ad un altro.

[assembly:InternalsVisibleTo("TAP-UnitTesting.Tests")]
Come faccio a sapere che ho testato abbastanza?
Ovviamente se non ho pezzi di codice che non sono stati eseguiti da nessun test non abbastanza.
Se non arriviamo a testare tutto il codice ci può essere qualsiasi cosa.
Visual studio analizza automaticamente il code coverage evidenziando le parti di codice pericolose perché non ancora testate.

7 Dependency Injection

7.1 Introduzione

Esempio, invio mail:

```
public class EmailSender{
public bool SendEmail(string to, string body) {
if (to == null)
throw new ArgumentNullException(nameof(to));
if (body == null)
throw new ArgumentNullException(nameof(body));
// ...send mail and return true
return false;} }
```

Uso EmailSender per creare BulkEmailSender:

```
public class BulkEmailSender {
private readonly EmailSender emailSender;
private readonly string footer;
public BulkEmailSender(string footer) {
this.emailSender = new EmailSender();
this.footer = footer; }
public void SendEmail(List<string> addresses,
string body) {
if (addresses == null)
throw new ArgumentNullException(nameof(addresses));
if (body == null) throw new
ArgumentNullException(nameof(body));
```

```
foreach (var a in addresses) {
if (!this.emailSender.SendEmail(a,
body + this.footer))
throw new Exception("Cannot send email"); } } }
```

Possibili problemi:
Se non ho ancora EmailSender o se non funziona, dipende da un DB o dall'esistenza di una particolare cartella nel filesystem oppure se il cliente volesse usare un'altra componente per spedire le singole email.
Un altro possibile problema sarebbe "Come possiamo testarla?", ad esempio se non vogliamo spammare durante il testing.
Il collegamento tra EmailSender e BulEmailSender è a livello di classi.
Usare il nome di una classe come tipo crea un legame (troppo) forte, infatti non si può distribuire/riusare (facilmente) una classe se dipende da un'altra oppure sostituire un'implementazione con un'altra.
Per risolvere il problema possiamo usare un'interfaccia ma come si creano gli oggetti? Usiamo delle factory (meno ovvio di quello che sembra, a versione "classica" a singleton, con costruttore privato, campo statico TheInstance NON va bene).
Non vogliamo/possiamo usare il costrutto "new" perchè non vogliamo accoppiare due classi che abbiamo definito noi, con le classi di sitema va bene! Voglio evitare dipendenze da cose definite dall'utente.
Vari approcci per evitare new:

- Factory (tipicamente un Singleton statico)
- Service locator
- Dependency injection

7.2 Factory Class

Minimizza i cambiamenti (una sola new, invece di tante sparse qua e là), ma non risolve alcune cose, infatti:

- Le classi dipendono dalla Factory invece che dalle implementazioni, ma lei dipende da loro... per transitività siamo al punto di partenza
- E' statica (se non lo fosse, chi la istanzierebbe e come? Si sposterebbe solo il problema...)
  il singleton non static va benissimo...

Il fatto che sia statica vuol dire che va bene per tutti

7.3 Service Locator

Istanzia qualunque interfaccia giri all'interno del sistema.
Scarica la responsabilità di istanziare gli oggetti su un oggetto, il Service Locator (o Registry, Context, Manager, Environment, ...), che si passa alle classi che devono "istanziare interfacce".
Le vere dipendenze sono nascoste dal fatto che ogni classe dipende dal Service Locator.
E' possibile, ma difficile il testing: cosa ridefinisco/modifico nel Service Locator per creare gli stub?
Se SL è una classe, praticamente ogni classe dipende da lei, che sa istanziare tutte le altre classi, quindi ogni classe dipende da tutte le altre... (peggio possibile).

7.4 Dependency Injection

Esistono vari tipi di DI, consideriamo la constructor injection (è quella da usare "di default") che "inietta" tramite i costruttori (Altre sono la method, property, field,...).
L'idea è estremamente semplice:
i costruttori richiedono gli oggetti che servono direttamente, invece di crearli.

```
public BulkEmailSender(string footer){
this.emailSender = new EmailSender();
this.footer = footer;}
public BulkEmailSender(IEmailSender emailSender,
string footer){
this.emailSender = emailSender;
this.footer = footer;}
Ricevo l'oggetto perche' e' un parametro
del costruttore.
Se voglio usare la factory:
this.emailSender = factory.CreateNew();
Dove:
interface IEmailSender {
bool SendEmail(string to, string body);}
interface IEmailSenderFactory{
IEmailSender CreateNew();}
// e, per esempio, una factory    :
```

```
class EmailSenderFactory : IEmailSenderFactory{
public IEmailSender CreateNew(){
return new EmailSender();} }
```

Facendo così diventa facile la sostituzione di un implementazione con un'altra e di conseguenza il testing è più facile ed esende le funzionalità, ad esempio tramite Decorator.
Esempio di Decorator:

```
class LoggingEmailSender : IEmailSender{
private readonly IEmailSender originalSender;
public LoggingEmailSender(IEmailSender originalSender){
if (originalSender == null)
throw new ArgumentNullException(nameof(originalSender);
this.originalSender = originalSender;}
public bool SendEmail(string to, string body){
Console.WriteLine("Sending mail to {0}", to);
return this.originalSender.SendEmail(to, body);} }
```

Prendo un oggetto e lo incapsulo così perchè ogni implementazione farà qualcosa in più e poi lo scarico sull'oggetto di partenza.
Un altro esempio: il testing, se durante un assert trovo un bug questo potrebbe essere ovunque nel sistema.
Per risolvere questa situazione uso degli stub/mock, cioè sostituiamo i sottosistemi (diretti) con dei fake semplificati che non possono sbagliare.
Se il costruttore istanzia gli oggetti o gli oggetti sono dei Singleton (statici)...
Come posso sostituirci degli stub/mock?
Per testare un metodo (d'istanza) la classe che lo contiene va istanziata.
Se il costruttore "fa troppo" può diventare difficile o troppo lento (connessione di rete, DB...).
La presenza di inizializzatori/campi statici peggiora le cose:

- Globali alla VM (in realtà all'Application Domain), non all'applicazione
  Ogni unit-test deve, logicamente, istanziare una (porzione) di nuova app.
- Devono fare/contenere cose diverse durante il testing
- L'ordine di esecuzione diventa importante
  I test non possono essere lanciati in parallelo!

La responsabilità di creare degli oggetti va separata!
Nei costruttori, invece di istanziare gli oggetti che servono, si richiedono tali

oggetti (Disinteressandosi sul come vengono istanziati) e si richiedono solo gli oggetti che servono direttamente (quelli che vengono copiati nei campi).
Idealmente, il corpo di un costruttore dovrebbe essere una sequenza di assegnazioni.
Ricevere x e poi fare this.f = x.qualcosa... è molto sospetto.
Bisogna usare sempre il principio di minima conoscenza (Legge di Demetra).
Nel nostro esempio:

```
public BulkEmailSender(string footer){
this.emailSender = new EmailSender();
this.footer = footer;}
diventa:
public BulkEmailSender(IEmailSender
emailSender, string footer){
this.emailSender = emailSender;
this.footer = footer;}
Il testing diventa semplice:
N.B.: uso deprecato di Assert
[TestFixture]
public class TestBulkEmailSender{
[Test]
public void TestSendMailEmailSenderSucceeds(){
var bulk = new BulkEmailSender(mock-ok, "bla bla");
bulk.SendEmail(new List<string> { "a@a.com", "b@b.com" },
"hi!"); }
[Test]
public void TestSendMailEmailSenderFails(){
var bulk = new BulkEmailSender(stub-fails, "bla bla");
Assert.Throws<Exception>(() =>
bulk.SendEmail(new List<string> { "a@a.it", "b@b.it"},
"hi!"));} }
```

Nei test è ragionevole usare new del SUT, il codice di test dipende da ciò che sta testando.
Analogamente la root di un sistema farà tutte le new dei pezzetti da passare ai vari costruttori per creare i sottosistemi.
Per TAP i test saranno gli stessi per tutte le diverse implementazioni, quindi si dovranno "istanziare interfacce" (come spiegheremo nel seguito).
- DI codice più facilmente testabile perché...
- DI codice con low coupling (diminuisce accoppiamento tra sistema e sottosistemi) codice con maggiore qualità (interna)

7.5 Dubbi sulla dependecy injection

Ma se ho 10 campi il costruttore deve richiedere 10 oggetti? Non sono troppi?

- Salvo eccezioni, sì deve richiedere 10 parametri
- Possibili eccezioni: le "classi di sistema" (come le collection), si possono istanziare senza problemi (ci giochiamo la possibilità di sostituirle... solitamente non è un problema)
- Se i parametri sono troppi, è probabile che la classe abbia troppe responsabilità, ma ce le aveva già prima di usare la DI!
- La Dependence Injection rende esplicite le dipendenze, non le diminuisce e non le aumenta

Quindi, se una classe di basso livello ha bisogno, diciamo, di un logger, lo devo passare lungo tutta la catena? no.

7.6 Dependency injection Container

Siccome le dipendenze sono esplicite, la composizione degli oggetti spesso si può automatizzare, è quello che fanno i container DI.
Si "programmano" associando implementazioni a interfacce:
Via codice, max flessibilità e controllo statico sui tipi, ma la riconfigurazione richiede ricompilazione
oppure
Via file di configurazione (tipicamente XML)
E poi loro gestiscono la creazione e lo scope (singleton, factory o container).
NON è necessario usare un DI container per usare la DI.
Come, per esempio, non è necessario usare una libreria di logging per loggare (Però è comodo).
Le implementazioni non mancano: Ninject, Autofac, Castle of Windsor, Guice, NanoContainer, PicoContainer, Spring.NET, Unity, Winter4net...

8 Iteratori

8.1 Introduzione

Servono per rappresentare in maniera finita qualcosa che potenzialmente è infinito.
Non mi interessa sapere come è fatta la sequenza ma sapere come posso visitarla.
Abbiamo un tipo IEnumerable che ci permette di identificare le sequenze che ha un unico metodo GetEnumerator che restituisce un IEnumerator (il modo in cui lo voglio visitare).
Poi abbiamo IEnumertor che ha un metodo per passare al valore successivo MoveNext, l'elemento corrente current e Dispose.

```
IEnumerable<T>
    IEnumerator<T> GetEnumerator()
IEnumerator<T>
    bool MoveNext()
    T Current { get; }
    void Dispose()
```

Nella prima versione di csharp non avevano i generici quindi current restituiva Object e si aveva anche il campo reset.

8.2 Classi intervallo

Possiamo quindi creare nuovi tipi intervallo che estendono l'interfaccia IEnumerable<> e poi utilizzarlo in un foreach ad esempio.
Per creare un intervallo non ci interessa sapere tutti i valori ma solo da dove parte e dove arriva.
Il resto lo fa la classe IEnumerator<>.
Il get enumerator deve essere definito sia nella versione generica sia non.
Quello generico restituirà un nuovo enumerator con new mentre mentre quello non generico faremo un semplice return del getEnumerator generico.

public IEnumerator< *int* > GetEnumerator() {
return new Enumerator(this); }

IEnumerator IEnumerable.GetEnumerator() {

return this.GetEnumerator(); }

La classe IEnumerator invece specifica in che modo i dati all'interno dell'intervallo devono essere visitati a seconda delle nostre esigenze.

8.3 Yield

Dalla seconda release di csharp possiamo scrivere del codice che sembra un metodo che se dato al compilatore crea in automatico quello che poi è l'implementazione del IEnumerator.
Questo nuovo costrutto è lo **Yield**.
Il getenumerator generico non fa più la new a un getEnumerator ma implemento direttamente che cosa farà.

public IEnumerator< *int* > GetEnumerator() {
for (var a = this.LowerBound; a <= this.UpperBound; ++a)
yield return a; }

Si basa sul fatto che esistono gli iteratori visti come blocchi-iteratori.
Quando invoco un metodo definito con yield viene restituita la classe creata dal compilatore ma non parte direttamente l'iteratore.
Per questo non posso avere dei parametri passati per riferimento (non saprei quando vengono inizializzati).

9 Unit Test di SUT

9.1 Introduzione

Nel mondo vero un sistema di testing non è semplice come abbiamo visto in precedenza.
Il modo giusto è utilizzare i principi della dependency injection.
Un'alternativa è utilizzare Microsoft Fakes, una libreria che serve per cambiare durante il testing i sottosistemi di codice scritto male e senza dependency injection.

9.2 Stub e Mock

Stub e mock sono oggetti che, durante il testing sostituiscono gli oggetti "veri"(le dipendenze del sut) facendo il minimo indispensabile.
La differenza tra i due è che lo stub non fa altro che rispondere mentre il mock si memorizza dove e perché l'hanno chiamato.
Usiamo gli stub tutte le volte che facciamo test dove ci interessa solo il risultato finale del test e usiamo i mock per tutto il resto.
Quando utilizzo degli stub posso fare solo assert sul valore di ritorno di un metodo o sullo stato del ST.
Se utilizzo il mock posso anche fare assert su aspettative che ha il mock object (ad esempio se è stato chiamato il numero giusto di volte).
Se facciamo test basati solo sullo stato ha più senso usare stub, in questo modo i test dipendono molto meno dall'implementazione.
Anche in caso di obbligo nell'utilizzo di mock usarne pochi (di solito un mock e tanti stub).

9.3 moq

Noi utilizzeremo un mock framework (una libreria che fornisce strumenti per creare mock e stub in modo semplice)
Utilizzeremo moq che si installa tramite NuGet.
All'interno di moq c'è la classe mock generica da estendere.
Quello che si chiama mock in moq non è proprio il mock ma un creatore di mock.
Questo "mock builder" ha due metodi essenziali:

- **SETUP:** indica cosa deve rispondere il mock nei vari casi.
  Se non la usiamo il nostro mock risponderà con il valore di default (ad esempio se è bool risponde con false).

- **VERIFY:** dice come sono andate le cose alla fine. Si può usare dentro le assert.

Entrambe possono utilizzare lambda per descrivere lo scenario, di quale metodo stiamo andando a fare una verifica.
Sia nel setup che nel Verify posso usare singoli valori (singleton), it.IsAny< T > (qualunque valore) o ancora it.Is< T >(predicato) (solo i valori che rispettano il predicato).

Posso anche usare intervalli con It.IsInRange o regex con It.IsRegex.

Mb.setup(...).Returns(true); ci permette di dire che per certi valori devo restituire vero.
Possiamo anche dire che per qualche valore si deve lanciare un'eccezione.

10 LINQ

10.1 Introduzione a LINQ

Serve ad interagire con il database.
Ci sono molti metodi ma quello adottato per .net è ADO.

ESEMPIO
Scrivere cognome e data di nascita su un database.

Ci possono essere diversi problemi con la comunicazione con un database tra cui sql injection.
Dal punto di vista di csharp una query non esiste, il compilatore non la conosce e l'ide non ti da una mano a scriverla.
Per usare una query dobbiamo scriverla all'interno di una stringa e inviarla con la funzione adatta.
In realtà utilizzare le stringhe non ha senso perché rende tutto complicato.
Linq cerca di portare il concetto di query sql in un linguaggio di programmazione.
Attraverso linq il linguaggio riconosce effettivamente le nostre query e le interpreta in modo corretto.
Il compilatore riconosce anche errori di ogni tipo.
Ovviamente questo rende linq essenziale e molto meglio della soluzione di utilizzare una stringa.
Linq è solo una maniera di interrogare una sorgente dati ad oggetti.
È stato inserito tra CSharp2 e CSharp3 (ora siamo al 9).

```
String[ ] string = { g, null, disi, genova};
var query=
from s in string
orderby s descending
select s.ToUpper();
```

La sintassi è molto simile a quella di sql anche se come si può notare alcune cose sono al contrario come il select alla fine.
Il tipo che sta dietro VAR che viene utilizzato per inizializzare la query è una Ienumerable<>.
La presenza di null da problemi ma essendo una Ienumerable<> non viene eseguita in modo autonomo.
Il risultato della nostra query può anche essere salvato in un'altra variabile per non utilizzare l'Ienumerable<>.
Ad esempio presenta i metodi .toArray() e .toList().
Possiamo anche chiaramente utilizzare direttamente la query con il suo risultato.
L'utilizzo di uno o dell'altro varia a seconda di cosa vogliamo fare.
Quella di sotto è come fare caching mentre quella di sopra cambia dinamicamente in base a come cambia il db.
Questo è un classico esempio in cui Var non è più opzionale ma obbligatorio.
Il nome del tipo non è noto al programmatore.
In una query posso anche creare nuovi Ienumerable<> con new nella select.

10.2 Implementazione di LINQ

Una query prende una sequenza di dati e restituisce un'altra sequenza di dati processata.
Come rappresentiamo una sequenza:
semplicemente come già detto una IEnumerable<> ovvero un insieme di dati.
Come funziona:
Esiste una traduzione che trasforma la query in una serie di chiamate di metodo.
Non c'è assolutamente una implementazioni di sintassi particolare ma solamente una traduzione.

```
var q = /* ... un IEnumerable/IQueryable ... */
.Where(user => user.firstname == "Giovanni")
.Select(user => user);
```

Where e Select sono viste come due chiamate di funzione normali.
Questi non fanno parte di IEnumerable e gli passiamo una labda.

10.3 Extension Method

Possiamo aggiungere un metodo a una classe o interfaccia.
In questo modo possiamo arricchire classi inserendo cose che prima non c'erano.
Queste cose vengono aggiunte ma separate in modo logico dalle altre.
Il compilatore accetta il primo argomento di una chiamata statica con this per creare i nuovi metodi.

public static int m(this T x, int k, ...)

In questo modo posso usare m nella classe T come se fosse nato con questo metodo.
Where, Select ecc non sono altro che extension method.

10.4 Lambda

Le lambda NON sono delegate ma sono tranquillamente convertibili in delegate.
In Questo caso non ci servono come delegate ma come Expression Tree.
È sempre codice .net ma crea una struttura dati che ci permetti di eseguire la labda.
Oltre a IEnumerable può essere utilizzata anche una IQueryable<T>.
Si comporta e risponde a IEnumerable ma è più complicato e completo.
IQueryable estende IEnumerable e viene usato perché abbiamo anche l'informazione di chi è il provider che esegue la query.
SI possono utilizzare entrambe.
Possiamo comunque usare i metodi senza utilizzare la sintassi alla sql.
Per il compilatore è uguale quindi possiamo anche usare direttamente i metodi .where(), .Select() ecce cc.

10.5 Decorator

Ogni parte della Query è un decorator.
La query si rivolge prima alla parte più esterna (select) che a sua volta si rivolge a quella più esterna apparte lei fino ad arrivare alla fine.

10.6 IQueryable e IEnumerable

IEnumerable<> e IQueryable<> sono due sorgenti di dati.
Tutti i contesti che si aspettano un IEnumerable possono avere anche un tipo IQuerable.
IQueryable come interfaccia di fatto estende l'interfaccia IEnumerable<>.
Esiste un metodo per IEnumerable che permette di trasformarlo in un IQuerable (.asQueryable).

11 Varie ed Eventuali

11.1 Tipi per impacchettare dati

CSharp offre varie opportunità per creare tipi che fanno viaggiare assieme varie informazioni.
Esempi sono le Struct (sono value types) e i record (tipi reference).
Un altro tipo di dato è la Tupla.
Molto utile per rappresentare punti sul piano cartesiano.
Possiamo installarla attraverso Nuget e consiste in una libreria chiamata System.ValueTuple.

11.2 Compilazione Condizionale

E' possibile in CSharp anche rendere alcune parti di codice opzionali da compilare.
Per far ciò si può usare la sintassi "alla C" con ♯ if condizione e ♯endif .
E' presente però un altro metodo molto più conforme alla sintassi degli attributi di CSharp ovvero l'utilizzo di [Conditional("condizione")].

11.3 Tipi valore Nullabili

Dato un tipo T il suo corrispondente valore nullabile è T?.
Corrisponde a quel determinato tipo T ma può anche essere null.
Il tipo T quindi diventa sottotipo di T?.
Questo è molto utile per interfacciarsi ad un database perchè vogliamo quasi sempre mettere dei valori nulli.
Esistono anche i valori nullabili di tipo reference.
Un tipo reference è un tipo che, a differenza dei tipi valore normale, archiviano i loro dati nei relativi oggetti.
Per questo mentre una variabile di tipo valore è unica potrebbero esserci

due variabili di tipo reference che rappresentano lo stesso oggetto e quindi se viene modificata una può indirettamente modificare anche l'altra.
I tipi reference predefiniti di CSharp sono Dynamic,Object e String.
Semplicemente come per i tipi valore quelli nullabili presentano la dicitura "?".
Tuttavia prima della creazione in CSharp dei tipi nullabili i reference potevano già avere il valore null.
Per questo c'è stato un cambiamento retroattivo e ora i tipi reference normali danno errore se vengono inizializzati con null.
Durante la compilazione variabili, campi etc di tipo reference [nullabile] vengono tracciati per capire se sono usati in accordocon l'intentodel programmatore.
Ogni espressione di tipo [nullable] reference può essere not-null o maybe-null.
il compilatore esegue un'analisi statica sulle variabili di qualsiasi tipo riferimento, sia nullable che non nullable.
Il compilatore tiene traccia dello stato Null di ogni variabile di riferimento come not-null o maybe-null.
Lo stato predefinito di un riferimento non nullable è diverso da Null.
Lo stato predefinito di un riferimento nullable è maybe-null.
Usare un maybe-null per assegnare una variabile/campo o come ritorno di metodo di tipo reference non nullabile dà warning.
Se exp è un'espressione che può valutarsi in null.

exp.BlaBla();

rischia di sollevare NullReferenceException
La soluzione ovvia è controllare nel metodo se exp è nulla.
Un metodo migliore è usare il null conditional operator "?" nella chiamata in questo modo:

exp?.blabla();

Questo corrisponde a IsNull(exp)?null:exp.BlaBla();
Altro importante operatore è il null-coalescing operator "??" che può essere molto utile per confrontare due espressioni che possono valutarsi in null o per assegnazioni dello stesso tipo.

exp1??exp2
che corrisponde a:
IsNull(exp1)?exp2:exp1

Può essere molto utile anche per le eccezioni:

var x = exp?? throw new ArgumentNullException;

null-coalescing assignment:

x??=exp
che corrisponde a:
if(null==x)x=exp;

11.4 Passaggio per riferimento

Una variabile di un tipo riferimento non contiene direttamente i dati, ma solo un riferimento a essi.
Quando si passa un parametro di tipo riferimento per valore, è possibile modificare i dati appartenenti all'oggetto di riferimento, ad esempio il valore del membro di una classe.
Non è tuttavia possibile modificare il valore del riferimento stesso.
Ad esempio, non è possibile usare lo stesso riferimento per allocare memoria per un nuovo oggetto e per renderlo persistente all'esterno del metodo. In questo caso, è necessario passare il parametro usando la parola chiave ref o out.
"in" ha senso solo quando il passaggio per valore andrebbe bene, ma per efficienza è meglio non fare copia.

11.5 Argomenti opzionali e con nome

E' possibile dare nomi o rendere argomenti opzionali nella chiamata di un metodo.
Alcuni esempi:
public void M(int x, int y = 5, int z = 7) ....
M(1, 2, 3); // ok
M(1, 2); // equivalente a M(1, 2, 7)
M(1); // equivalente a M(1, 5, 7)
M(z: 3, x: 1); // rovesciando l'ordine...

11.6 Nameof

nameof è un'espressione che produce il nome di una variabile, un tipo o un membro come costante stringa.

Console.WriteLine(nameof(System.Collections.Generic)); // output: Generic
Console.WriteLine(nameof(List< *int* >)); // output: List
Console.WriteLine(nameof(List< *int* >.Count)); // output: Count
Console.WriteLine(nameof(List< *int* >.Add)); // output: Add

Può essere utile in alcuni casi come ad esempio verificare la correttezza dei tipi.

11.7 Switch

L'idea dello switch è risolvere il problema di una cascata di if (scelta pessima).
Ho diversi casi e a seconda del caso eseguo parti diverse di codice.
Ho anche un caso default che rappresenta il caso che non voglio specificare.
I casi vuoti non hanno uscita mentre tutti quelli che contengono codice devono avere una condizione di terminazion(return,break,...).
la sintassi è:
case pattern when guardia:
La guardia è opzionale mentre il pattern no.
Ci sono altri metodi meno classici per usare lo switch(switch expression), ad esempio:

```
Point switch{
    {X: 0, Y: 0} => new Point(0,0),
    {X: var x, Y: var y} when x < y => new Point(x+y, y),
    {X: var x, Y: var y} when x > y => new Point(x, x+y),
    {X: var x, Y: var y} => new Point(2+x, 2+y),
};
```

11.8 Pattern

Si possono usare in switch, switch expression e col costrutto is.
Ci sono molte tipologie di pattern diversi:

- Declaration Pattern: controlla il tipo e se compatibile inizializza una variabile
- Type pattern: controlla il tipo
- Costant pattern: verifica se il valore corrisponde ad una costante
- Relational pattern: controlla una relazione d'ordine
- Logical patterns: combina pattern mediante operatori logici
- var pattern:per introdurre variabili
- Property pattern: verifica che property/campi soddisfino pattern annidati
- Positional pattern: usa le posizioni per decostruire oggetti e verificare se i pezzi soddisfano pattern annidati
- Discard pattern: sempre soddisfatto

```
object o ="ghidf";
int y = o switch{
    strings=>s.Length,   //declaration
    C=>0,   //type
    null=>1,   //constant
    >=0=>2,   //relational
    2 and >0 or <7 =>3,//logical
    var d =>4,    //var
};

var p = (X:4, Y:7);
int z = p switch{
    {Y:-1,X:7}=>-1,   //property
    (6,8)=>-6,   //positional
    _ => 42,   //discard
};
```

12 Entity Framework Core

12.1 Introduzione

È un orm è uno strumento che fa in automatico la mappatura tra la traduzione di un concetto dal del database a quello del codice OO.

Ci permette di restare ad un livello più astratto.
Usando un orm lavoriamo solo nel mondo object oriented e non dobbiamo pensare alle query che vengono generate dal sistema.
Una volta fatto il design delle classi il framework crea in automatico tutti i costrutti sql (quindi di per se sql diventa obsoleto).
In più se dobbiamo creare un nuovo database perché quello vecchio non è più aggiornato alle esigenze il framework offre un sistema di migrazione dei dati che permettono di non perdere nulla.
L'uso di un framework diminuisce di gran lunga tutti gli errori più comuni.
Fino alla versione di .NET 4 l'entity framework venivano in automatico distribuiti assieme.
Oggi non più ed è anche un progetto open source dalla versione 6.0.
In parallelo è stata creata una versione diversa "core" che si rivolgeva a tutti i sistemi operativi e non più solo a windows con il nome entity framework core.
(noi usiamo entity framework core 5 anche se è già uscita la 6).

12.2 Framework

La prima parte del framework è il modello concettuale (con diagrammi come in UML).
Questo poi si traduce in codice come entità.
Solitamente si aggiunge ad ogni oggetto un intero auto incrementale che servirà come chiave primaria.
In più se due entità sono collegate chiaramente troveremo in una classe un elemento dell'altra e viceversa.
Una volta fatto questo il nostro lavoro di programmatore è quasi finito perché il resto lo farà il nostro framework.
A questo punto basterà installare il framework con NUget.
Esistono delle funzioni che ci permettono di creare o cancellare db.

```
Using (var c = new bloggingContext()){
    C.database.ensuredeleted();
    c.database.ensurecreated();
}
```

In questo modo cerco qualcosa da cancellare e poi ne creo uno nuovo.
In questo modo perdiamo tutto quello che avevamo in precedenza.
Se non vogliamo servono funzioni di migration.

12.3 Framework in particolare

Il Dbcontext è l'interfaccia verso lo strato di gestione dei dati.
Quando noi utilizziamo il framework tutto passa dal Dbcontext.
Incapsula modelli tra cui tre pattern fondamentali (gestione delle identità) che quindi vengono forzatamente seguiti:

- Modello dei dati
- Gestione del DB
- Gestione delle entità:
  - Identity
  - Repository
  - Unit of work
    (non lavoro su un singolo dato ma su una collezione e rendo le modifiche consistenti solo se vanno a buon fine per tutti gli elementi della collezione e devo tenere assieme solo dati indispensabili che non possono essere divisi).

12.4 Modello Dei Dati

Come faccio a dire al framework che schema del database voglio?
La scelta dei nomi all'interno del codice definisci cosa saranno gli oggetti.
Ad esempio se un elemento è un intero e si chiama IDqualcosa il framework deduce che questa è la chiave.
Quando le informazioni del nome non sono sufficienti bisogna dare delle informazioni esplicite.
Si può anche aggiungere esplicitamente delle ulteriori entità ma non ci interessa.
Ci sono due modi per dare indicazioni esplicite: attraverso attributi o attraverso la libreria Fluent API (utilizzabile solo nel metodo OnModelCreating).
I Validation Attribute esprimono vincoli sulle entità (sono custom attribute).
Questi vincoli vengono utilizzati per tradurli quando si può in qualcosa che modifica la creazione del db nei vincoli dei suoi elementi.
Le fluent API sono molto più espressive e hanno tutte le cose che posso fare anche con i validation attribute più altre vunzioni.
Volendo si può solo usare fluent API.

[Maura Cerioli, Slide del corso TAP]

[Documentazione Microsoft Docs su CSharp]

[Libro Csharp]