ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Ingegneria del Software T

UTILIZZO DI ROSLYN PER L'ANALISI E IL REFACTORING DI CODICE C#

CANDIDATO:
Francesco Bonacci

RELATORE:
Chiar.mo Prof. Ing. Giuseppe Bellavia

Anno Accademico 2015/16

Sessione I

INDICE

INTRODUZIONE **1**

1. .NET COMPILER PLATFORM **3**

1.1. PIPELINE 4
1.2. ROSLYN AS A PLATFORM 5
1.3. IMMUTABILITÀ 6
1.4. PROGRAMMAZIONE ASINCRONA 7
1.5. STRATI DELLE API 9
1.4.1 Compiler API *9*
1.4.2 Workspace API *10*
1.4.3 Features API *11*
1.6. LIVELLO SINTATTICO 12
1.5.1 Syntax Tree *12*
1.5.2 Syntax Node *14*
1.5.3 Syntax Token *16*
1.5.4 Syntax Trivia *16*
1.5.5 Span *17*
1.5.6 Kind *17*
1.5.7 Syntax Factory & Syntax Generator *18*
1.5.8 Syntax Visitor *20*
1.7. LIVELLO SEMANTICO 24
3.1.1 Compilation *24*
3.1.2 Semantic Model *26*
3.1.3 Symbol Visitor *27*

2. ANALYZER WITH CODE FIX **29**

2.1. STRUTTURA DEL PROGETTO 30
2.1.1 File di Analisi *31*
2.1.2 File di Code Fix *34*
2.2. ANALISI DEI VAR 38
3.2.1 Specifiche del problema *38*
3.2.2 Soluzione *40*

3. CODE REFACTORING **48**

3.1. STRUTTURA DEL PROGETTO 49
3.1.1 File di Refactoring *50*
3.2. REFACTORING DEI METATYPE 53
3.2.1 Specifiche del problema *53*
3.2.2 Soluzione *55*

CONCLUSIONI **75**

BIBLIOGRAFIA **76**

APPENDICE ..... **I**

PROGETTO 1 I
PROGETTO 2 VI

INTRODUZIONE

Fin dalla pubblicazione dei primi sistemi integrati di sviluppo la preferenza della comunità di sviluppatori verso uno dei tanti è stata dettata da svariati fattori quali la popolarità del sistema nel periodo, i linguaggi supportati, il tipo di licenza per l'utilizzo e la strumentazione disponibile. Negli ultimi anni, data la sempre più esigente necessità da parte dei clienti di vedere sviluppate applicazioni in tempi brevi, e quindi con il minor costo possibile, sta diventando influente nella scelta del sistema anche la disponibilità di tool per l'autogenerazione di codice. Un popolare esempio tra tali strumenti è il sistema IntelliSense di Visual Studio che, a partire dalla versione 2015, sfrutta i servizi offerti dalla .NET Compiler Platform.

L'obiettivo della tesi vuole essere lo studio delle potenzialità della suddetta piattaforma e l'applicazione dei suoi servizi per lo sviluppo, in linguaggio C#, di tool integrati in Visual Studio. La tesi si pone pertanto nel contesto delle tecniche di analisi e generazione di codice sorgente.

Durante lo svolgimento della tesi sono stati sviluppati alcuni progetti finalizzati alla comprensione della piattaforma e altri due progetti più significativi dal punto di vista dell'utilità per lo sviluppatore.

L'impianto del testo è stato organizzato in tre capitoli.

Il primo capitolo introduce la .NET Compiler Platform e ne descrive esaustivamente funzionalità e servizi offerti. Il capitolo è diviso in sette sezioni in ordine di propedeuticità degli argomenti. Ogni sezione riporta esempi e figure utili a favorire una migliore comprensione della piattaforma.

Il secondo capitolo concerne lo studio del template di progetto "Analyzer with Code Fix". Nello specifico, la prima sezione descrive la struttura del progetto e analizza un semplice progetto di esempio. La seconda sezione descrive invece il primo progetto sviluppato, inerente la deduzione dei tipi impliciti C#.

Infine, il terzo capitolo analizza il template di progetto “Code Refactoring”. In particolare, la prima sezione descrive la struttura del progetto richiamando un semplice esempio. La seconda sezione riporta il secondo progetto sviluppato, riguardante un tool di ausilio per la modifica del codice sorgente nell’ambito del framework Phoenix.

Il codice sorgente completo dei due progetti sviluppati è stato riportato nell’Appendice.

1. .NET COMPILER PLATFORM

Nel passato i compilatori venivano sviluppati unicamente per tradurre il codice sorgente da un linguaggio di alto livello ad uno di più basso livello interpretabile direttamente dal calcolatore o per mezzo di un'infrastruttura di supporto. Interazioni con la fase di compilazione potevano avvenire da parte del programmatore solamente in input allo stesso senza nessuna possibilità di poter avere accesso all'elaborato intermedio prodotto dal compilatore.

Tuttavia questo approccio e visione di un compilatore come black-box risultano inadatti nel momento in cui le funzionalità di un linguaggio crescono così come le manipolazioni intermedie che portano alla produzione del codice nativo.

.NET Compiler Platform, meglio conosciuto con il nome in codice "Project Roslyn", da qui in avanti semplicemente **Roslyn**, rivoluziona il concetto di compilatore tradizionale rendendo fruibili al programmatore molte delle metainformazioni prodotte nelle varie fasi della compilazione del sorgente per due linguaggi specifici del Common Language Runtime: **C#** (versione **6.0**) e **Visual Basic** (versione **14.0**).

Queste informazioni possono essere usate per creare tool e applicazioni per la generazione e trasformazione di codice, operazione che viene comunemente chiamata **"code refactoring"**.

Da scatola nera il compilatore diventa quindi una piattaforma o servizio da consultare. Si parla in questo caso di "compiler as a platform".

1.1. PIPELINE

Come molti compilatori tradizionali, Roslyn processa il codice sorgente in input tramite una pipeline di comandi così da migliorare il throughput e formalizzare la divisione delle responsabilità in unità funzionali.

Si distinguono 4 fasi della pipeline:

1. **Parsing**: il compilatore suddivide il sorgente in **elementi sintattici**, unità logica per questo livello, propri della grammatica del linguaggio C# e Visual Basic. In altre parole viene definita quella che sarà la struttura del codice: il **livello sintattico**.
2. **Declaration**: a partire dagli elementi sintattici e dall'estrazione di dichiarazioni e metadati, vengono prodotti i **simboli**. Questi sono le unità logiche del livello e danno significato agli elementi sintattici permettendone l'identificazione e la loro correlazione attraverso riferimenti. Si parla in questo caso di **livello semantico**.
3. **Binder**: ad ogni **nome** ed **espressione** del codice sorgente viene associato l'elaborato del livello precedente cioè il simbolo.
4. **IL Emitter**: a partire da tutte le informazioni precedenti viene "emesso" l'**IL** o CIL (Common Intermediate Language).

1.2. ROSLYN AS A PLATFORM

Come accennato, Roslyn espone delle API per interagire con ognuna delle fasi della compilazione. Ciò è reso possibile grazie a specifici **object model**, elementi del framework .NET, che ne espongono proprietà e servizi.

Gli object model incapsulano rispettivamente:

1. API del **Syntax Tree**: permettono di interagire con la fase di parsing.
2. **Symbol API**: offrono una vista su metadati e simboli tramite una tabella gerarchica sotto forma di grafo.
3. **Binder e Flow Analysis API**: espongono il risultato prodotto dal binding tra identificatori e simboli.
4. **Emit API**: permettono di interagire con la compilazione del sorgente in IL e richiederne l'emissione "on demand" a partire da codice C# o Visual Basic.

Questi object model sono gli stessi utilizzati da API di livello superiore quali **Language Service** per la realizzazione di servizi tipici di IntelliSense come l'outlining e il renaming.

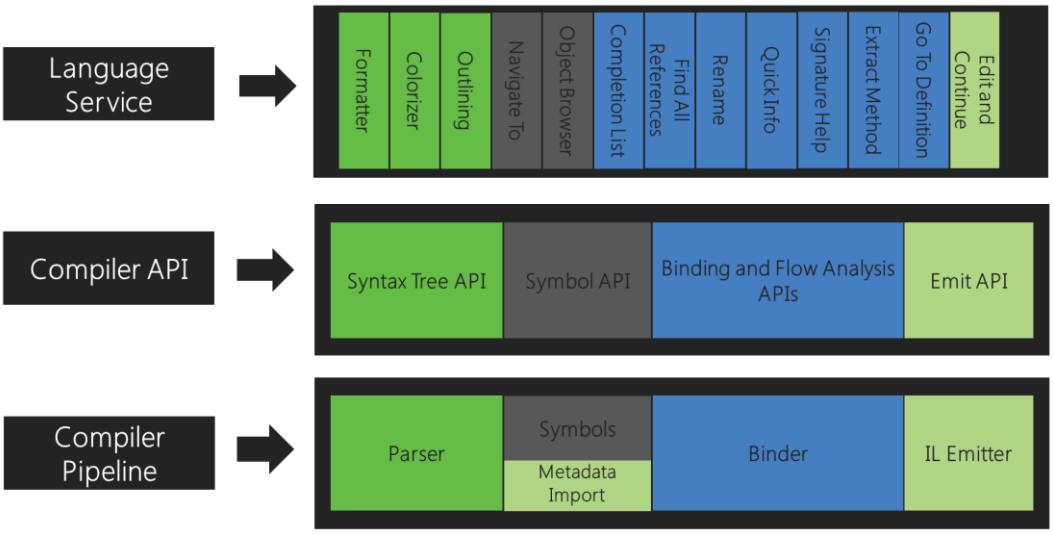

Figura 1 Ogni fase della Compiler Pipeline è accessibile attraverso le Compiler API. Queste sono usate dai servizi applicativi di Language Service.

1.3. IMMUTABILITÀ

Molte delle strutture dati racchiuse nei vari object model sono **immutabili**. Come vedremo, il concetto di immutabilità è molto importante in Roslyn in quanto garantisce di poter lavorare su una struttura dati con la garanzia che nessuna entità possa apportarne parallelamente delle modifiche.

Pertanto, molti dei servizi che manipolano una struttura dati in Roslyn, lo fanno creandone una nuova, operazione che causa overhead in termini di impiego di risorse ma che apporta anche dei vantaggi quali la thread-safety.

La creazione di nuovi oggetti immutabili è spesso delegata a metodi **Factory** che ne facilitano la creazione nascondendo i bassi dettagli implementativi della struttura.

Infine, molti degli oggetti immutabili espongono metodi con prefisso `With` che permettono la costruzione di un nuovo oggetto a partire dal precedente e con nuove caratteristiche.

1.4. PROGRAMMAZIONE ASINCRONA

Pur non essendo la programmazione asincrona stata introdotta con Roslyn, molte delle nuove API, specialmente quelle che hanno a che fare con feedback di risposta all'utente, sfruttano il pattern asincrono per l'esecuzione di servizi.

La programmazione asincrona è stata introdotta per rendere più performanti le chiamate a servizi tipicamente I/O bound quali operazioni di lettura e scrittura da file, di accesso al Web e in generale qualsiasi tipo di operazione possa impiegare più di 50 millisecondi per restituire il controllo al chiamante.

Il pattern asincrono riveste particolare importanza quando il contesto di sincronizzazione di un metodo è la **message pump** (e dunque risiede nel thread della UI). Infatti se si eseguisse una tipica operazione I/O bound sul thread della UI, quest'ultima rimarrebbe insensibile agli input dell'utente per tutta la durata del servizio. La programmazione asincrona risolve questo problema congelando, all'invocazione di un metodo asincrono, l'attività corrente (**Task**), salvando il contesto di sincronizzazione corrente e restituendo il controllo al chiamante. Una volta che l'operazione legata all'I/O termina, il Task viene "riesumato" e continua il suo flusso di esecuzione nello stesso contesto di sincronizzazione catturato in precedenza (il Thread Pool o la Message Pump). Infine il Task termina restituendo il risultato effettivo al cliente. In questa maniera, il Thread del chiamante (eventualmente il Thread della UI) non viene mai bloccato. Se ad esempio il chiamante fosse un controllo della UI, quest'ultima rimarrebbe sensibile ad altri input dell'utente anche per tutta la durata del servizio.

Tutti i metodi asincroni nativi in .NET antepongono nella firma del metodo la keyword `async` e restituiscono un `Task` o `Task<T>` (del namespace `System.Threading.Tasks`) rispettivamente se l'operazione termina senza valori di ritorno o meno. In questo secondo caso `T` rappresenta il tipo del risultato.

Esiste una sola eccezione a questa seconda regola. Infatti, per supportare la piena compatibilità degli event handler con il codice asincrono, è permesso l'utilizzo di `async void` anziché `async Task`. In tutti gli altri casi l'utilizzo di `async void` è sconsigliato in quanto se si presentassero eccezioni non catturate nel corpo di un metodo asincrono, non

esistendo alcun contesto di sincronizzazione catturato per un metodo che non restituisce un Task, le eccezioni non gestite “rimbalzerebbero” direttamente nel Thread Pool di sistema causando la terminazione del processo.

Dall’altra parte, il cliente che vuole invocare un metodo asincrono deve anteporre la keyword `await` all’invocazione dello stesso. In caso contrario l’operazione procede in modo sincrono.

È inoltre importante capire che un metodo asincrono non implica necessariamente la creazione di un nuovo Thread, cosa che è gestita automaticamente dal Thread Pool di sistema in base alle caratteristiche dell’host. Tuttavia, è possibile garantire un minimo di parallelismo tra Task chiamando `ConfigureAwait(false)` (a default `true`) sul Task asincrono e anteponendo normalmente la keyword `await`. In questa maniera, quando il Task viene congelato, non viene catturato il contesto di sincronizzazione corrente. Pertanto, una volta che il Task è stato riesumato, il flusso di esecuzione può procedere su qualsiasi contesto di sincronizzazione (il Thread Pool o la Message Pump). Questo meccanismo consente inoltre di evitare deadlock poiché, se il contesto di sincronizzazione fosse la Message Pump, non costringerebbe il Task ad essere riesumato sulla stessa, che potrebbe essere congelata a causa di un’operazione bloccante sul Task come `Wait()` o `Result`.

1.5. STRATI DELLE API

Oltre alle già citate Compiler API, che inglobano le varie fasi della compilazione, Roslyn mette a disposizione degli sviluppatori anche le **Workspaces API** e **Features API**.

Le API offerte da Roslyn sono tutte contenute nell'assembly `Microsoft.CodeAnalysis`.

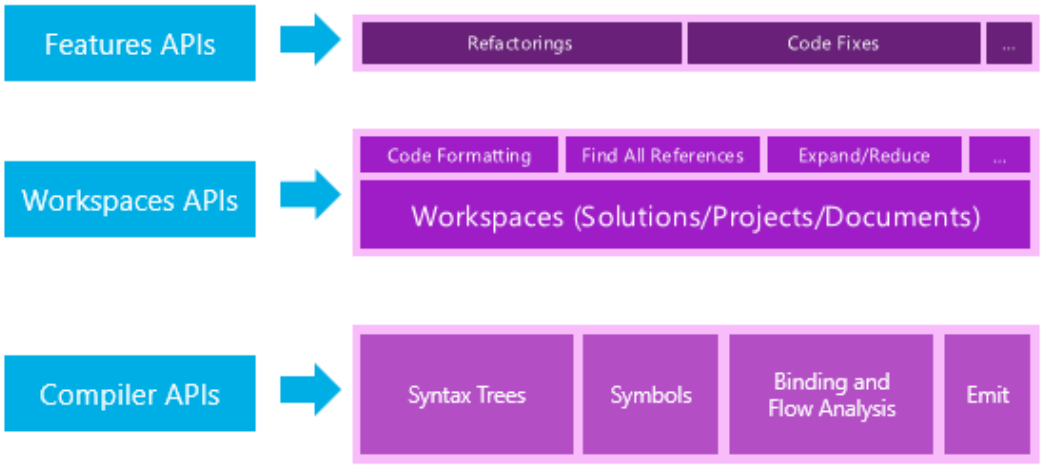

Figura 2 In basso le già citate compiler API. Al centro le Workspace API per la gestione di Solution e Project. In alto le Features API utilizzate nei template di progetto Code Refactoring e Analyzer.

1.4.1 Compiler API

Le Compiler API espongono la struttura sintattica del codice sorgente tramite i **Syntax Tree** e quella semantica per mezzo del **Semantic Model**. Quest'ultimo offre una istanza immutabile della compilazione tramite cui accedere a file sorgenti, opzioni e riferimenti agli assembly. Essendo queste API strettamente dipendenti dal linguaggio, esse sono esposte da object model diversi per C# e Visual Basic.

Le Compiler API possono essere ulteriormente suddivise in due categorie:

1. **Diagnostic API**: permettono al programmatore di avere accesso a tutte le diagnostiche quali errori e warning, prodotte o meno in fase di compilazione, e di creare delle diagnostiche personalizzate che possono essere integrate, al pari di quelle native, in un'istanza di Visual Studio. L'**Esempio 1** riporta un frammento di codice che sfrutta le Diagnostic API per ricercare le diagnostiche aventi identificativo `CS0162` corrispondente al warning *"Unreachable code detected"*.

1. **Scripting API**: permettono di valutare ed eseguire dei frammenti di codice C# o Visual Basic sfruttando la grammatica propria del linguaggio in un ambiente di esecuzione controllato dallo sviluppatore**.** Sono esposte dalle classi `CSharpScript` e `VisualBasicScript` dei namespace `Microsoft.CodeAnalysis.CSharp.Scripting` e `Microsoft.CodeAnalysis.VisualBasic.Scripting` come riportato nell'**Esempio 2**.

```
// CS0162 - Unreachable code detected
IEnumerable<Diagnostic> diagnostics =
       compilation.GetDiagnostics();
IEnumerable<Diagnostic> unreachableCodeWarnings =
       diagnostics.Where(d => d.Descriptor.Id == "CS0162");
```

Esempio 1 Utilizzo delle Diagnostic API per trovare le diagnostiche prodotte in fase di compilazione. Nello specifico vengono trovati i warning corrispondenti al descrittore "*Codice non raggiungibile*".

```
string result = await CSharpScript.EvaluateAsync(@""sample"");
Console.WriteLine(result);
```

Esempio 2 Semplice utilizzo delle Scripting API per la creazione di una stringa.

1.4.2 Workspace API

Le Workspace API offrono informazioni e servizi per poter lavorare direttamente su Workspace, Soluzioni, Progetti e Documenti senza dover utilizzare API di sistema per file generici del File System (FS), provvedere a parsificazioni o gestire dipendenze tra progetti. Nello specifico:

- **Workspace**: così come in un template di progetto di Visual Studio, rappresenta in Roslyn la radice di una gerarchia che ha come nodi figlio una o più Solution.
  È rappresentato dalla classe astratta `Workspace` e dalle sue concretizzazioni `MSBuildWorkspace`, `AdhocWorkspace` e `VisualStudioWorkspace`.
  `MSBuildWorkspace` rappresenta un workspace creato per gestire Solution (.sln) e Project (.csproj, .vbproj). Non consente tuttavia di aggiungere dinamicamente solution, progetti e documenti.
  `AdhocWorkspace` rappresenta un workspace su cui poter eseguire manualmente operazioni di aggiunta, modifica e cancellazione di solution, progetti e documenti.
  `VisualStudioWorkspace` rappresenta infine un concetto di Workspace fortemente integrato con l'ambiente Visual Studio e utile per lo sviluppo di estensioni per Visual Studio (.vsix) come ToolBox in WPF e WinForms.

- **Solution** e **Project**: sono equivalente ai concetti rispettivi in un template di progetto di Visual Studio e vengono rappresentate rispettivamente dalle classi `Solution` e `Project`.
- **Document**: rappresenta un'astrazione del documento file sorgente .cs o .vb rispetto al FS ed è mappato nella casse `Document`.

Queste classi sono disponibili nel namespace `Microsoft.CodeAnalysis` e risultano indipendenti dal linguaggio utilizzato.

L'**Esempio 3** dimostra come a partire dal percorso di una solution nel FS sia possibile creare un'istanza immutabile di Workspace usando il metodo statico `Create` di `MSBuildWorkspace` per poi effettuare una ricerca a livello di progetto e documento.

```
string path = @"C:\Users\Francesco\SampleProject.sln";

// Create a new Workspace
MSBuildWorkspace workspace = MSBuildWorkspace.Create();

// Open a solution
Solution solution = workspace.OpenSolutionAsync(path).Result;

foreach (Project project in solution.Projects)
{
    Console.WriteLine("Progetto " + project.Name);

    foreach (Document document in project.Documents)
    {
        Console.WriteLine("Documento " + document.Name);
    }
}
```

Esempio 3 Iterazione a Livello di Documento e Progetto a partire da una solution nel FS.

Così come tutti gli oggetti in Roslyn, anche i precedenti sono immutabili. Conseguentemente, se si vuole modificare un elemento discendente di workspace occore sostituire l'elemento di livello opportuno (Documento, Progetto, Soluzione o intero Workspace) con uno nuovo creato a partire dal precedente.

1.4.3 Features API

Le Features API sono API specifiche utilizzate nei template di progetto "Analyzer with Code Fix" e "Code Refactoring". Comprendono la validazione e registrazione di Analizzatori e Refactoring. Verranno trattate pertanto nel **Capitolo 2** e nel **Capitolo 3**.

1.6. LIVELLO SINTATTICO

Le API più utilizzate in Roslyn sono sicuramente le Compiler API, nello specifico le **SyntaxTree API**. Queste ultime permettono l'analisi e la manipolazione della struttura sintattica del codice sorgente: il **livello sintattico**.

1.5.1 Syntax Tree

Il livello sintattico di un documento C# o Visual Basic viene rappresentato da un albero con un nodo progenitore radice (albero radicato) i cui nodi sono elementi sintattici propri della grammatica del linguaggio. Questo albero viene comunemente chiamato **Syntax Tree** (**ST**) o **Abstract Syntax Tree** (**AST**). Per brevità lo riferiremo d'ora in poi come ST.

L'ST ha cinque proprietà fondamentali:

- Rappresenta fedelmente la struttura sintattica del codice sorgente, comprensiva di trivia (spazi, tabulazioni, errori), commenti o direttive al preprocessore.
- Tramite un'operazione biunivoca è possibile risalire in ogni momento dal dominio sintattico (l'albero sintattico) a quello reale (il testo del documento che rappresenta) e viceversa. Ogni modifica su uno dei due domini si riflette sull'altro.
- È immutabile e quindi thread-safe. Rappresenta cioè una fotografia della sua struttura al momento della creazione.
- È un albero ottimizzato. Infatti, per sopperire all'overhead causato dalla continua creazione di nuovi oggetti dalla natura immutabile, ad ogni nuova parsificazione di ST viene valutata l'esistenza di alberi, sottoalberi o elementi (nodi) uguali e non più utilizzati. In caso positivo questi vengono utilizzati nel nuovo albero. Similmente, se una modifica coinvolge parte di un sottoalbero, viene sostituito solo questa mentre rimangono validi eventuali riferimenti agli alberi contenitori.
- Se la struttura dell'ST presenta degli errori sintattici, l'ST viene comunque creato. Tuttavia, al momento della parsificazione, il parser può decidere se aggiungere un eventuale elemento mancante tramite un'operazione di stima o se saltare tutti gli elementi fino al primo elemento ben formato.

L’object model che mappa un ST è rappresentatato dalla classe astratta `SyntaxTree`. Poiché la struttura di un ST presenta delle forti dipendenze dalla sintassi specifica del linguaggio, le sottoclassi `CSharpSyntaxTree` e `VisualBasicSyntaxTree` vengono comunemente utilizzate.

L'**Esempio 4** mostra l'utilizzo del metodo statico `ParseText` della classe `CSharpSyntaxTree` per generare l'ST del frammento di codice passato come stringa. Il metodo `ToString` di `CSharpSyntaxTree` è opportunatamente ridefinito per restituire la rappresentazione testuale dell'albero sotto forma di codice sorgente.

```
string code = @"namespace Sample
                {
                    class Car
                    {
                        public string Plate { get; set; }
                        public string Model { get; set; }
                    }
                }";
SyntaxTree syntaxTree = CSharpSyntaxTree.ParseText(code);
string sameCode = syntaxTree.ToString();
```

Esempio 4 Generazione di un ST a partire dal codice sorgente e viceversa.

Visual Studio mette a disposizione degli sviluppatori un componente integrato di tipo Toolbox Control, chiamato **Syntax Visualizer**, per permette di analizzare l’ST del documento aperto nel Code Editor di Visual Studio. In **Figura 3** viene riportato l’ST corrispondente alla classe `Car`, composta da due dichiarazioni di proprietà.

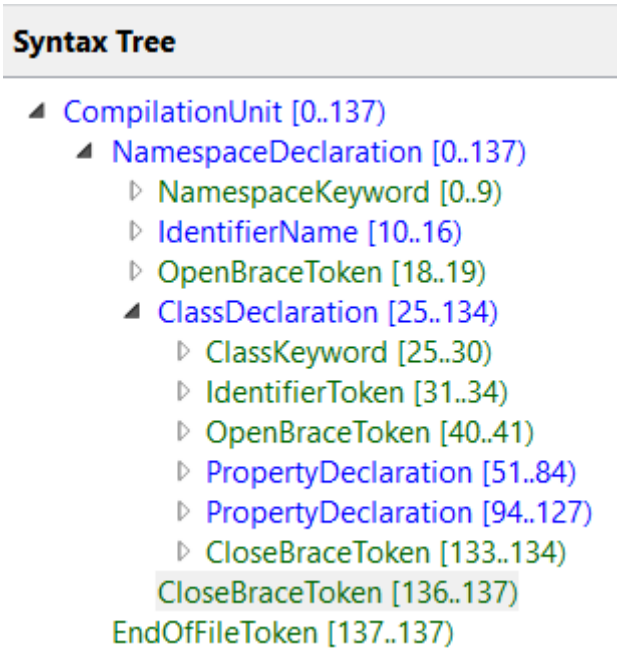

Figura 3 Finestra Syntax Visualizer per la visualizzazione dell’ST del documento aperto.

A partire dalla visualizzazione dell'ST nel Syntax Visualizer, è possibile generare un file in formato dmgl per la visualizzazione completa del grafo come riportato in **Figura 4**.

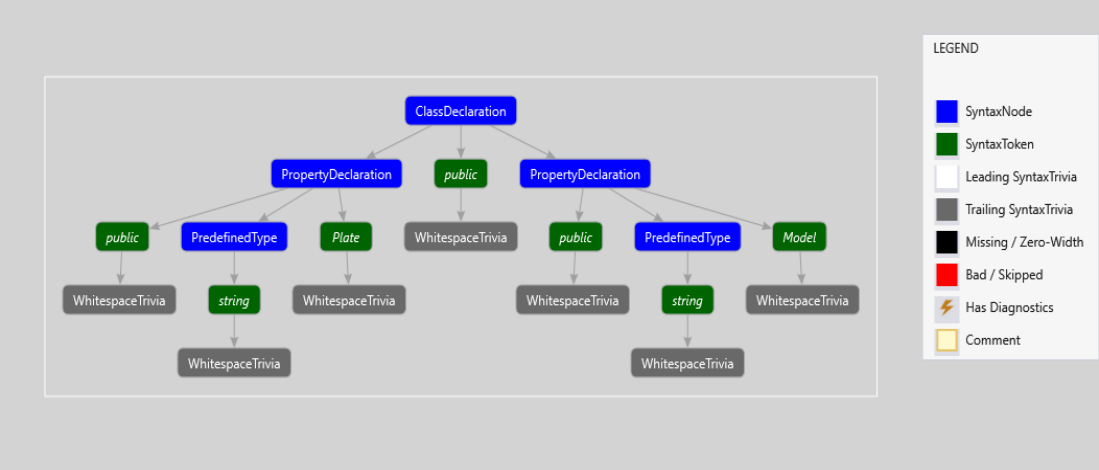

Figura 4 Grafo Semplificato in formato dmgl corrispondente all'ST della classe Car.

Come si può notare dall'ultima figura, esistono diverse tipologie di elementi in un ST. Tali elementi sono classificabili in base al ruolo del nodo nella grammatica del linguaggio utilizzato.

Gli elementi sintattici si dividono in:

1. **Syntax Nodes**
2. **Syntax Tokens**
3. **Syntax Trivia**

1.5.2 Syntax Node

Sono gli elementi principali dell'ST e rappresentano costrutti sintattici propri del linguaggio. Ve ne sono di vario tipo a seconda del ruolo dell'elemento nel codice sorgente ma tutti ereditanto dalla classe astratta `SyntaxNode`, concretizzata dalle due classi `CSharpSyntaxNode` e `VisualBasicSyntaxNode` dipendentemente dal linguaggio utilizzato.

Esempi di Syntax Node sono:

- `ClassDeclarationSyntax`: rappresenta il nodo sintattico relativo alla dichiarazione di una classe. Costituisce il nodo radice del sottoalbero di **Figura 4**.

- `PropertyDeclarationSyntax`: rappresenta il nodo sintattico relativo alla dichiarazione di una proprietà. Ne sono esempi i due nodi figlio di `ClassDeclarationSyntax` in **Figura 4** relativi alle proprietà `Plate` e `Model`.

Un Syntax Node si differenzia da tutti gli altri elementi sintattici per il fatto che rappresenta un nodo non terminale di un ST. Pertanto, possiede sempre almeno un elemento figlio (Node o Token). Inoltre, ad esclusione del nodo radice `CompilationUnitSyntax`, può avere un unico nodo genitore.

Per accedere a una lista sequenziale di nodi figlio e discendenti si utilizzano rispettivamente i metodi `ChildNodes` e `DescendantNodes`. Similmente, per l'accesso ai token si utilizzano i metodi `ChildTokens` e `DescendantTokens` e per l'accesso ai Trivia univocamente il metodo `DescendantTrivia` dato che un Trivia non può essere discendente di primo grado di un SyntaxNode.

L'**Esempio 5** dimostra come a partire dall'ST della classe `Car` definito precedentemente sia possibile ricavare il node radice della gerarchia per poi trovare il nodo corrispondente alla proprietà `Plate` tramite una query LINQ sui soli nodi di tipo `PropertyDeclarationSyntax` (metodo di filtraggio `OfType<T>`).

```
SyntaxNode syntaxRoot = syntaxTree.GetRoot();
PropertyDeclarationSyntax plateProperty =
                    syntaxRoot.DescendantNodes().
                           OfType<PropertyDeclarationSyntax>().First();
```

Esempio 5 Utilizzo del metodo DescendantNodes unitamente ad una query LINQ per trovare il nodo PropertyDeclarationSyntax corrispondente alla proprietà Plate.

Esistono molti altri metodi di utilità per i Syntax Node quali ad esempio:

- `ReplaceNode`: metodo di estensione per sostituire un nodo nel sottoalbero dell'istanza corrente con un nuovo nodo.
- `InsertNodesBefore`: metodo di estensione per aggiungere una lista di nodi precedentemente ad un nodo di riferimento nel sottoalbero dell'istanza corrente.

Per l'immutabilità degli oggetti in Roslyn, entrambi i metodi soprariportati restituiscono un nuovo nodo radice.

1.5.3 Syntax Token

I Syntax Token sono elementi sintattici caratterizzati dalla particolarità di non essere padri di una sottogerarchia. Costituiscono pertanto gli elementi terminali di un ST.

I Syntax Token modellano elementi di codice sorgente corrispondenti a keyword, identificatori, valori letterali e caratteri di punteggiatura. Al contrario dei Syntax Node che sono dei Reference Type, i Syntax Token sono stati progettati per essere delle strutture e quindi dei Value Type. Per questo motivo esiste un unico tipo `SyntaxToken` che raggruppa le proprietà comuni ad ogni elemento token del linguaggio. Tra le proprietà più importanti esposte dai SyntaxToken troviamo:

- `Value`: rappresenta il valore grezzo dell'oggetto incapsulato in un Object. Se ad esempio Value fosse riferito ad un Integer Token, questi rappresenterebbe l'intero esatto decodificato.
- `ValueText`: restituisce la rappresentazione testuale del SyntaxToken in caratteri Unicode considerando anche eventuali caratteri di escape.

1.5.4 Syntax Trivia

I Syntax Trivia, mappati nella struttura `SyntaxTrivia`, rappresentano quelle parti di testo che decorano il codice sorgente con informazioni aggiuntive, a volte non strettamente necessarie per la corretta compilazione del sorgente.

I Trivia possono essere classificati in:

- Trivia non strutturali (**Unstructured Trivia**): sono trattati dal parser come puro testo. Tra di questi troviamo commenti, spazi bianchi e tabulazioni. Questa categoria di Trivia viene ignorata durante la fase di compilazione.
- Trivia strutturali (**Structured Trivia**): possiedono del contenuto e quindi un sottoalbero. Ne sono esempi i commenti XML e le direttive al preprocessore, pertanto non ignorati durante la fase di compilazione.

Per verificare se un Trivia sia strutturale o meno, e quindi contenga un sottoalbero, si può ricorre alla proprietà `HasStructure`, positiva in caso di Structured Trivia.

L’esistenza dei Syntax Trivia è fondamentale per rispettare la seconda proprietà dell’ST di essere mappabile in maniera biunivoca con il codice sorgente da cui è generato.

Inoltre, i Syntax Trivia non possono essere discendenti né dei Syntax Node né dei Syntax Token ma solamente essere “agganciati” a questi ultimi.

Sia i Syntax Node che i Syntax Token espongono dei servizi per accedere ai Trivia a monte e a valle del Token:

- Le proprietà `LeadingTrivia` e `TrailingTrivia` di Syntax Token per l’accesso diretto al Trivia.
- I metodi `GetLeadingTrivia` e `GetTrailingTrivia` di Syntax Node per i Trivia associati ai Token che sono discendenti del nodo.

Infine, a partire da un Syntax Trivia è possibile risalire al Syntax Token ad esso associato per mezzo della proprietà `Token`.

1.5.5 Span

Lo span indica la posizione di un nodo, token o trivia all’interno di un documento file sorgente ed è rappresentato dal tipo `TextSpan`, un Value Type struttura. La posizione dell’elemento è espressa come intero a 32 bit in termini di offset dall’inizio del file (`Start`) e lunghezza dell’elemento (`Length`). La posizione finale (`End`) è ricavata a partire dalle precedenti.

È possibile risalire alla posizione di un elemento sintattico per mezzo di due proprietà comuni ad ogni nodo, token e trivia:

- `Span`: rappresenta il `TextSpan` assoluto del nodo in termini di caratteri occorsi. Questo conteggio non è comprensivo dei Trivia associati al primo SyntaxToken nel sottoalbero del nodo.
- `FullSpan`: contrariamente a `Span` tiene conto anche dei Trivia.

1.5.6 Kind

Il concetto di Syntax Kind rappresenta in Roslyn una modalità per poter distinguere la tipologia degli elementi sintattici. Risulta estremamente utile nel caso di Syntax Token e Syntax Trivia che, essendo Value Type, non possono essere estesi o differenziati per

tipologia. Ogni elemento sintattico possiede infatti una proprietà chiamata `RawKind` di tipo `Int32` che può essere confrontata con i valori dell'enumerativo `SyntaxKind` del namespace `Microsoft.CodeAnalysis.CSharp` (o `Microsoft.CodeAnalysis.VisualBasic`) in modo da risalire alla tipologia dell'elemento. Esempi di valori dell'enumerativo sono `SemicolonToken`, `IfKeyword` e `WhitespaceTrivia`.

1.5.7 Syntax Factory & Syntax Generator

Roslyn fornisce due classi di servizio per la creazione di Syntax Node, Syntax Token e Syntax Trivia:

- la classe statica `SyntaxFactory`.
- la classe astratta `SyntaxGenerator`.

`SyntaxFactory` è disponibile in due namespace diversi a seconda che si crei un elemento sintattico C# (`Microsoft.CodeAnalysis.CSharp`) o Visual Basic (`Microsoft.CodeAnalysis.VisualBasic`).

Se si decide di utilizzare la classe `SyntaxFactory` esistono due approcci per creare un elemento sintattico:

- Utilizzando i metodi di costruzione del nodo.
- Utilizzando i metodi di parsificazione del codice sorgente.

L'**Esempio 6a** mostra la costruzione del nodo `PropertyDeclarationSyntax` corrispondente alla proprietà `Plate` di `Car` sfruttando entrambi gli approcci.

```
PredefinedTypeSyntax predefinedType =
                SyntaxFactory.PredefinedType(
                    SyntaxFactory.Token(SyntaxKind.StringKeyword)
                    /*Con Parse: SyntaxFactory.ParseToken("string")*/);
SyntaxToken plateToken =
                SyntaxFactory.Identifier("Plate");
SyntaxTokenList publicTokenList =
                SyntaxFactory.TokenList(
                    SyntaxFactory.Token(SyntaxKind.PublicKeyword)
                        /*Con Parse: SyntaxFactory.ParseToken("public")*/);
AccessorListSyntax accessorList =
    SyntaxFactory.AccessorList(
        SyntaxFactory.List(
        new[]
        {
SyntaxFactory.AccessorDeclaration(
    SyntaxKind.GetAccessorDeclaration)
    .WithSemicolonToken(
```

```
    SyntaxFactory.Token(SyntaxKind.SemicolonToken)
    /*Con Parse: SyntaxFactory.ParseToken(";")*/),
SyntaxFactory.AccessorDeclaration(
    SyntaxKind.SetAccessorDeclaration)
    .WithSemicolonToken(
    SyntaxFactory.Token(SyntaxKind.SemicolonToken))
        }));

PropertyDeclarationSyntax plateDeclaration =
    SyntaxFactory.PropertyDeclaration(
        predefinedType,
            plateToken).
            WithModifiers(publicTokenList).
            WithAccessorList(accessorList);
```

Esempio 6a Utilizzo dei metodi di costruzione e parsificazione di SyntaxFactory per la creazione di una PropertyDeclarationSyntax corrispondente alla proprietà Plate.

Parallelamente a `SyntaxFactory` è possibile utilizzare la classe astratta `SyntaxGenerator` del namespace `Microsoft.CodeAnalysis.Editing`.

Questa classe espone una serie di servizi indipendenti dal linguaggio e quindi API condivise tra C# e Visual Basic. Poiché una classe astratta non è instanziabile, è possibile ottenerne un riferimento solo tramite il metodo statico `SyntaxGenerator.GetGenerator` disponibile nelle due varianti overloaded:

- `SyntaxGenerator GetGenerator(Workspace workspace, string language)`
- `SyntaxGenerator GetGenerator(Document document)`: inferisce il linguaggio a partire dal documento .cs o .vb

La stessa operazione di creazione del nodo `PropertyDeclarationSyntax` può essere in parte semplificata utilizzando la classe `SyntaxGenerator` come riportato nell'**Esempio 6b**.

```
MSBuildWorkspace workspace = MSBuildWorkspace.Create();
SyntaxGenerator syntaxGenerator =
                SyntaxGenerator.GetGenerator(workspace, "C#");

plateDeclaration = (PropertyDeclarationSyntax)syntaxGenerator.
                        PropertyDeclaration(
                            "Plate",
                            SyntaxFactory.ParseTypeName("string"),
                            Accessibility.Public,
                            DeclarationModifiers.None);
plateDeclaration = plateDeclaration.WithAccessorList(accessorList);
```

Esempio 6b Utilizzo della classe SyntaxGenerator unitamente a SyntaxFactory per la creazione di una PropertyDeclarationSyntax corrispondente alla proprietà Plate.

Il vantaggio di astrazione in termini di indipendenza del linguaggio porta però `SyntaxGenerator` ad avere un set inferiore di servizi specifici rispetto a `SyntaxFactory`. Per questo motivo, in alcuni scenari come quello dell'**Esempio 6b** è necessario combinare

l'utilizzo di entrambe le classi `SyntaxFactory` e `SyntaxGenerator` per ottenere il risultato corretto.

1.5.8 Syntax Visitor

Roslyn offre un ulteriore approccio alla navigazione dell'ST oltre alla navigazione standard tramite cicli con metodi quali `DescendantNodes`, `DescendantTokens` e `DescendantTrivia`.

```
IEnumerable<FieldDeclarationSyntax> fieldDeclarations =
    syntaxRoot.DescendantNodes().OfType<FieldDeclarationSyntax>();

// Navigazione in sola lettura mediante l'utilizzo di DescendantNodes
foreach (FieldDeclarationSyntax fieldDeclaration in fieldDeclarations)
{
    Console.WriteLine("Trovato nodo di tipo field: " + fieldDeclaration);
}
```

Esempio 7 Utilizzo del metodo DescendantNodes per la navigazione standard tra i soli nodi di tipo FieldDeclarationSyntax dell'ST.

L'approccio descritto in questa sottosezione consiste nell'utilizzo di due classi introdotte con Roslyn le quali implementano automaticamente il pattern visitor:

- `CSharpSyntaxWalker` (e `VisualBasicSyntaxWalker`).
- `CSharpSyntaxRewriter` (e `VisualBasicSyntaxRewriter`).

Tali classi ereditano i servizi di visiting basilari rispettivamente da `CSharpSyntaxVisitor` e `CSharpSyntaxVisitor<T>`.

Estendendo la classe `CSharpSyntaxWalker` e ridefinendo opportunamente i metodi con prefisso `Visit` è possibile creare Syntax Walker personalizzati per la navigazione in sola lettura tra nodi, token e trivia di un ST. È bene notare che la navigazione tramite Syntax Walker raggiunge a default solamente il livello di nodo. Per specificare altrimenti occore invocare il costruttore di base nel costruttore della classe estesa passando come argomento uno dei possibili valori dell'enumerativo `SyntaxWalkerDepth` tra `Node`, `Token`, `Trivia` e `StructuredTrivia`.

L'**Esempio 8** mostra come sia possibile implementare un Syntax Walker che ricostruisce l'ST stampandolo a Console con un comportamento specifico qualora il nodo sia di tipo `ClassDeclarationSyntax`.

```
public class SimpleWalker : CSharpSyntaxWalker
{
    public int NumTabulazioni { get; set; }

    public override void Visit(SyntaxNode node)
    {
        NumTabulazioni++;
        string indentazione = new string('\t', NumTabulazioni);
        Console.WriteLine(indentazione + node.Kind());
        base.Visit(node);
        NumTabulazioni--;
    }

    public override void VisitClassDeclaration(ClassDeclarationSyntax node)
    {
        Console.WriteLine("Trovato Nodo di Tipo ClassDeclarationSyntax.");
        base.VisitClassDeclaration(node);
    }
}

// Cliente
SimpleWalker simpleWalker = new SimpleWalker();
simpleWalker.Visit(syntaxRoot);
```

Esempio 8 Utilizzo di SimpleWalker, esteso a partire da CSharpSyntaxWalker, per la stampa a console dell'ST navigato.

La classe astratta `CSharpSyntaxRewriter` adotta lo stesso pattern Visitor del Syntax Walker ma è ottimizzata per apportare modifiche a elementi sintattici durante la navigazione dell'ST. Infatti, per via dell'immutabilità degli oggetti in Roslyn non è possibile, durante un'iterazione, apportare modifiche a un nodo senza invalidarne il sottoalbero come mostrato nell'**Esempio 9a**.

```
IEnumerable<ClassDeclarationSyntax> classDeclarations =
    syntaxRoot.DescendantNodes().OfType<ClassDeclarationSyntax>();

foreach (ClassDeclarationSyntax classDeclaration in classDeclarations)
{
    // Nuovo nodo da sostituire all'originale
    ClassDeclarationSyntax newClassDeclaration =
        classDeclaration.WithModifiers(
            SyntaxTokenList.Create(SyntaxFactory.ParseToken("private")));

    // La sostituzione del nodo provoca la creazione di un nuovo
    // sottoalbero di syntaxRoot
    syntaxRoot = syntaxRoot.ReplaceNode(
                    classDeclaration, newClassDeclaration);
}
```

Esempio 9a Approccio errato alla sostituzione di nodi durante la navigazione dell'ST.

Il ciclo di iterazione attorno ai nodi discendenti da `syntaxRoot` dell'**Esempio 9a** non è l'approccio corretto per navigare l'ST apportando modifiche ai suoi nodi. Infatti, mentre l'operazione di modifica è sintatticamente valida nel caso in cui venga trovato tra i nodi discendenti della radice dell'albero un solo nodo `ClassDeclarationSyntax`, in caso

contrario la modifica coinvolge comunque solamente il primo nodo. Questo comportamento è causato dal fatto che `ReplaceNode` opera su di un tipo immutabile e pertanto restituisce un nuovo nodo radice avente come sottoalbero non più l'albero originale ma una nuova copia con il vecchio nodo rimpiazzato. Da questo momento in poi l'iterazione del ciclo prosegue su dei riferimenti a nodi del vecchio albero e `ReplaceNode` non è più in grado di trovare il nodo da sostituire, ritornando pertanto lo stesso nodo `syntaxRoot`.

Per ovviare a questo problema, `CSharpSyntaxRewriter` gestisce automaticamente tale scenario e prosegue la visita nel nodo successivo del nuovo albero creato.

L'**Esempio 9b** mostra il corretto approccio alla modifica dei nodi dell'ST durante la sua navigazione.

```
public class SimpleRewriter : CSharpSyntaxRewriter
{
    public override SyntaxNode VisitClassDeclaration(
       ClassDeclarationSyntax node)
    {
        // Nuovo nodo da sostituire all'originale
        ClassDeclarationSyntax newClassDeclaration =
            node.WithModifiers(
                SyntaxTokenList.Create(
                      SyntaxFactory.ParseToken("private")));

        return base.VisitClassDeclaration(newClassDeclaration);
    }
}

// Cliente
SimpleRewriter simpleRewriter = new SimpleRewriter();
simpleRewriter.Visit(syntaxRoot);
```

Esempio 9b Approccio corretto alla sostituzione di nodi durante la navigazione dell'ST sfruttando la classe SimpleRewriter.

I metodi con firma `Visit`, contrariamente al Syntax Walker, restituiscono sempre un riferimento a `SyntaxNode`, `SyntaxToken` o `SyntaxTrivia`. Questo può essere:

- L'elemento originale: il nodo non viene modificato né viene creato un nuovo albero.
- Un riferimento a `null`: l'elemento viene eliminato dall'ST.
- Un nuovo elemento sintattico: l'elemento viene sostituito e creato un nuovo albero su cui continuare la navigazione.

Se, nell'esplorazione degli elementi dell'ST, si necessita un livello di dettaglio fino agli Structured Trivia, occorre invocare il costruttore base nel costruttore della classe estesa passando come parametro `true`.

Seppure il modello sintattico di cui si è parlato in questo capitolo sia uno strumento molto potento offerto da Roslyn, da solo non basta per poter effettuare operazioni di refactoring sofisticate.

Possiamo infatti usare il livello sintattico per risalire all'invocazione di un metodo a partire ad esempio dal suo span iniziale e finale ma non possiamo poi usare questa informazione per risalire ai sui membri o ad eventuali versioni del metodo in overload.

Per risolvere questo genere di problema occore introdurre il concetto di livello semantico.

1.7. LIVELLO SEMANTICO

Il livello semantico permette di avere accesso a molte delle informazioni prodotte nella fase di Declaration della pipeline di Roslyn tra cui i **simboli**.

Queste informazioni possono essere utilizzate per risolvere problemi di refactoring quali:

- risalire ai riferimenti di una variabile a partire dalla stessa.
- trovare metodi in overload a partire dall'invocazione di un metodo.
- ricavare i membri accessibili da un metodo.
- qualsiasi altro problema coinvolga l'utilizzo obbligatorio di simboli.

Ogni simbolo identifica univocamente una variabile, un campo, proprietà, parametro, tipo, metodo, evento o namespace all'interno del dominio della compilazione e ne espone informazioni e proprietà specifiche per la sua tipologia.

Un simbolo è genericamente rappresentato in Roslyn dall'interfaccia `ISymbol`, estesa opportunatamente a seconda della tipologia di elemento semantico come `IFieldSymbol`, `IMethodSymbol` e `IPropertySymbol` rispettivamente per un campo, metodo e proprietà. Similmente agli elementi sintattici, è possibile distinguere la tipologia di un simbolo anche attraverso la proprietà in sola lettura `Kind` di `ISymbol` la quale restituisce un valore dell'enumerativo `SymbolKind`. Esempi di valori di `SymbolKind` sono `Field`, `Method`, `Namespace` e `Parameter`.

Come vedremo, è possibile ricavare i simboli accedendo direttamente al prodotto della compilazione, `Compilation`, o consultando il modello semantico, `SemanticModel`.

3.1.1 Compilation

Una compilazione è una fotografia immutabile di tutto quello che ha raccolto Roslyn per effettuare una compilazione. Oltre ai simboli vengono raccolte altre informazioni tra cui:

- Riferimenti agli assembly prodotti.
- Eventuali opzioni sul comportamento del compilatore.
- I Syntax Tree relativi ad ogni documento compilato.

- Risultati della fase di Emit (IL).

Una Compilation è mappata dalla classe astratta `Microsoft.CodeAnalysis.Compilation` concretizzata a seconda del linguaggio in `CSharpCompilation` e `VisualBasicCompilation`.

Tra i servizi offerti da `Compilation` esiste anche la possibilità di avviare una compilazione "on demand" a partire da uno o più ST rappresentanti il codice sorgente. Tale compilazione è ottimizzata attraverso l'utilizzo estensivo di cache che permettono il riutilizzo di frammenti di compilazione relativi a richieste diverse.

L'**Esempio 10** mostra come sia possibile generare "on demand" una compilazione. Nello specifico viene utilizzato il metodo `CSharpCompilation.Create` a cui vengono passati quattro parametri:

- Il nome dell'assembly in uscita.
- Una lista di ST corrispondente ai Documenti che si vuole compilare.
- Una lista di riferimenti ai metadati. Nel caso dell'esempio è necessario che venga creato un solo `MetadataReference` corrispondente a mscorlib.dll.
- Una lista di opzioni al compilatore `CSharpCompilationOptions`, concretizzazione di `CompilationOptions` nel caso di C#. Nel caso dell'esempio si richiede tramite il valore `DynamicallyLinkedLibrary` dell'enumerativo `OutputKind` che il risultato della compilazione sia una dll. Altri valori notevoli dell'enumerativo sono `ConsoleApplication` e `WindowsApplication` rispettivamente per la compilazione di un eseguibile exe per applicazioni di tipo Console, Windows Forms o WPF e `WindowsRuntimeApplication` per la generazione di un eseguibile appx per applicazioni di tipo WinRT o UWP.

```
MetadataReference[] referenceList = new MetadataReference[]
{
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
};

CSharpCompilationOptions option =
    new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary);

CSharpCompilation compilation =
    CSharpCompilation.Create(
        "SampleAssembly",
        syntaxTrees: new[] { syntaxTree },
        references: referenceList,
```

```
        options: option);
```

Esempio 10 Creazione “on demand” di una compilazione a partire da un ST.

3.1.2 Semantic Model

Il Modello Semantico rappresenta il ponte tra l’ST di un Documento e la Compilazione ad esso relativa ed è mappato nel tipo `Microsoft.CodeAnalysis.SemanticModel`.

Oltre ai simboli, offre una serie di informazioni semantiche tra cui:

- Informazioni sul tipo di un nodo sintattico sotto forma di oggetto `TypeInfo`.
- Informazioni sul simbolo di un nodo sintattico sotto forma di oggetto `SymbolInfo`.
- Collezioni di diagnostiche come errori e warning.
- Un’istanza immutabile di Compilazione dalla quale può essere ottenuto.
- L’ST da cui è ricavato il modello.

Esistono due modi per ricavare il modello semantico:

- A partire da un’istanza della classe `Document` utilizzando i metodi `GetSemanticModelAsync` e `TryGetSemanticModel` come da **Esempio 11**.
- A partire da un’istanza di `Compilation` attraverso il metodo `GetSemanticModel` passando un riferimento al `SyntaxTree`.

```
SyntaxNode syntaxRoot = document.GetSyntaxRootAsync().Result;
SemanticModel model = document.GetSemanticModelAsync().Result;

PropertyDeclarationSyntax plateDecl =
    syntaxRoot.DescendantNodes().
        OfType<PropertyDeclarationSyntax>().First();
IPropertySymbol plateSymbol = model.GetDeclaredSymbol(plateDecl);
```

Esempio 11 Utilizzo del Modello Semantico per trovare il simbolo corrispondente alla proprietà Plate di Car a partire dal suo nodo sintattico.

Al pari di Compilation, la consultazione del modello semantico ha un costo molto elevato se paragonato a ricerche sul modello sintattico. Di conseguenza anche il Semantic Model mantiene in cache simboli e informazioni semantiche così che queste siano disponibili per una successiva richiesta. D’altro canto, questa occupazione di risorse non permette al Garbage Collector di liberare tali porzioni di memoria. È pertanto sconsigliato l’utilizzo simultaneo di più istanze dello stesso Semantic Model per evitare fenomeni di memory leakage.

3.1.3 Symbol Visitor

La classe `SymbolVisitor` rappresenta il concetto equivalente di visitor nel livello semantico. Tuttavia, al contrario di `CSharpSyntaxWalker` e `CSharpSyntaxRewriter`, occorre reimplementare il pattern visitor all'interno dei metodi di visiting per navigare il grafo dei simboli.

L'**Esempio 12** mostra una possibile implementazione del pattern Visitor. Come si può notare, occorre invocare il metodo `Accept(SymbolVisitor visitor)` dell'interfaccia `ISymbol` per ogni simbolo fino al livello di simbolo desiderato così da permettere al visitor di continuare la navigazione. L'applicazione di tale pattern è necessaria poichè, mentre la struttuta di albero di un ST consente la navigazione top-bottom fino agli elementi foglia, la struttura che rappresenta i simboli è un grafo ed è necessario prevedere un meccanismo che eviti situazioni di ricorsione.

Nel caso dell'esempio, il cliente passa al metodo `Visit` di `SimpleSymbolVisitor` un riferimento a `compilation.GlobalNamespace` contenente tutti i tipi definiti nel codice sorgente e da cui comincia la navigazione.

```
public class SimpleSymbolVisitor : SymbolVisitor
{
    public override void VisitNamespace(INamespaceSymbol symbol)
    {
        Console.WriteLine(symbol);

        foreach (INamespaceOrTypeSymbol memberSymbol in symbol.GetMembers())
        {
            memberSymbol.Accept(this);
        }
    }

    public override void VisitNamedType(INamedTypeSymbol symbol)
    {
        // INamedTypeSymbol Represents a type other than an array,
        // a pointer, a type parameter

        Console.WriteLine(symbol);

        foreach (INamedTypeSymbol memberSymbol in symbol.GetTypeMembers())
        {
            memberSymbol.Accept(this);
        }
    }
}


// Cliente
SemanticModel model = document.GetSemanticModelAsync().Result;
Compilation compilation = model.Compilation;
```

```
INamespaceSymbol namespaceSymbol = compilation.GlobalNamespace;

SimpleSymbolVisitor symbolVisitor = new SimpleSymbolVisitor();
symbolVisitor.Visit(namespaceSymbol);
```

Esempio 12 Utilizzo di SimpleSymbolVisitor, esteso a partire da SymbolVisitor, per la navigazione tra i simboli di tipo INamespaceSymbol e INamedTypeSymbol.

2. ANALYZER WITH CODE FIX

Come visto nell'**Esempio 1**, Roslyn ha introdotto la possibilità per gli sviluppatori di aver accesso, tramite le Diagnostic API, alle diagnostiche relative a errori e warning prodotti in fase di compilazione.

Oltre a quelle native della piattaforma, Roslyn permette di estendere il set di "regole" che portano alla produzione di diagnostiche. Per fare questo è stato introdotto con Roslyn un nuovo template di progetto chiamato "**Analyzer with Code Fix**", d'ora in poi abbreviato Analyzer.

Tale template di progetto fa parte della famiglia di progetti **Extensibility** che hanno come scopo lo sviluppo di tool che estendono le funzionalità native di Visual Studio. L'unità di distribuzione per questo tipo di progetti sono file in formato **vsix** che possono essere installati su un'istanza di Visual Studio con modalità simili a quelle di deployment di un file exe.

Un progetto Analyzer risulta adatto per tutte quelle situazioni in cui esiste una porzione di codice sintatticamente o semanticamente errata e si vuole suggerire un'opzione di refactoring utile a risolvere tale problema.

2.1. STRUTTURA DEL PROGETTO

Un template di progetto di tipo Analyzer si compone di una solution con tre progetti:

- Una **Portable Class Library** (PCL): contiene i riferimenti alle librerie indispensabili tra cui `Microsoft.CodeAnalysis`. In questo progetto deve essere definita la logica di Analisi e Refactoring del codice sorgente.
- Una Libreria di Test: è indispensabile per lo Unit Testing della PCL e pertanto ne mantiene un riferimento.
- Un Progetto di deployment: mantiene un riferimento alla PCL e tramite il suo descrittore di deployment **vsixmanifest** consente il packaging e la distribuzione del vsix prodotto all'interno di un'istanza sperimentale di Visual Studio.

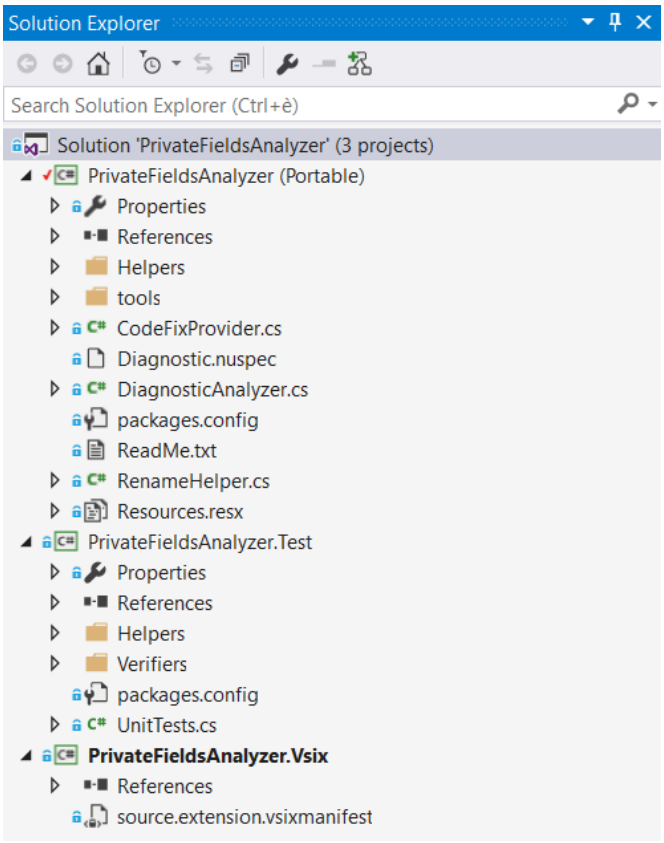

Figura 5 Struttura di un progetto Analyzer. In alto la PCL. Al centro il progetto per lo Unit Testing. In basso il progetto per il deployment del vsix.

La parte più significativa del progetto si sviluppa all'interno della PCL. Di default Visual Studio crea due file all'interno del progetto in modo da suddividere la logica di analisi del codice da quella di applicazione del refactoring. Nello specifico devono essere creati:

- Uno o più file di **Analisi**: a default DiagnosticAnalyzer.cs (o .vb).
- Uno o più file per l'applicazione del Refactoring (chiamato in questo contesto **Code Fix**): a default CodeFixProvider.cs (o .vb).

Per facilitare l'illustrazione della struttura di questa tipologia di progetto, questa sezione richiamerà più volte come **Esempio 13** un semplice Analyzer relativo all'analisi e al refactoring di campi privati che non rispettano la "naming convention" di C# _nomeCampo.

2.1.1 File di Analisi

La parte di analisi deve implementare la logica di monitoraggio del codice sorgente e può portare o meno alla visualizzazione di un messaggio di errore o warning nella Error List di Visual Studio come mostrato in **Figura 6**.

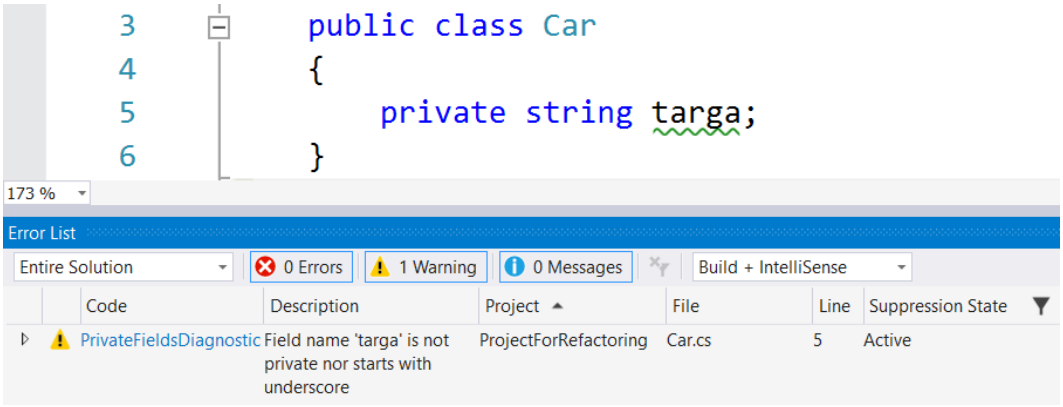

Figura 6 La diagnostica prodotta dall'Analyzer sviluppato in questa sottosezione viene mostrata come warning nella error list di Visual Studio in quanto il campo targa non rispetta la naming convention di C# (_targa).

La classe a cui si delega l'operazione di monitoraggio del codice sorgente deve estendere la classe astratta `DiagnosticAnalyzer` e deve essere decorata dell'attributo `DiagnosticAnalyzerAttribute` che specifica il linguaggio di riferimento (nel nostro caso `LanguageNames.CSharp`).

Per identificare univocamente la diagnostica che si vuole integrare, occorre definire un campo (o proprietà) di tipo `DiagnosticDescriptor` inizializzato a partire da una stringa

identificativa di diagnostica, un titolo, un messaggio, una categoria e un livello di severità come valore dell'enumerativo `DiagnosticSeverity`. I campi titolo, messaggio e descrizione possono essere modellati sia come normali stringhe che come `LocalizableString` qualora si preveda l'utilizzo di file di risorse resx per la distribuzione in più lingue. Infine, il descrittore di diagnostica deve essere restituito come `ImmutableArray<DiagnosticDescriptor>` nella ridefinizione della proprietà astratta `SupportedDiagnostics`.

Il frammento di codice dell'**Esempio 13a** riporta la definizione della classe `PrivateFieldsDiagnosticAnalyzer` e dei suoi campi per la descrizione della diagnostica.

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class PrivateFieldsDiagnosticAnalyzer : DiagnosticAnalyzer
{
    #region Analyzer Description
    public const string DiagnosticId = "PrivateFieldsDiagnosticAnalyzer";
    public const string Category = "Naming";

    private static readonly LocalizableString Title =
        new LocalizableResourceString(
              nameof(Resources.AnalyzerTitle),
                    Resources.ResourceManager,
                    typeof (Resources));

    private static readonly LocalizableString MessageFormat =
        new LocalizableResourceString(
              nameof(Resources.AnalyzerMessageFormat),
                    Resources.ResourceManager,
                    typeof (Resources));

    private static readonly LocalizableString Description =
        new LocalizableResourceString(
              nameof(Resources.AnalyzerDescription),
              Resources.ResourceManager,
              typeof (Resources));
    #endregion

    private static readonly DiagnosticDescriptor Rule =
        new DiagnosticDescriptor(
              DiagnosticId, Title,
              MessageFormat, Category,
              DiagnosticSeverity.Warning,
              isEnabledByDefault: true,
              description: Description);

    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics
    {
        get { return ImmutableArray.Create(Rule); }
    }

    // Continua...
}
```

Esempio 13a Definizione della classe PrivateFieldsDiagnosticAnalyzer e dei suoi campi per la descrizione della diagnostica.

L'entry point della classe di analisi è costituito dal metodo `Initialize(AnalysisContext context)` a partire dal quale si utilizza l'oggetto contesto per la registrazione di un'azione. Nello specifico occorre invocare il metodo opportuno con prefisso `Register` al quale viene passato un delegato di sistema `Action<T>` dove `T` rappresenta la struttura contesto dipendente dal metodo `Register` utilizzato.

Esempi notevoli di metodi di registrazione sono:

- `abstract void RegisterSyntaxNodeAction<TLanguageKindEnum>(Action<SyntaxNodeAnalysisContext> action, ImmutableArray<TLanguageKindEnum> syntaxKinds)`: il delegato `action` viene invocato ogni volta che il parser dell'ST incontra un nodo il cui `SyntaxKind` combaci con uno tra i `syntaxKinds` passati durante la registrazione.
- `abstract void RegisterSymbolAction(Action<SymbolAnalysisContext> action, ImmutableArray<SymbolKind> symbolKinds)`: il delegato `action` viene invocato ogni volta che nella fase di compilazione viene prodotto un simbolo il cui `SymbolKind` è uguale a uno tra i `symbolKinds` passati durante la registrazione.

Nel metodo delegato si devono verificare le condizioni (regole) di applicabilità della diagnostica in base all'analisi del contesto passato come parametro. Se tali regole sono soddisfatte, si procede con la creazione delle diagnostiche attraverso il metodo statico `Diagnostic.Create` avente come parametri:

- Il descrittore di diagnostica precedentemente creato.
- Un oggetto `Location` che permette di identificare il nodo target del refactoring in termini di Span, ST e modulo dell'assembly.
- Una serie di parametri di tipo `object` utili al fine di passare oggetti alla classe responsabile del Code Fix.

Una volta che la diagnostica è stata creata, la si riporta nella Error List di Visual Studio utilizzando il metodo `ReportDiagnostic` dell'oggetto contesto.

L'**Esempio 13b** riporta la ridefinizione del metodo `Initialize` assieme al metodo delegato alla verifica di applicabilità della diagnostica. Come si può notare, viene inizialmente verificato che il campo non sia privato e che non rispetti già la naming

convention. Se si superano queste condizioni si procede a creare e riportare la diagnostica nella console di errori.

```
// Entry Point
public override void Initialize(AnalysisContext context)
{
    context.RegisterSymbolAction(AnalyzeField, SymbolKind.Field);
}

private void AnalyzeField(SymbolAnalysisContext context)
{
    IFieldSymbol fieldSymbol = context.Symbol as IFieldSymbol;

    // Se non si tratta di un campo private
    if (fieldSymbol == null ||
            fieldSymbol.DeclaredAccessibility != Accessibility.Private)
    {
        return;
    }

    // Se il nome di variabile inizia per _lower
    if (fieldSymbol.Name.Length >= 2 &&
            StarsWithUnderscoreLower(fieldSymbol.Name))
    {
        return;
    }

    // Se qui, creare e riportare la diagnostica
    Diagnostic diagnostic =
        Diagnostic.Create(
            Rule, fieldSymbol.Locations[0], fieldSymbol.Name);
    context.ReportDiagnostic(diagnostic);
}

private bool StarsWithUnderscoreLower(string name)
{
    char[] array = name.ToCharArray();
    if (array[0] == '_' && char.IsLower(array[1]))
        return true;

    return false;
}
```

Esempio 13b In alto la ridefinizione del metodo Initialize. A seguire il metodo per l'analisi dell'oggetto contesto. In basso un metodo di utilità per verificare che il campo non rispetti già la naming convention da applicare.

Per concludere, per uno stesso progetto possono essere registrate più azioni e offerte più diagnostiche così da avere validazioni e diagnostiche diverse a seconda del nodo o simbolo target dell'analisi.

2.1.2 File di Code Fix

La controparte Code Fix è delegata all'esecuzione della logica di refactoring vera e propria. Un Code Fix si scatena solo quando l'utente seleziona dopo un click sul Light Bulb (accessibile anche da scorciatoia CTRL + .) l'operazione di Code Fix desiderata tra

le disponibili in una certa locazione. Nello specifico, viene presa come locazione di riferimento lo Span del cursore di inserimento testo nel Code Editor di Visual Studio.

L'immagine di **Figura 7** cerca di chiarire questo concetto mostrando il menu a comparsa contenente la lista dei Code Fix disponibili. Alla selezione dell'opzione, viene visualizzata la preview degli effetti del refactoring sul codice sorgente. In questo caso il campo targa viene trasformato in _targa per rispettare la naming convention di C#.

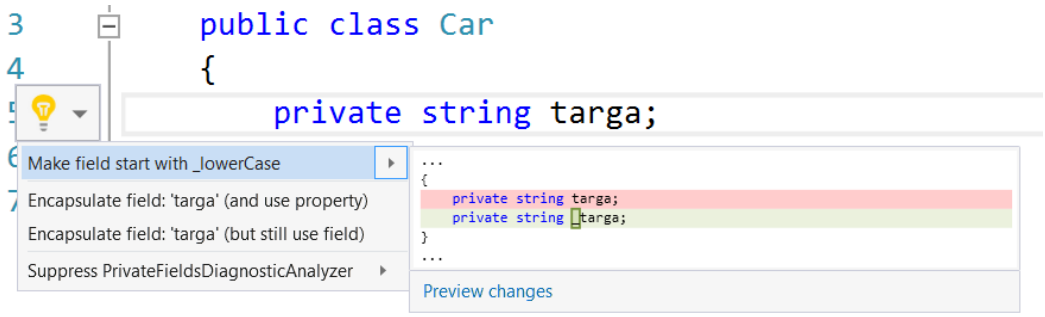

Figura 7 Al click sul Light Bulb viene mostrato un menu con la lista di Code Fix disponibili. Il Code Fix che sarà sviluppato in questa sezione è il primo della lista.

La classe responsabile del Code Fix deve estendere la classe `CodeFixProvider` ed essere decorata con l'attributo `ExportCodeFixProviderAttribute` che specifica il linguaggio di riferimento ed il nome della classe decorata. Tale attributo viene utilizzato dall'host environment per trovare tramite reflection la classe di Code Fix e permettere così di mostrare le varie opzioni nella casella del Light Bulb.

Infine occorre ridefinire la proprietà astratta `FixableDiagnosticIds` in modo tale da restituire un `ImmutableArray` di stringhe identificative relative ai descrittori di diagnostica dell'analizzatore creato nella sottosezione precedente. Questo meccanismo consente di collegare una diagnostica ad un Code Fix così che nell'oggetto contesto, parametro del metodo di registrazione del Code Fix, siano fruibili le diagnostiche relative agli id in `FixableDiagnosticIds`.

L'**Esempio 13c** riporta la definizione della classe `PrivateFieldsCodeFixProvider` assieme alla ridefinizione della proprietà `FixableDiagnosticIds`.

```
[ExportCodeFixProvider(LanguageNames.CSharp, Name =
                nameof(PrivateFieldsCodeFixProvider)), Shared]
public class PrivateFieldsCodeFixProvider : CodeFixProvider
{
    public sealed override ImmutableArray<string> FixableDiagnosticIds
```

```
    {
        get
        {
                return ImmutableArray.Create(
                        PrivateFieldsDiagnosticAnalyzer.DiagnosticId);
        }
    }
}
```

Esempio 13c Definizione della classe PrivateFieldsCodeFixProvider e ridefinizione della proprietà FixableDiagnosticIds.

L'entry point del Code Fix è costituito dal metodo asincrono `RegisterCodeFixesAsync`. Tale metodo deve contenere:

- La logica di reperimento dei nodi sintattici o simboli relativi alle diagnostiche dell'oggetto contesto `CodeFixContext`.
- La definizione di un oggetto di tipo `CodeAction` a partire da un titolo per l'operazione (visualizzato nel menu a comparsa) e un delegato di sistema `Func<CancellationToken, Task<T>>` dove `CancellationToken` rappresenta l'handler per l'interruzione dell'operazione di Code Fix e `Task<T>` è il tipo di ritorno del delegato contenente la logica del Refactorig con tipo generico `Document` o `Solution` a seconda che l'operazione coinvolga il solo Documento o l'intera Solution.
- La registrazione del Code Fix tramite il metodo `RegisterCodeFix` dell'oggetto contesto avente come parametri l'oggetto di tipo `CodeAction` precedentemente definito e una o più diagnostiche ad esso collegate.

L'**Esempio 13d** mostra come a partire dallo span relativo alla locazione della diagnostica, reperito a partire dal contesto `CodeFixContext`, sia possibile risalire al nodo sintattico `FieldDeclarationSyntax` sul quale il delegato può eseguire la logica di code fix restituendo un Documento avente un nuovo sottoalbero a partire dal nodo `VariableDeclarationSyntax`.

```
public sealed override async Task RegisterCodeFixesAsync(CodeFixContext context)
{
    SyntaxNode root =
        await context.Document.GetSyntaxRootAsync(
            context.CancellationToken).ConfigureAwait(false);
    Diagnostic diagnostic = context.Diagnostics.First();
    TextSpan diagnosticSpan = diagnostic.Location.SourceSpan;

    FieldDeclarationSyntax fieldDecl =
        root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().
        OfType<FieldDeclarationSyntax>().First();
```

```
    CodeAction codeAction = CodeAction.Create(
        title: Title,
        createChangedDocument: c =>
            RefactorWithUnderscoreLowerAsync(context, fieldDecl, c));

    context.RegisterCodeFix(codeAction, diagnostic);
}

private async Task<Document> RefactorWithUnderscoreLowerAsync(
       CodeFixContext context, FieldDeclarationSyntax fieldDecl,
              CancellationToken token)
{
    Document document = context.Document;
    SyntaxNode root =
        await document.GetSyntaxRootAsync(token).ConfigureAwait(false);
    VariableDeclarationSyntax oldVariableDecl = fieldDecl.Declaration;
    SeparatedSyntaxList<VariableDeclaratorSyntax> oldVariables =
        oldVariableDecl.Variables;
    List<VariableDeclaratorSyntax> newVariables =
        oldVariables.Select(
            oldVariable => oldVariable.WithIdentifier(
                SyntaxFactory.ParseToken(
                    ToUnderscoreLower(
                        oldVariable.Identifier.ValueText)))).ToList();

    VariableDeclarationSyntax newVariableDecl =
        oldVariableDecl.WithVariables(
            SyntaxFactory.SeparatedList(newVariables));

    // Replace the old VariableDeclarationSyntax with the new one
    SyntaxNode newRoot = root.ReplaceNode(oldVariableDecl, newVariableDecl);
    Document newDocument = document.WithSyntaxRoot(newRoot);

    return newDocument;
}

private static string ToUnderscoreLower(string valueText)
{
    char[] array = valueText.ToCharArray();
    if (array[0] == '_')
    {
        array[1] = char.ToLower(array[1]);
        return new string(array);
    }
    array[0] = char.ToLower(array[0]);
    return "_" + new string(array);
}
```

Esempio 13d In alto il metodo di registrazione per il Code Fix. A seguire il metodo di callback per l'esecuzione della logica di refactoring assieme ad una funzione di utilità per l'applicazione della naming convention corretta.

2.2. ANALISI DEI VAR

3.2.1 Specifiche del problema

La keyword `var`, comunemente detta **variabile implicita**, è stata introdotta nella versione 3.0 del linguaggio C# per supportare la memorizzazione di tipi anonimi. Un **tipo anonimo** rappresenta un tipo che viene creato inline senza avere nome di tipo (in realtà un nome di tipo gli è assegnato automaticamente durante la fase di compilazione).

L'**Esempio 14** e l'**Esempio 15a** mostrano rispettivamente due casi di utilizzo di tipi anonimi assegnati a variabili implicite rispettivamente a partire da una definizione di tipo anonimo inline e dal risultato di una query LINQ.

```
var car = new { Plate = "2SAM123", Model = "Bentley" };
```

Esempio 14 Il tipo anonimo avente due proprietà Plate e Model viene assegnato ad una variabile implicita.

```
List<Car> cars = new List<Car>();
cars.Add(new Car { Plate = "1SAM123", Model = "Punto" });
cars.Add(new Car { Plate = "2SAM123", Model = "Bentley" });
cars.Add(new Car { Plate = "3SAM123", Model = "Porsche" });

var anonymousCars = from entry in cars
                    select new { entry.Plate, entry.Model };
```

Esempio 15a A partire da una lista di tipi non anonimi, la query LINQ restituisce una lista di tipi anonimi che viene memorizzata in anonymousCars.

Il vantaggio dell'introduzione di var sta nel fatto che questo costrutto può incapsulare un tipo senza nome permettendo di accedere alle sue proprietà come un normale tipo con nome. Questo meccanismo consente inoltre di utilizzare query LINQ su liste di tipi anonimi come nell'**Esempio 15b**.

```
IEnumerable<string> plates = from entry in anonymousCars
                             select entry.Plate;
```

Esempio 15b A partire da una lista di tipi anonimi, la query LINQ restituisce una lista di stringhe corrispondenti alla proprietà Plate del tipo anonimo.

La keyword var può essere inoltre utilizzata come shortcut di scrittura per tipi non anonimi come nel caso dell'**Esempio 16a**.

```
var implicitCars = new List<Car>();
// Is the same as
List<Car> explicitCars = new List<Car>();
```

Esempio 16a L'utilizzo di var per un tipo non anonimo è del tutto equivalente all'utilizzo del tipo esplicito.

Non esiste infatti alcuna differenza né a livello di semantica né a livello di performance nell'utilizzo delle due variabili. Il compilatore è infatti in grado di inferire automaticamente il tipo esplicito della variabile. L'**Esempio 16b** riporta il compilato CIL dell'**Esempio 16a**. Come si può notare, la prima variabile è stata automaticamente esplicitata e non esiste alcuna differenza tra le due definizioni di variabili.

```
.locals init (
      [0] class [mscorlib]System.Collections.Generic.List`1<class
              SampleNamespace.Car> list1,
      [1] class [mscorlib]System.Collections.Generic.List`1<class
              SampleNamespace.Car> list2
    )
```

Esempio 16b Compilato IL della porzione di codice precedente. Come si voleva dimostrare il compilatore inferisce automaticamente il tipo della classe Car.

L'utilizzo della keyword var ha però delle limitazioni. Infatti questa può essere utilizzata solamente con scope locale e deve essere dichiarata ed inizializzata in uno stesso statement. Inoltre non può essere utilizzata per campi a livello di classe.

I casi di utilizzo di var si possono quindi ricondurre a quattro:

1. Utilizzo come tipo implicito per una **variabile locale**.
2. Utilizzo come tipo implicito del campo inizializzatore di un **for**.
3. Utilizzo come tipo implicito del valore corrente dell'iteratore di un **foreach**.
4. Utilizzo come tipo implicito di un oggetto da disporre all'interno del costrutto **using**.

L'**Esempio 17** mostra i quattro casi di utilizzo di var soprariportati.

```
// var come variabile locale
var range = Enumerable.Range(0, 10);

// var come tipo implicito dell'inizializzatore di un for
for (var i = 0; i < 10; i++)
{
    // Iteration
}

// var come tipo implicito del valore corrente dell'iteratore di un foreach
foreach (var i in range)
{
    // Iteration
}

// var come tipo implicito di uno Stream da disporre al termine del blocco
using (var stream = new FileStream("sampleFile.txt", FileMode.Open))
{
    // Do Stuff with stream
```

```
}
```

Esempio 17 I quattro casi di utilizzo di var.

Seppure l'utilizzo di var faciliti la definizione di una variabile, viceversa ne penalizza la leggibilità e manutenibilità del codice. Infatti, qualora l'assegnamento di una variabile renda difficile capirne il tipo, occorre posizionarsi con il cursore precisamente sul nome della variabile, come indicato in **Figura 8**.

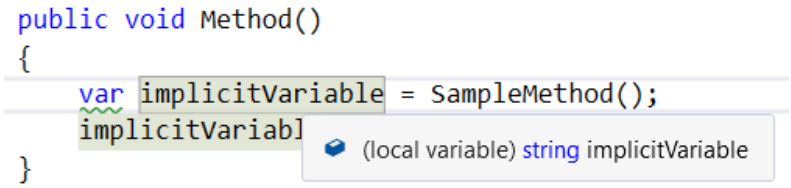

Figura 8 Per scoprire il tipo della variabile implicitVariable occore posizionarsi sul suo nome. Il tooltip che si apre mostra il tipo della variabile, in questo caso string.

L'utilizzo del tipo implicito var è pertanto da considerarsi "lecito" solamente quando la variabile implicita incapsula un tipo anonimo. Al contrario, il suo utilizzo per tipi con nome è da scoraggiare.

Si vuole dunque creare un'estensione per Visual Studio che notifichi l'uso errato della keyword var secondo le regole sopraindicate e ne suggerisca il refactoring a livello di documento, qualora possibile.

3.2.2 Soluzione

Sulla base delle specifiche introdotte nella sottosezione precedente si è proceduto a creare un progetto di tipo Analyzer.

Questa sottosezione richiamerà per parti il progetto sviluppato, etichettato nell'Appendice come **Progetto 1**.

Il **Progetto 1a** mostra la definizione della classe `VarAnalyzer` e dei suoi campi per la descrizione della diagnostica.

```
[DiagnosticAnalyzer(LanguageNames.CSharp)]
public class VarAnalyzer : DiagnosticAnalyzer
{
    #region Analyzer Description

    public const string DiagnosticId = "PH0001";

    private static readonly LocalizableString Title =
```

```
        "Trasformazione di var in tipo specifico";
    private static readonly LocalizableString MessageFormat =
        "Utilizzo di var invece di tipo specifico";

    private const string Category = "Types";

    #endregion

    public static DiagnosticDescriptor Rule =
        new DiagnosticDescriptor(DiagnosticId, Title,
                                 MessageFormat, Category,
                                 DiagnosticSeverity.Warning,
                                 isEnabledByDefault: true);

    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics
    {
         get { return ImmutableArray.Create(Rule); }
    }

    // Continua...
}
```

Progetto 1a Definizione della classe VarAnalyzer e dei suoi campi per la descrizione della diagnostica.

Come si può evincere dagli screenshot dei Syntax Visualizer in **Figura 9**, **10**, **11** e **12**, il nodo sintattico corrispondente alla keyword var è in tutti i casi possibili di tipo `IdentifierNameSyntax`, nodo che rappresenta un nome di identificatore.

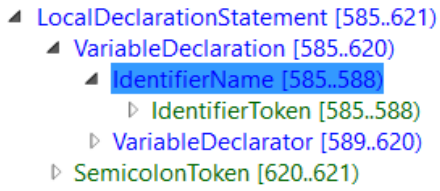

Figura 9 ST dal nodo LocalDeclarationStatementSyntax.

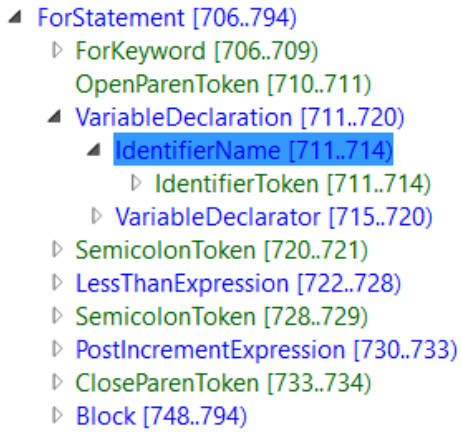

Figura 10 ST dal nodo ForStatementSyntax.

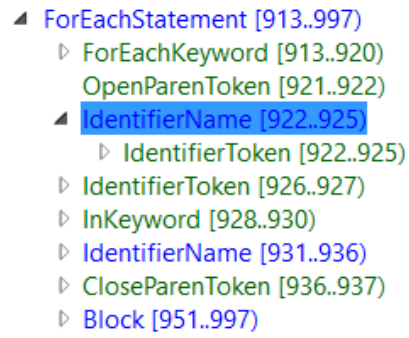

Figura 11 ST dal nodo ForEachStatementSyntax.

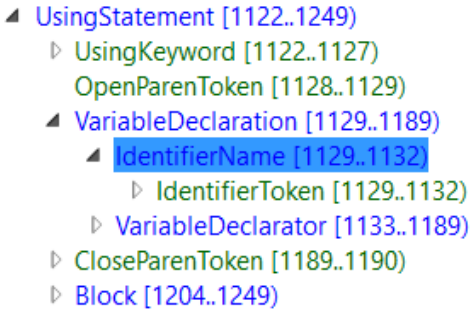

Figura 12 ST dal nodo UsingStatementSyntax.

Per evitare di registrare un'azione (e quindi invocare un delegato) per ogni nodo di tipo `IdentifierNameSyntax` e per poter analizzare il sottoalbero completo prima di questo, è stato scelto di registrare un'azione per ognuna delle quattro tipologie di nodo da analizzare ovvero:

- `LocalDeclarationStatementSyntax`
- `ForStatementSyntax`
- `ForEachStatementSyntax`
- `UsingStatementSyntax`

Il **Progetto 1b** riporta il metodo `Initialize` consistente nella registrazione delle quattro azioni su nodi.

```
/// <summary>
/// The 4 var use cases:
/// 1) local variables
/// 2) for statement
/// 3) foreach statement
/// 4) using statement
/// </summary>
public override void Initialize(AnalysisContext context)
{
    context.RegisterSyntaxNodeAction(
       AnalyzeLocalDeclarationStatement, SyntaxKind.LocalDeclarationStatement);
    context.RegisterSyntaxNodeAction(
       AnalyzeForStatement, SyntaxKind.ForStatement);
    context.RegisterSyntaxNodeAction(
       AnalyzeForEachStatement, SyntaxKind.ForEachStatement);
    context.RegisterSyntaxNodeAction(
       AnalyzeUsingStatement, SyntaxKind.UsingStatement);
}
```

Progetto 1b Metodo Initialize contenente la registrazione delle diverse azioni sui nodi.

Ogni azione sfrutta il contesto passato come argomento per ritrovare il nodo da analizzare e delega immediatamente l'analisi di tale nodo ad un metodo di utilità `AnalyzeTypeIdentifierName` come mostrato in **Progetto 1c**.

```
private static void AnalyzeLocalDeclarationStatement(
       SyntaxNodeAnalysisContext context)
{
    LocalDeclarationStatementSyntax localDeclaration =
       context.Node as LocalDeclarationStatementSyntax;
    AnalyzeTypeIdentifierName(context,
              localDeclaration?.Declaration.Type as IdentifierNameSyntax);
}

private static void AnalyzeForStatement(SyntaxNodeAnalysisContext context)
{
    ForStatementSyntax forStatement = context.Node as ForStatementSyntax;
```

```
    AnalyzeTypeIdentifierName(context,
       forStatement?.Declaration.Type as IdentifierNameSyntax);
}

private static void AnalyzeForEachStatement(SyntaxNodeAnalysisContext context)
{
    ForEachStatementSyntax forEachStatement =
       context.Node as ForEachStatementSyntax;
    AnalyzeTypeIdentifierName(context,
       forEachStatement?.Type as IdentifierNameSyntax);
}

private static void AnalyzeUsingStatement(SyntaxNodeAnalysisContext context)
{
    UsingStatementSyntax usingStatement =
       context.Node as UsingStatementSyntax;
    AnalyzeTypeIdentifierName(context,
       usingStatement?.Declaration.Type as IdentifierNameSyntax);
}
```

Progetto 1c Metodi di azione sui nodi.

Il metodo di utilità `AnalyzeTypeIdentifierName` sfrutta il metodo `IsInferableTypeIdentifier` per verificare la correttezza nell'uso del costrutto var. In caso di uso errato crea e riporta la diagnostica nella Error List di Visual Studio.

Nello specifico `IsInferableTypeIdentifier` verifica inizialmente se l'`IdentifierNameSyntax` sia relativo alla keyword var. Successivamente fa uso del modello semantico per inferire il tipo della variabile implicita ottenendo un riferimento a `ITypeSymbol`, interfaccia che descrive il simbolo relativo ad un tipo. Questa interfaccia viene poi utilizzata per verificare se il tipo inferito è un tipo anonimo o un tipo contenitore tra i cui tipi generici esiste almeno un tipo anonimo. In questi ultimi due casi il costrutto var è considerato "lecito" e il warning non prodotto.

Il **Progetto 1d** mostra entrambi i metodi.

```
private static void AnalyzeTypeIdentifierName(
       SyntaxNodeAnalysisContext context,
       IdentifierNameSyntax typeIdentifierName)
{
    if (typeIdentifierName != null &&
       IsInferableTypeIdentifier(context, typeIdentifierName))
    {
        Rule = new DiagnosticDescriptor(DiagnosticId, Title,
                                        MessageFormat, Category,
                                        DiagnosticSeverity.Warning,
                                        isEnabledByDefault: true);
        Diagnostic diagnostic =
              Diagnostic.Create(Rule, typeIdentifierName.GetLocation());
        context.ReportDiagnostic(diagnostic);
    }
}
```

```
private static bool IsInferableTypeIdentifier(
       SyntaxNodeAnalysisContext context,
       IdentifierNameSyntax typeIdentifierName)
{
    if (!typeIdentifierName.IsVar)
    {
        return false;
    }

    // Get the inferred Type
    SemanticModel semanticModel = context.SemanticModel;
    TypeInfo typeInfo = semanticModel.GetTypeInfo(typeIdentifierName);
    ITypeSymbol inferredType = typeInfo.ConvertedType;

    if (inferredType.Name == "var")
    {
        // It may be a variable of a type named 'var'.
        // In this case we don't produce a diagnostic.
        return false;
    }

    if (inferredType.IsAnonymousType)
    {
        // var construct is considered legit for an anonymous type.
        return false;
    }

    INamedTypeSymbol inferredNamedtype = inferredType as INamedTypeSymbol;
    if (inferredNamedtype != null && inferredNamedtype.IsGenericType)
    {
        ImmutableArray<ITypeSymbol> typeArguments =
              inferredNamedtype.TypeArguments;
        if (typeArguments != null &&
              typeArguments.Any(ta => ta.IsAnonymousType))
        {
            // In this case we have an IEnumerable<T> where T
            // is an anonymous type.
            // var construct is condidered legit for an anonymous list.
            return false;
        }
    }
    return true;
}
```

Progetto 1d Metodi AnalyzeTypeIdentifierName e IsInferableTypeIdentifier per l'analisi del corretto utilizzo di var e l'eventuale registrazione della diagnostica.

La **Figura 13** evidenzia il risultato finale dell'analisi dei var. Come si può notare, viene prodotto un warning esclusivamente per le prime due variabili implicite. L'ultima variabile, incapsulando un tipo anonimo, non viene segnalata nella Error List.

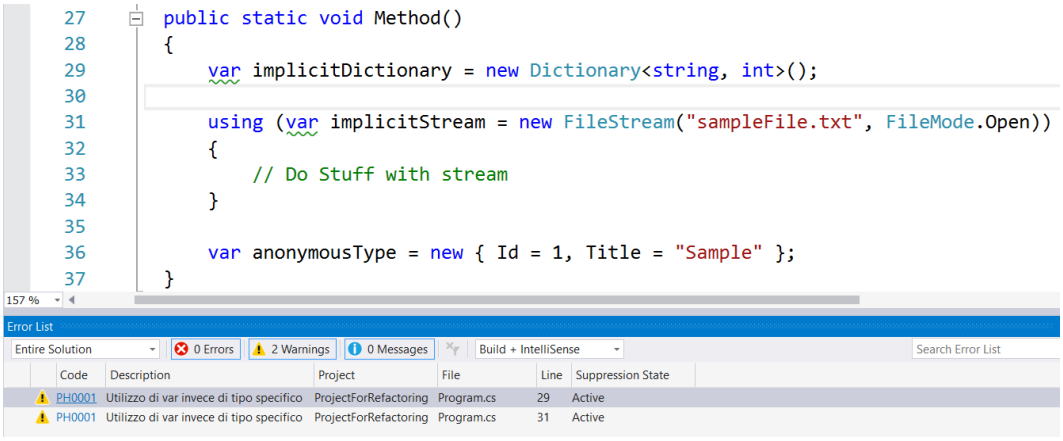

Figura 13 Warning dovuti all'errato utilizzo del costrutto var.

Per quanto riguarda la logica di applicazione del Code Fix, questa è stata confinata nella classe `VarCodeFix` come riportato in **Progetto 1e**.

```
[ExportCodeFixProvider(LanguageNames.CSharp, Name = nameof(VarCodeFix)), Shared]
public class VarCodeFix : CodeFixProvider
{
    public sealed override ImmutableArray<string> FixableDiagnosticIds
    {
        get { return ImmutableArray.Create(VarAnalyzer.DiagnosticId); }
    }

    // Continua...
}
```

Progetto 1e Definizione della classe VarCodeFix e ridefinizione della proprietà FixableDiagnosticIds.

Come si può inoltre notare da **Progetto 1f**, il metodo `RegisterCodeFixesAsync` è stato progettato per fungere da dispatcher per diverse possibili registrazioni di Code Fix sulla base dell'identificatore di diagnostica. Inoltre, avendo ridefinito `FixableDiagnosticIds`, le uniche diagnostiche che compaiono nel contesto `CodeFixContext` sono quelle relative alle diagnostiche della fase di analisi. Per questo motivo, nel metodo `RegisterVarCodeFix` è possibile ricavare l'`IdentifierNameSyntax` corrispondente a var, a questo punto necessariamente da rifattorizzare, a partire dallo span della diagnostica (metodo `root.FindNode(context.Span)`).

```
private const string CodeFixTitle = "Trasforma var nel tipo specifico";

public sealed override async Task RegisterCodeFixesAsync(CodeFixContext context)
{
    SyntaxNode root = await context.Document.
       GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);
    Diagnostic diagnostic = context.Diagnostics.First();
    switch (diagnostic.Id)
    {
```

```
        case VarAnalyzer.DiagnosticId:
            RegisterVarCodeFix(context, root, diagnostic);
            break;
    }
}

private static void RegisterVarCodeFix(CodeFixContext context, SyntaxNode root,
Diagnostic diagnostic)
{
    IdentifierNameSyntax typeIdentifierName =
       root.FindNode(context.Span) as IdentifierNameSyntax;
    CodeAction action = CodeAction.Create(
       title: CodeFixTitle,
       createChangedDocument: c =>
              VarCodeFixAsync(context.Document, typeIdentifierName, c),
       equivalenceKey: CodeFixTitle);
    context.RegisterCodeFix(action, diagnostic);
}
```

Progetto 1f In alto il titolo del Code Fix. A seguire il metodo dispatcher per le possibili registrazioni di Code Fix. In basso la registrazione del Code Fix per l'inferimento di var.

Per completare la trattazione del Code Fix, si riporta in **Progetto 1g** il metodo delegato, di tipo `Func<CancellationToken, Task<Document>>`, relativo alla `CodeAction` registrata. Similmente a `IsInferableTypeIdentifier` dell'analizzatore, viene consultato il modello semantico per ricavare il simbolo relativo al tipo esplicitato, sotto forma di riferimento all'interfaccia `ITypeSymbol`. A partire dalle informazioni sul simbolo si è in grado, utilizzando la classe `SyntaxFactory`, di creare un nodo sintattico `TypeSyntax` corrispondente al tipo esplicitato. Essendo inoltre la classe `IdentifierNameSyntax` estesa da `TypeSyntax`, è poi possibile sostituire il vecchio nodo `IdentifierNameSyntax`, riferito a var, con un nuovo nodo `TypeSyntax` corrispondente al tipo esplicitato.

```
private static async Task<Document> VarCodeFixAsync(
       Document document, IdentifierNameSyntax typeIdentifierName,
              CancellationToken cancellationToken)
{
    SyntaxNode syntaxRoot = await document.
       GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
    SemanticModel semanticModel = await document.
       GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    TypeInfo typeInfo = semanticModel.GetTypeInfo(typeIdentifierName);
    ITypeSymbol inferredType = typeInfo.ConvertedType;

    TypeSyntax inferredVariableTypeName =
       SyntaxFactory.ParseTypeName(
              inferredType.ToDisplayString(
                     SymbolDisplayFormat.MinimallyQualifiedFormat)).
                     WithLeadingTrivia(
                            typeIdentifierName.GetLeadingTrivia()).
                     WithTrailingTrivia(
                            typeIdentifierName.GetTrailingTrivia());

    SyntaxNode newSyntaxRoot =
       syntaxRoot.ReplaceNode(typeIdentifierName, inferredVariableTypeName);
```

```
    Document newDocument = document.WithSyntaxRoot(newSyntaxRoot);
    return newDocument;
}
```

Progetto 1g Metodo di callback per l'esecuzione della logica di refactoring.

Il risultato finale del Code Fix è riportato in **Figura 14**. Cliccando sul Light Bulb è possibile visualizzare la preview del refactoring che in questo caso porta alla esplicitazione del tipo generico `Dictionary<string, int>`.

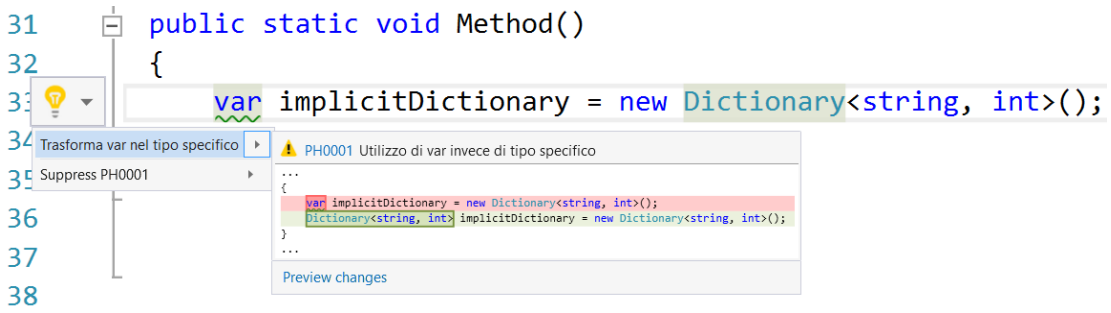

Figura 14 Cliccando sul Light Bulb e selezionando il primo Code Fix, scatta la preview del refactoring. In questo caso viene inferito (in verde) il tipo Dictionary<string, int>.

3. CODE REFACTORING

Il secondo template di progetto della famiglia Extensibility introdotto con Roslyn è il template "**Code Refactoring (VSIX)**". Un progetto di Code Refactoring rappresenta logicamente la sottoparte di Code Fix (in questo contesto chiamata **Refactoring**) di un progetto "Analyzer with Code Fix". Di conseguenza non deve essere segnalata alcuna diagnostica nella Error List di Visual Studio.

Il Code Refactoring è pertanto adatto per tutte quelle situazioni in cui si vuole rifattorizzare una porzione di codice; non perché questa presenti errori sintattici o semantici, ma bensì perché si vuole fornire un meccanismo di automazione che permetta all'utente (sviluppatore) di avere del codice autogenerato.

3.1. STRUTTURA DEL PROGETTO

In maniera molto simile al template Analyzer, un progetto di tipo Code Refactoring si compone di una solution con due progetti:

- Una **Portable Class Library** (PCL): contiene i riferimenti alle librerie indispensabili per un progetto di Code Refactoring tra cui `Microsoft.CodeAnalysis`. In essa va definita esclusivamente la logica di applicazione del Refactoring.
- Un Progetto di deployment: mantiene un riferimento alla PCL e tramite il suo descrittore di deployment **vsixmanifest** consente il packaging e la distribuzione del vsix prodotto all'interno di un'istanza sperimentale di Visual Studio.

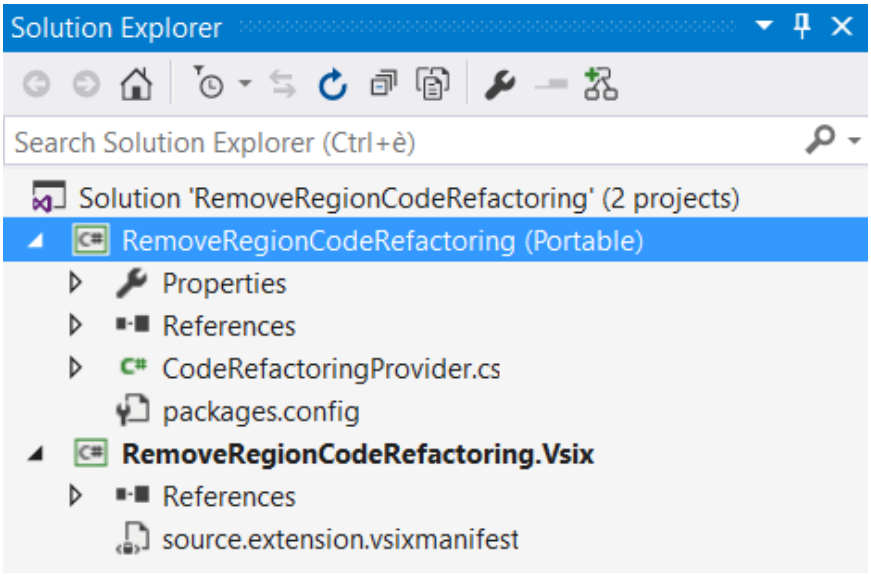

Figura 15 Struttura di un progetto Code Refactoring (VSIX). In alto la PCL. In basso il progetto per il deployment del vsix.

Anche in questo caso, la parte più significativa del progetto si sviluppa all'interno della PCL. Visual Studio crea di default il file CodeRefactoringProvider.cs (o .vb) in cui inserire la logica di refactoring.

Prima di passare al progetto di Refactoring dei MetaType della **Sezione 3.2** occorre illustrare con un esempio meno sofisticato la struttura della classe di Code Refactoring. Per fare questo, in questa sezione richiameremo più volte come **Esempio 18** un semplice progetto di Code Refactoring relativo alla eliminazione delle region C#.

3.1.1 File di Refactoring

La classe responsabile del Code Refactoring deve estendere la classe astratta `CodeRefactoringProvider` ed essere decorata con l'attributo `ExportCodeRefactoringProviderAttribute` con uso del tutto analogo a quello di `ExportCodeFixProviderAttribute` di un Code Fix.

```
[ExportCodeRefactoringProvider(LanguageNames.CSharp, Name
       =nameof(RemoveRegionCodeRefactoringProvider)), Shared]
internal class RemoveRegionCodeRefactoringProvider : CodeRefactoringProvider
{
    // Continua...
}
```

Esempio 18a Definizione della classe RemoveRegionCodeRefactoringProvider.

Inoltre, tale classe deve ridefinire il metodo `ComputeRefactoringsAsync(CodeRefactoringContext context)` per la registrazione del Code Refactoring. L'oggetto contesto, di tipo `CodeRefactoringContext`, in questo caso non è legato ad alcuna diagnostica. Pertanto lo span è unicamente riferito al cursore di inserimento testo nel Code Editor di Visual Studio.

L'**Esempio 18b** riporta l'utilizzo di tale metodo. Nello specifico viene ricavato il `SyntaxTrivia` alla locazione del cursore (`context.Span.Start`), si verifica che il Trivia sia di tipo `RegionDirectiveTrivia`, valore corrispondente all'inizio della direttiva `#region` e si crea e registra la `CodeAction` a partire dal titolo del Refactoring e dal delegato `RemoveRegionAsync`.

```
public sealed override async Task
       ComputeRefactoringsAsync(CodeRefactoringContext context)
{
    SyntaxNode root = await context.Document.
       GetSyntaxRootAsync(context.CancellationToken).ConfigureAwait(false);

    SyntaxTrivia startRegionTrivia = root.FindTrivia(context.Span.Start);
    if (!startRegionTrivia.IsKind(SyntaxKind.RegionDirectiveTrivia))
    {
        return;
    }

    CodeAction action = CodeAction.Create(
      "Remove Region",
      c => RemoveRegionAsync(context.Document, startRegionTrivia, c));
    context.RegisterRefactoring(action);
}
```

Esempio 18c Metodo di registrazione del Refactoring.

Il metodo `RemoveRegionAsync` contiene invece la logica effettiva di applicazione del refactoring. Nello specifico, viene utilizzato il Syntax Rewriter `RegionRewriter`, a cui viene passato tramite costruttore il Trivia corrispondente all'inizio della region, per navigare l'ST in modo da eliminare il livello di regione su cui è posizionato il cursore. Per fare questo, si è ridefinito il metodo `VisitTrivia` di `RegionRewriter` in modo che, per ogni Trivia incontrato nella navigazione dell'ST, si distingui tra il Trivia relativo all'inizio della region (`SyntaxKind.RegionDirectiveTrivia`) e quello relativo alla fine della stessa (`SyntaxKind.EndRegionDirectiveTrivia`). Infine, per risolvere il problema di più region innestate, e dunque più Trivia associati allo stesso Token, è stato previsto un contatore di livello `_level` incrementato per ogni `RegionDirectiveTrivia` successivo a `_startRegionTrivia` e decrementato per ogni `EndRegionDirectiveTrivia` trovato. L'**Esempio 18d** riporta la classe `RegionRewriter` ed il suo utilizzo in `RemoveRegionAsync`.

```
// In RegionRewriter.cs
public class RegionRewriter : CSharpSyntaxRewriter
{
    private readonly SyntaxTrivia _startRegionTrivia;
    private int _level = -1;

    public RegionRewriter(SyntaxTrivia startRegionTrivia) : base(true)
    {
        // L'invocazione del costruttore di base è indispensabile
        // per poter raggiungere i Trivia. Infatti a default
        // (visitIntoStructuredTrivia = false) la
        // navigazione non raggiunge i Syntax Trivia in quanto
        // si ferma al livello di Syntax Node.
        _startRegionTrivia = startRegionTrivia;
    }

    public override SyntaxTrivia VisitTrivia(SyntaxTrivia trivia)
    {
        if (trivia.IsKind(SyntaxKind.RegionDirectiveTrivia))
        {
            if (trivia == _startRegionTrivia)
            {
                _level++;
                return SyntaxFactory.ElasticSpace;
            }
            // Blocking till _startRegionTrivia
            if (_level >= 0)
            {
                _level++;
            }
        }
        else if (trivia.IsKind(SyntaxKind.EndRegionDirectiveTrivia))
        {
            if (_level == 0)
            {
                _level--;
                return SyntaxFactory.ElasticSpace;
            }
            if (_level >= 0)
```

```
            {
                _level--;
            }
        }

        return base.VisitTrivia(trivia);
    }
}

// In CodeRefactoringProvider.cs
private async Task<Document> RemoveRegionAsync(Document document, SyntaxTrivia
startRegionTrivia, CancellationToken c)
{
    SyntaxNode root = await document.
       GetSyntaxRootAsync(c).ConfigureAwait(false);
    RegionRewriter rewriter = new RegionRewriter(startRegionTrivia);
    SyntaxNode newRoot = rewriter.Visit(root);
    Document newDocument = document.WithSyntaxRoot(newRoot);
    return newDocument;
}
```

Esempio 18d Classe RegionRewriter e metodo RemoveRegionAsync.

Per concludere, la **Figura 16** mostra una possibile applicazione del progetto Code Refactoring sviluppato come **Esempio 18**. Alla selezione del Code Refactoring "Remove Region" viene eliminato il livello di regione intermedio poichè il cursore si trova posizionato all'inizio della regione `FullProperties`.

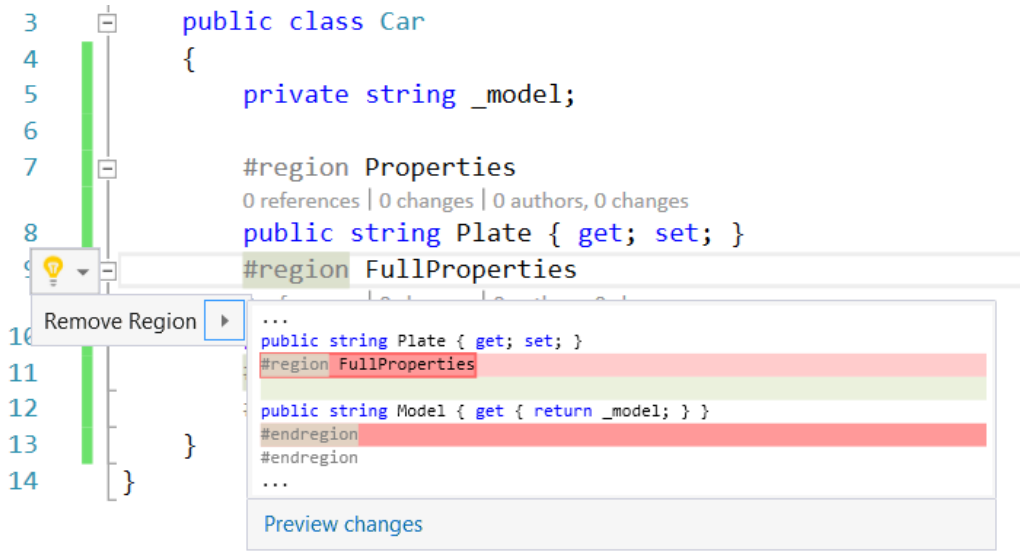

Figura 16 Cliccando sul Light Bulb e selezionando il primo Refactoring, viene mostrato il riquadro relativo alla preview delle modifiche. In questo caso viene eliminata (in rosso) la regione di nome FullProperties.

3.2. REFACTORING DEI METATYPE

Il framework Phoenix costituisce un'estensione del framework .NET atta a fornire una serie di componenti e servizi per semplificare lo sviluppo di applicazioni per ambienti desktop tra cui applicazioni Windows Forms e WPF (Windows Presentation Foundation). Per garantire il riutilizzo dei componenti, l'estendibilità del framework e la sua manutenibilità, Phoenix è stato progettato seguendo i design pattern architetturali Model-View-Presenter (MVP) e Model-View-ViewModel (MVVM).

Più nel dettaglio, lo strato ViewModel di MVVM, contenuto nel namespace `Phoenix.Virtuals`, funge da livello intermedio tra View e Model, garantendone il totale disaccoppiamento. Tra le classi presenti a questo livello, la classe `VirtualObject` permette di incapsulare entità del modello (`target`), facenti parte del namespace `Phoenix.Virtuals.Models`. Inoltre, ogni `VirtualObject` è costituito da proprietà di tipo `VirtualProperty`.

Per automatizzare la creazione di alcuni servizi, Phoenix utilizza una complessa struttura di metadati. Tra di questi esistono metadati particolari chiamati **MetaType**. In generale, una tipologia di MetaType può rappresentare uno o più tipi .NET (e viceversa), arricchendoli di informazioni utili per la trattazione di `VirtualObject` e `VirtualProperty`. Le classi corrispondenti ai MetaType sono tutte contenute nel namespace `Phoenix.Virtuals.MetaTypes` ed ereditano i propri servizi dalla classe astratta `MetaType`.

3.2.1 Specifiche del problema

In Phoenix ogni proprietà delle classi modello può essere associata ad un `MetaType`, il cui tipo concreto dipende dal tipo della proprietà stessa. Il mapping della proprietà al MetaType avviene grazie all'invocazione del metodo statico `VirtualsRepository.SetMetaType` nel costruttore statico della classe contenente la proprietà.

L'**Esempio 19a** mostra il template di utilizzo di tale metodo.

Nello specifico, il metodo accetta come tipo parametrico `T` il tipo della classe in cui è definita la proprietà da associare al MetaType e prende in ingresso due argomenti:

- Un delegato di sistema di tipo `Func<T, object>`.

- Un oggetto MetaType (`TipoMetaType`) istanziato a partite dal tipo della proprietà e inizializzato con la proprietà `Caption`, contenente, di default, il nome trasformato della `VirtualProperty`. Il prefisso `Tipo` di `TipoMetaType` è sempre riferito al nome della classe contenente la proprietà. Ne sono esempi `StringMetaType`, `BooleanMetaType` e `SingleMetaType` rispettivamente per proprietà di tipo `string` (`System.String`), `boolean` (`System.Boolean`) e `float` (`System.Single`).

```
VirtualsRepository.SetMetaType<NomeDellaClasse>(
        target => target.NomeDellaProprietà,
        new TipoMetaType(typeof(TipoDellaProprietà))
        {
                Caption = "NomeDellaProprietàTRASFORMATA",
        });
```

Esempio 19a Template di utilizzo del metodo SetMetaType per l'associazione di un MetaType ad una Proprietà.

Per `NomeDellaProprietàTRASFORMATA` si intende la stringa ottenuta a partire dal nome della proprietà nella notazione "CamelCase" separando ogni occorrenza di carattere minuscolo-maiuscolo con uno spazio. Inoltre, anche eventuali caratteri '_' devono essere rimpiazzati da spazi bianchi.

L'**Esempio 19b** riporta invece un caso reale di associazione tra la proprietà `ItalianPlate` della classe `Car` ed il MetaType `StringMetaType`.

```
public class Car
{
        static Car()
        {
                VirtualsRepository.SetMetaType<Car>(
                        target => target.ItalianPlate,
                        new StringMetaType(typeof(String))
                        {
                                Caption = "Italian Plate",
                        });
        }

        public string Plate { get; set; }
}
```

Esempio 19b Utilizzo del metodo SetMetaType per l'associazione del metadato StringMetaType alla proprietà ItalianPlate di Car.

L'operazione di associazione di una proprietà ad un MetaType può essere parzialmente automatizzata attraverso un'operazione di Refactoring.

Si richiede pertanto che a partire dalla definizione della proprietà venga automaticamente creata l'istruzione di associazione al MetaType e che questa venga aggiunta in fondo al costruttore statico della classe. Se tale costruttore non esiste, deve essere creato e

posizionato all'inizio della classe, dopo eventuali campi ed eventi e prima di costruttori d'istanza, proprietà e metodi. Inoltre, nel caso in cui il costruttore statico contenga già la definizione del MetaType per la proprietà, l'operazione non deve essere consentita. In caso di classi parziali, se il costruttore statico esiste ma è contenuto in un file diverso, l'operazione deve essere effettuata con le stesse modalità sopraripportate facendo in modo che il documento contenente il costruttore statico venga aperto nel Code Editor di Visual Studio. Se il costruttore statico non dovesse esistere in alcun documento relativo alla classe parziale, allora questo deve essere creato all'interno del documento in cui è dichiarata la proprietà.

Inoltre, Qualora non esistessero le direttive using per `Phoenix.Virtuals.MetaTypes` o `Phoenix.Virtuals.Models`, queste devono essere aggiunte in fondo alle direttive esistenti. Inoltre, ad ogni definizione di MetaType, devono sempre essere effettuare le seguenti operazioni:

1. Eliminazione delle direttive non utilizzate.
2. Aggiunta dei namespace di default `System`, `System.Linq`, `System.Text` e `System.Collections.Generic`.
3. Riordinamento degli using secondo lo standard di Phoenix.
4. Raggruppamento delle direttive using e divisione dei gruppi tramite una riga vuota. Un gruppo è costituito da tutte le direttive using aventi lo stesso nome di primo livello nel namespace.

Le quattro operazioni sopraripportate devono essere inoltre offerte come unica opzione di refactoring qualora il cursore dell'utente sia posizionato nella regione di definizione delle direttive.

Per ultimo, si deve dare la possibilità all'utente di scegliere se effettuare l'operazione di definizione dei MetaType solo per la proprietà selezionata o per tutte le proprietà della classe associabili ad un MetaType.

3.2.2 Soluzione

Sulla base delle specifiche introdotte nella sottosezione precedente, si è proceduto a creare un progetto di tipo Code Refactoring.

Questa sottosezione richiamerà per parti il progetto sviluppato, etichettato nell'Appendice come **Progetto 2**, analizzandone le parti più importanti.

Similmente all'**Esempio 18**, la logica di Refactoring è contenuta all'interno della classe `MetaTypeDefinition` estesa a partire da `CodeRefactoringProvider`. Il **Progetto 2a** riporta la prima parte del metodo `ComputeRefactoringsAsync`. Come si può notare, viene inizialmente ottenuto un riferimento al `SyntaxNode` corrispondente alla proprietà di cui definire il MetaType così da estrapolarne le informazioni necessarie per il refactoring. Tuttavia, essendo necessarie alcune informazioni note solo in fase di compilazione, si è dovuti ricorre all'uso del modello semantico (`semanticModel.GetDeclaredSymbol()`) per ricavare il simbolo associato alla proprietà (`IPropertySymbol`). In questo modo, analizzando il suo `ContainingType`, di tipo `INamedTypeSymbol`, si è in grado di verificare se il tipo contenitore della proprietà dichiarata sia una classe o una struttura. In caso negativo (un'interfaccia), il refactoring non viene offerto.

```
[ExportCodeRefactoringProvider(LanguageNames.CSharp, Name =
nameof(MetaTypeDefinition)), Shared]
internal class MetaTypeDefinition : CodeRefactoringProvider
{
    public sealed override async Task ComputeRefactoringsAsync(
       CodeRefactoringContext context)
    {
        SyntaxNode root = await context.Document.GetSyntaxRootAsync(
              context.CancellationToken).ConfigureAwait(false);
        SyntaxNode node = root.FindNode(context.Span);
        PropertyDeclarationSyntax propertyDeclaration =
              node as PropertyDeclarationSyntax;
        if (propertyDeclaration == null)
        {
            return;
        }

        SemanticModel semanticModel = await context.Document.
              GetSemanticModelAsync(context.CancellationToken).
                    ConfigureAwait(false);
        IPropertySymbol propertySymbol = semanticModel.
              GetDeclaredSymbol(propertyDeclaration);
        // Get class/struct node where propertyDeclaration is declared
        INamedTypeSymbol namedTypeSymbol = propertySymbol.ContainingType;
        if (namedTypeSymbol.TypeKind != TypeKind.Class &&
              namedTypeSymbol.TypeKind != TypeKind.Struct)
        {
            // If here we have an interface and we do not offer refactoring
            return;
        }
        // Continua...
    }
}
```

Progetto 2a Prima parte del metodo ComputeRefactoringsAsync.

Per verificare il rispetto di tutte le condizioni di applicabilità del refactoring prima della registrazione della `CodeAction`, si è deciso di creare una classe di utilità `MetaTypeRefactoringInfoProvider`. Passando al costruttore di tale classe il documento corrente ed il simbolo riferito alla proprietà, la classe calcola:

- `NamedTypeSymbol`: interfaccia `INamedTypeSymbol` relativa al tipo contenitore.
- `PropertySymbols`: collezione di `IPropertySymbol`, relativi alle proprietà nella classe, ricavata attraverso il metodo `GetMembers()` di `NamedTypeSymbol`.
- `TypeDeclarations`: lista di eventuali nodi `TypeDeclarationSyntax` riferiti alla stessa classe.
- `CctorSymbol`: `IMethodSymbol` riferito al costruttore statico della classe.
- `CctorDeclaration`: nodo `ConstructorDeclarationSyntax` corrispondente al costruttore statico della classe, se presente.

```
class MetaTypeRefactoringInfoProvider
{
    public MetaTypeRefactoringInfoProvider(Document document,
       IPropertySymbol propertySymbol)
    {
        Document = document;
        PropertySymbol = propertySymbol;
        NamedTypeSymbol = propertySymbol.ContainingType;
        PropertySymbols = NamedTypeSymbol.GetMembers().
            OfType<IPropertySymbol>().Where(s => !s.IsIndexer).ToList();
        TypeDeclarations =
              (from syntaxReference in
                    NamedTypeSymbol.DeclaringSyntaxReferences
               select syntaxReference.GetSyntax() as
                    TypeDeclarationSyntax).ToList();
        CctorSymbol = NamedTypeSymbol.StaticConstructors.SingleOrDefault();
        CctorDeclaration = CctorSymbol?.DeclaringSyntaxReferences.Single().
                           GetSyntax() as ConstructorDeclarationSyntax;
    }
    public Document Document { get; }
    public ConstructorDeclarationSyntax CctorDeclaration { get; }
    private IPropertySymbol PropertySymbol { get; }
    private INamedTypeSymbol NamedTypeSymbol { get; }
    private IMethodSymbol CctorSymbol { get; }
    private IEnumerable<IPropertySymbol> PropertySymbols { get; }
    private IEnumerable<TypeDeclarationSyntax> TypeDeclarations { get; }
    // Continua...
}
```

Progetto 2b Prima parte della classe MetaTypeRefactoringInfoProvider contenente costruttore e proprietà.

Inoltre, la classe `MetaTypeRefactoringInfoProvider` espone dei servizi tra cui:

- `IsPartialType`: verifica che la classe sia parziale e dunque che esistano più nodi `TypeDeclarationSyntax` riferiti alla stessa classe.

- `ExistsStaticConstructor`: verifica la presenza di un costruttore statico.
- `CctorTypeDeclaration`: utile nel caso di classi parziali per risalire al nodo `TypeDeclarationSyntax` (eventualmente in un altro ST) contenente il costruttore statico.
- `CctorSyntaxTree`: permette di risalire all'ST del nodo `ConstructorDeclarationSyntax` riferito al costruttore statico.
- `GetCctorDocument`: ricava il `Document` in cui è contenuto il costruttore statico.
- `GetPropertiesWithMetaType`: permette di trovare i nomi delle proprietà per cui è già stato definito un MetaType.

  Per fare questo, occorre ricercare tra i nodi di tipo `ExpressionStatementSyntax` quelli contenenti una `SimpleLambdaExpressionSyntax` (riferita al delegato `Func<T, object>`) come primo argomento dell'invocazione e accedere al nodo `MemberAccessExpressionSyntax` rappresentante il suo parametro. Infine, a partire da questi si ottengono i nomi delle proprietà.
- `GetPropertySymbolsWithoutMetaType`: permette di trovare gli `IPropertySymbol` riferiti alle proprietà per cui non è già stato definito un MetaType.
- `ExistsMetaTypeForProperty`: permette a partire da un nome di proprietà di verificare se ne è già stato definito il MetaType, sfruttando il metodo `GetPropertiesWithMetaType`.

Tali proprietà e metodi di servizio sono riportati in **Progetto 2c**.

```
class MetaTypeRefactoringInfoProvider
{
    // ...
    public bool IsPartialType { get { return TypeDeclarations.Count() > 1; } }
    public bool ExistsStaticConstructor { get { return CctorSymbol != null; } }
    public TypeDeclarationSyntax CctorTypeDeclaration
    {
        get
        {
            return (from typeDeclaration in TypeDeclarations
                    where typeDeclaration.SyntaxTree == CctorSyntaxTree
                    select typeDeclaration).Single();
        }
    }
    public SyntaxTree CctorSyntaxTree
    {
        get
        {
            if (ExistsStaticConstructor)
            {
                return CctorDeclaration.SyntaxTree;
```

```
            }

            return Document.GetSyntaxTreeAsync().Result;
        }
    }
    public Document GetCctorDocument()
    {
        if (ExistsStaticConstructor && IsPartialType)
        {
            return (from document in Document.Project.Documents
                    where document.SourceCodeKind == SourceCodeKind.Regular
                    where document.GetSyntaxTreeAsync().
                            Result == CctorSyntaxTree
                    select document).Single();
        }
        return Document;
    }
    private IEnumerable<string> GetPropertiesWithMetaType()
    {
        if (ExistsStaticConstructor)
        {
            return from e in CctorDeclaration.Body.Statements.
                                OfType<ExpressionStatementSyntax>()
                   where e.Expression is InvocationExpressionSyntax
                   let firstArgument =
                                (e.Expression as InvocationExpressionSyntax).
                             ArgumentList.Arguments[0].Expression
                   where firstArgument is SimpleLambdaExpressionSyntax
                   let firstArgumentBody =
                          (firstArgument as SimpleLambdaExpressionSyntax).Body
                   where firstArgumentBody is MemberAccessExpressionSyntax
                   select (firstArgumentBody as MemberAccessExpressionSyntax).
                     Name.ToString();
        }
        return Enumerable.Empty<string>();
    }
    public IEnumerable<IPropertySymbol> GetPropertySymbolsWithoutMetaType()
    {
        if (ExistsStaticConstructor)
        {
            return from propertySymbol in PropertySymbols
                   where !GetPropertiesWithMetaType().
                             Contains(propertySymbol.Name)
                   select propertySymbol;
        }
        return PropertySymbols;
    }
    public bool ExistsMetaTypeForProperty(string propertyName)
    {
        if (ExistsStaticConstructor)
        {
            return GetPropertiesWithMetaType().Contains(propertyName);
        }
        return false;
    }
}
```

Progetto 2c Seconda parte della classe MetaTypeRefactoringInfoProvider contenente proprietà e metodi di servizio.

Come già detto, la classe `MetaTypeRefactoringInfoProvider` viene utilizzata all'interno del metodo `ComputeRefactoringsAsync` per verificare se registrare o meno le azioni di Refactoring. Nello specifico possono essere registrate due azioni:

1. `CodeAction` per la proprietà corrente, qualora non sia già stato definito il suo MetaType.
2. `CodeAction` per tutte le proprietà nel caso esista almeno una proprietà per cui è definibile un MetaType.

Se la verifica ha successo, in entrambi i casi viene registrata un'azione a livello di Solution passando alla `CodeAction` il metodo di callback `Task<Solution> DefineMetaTypeMethodAsync` come riportato in **Progetto 2d**.

```
public sealed override async Task ComputeRefactoringsAsync(
       CodeRefactoringContext context)
{
    // ...
    MetaTypeRefactoringInfoProvider infoProvider =
        new MetaTypeRefactoringInfoProvider(
              context.Document, propertySymbol);

    // Registrations:
    // 1) For current property
    if (!infoProvider.ExistsMetaTypeForProperty(propertySymbol.Name))
    {
        CodeAction action = CodeAction.Create(
            $"Define Default MetaType for Property '{propertySymbol.Name}'",
            c => DefineMetaTypeMethodAsync(
                      infoProvider, new[] { propertySymbol }, c));
        context.RegisterRefactoring(action);
    }

    // 2) For all properties
    if (infoProvider.GetPropertySymbolsWithoutMetaType().Any())
    {
        CodeAction action = CodeAction.Create(
            "Define Default MetaType for all Properties",
            c => DefineMetaTypeMethodAsync(infoProvider,
                      infoProvider.GetPropertySymbolsWithoutMetaType(), c));
        context.RegisterRefactoring(action);
    }
}
```

Progetto 2d Seconda parte del metodo ComputeRefactoringsAsync.

Il metodo `DefineMetaTypeMethodAsync` sfrutta la classe `MetaTypeRefactoringInfoProvider` precedentemente descritta per ottenere il `Document` e i nodi `TypeDeclarationSyntax` e `ConstructorDeclarationSyntax` associati al costruttore statico come riportato in **Progetto 2e**.

```
private async Task<Solution> DefineMetaTypeMethodAsync(
       MetaTypeRefactoringInfoProvider infoProvider,
              IEnumerable<IPropertySymbol> propertySymbols,
                     CancellationToken cancellationToken)
{
    Document cctorDocument = infoProvider.GetCctorDocument();
    TypeDeclarationSyntax typeDeclaration = infoProvider.CctorTypeDeclaration;
    ConstructorDeclarationSyntax cctorDeclaration =
       infoProvider.CctorDeclaration;

    CompilationUnitSyntax compilationUnit = (CompilationUnitSyntax)await
      cctorDocument.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);

    UsingDirectivesManager usingDirectivesManager =
      new UsingDirectivesManager(cctorDocument).
        AddNamespaces("Phoenix.Virtuals.Model", "Phoenix.Virtuals.MetaTypes");
    // Continua...
}
```

Progetto 2f Prima parte del metodo DefineMetaTypeMethodAsync.

Per soddisfare la specifica dei requisiti riguardante l'aggiunta dei namespace per i MetaType di Phoenix, è stata progettata la classe di utilità `UsingDirectivesManager`. Il costruttore di tale classe utilizza il documento corrente per memorizzare il nodo radice del documento (`CompilationUnit`) ed il modello semantico (`SemanticModel`) relativo al suo ST. Infine aggiunge gli using e alias trovati nel nodo `CompilationUnitSyntax` alle corrispettive liste interne `_usingDirectives` e `_aliasDirectives`. La lista `_fixedNamespaces` viene utilizzata come sorgente di namespace da aggiungere, se assenti, ad ogni refactoring. La prima parte della classe è riportata in **Progetto 2f**.

```
class UsingDirectivesManager
{
    private readonly List<UsingDirectiveSyntax> _usingDirectives =
       new List<UsingDirectiveSyntax>();
    private readonly List<UsingDirectiveSyntax> _aliasDirectives =
       new List<UsingDirectiveSyntax>();
    private readonly List<string> _fixedNamespaces = new List<string>
    {
        "System", "System.Linq", "System.Text", "System.Collections.Generic",
    };

    public UsingDirectivesManager(Document document)
    {
        Document = document;
        CompilationUnit = (CompilationUnitSyntax)document.
              GetSyntaxRootAsync().Result;
        SemanticModel = document.GetSemanticModelAsync().Result;
        _usingDirectives.AddRange(CompilationUnit.Usings.
              Where(ud => ud.Alias == null));
        _aliasDirectives.AddRange(CompilationUnit.Usings.
              Where(ud => ud.Alias != null));
    }

    private Document Document { get; }
    private CompilationUnitSyntax CompilationUnit { get; }
    private SemanticModel SemanticModel { get; }
```

```
    private IEnumerable<string> Namespaces
    {
        get
        {
            return from usingDirective in _usingDirectives
                   select usingDirective.Name.ToString();
        }
    }
    //Continua...
}
```

Progetto 2f Prima parte della classe UsingDirectivesManager.

La classe `UsingDirectivesManager` espone inoltre due metodi di servizio:

- `AddNamespaces`
- `GetUsingDirectives`

Il primo consente di aggiungere alla lista interna di using le direttive corrispondenti al vettore variabile di stringhe (`namespaces`). Nello specifico, per ogni namespace da aggiungere, si verifica che questo non compaia già tra i namespace salvati in precedenza. In caso positivo si utilizza il metodo di parsificazione `SyntaxFactory.ParseName(@namespace)` per creare un nodo di tipo `NameSyntax` e, a partire da questo, si costruisce il nodo corrispondente alla direttiva using per mezzo del metodo di costruzione `SyntaxFactory.UsingDirective(nameSyntax)`. Una volta creato il nodo `UsingDirectiveSyntax`, questo viene aggiunto alla lista interna `_usingDirectives`.

`GetUsingDirectives` restituisce invece una lista immutabile `SyntaxList` di nodi di tipo `UsingDirectiveSyntax`. Per prima cosa, il metodo sfrutta l'istanza di compilazione accessibile attraverso il modello semantico per ricercare le diagnostiche con identificativo "CS8019", corrispondente al warning *"Unnecessary using directive"*. A partire dallo span di queste diagnostiche vengono ritrovati i nodi `UsingDirectiveSyntax` non necessari ed eliminati dalla lista interna. Successivamente, si aggiungono gli `UsingDirectiveSyntax` di default, ricavati a partire da `_fixedNamespaces`, alla lista interna `_usingDirectives`. La lista `_usingDirectives` viene poi ordinata in base ad un coefficiente di peso stabilito dal metodo di utilità `int WeightOf(string @namespace)`. In caso di nodi con lo stesso peso, si considerano altri parametri (e quindi diversi comparatori) per l'ordinamento della lista. Per ultimo, viene creato il `SyntaxTrivia` per il carattere di nuova linea (`SyntaxFactory.ParseTrailingTrivia(Environment.NewLine)`) e aggiunto in coda a tutti quei nodi `UsingDirectiveSyntax` corrispondenti a direttive il cui

namespace abbia un nome di primo livello differente da quello della direttiva precedente.

La parte rimanente della classe `UsingDirectivesManager` è riportata in **Progetto 2g**.

```
class UsingDirectivesManager
{
    // ...
    public UsingDirectivesManager AddNamespaces(params string[] namespaces)
    {
        foreach (string @namespace in namespaces)
        {
            if (!Namespaces.Contains(@namespace))
            {
                NameSyntax nameSyntax = SyntaxFactory.ParseName(@namespace);
                UsingDirectiveSyntax usingDirective =
                      SyntaxFactory.UsingDirective(nameSyntax).
                             NormalizeWhitespace();
                _usingDirectives.Add(usingDirective);
            }
        }

        return this;
    }
    public SyntaxList<UsingDirectiveSyntax> GetUsingDirectives()
    {
        // 1. Eliminazione dei namespace non utilizzati
        foreach (UsingDirectiveSyntax unusedUsingDirective
              in from d in SemanticModel.Compilation.GetDiagnostics()
              where d.Descriptor.Id == "CS8019"
              where d.Location.SourceTree == CompilationUnit.SyntaxTree
              let ud = (UsingDirectiveSyntax)CompilationUnit.
                     FindNode(d.Location.SourceSpan)
              where ud.Alias == null
              where !_fixedNamespaces.Contains(ud.Name.ToString())
              select ud)
        {
            _usingDirectives.Remove(unusedUsingDirective);
        }

        // 2. Aggiunta dei 4 namespace fissi
        AddNamespaces(_fixedNamespaces.ToArray());

        // 3. Ordinamento dei namespace
        List<UsingDirectiveSyntax> usingDirectives =
            (from ud in _usingDirectives
             let @namespace = ud.Name.ToString()
             let identifiers = @namespace.Split('.')
             orderby WeightOf(identifiers.First()), identifiers.First(),
                     identifiers.ElementAtOrDefault(1)?.Length,
              identifiers.ElementAtOrDefault(1),
                     identifiers.ElementAtOrDefault(2)?.Length,
                     identifiers.ElementAtOrDefault(2),
                     identifiers.ElementAtOrDefault(3)?.Length,
                     identifiers.ElementAtOrDefault(3)
             select ud.WithoutLeadingTrivia().WithTrailingTrivia(
              SyntaxFactory.ParseTrailingTrivia(
                           Environment.NewLine))).ToList();

        // 4. Suddivisione in blocchi aggiungendo una riga vuota tra i diversi
        //    gruppi
        string lastPrimaryIdentifier = null;
        for (int k = 0; k < usingDirectives.Count; k++)
```

```
        {
            string primaryIdentifier =
              usingDirectives[k].Name.ToString().Split('.').First();
            if (primaryIdentifier != lastPrimaryIdentifier)
            {
                lastPrimaryIdentifier = primaryIdentifier;
                if (k > 0)
                {
                   usingDirectives[k] =
                    usingDirectives[k].WithLeadingTrivia(Environment.NewLine));
                }
            }
        }

        return SyntaxFactory.List(usingDirectives).AddRange(_aliasDirectives);
    }
    private int WeightOf(string @namespace)
    {
        switch (@namespace)
        {
            case "System":
                return 0;
            case "Microsoft":
                return 1;
            case "Phoenix":
                return 3;
            case "Sifib":
                return 4;
            default:
                return 2;
        }
    }
}
```

Progetto 2g Seconda parte della classe UsingDirectivesManager.

Il metodo `DefineMetaTypeMethodAsync` verifica in seguito se è stato trovato un costruttore. In caso negativo, procede alla creazione di un nodo `ConstructorDeclarationSyntax`. Nello specifico, viene ricavata la stringa corrispondente all'identificatore del costruttore (`typeDeclaration.Identifier.Text`), creato il suo `SyntaxToken` (`SyntaxFactory.Identifier`) e, a partire da quest'ultimo, si costruisce il nodo `ConstructorDeclarationSyntax` per mezzo del metodo `SyntaxFactory.ConstructorDeclaration`. Infine, viene aggiunta al nodo costruttore una lista immutabile `SyntaxTokenList` di `SyntaxToken`, contenente il token per il modificatore `static`, e un nodo `BlockSyntax` corrispondente al body del costruttore (`WithBody(SyntaxFactory.Block())`).

Una volta creato il nodo per il costruttore statico, lo si aggiunge all'ST, tramite il metodo di estensione `InsertNodesBefore`, alla posizione precedente al primo nodo `PropertyDeclarationSyntax` trovato, sicuramente esistente data l'origine del Refactoring.

Come già detto, per via della natura immutabile degli oggetti in Roslyn, una modifica su un albero, in questo caso il sottoalbero a partire da `typeDeclaration`, ne crea uno nuovo. È pertanto necessario memorizzare il nuovo nodo corrispondente al tipo contenitore (`newTypeDeclaration`) e, a partire da questo, ricavare nuovamente il `ConstructorDeclarationSyntax` per il costruttore statico. Nello specifico, quest'ultima operazione si rende necessaria poiché il vecchio riferimento `cctorDeclaration` risulta ancora puntare ad un nodo `ConstructorDeclarationSyntax` svincolato dal nuovo sottoalbero di `typeDeclaration`.

Nel caso in cui invece il costruttore statico sia stato trovato, si verifica se questo sia contenuto nella classe di definizione della proprietà o in una classe parziale relativa allo stesso tipo. Nel secondo caso, si ricava il `Workspace` relativo al documento contenente il costruttore statico e si utilizza il suo metodo `OpenDocument(DocumentId documentId)` per far si che Visual Studio apri quel documento, come da specifiche.

Poiché questa tipologia di **Workspace API** richiede che il contesto di sincronizzazione del cliente sia la Message Pump, residente dunque nel Thread della UI dell'istanza di Visual Studio, si è utilizzato un event handler per l'evento `WorkspaceChanged`. L'evento viene agganciato nel cliente e sganciato nello stesso event handler e permette di eseguire l'API in oggetto nel contesto di sincronizzazione della Message Pump.

Una volta ottenuto il riferimento al nodo `ConstructorDeclarationSyntax`, sia esso stato creato o meno, si procede alla creazione di un nuovo nodo `BlockSyntax` da sostituire al nodo attuale in `cctorDeclaration.Body`.

Il **Progetto 2h** riporta la seconda parte del metodo `DefineMetaTypeMethodAsync` assieme all'utilizzo dell'event handler `Workspace_WorkspaceChanged`.

```
private async Task<Solution> DefineMetaTypeMethodAsync(
       MetaTypeRefactoringInfoProvider infoProvider,
              IEnumerable<IPropertySymbol> propertySymbols,
                     CancellationToken cancellationToken)
{
    // ...
    SyntaxNode newTypeDeclaration = typeDeclaration;
    if (cctorDeclaration == null)
    {
        cctorDeclaration = SyntaxFactory.ConstructorDeclaration(
            SyntaxFactory.Identifier(typeDeclaration.Identifier.Text)).
              WithModifiers(SyntaxFactory.TokenList(
                     SyntaxFactory.Token(SyntaxKind.StaticKeyword))).
```

```
                            WithBody(SyntaxFactory.Block());
        SyntaxNode firstNode = typeDeclaration.ChildNodes().
                                FirstOrDefault(n =>
                                  n.IsKind(SyntaxKind.PropertyDeclaration));
        newTypeDeclaration = typeDeclaration.InsertNodesBefore(
                                  firstNode, new[] { cctorDeclaration });
        cctorDeclaration = newTypeDeclaration.ChildNodes().
                            OfType<ConstructorDeclarationSyntax>().
                                  Single(c => c.Modifiers.Any(m =>
                                        m.IsKind(SyntaxKind.StaticKeyword)));
    }
    else if (cctorDocument != infoProvider.Document)
    {
        Workspace workspace = cctorDocument.Project.Solution.Workspace;
        workspace.WorkspaceChanged -= Workspace_WorkspaceChanged;
        workspace.WorkspaceChanged += Workspace_WorkspaceChanged;
    }
    BlockSyntax cctorBlock = cctorDeclaration.Body;
    BlockSyntax newCctorBlock = cctorBlock;
    // Continua...
}
private void Workspace_WorkspaceChanged(object sender,
       WorkspaceChangeEventArgs e)
{
    Workspace workspace = (Workspace)sender;
    workspace.WorkspaceChanged -= Workspace_WorkspaceChanged;
    // Invoked on UI Thread
    workspace.OpenDocument(e.DocumentId);
}
```

Progetto 2h In alto la seconda parte del metodo DefineMetaTypeMethodAsync.In basso l'event handler Workspace_WorkspaceChanged.

Dopodiché, per ogni `IPropertySymbol` passato come argomento, si utilizza la classe di utilità `SetMetaTypeStatementBuilder` per la costruzione dello `ExpressionStatementSyntax` relativo alla statement di definizione del MetaType specifico per la proprietà.

La classe `SetMetaTypeStatementBuilder` implementa il pattern creazionale "Builder" per facilitare la creazione dello statement. Nello specifico, viene istanziata passando come argomento del costruttore il simbolo della proprietà (`IPropertySymbol`), memorizzato nella proprietà in sola lettura `Property`. Questa classe espone il metodo `BuildStatement` che calcola:

- `propertyTypeName`: il nome del tipo della proprietà.
- `metaTypeTypeName`: il nome del tipo di MetaType, calcolato come concatenazione di `propertyTypeName` con la stringa `MetaType`.
- `caption`: il nome della proprietà "trasformato" secondo la convenzione di Phoenix. Per calcolare tale campo si è ricorso all'utilizzo di un metodo di utilità

`GetPropertyCaption` facente uso di una regular expression per la separazione delle occorrenze minuscolo-maiuscolo con spazi e del metodo `Replace` di `String` per la sostituzione del carattere '_' con uno spazio bianco.

Infine, viene utilizzato il metodo di parsificazione `SyntaxFactory.ParseStatement` per costruire l'`ExpressionStatementSyntax` relativo alla definizione del MetaType concatenando opportunamente le stringhe sopra calcolate ad una stringa avente ruolo di "template".

Il **Progetto 2i** riporta l'implementazione di questa classe.

```
class SetMetaTypeStatementBuilder
{
    public SetMetaTypeStatementBuilder(IPropertySymbol property)
    {
        if (property == null)
            throw new ArgumentNullException(nameof(property));
        Property = property;
    }

    public IPropertySymbol Property { get; }

    public ExpressionStatementSyntax BuildStatement()
    {
        string propertyTypeName = Property.Type.Name;
        string metaTypeTypeName = propertyTypeName + "MetaType";
        string caption = GetPropertyCaption();
        return SyntaxFactory.ParseStatement($@"
        VirtualsRepository.SetMetaType<{Property.ContainingType.Name}>(target =>
target.{Property.Name}, new {metaTypeTypeName}(typeof({propertyTypeName}))
        {{
            Caption = ""{caption}"",
        }});
") as ExpressionStatementSyntax;
    }

    private string GetPropertyCaption()
    {
        return Regex.Replace(Property.Name.Replace("_", " "),
            "(((?<!^)[A-Z](?=[a-z]))|((?<=[a-z])[A-Z]))", " $1");
    }
}
```

Progetto 2i Classe di utilità SetMetaTypeStatementBuilder per la costruzione dello statement di definizione del MetaType.

Una volta che in `DefineMetaTypeMethodAsync` sono stati ricavati tutti gli `ExpressionStatementSyntax` e aggiunti al nodo `BlockSyntax` `newCctorBlock` del costruttore statico, si rimpiazza il vecchio nodo `cctorBlock` con `newCctorBlock` e si ottiene un nuovo riferimento al nodo del tipo contenitore `newTypeDeclaration`. Inoltre, si ridefinisce un nuovo nodo `CompilationUnitSyntax` sostituendo il vecchio nodo

`typeDeclaration` con `newTypeDeclaration` e aggiungendo (`WithUsings`) le direttive ricavate con `GetUsingDirectives` della classe di utilità `UsingDirectivesManager`. Infine, viene sostituita la radice del documento con il nodo `CompilationUnitSyntax` appena creato e restituita l'intera `Solution` modificata come riportato in **Progetto 2j**.

```
private async Task<Solution> DefineMetaTypeMethodAsync(
       MetaTypeRefactoringInfoProvider infoProvider,
              IEnumerable<IPropertySymbol> propertySymbols,
                     CancellationToken cancellationToken)
{
    // ...
    foreach (IPropertySymbol propertySymbol in propertySymbols)
    {
        SetMetaTypeStatementBuilder builder =
              new SetMetaTypeStatementBuilder(propertySymbol);
        newCctorBlock = newCctorBlock.AddStatements(builder.BuildStatement());
    }
    newTypeDeclaration = newTypeDeclaration.
                              ReplaceNode(cctorBlock, newCctorBlock);
    compilationUnit =
       compilationUnit.ReplaceNode(typeDeclaration, newTypeDeclaration).
              WithUsings(usingDirectivesManager.GetUsingDirectives());
    cctorDocument = cctorDocument.WithSyntaxRoot(compilationUnit);
    Project project = cctorDocument.Project;
    return project.Solution;
}
```

Progetto 2j Terza ed ultima parte del metodo DefineMetaTypeMethodAsync.

Come indicato da specifica, si è proceduto anche a creare un'opzione di Refactoring che comprenda l'eliminazione delle direttive using superflue, l'aggiunta dei namespace di default, l'ordinamento secondo lo standard di Phoenix e la loro divisione in gruppi. Pertanto, si è creato una classe autonoma `UsingDirectivesRefactoring` riportata in **Progetto 2k**.

```
[ExportCodeRefactoringProvider(LanguageNames.CSharp, Name =
nameof(UsingDirectivesRefactoring)), Shared]
internal class UsingDirectivesRefactoring : CodeRefactoringProvider
{
    public sealed override async Task ComputeRefactoringsAsync(
       CodeRefactoringContext context)
    {
        SyntaxNode root = await context.Document.GetSyntaxRootAsync(
              context.CancellationToken).ConfigureAwait(false);
        // Find the node at the selection.
        SyntaxNode node = root.FindNode(context.Span);
        UsingDirectiveSyntax usingDirective = node as UsingDirectiveSyntax;
        if (usingDirective == null)
        {
            return;
        }
        CodeAction action = CodeAction.Create(
              "Ordina gli using con lo standard di Phoenix",
              c => ReorderUsingMethodAsync(context.Document, c));
        context.RegisterRefactoring(action);
```

```
    }

    private async Task<Document> ReorderUsingMethodAsync(Document document,
       CancellationToken cancellationToken)
    {
        CompilationUnitSyntax compilationUnit =
              (CompilationUnitSyntax)await document.GetSyntaxRootAsync(
              cancellationToken).ConfigureAwait(false);
        UsingDirectivesManager usingDirectivesManager =
              new UsingDirectivesManager(document);
        compilationUnit = compilationUnit.WithUsings(usingDirectivesManager.
                          GetUsingDirectives());
        document = document.WithSyntaxRoot(compilationUnit);
        return document;
    }
}
```

Progetto 2k Classe UsingDirectivesRefactoring per il refactoring delle direttive using.

Come si può notare, in `ComputeRefactoringsAsync` si verifica che la locazione del cursore di inserimento del testo nel Code Editor sia corrispondente ad una direttiva using; viceversa non si procede con il refactoring. In caso positivo, a partire dal titolo del Refactoring e dal delegato `ReorderUsingMethodAsync` viene creata e registrata la `CodeAction`.

Il metodo `ReorderUsingMethodAsync` ottiene un riferimento al nodo radice `CompilationUnitSyntax` e, facendo uso della classe di utilità `UsingDirectivesManager`, crea una nuova radice `CompilationUnitSyntax` avente come direttive using tutte quelle ottenute con il metodo `GetUsingDirectives`. Infine, crea e restituisce un documento costituito dalla nuova radice `CompilationUnitSyntax` appena creata.

Il risultato finale dei Refactoring sviluppati per il **Progetto 2** è riportato qui di seguito attraverso quattro esempi significativi corrispondenti alle **Figure 17**, **18**, **19** e **20**.

Nello specifico, la **Figura 17** riporta un semplice scenario di definizione di un MetaType per una singola proprietà. In questo caso esiste già il costruttore statico della classe `Car`, contenente una definizione di MetaType per la proprietà `Id`. Inoltre, tutte le direttive using necessarie sono già presenti, ordinate e partizionate. Pertanto, il risultato del Refactoring è la semplice aggiunta, in fondo al costruttore statico, dello statement di definizione del MetaType per la proprietà `Plate`.

La **Figura 18** riporta uno scenario più complesso in cui il tipo `Car` è una classe parziale. In questo caso, selezionando l'opzione di refactoring per la proprietà `Id`, poichè il costruttore statico si trova in un altro documento questo deve essere aperto (nella Figura

lo è già) e il MetaType per la proprietà aggiunto in fondo al costruttore. Inoltre, poiché nel file Car2.cs non sono definite le direttive necessarie, queste vengono aggiunte, riordinate e partizionate in sezioni.

La **Figura 19** illustra il refactoring per tutte le proprietà contenute nella classe innestata `NestedCar`. Come mostrato nel risultato finale, l'aggiunta degli statement di definizione dei MetaType coinvolge solo il costruttore statico della classe contenente la proprietà. Inoltre, poiché anche in questo caso non sono presenti le direttive using necessarie, queste vengono aggiunte, riordinate e partizionate in sezioni.

La **Figura 20** mostra infine il refactoring dei namespace secondo lo standard di Phoenix. In particolare, vengono cancellate le direttive superflue, aggiunte quelle di default, ordinati i namespace in base ai coefficienti di peso per il nome di primo livello (in ordine: `System`, `Microsoft` e `Phoenix`) e separati i gruppi.

```
using System;
using System.Linq;
using System.Text;
using System.Collections.Generic;

using Phoenix.Virtuals.Model;
using Phoenix.Virtuals.MetaTypes;

namespace SampleNamespace {
    2 references | 0 changes | 0 authors, 0 changes
    class Car
    {
        0 references | 0 changes | 0 authors, 0 changes
        static Car()
        {
            VirtualsRepository.SetMetaType<Car>(target => target.Id, new Int32MetaType(typeof(Int32))
            {
                Caption = "Id",
            });
        }
        1 reference | 0 changes | 0 authors, 0 changes
        public int Id { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        public string Plate { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        public string Model { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        public double Price { get; set; }
    }
}
```

Figura 17a Situazione iniziale. Il costruttore statico comprende già la definizione del MetaType della proprietà Id. Sono già presenti tutte le direttive using necessarie.

Figura 17b Selezione del Refactoring “Define Default MetaType for Property ‘Plate’” posizionandosi su Plate.

```
using System;
using System.Linq;
using System.Text;
using System.Collections.Generic;

using Phoenix.Virtuals.Model;
using Phoenix.Virtuals.MetaTypes;

namespace SampleNamespace {
    3 references | 0 changes | 0 authors, 0 changes
    class Car
    {
        0 references | 0 changes | 0 authors, 0 changes
        static Car()
        {
            VirtualsRepository.SetMetaType<Car>(target => target.Id, new Int32MetaType(typeof(Int32))
            {
                Caption = "Id",
            });
            VirtualsRepository.SetMetaType<Car>(target => target.Plate, new StringMetaType(typeof(String))
            {
                Caption = "Plate",
            });
        }
        1 reference | 0 changes | 0 authors, 0 changes
        public int Id { get; set; }
        1 reference | 0 changes | 0 authors, 0 changes
        public string Plate { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        public string Model { get; set; }
        0 references | 0 changes | 0 authors, 0 changes
        public double Price { get; set; }
    }
}
```

Figura 17c Risultato Finale. Lo statement di definizione del MetaType per la proprietà Plate è stato aggiunto in fondo al costruttore statico.

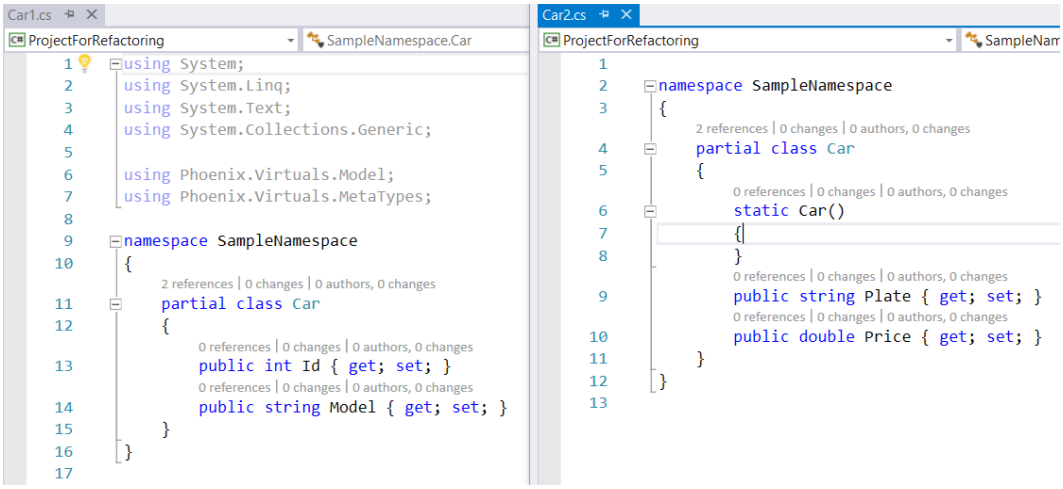

Figura 18a Situazione iniziale. Car è una classe parziale definita sui due file Car1.cs e Car2.cs. Il secondo file contiene il costruttore statico.

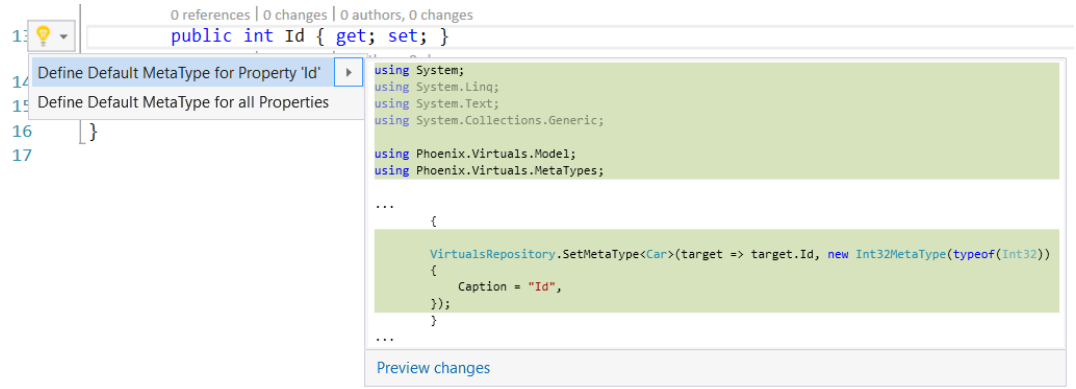

Figura 18b Selezione del Refactoring “Define Default MetaType for Property ‘Id’” posizionandosi su Id.

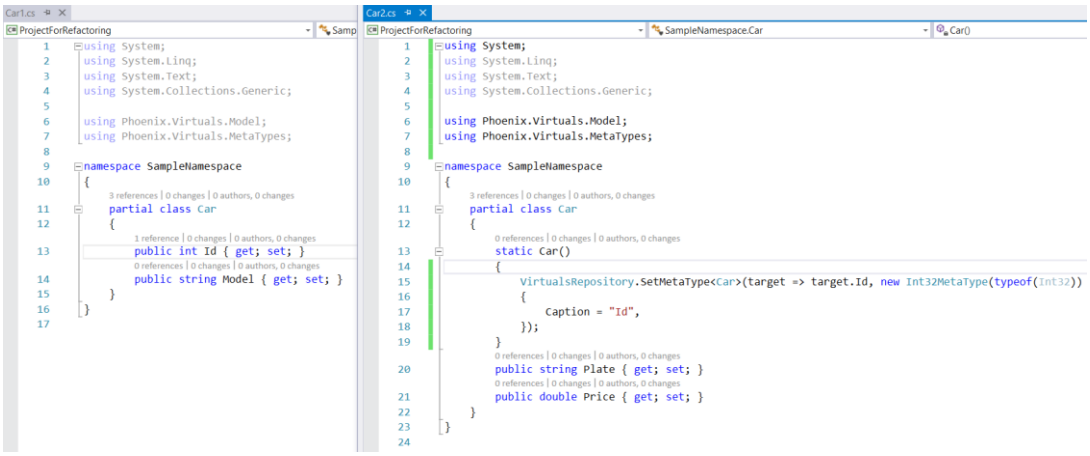

Figura 18c Risultato Finale. Lo statement di definizione del MetaType per la proprietà Id è stato aggiunto in fondo al costruttore statico di Car2.cs. Inoltre sono stati aggiunti i namespace necessari.

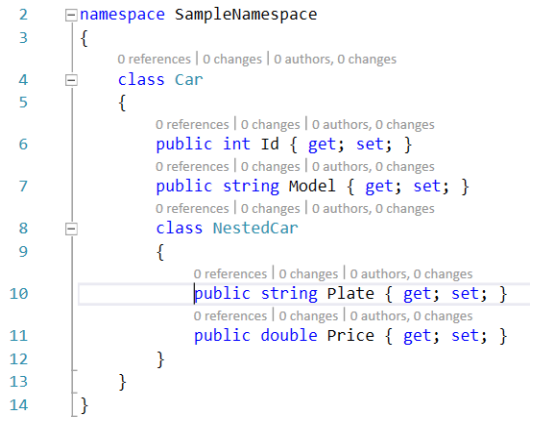

Figura 19a Situazione iniziale. Car è la classe contenitore della classe innestata NestedCar.

Figura 19b Selezione del Refactoring “Define Default MetaType for all Properties” posizionandosi su Plate.

Figura 19c Risultato Finale. Viene aggiunto il costruttore statico nella classe NestedCar e inseriti nel suo body tutti gli statement di definizione di MetaType per le proprietà della classe innestata.

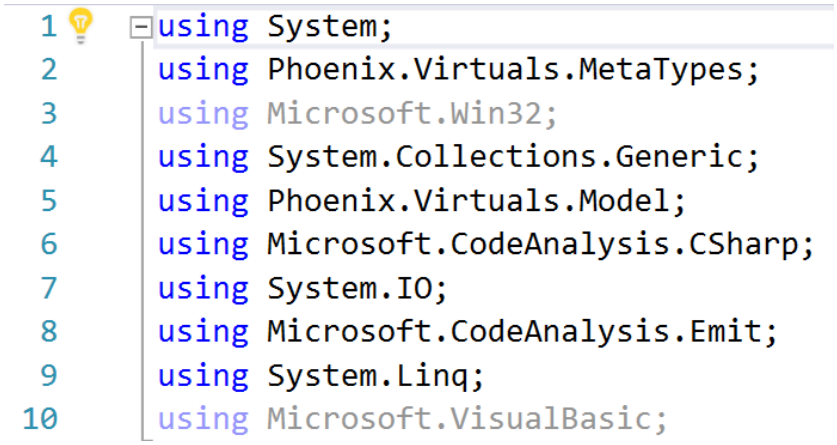

Figura 20a Situazione iniziale. I namespace non sono né ordinati né raggruppati. Inoltre Microsoft.Win32 e Microsoft.VisualBasic sono superflui.

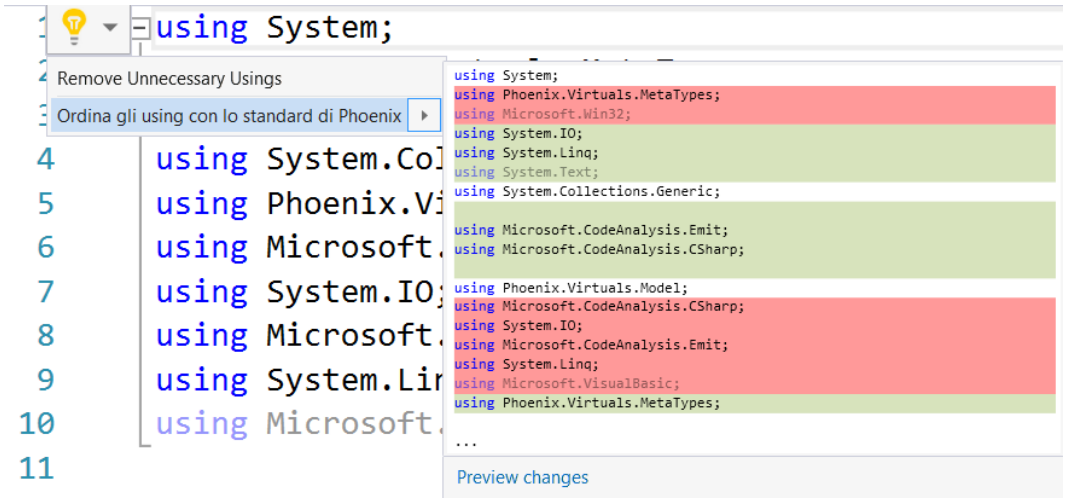

Figura 20b Selezione del Refactoring “Ordina gli using con lo standard di Phoenix” posizionandosi su una qualunque delle direttive.

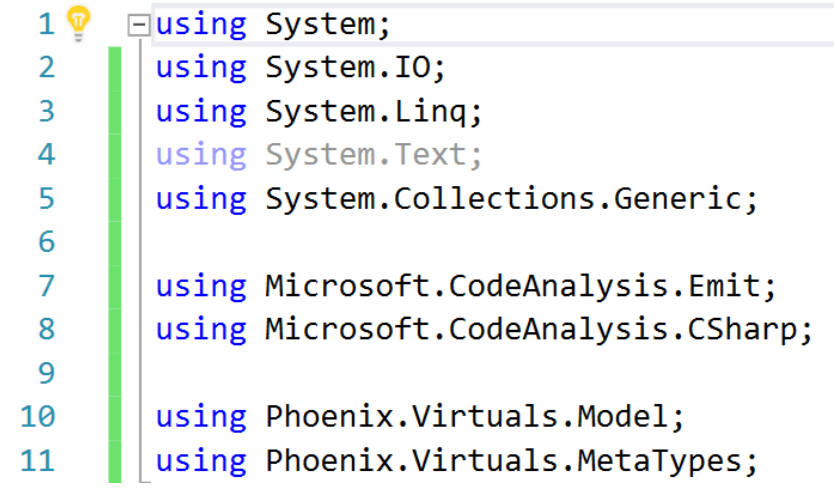

Figura 20c Risultato Finale. Vengono eliminati i namespace Microsoft.Win32 e Microsoft.VisualBasic. Viene aggiunto il namespace System.Text. Infine i namespace vengono riordinati e partizionati in gruppi.

CONCLUSIONI

Il nostro lavoro ha cercato di analizzare le nuove funzionalità introdotte nella .NET Compiler Platform a partire dalla versione 2015 di Visual Studio. Nello specifico, gli esempi sviluppati durante la fase di progetto della tesi sono serviti come utile campo di prova per evidenziare le potenzialità offerte dalla piattaforma per lo sviluppo di tool per l'autogenerazione di codice.

Il progetto della **Sezione 2.2** "Analisi dei var" ci ha permesso di analizzare un problema concettuale molto discusso e controverso nella comunità di sviluppatori C# relativo alle modalità di utilizzo della keyword var e di fornire una nostra interpretazione di soluzione. Tale soluzione ci ha permesso oltretutto di capire come riesce il compilatore a esplicitare i tipi impliciti nella traduzione del codice sorgente in IL. Inoltre è servito come punto di partenza nello studio della piattaforma e nella preparazione allo sviluppo di un tool di refactoring più sofisticato.

Il progetto della **Sezione 3.2** "Refactoring dei MetaType" vuole infatti essere uno strumento integrato nel framework Phoenix per facilitare la scrittura di parti di codice ripetitive come la definizione di un metadato "MetaType" per una proprietà e l'importazione dei tipici namespace del framework. Data la modularità ed estendibilità del codice, molte delle componenti sviluppate per questo progetto potranno essere facilmente riutilizzate ed estese per problemi di refactoring simili, sia interni che esterni al framework Phoenix.

Per concludere, .NET Compiler Platform è un progetto in continua evoluzione; anche grazie al contributo open-source della comunità di sviluppatori. Ad oggi, dopo il rilascio di Visual Studio Update 3, la piattaforma è giunta alla versione 1.3.2 introducendo nuove funzionalità e bug fix. Pertanto, si possono certamente prevedere degli sviluppi futuri di applicazioni per l'analisi e il refactoring di codice sorgente, non soltanto a livello accademico ma anche in un contesto aziendale.

BIBLIOGRAFIA

[1] Microsoft, «Roslyn Overview,» 2016. [Online]. Available at: https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview.

[2] A. Del Sole, Roslyn Succinctly, Syncfusion, 2016.

[3] A. Del Sole, «Roslyn Succinctly GitHub,» 2016. [Online]. Available at: https://github.com/AlessandroDelSole/RoslynSuccinctly.

[4] J. Varty, «Learn Roslyn Now,» 2016. [Online]. Available at: https://joshvarty.wordpress.com/.

[5] A. Turner, «Use Roslyn to Write a Live Code Analyzer for Your API,» 2015. [Online]. Available at: https://msdn.microsoft.com/en-us/magazine/dn879356.aspx.

[6] A. Turner, «Adding a Code Fix to Your Roslyn Analyzer,» 2015. [Online]. Available at: https://msdn.microsoft.com/magazine/dn904670.aspx.

[7] J. Clark, «Demystifying the "var" Keyword in C#,» 2014. [Online]. Available at: http://jeremybytes.blogspot.it/2014/02/demystifying-var-keyword-in-c.html.

[8] S. Cleary, «Async/Await - Best Practices in Asynchronous Programming,» 2013. [Online]. Available at: https://msdn.microsoft.com/en-us/magazine/jj991977.aspx.

RINGRAZIAMENTI

Desidero innanzitutto esprimere un sentito ringraziamento al professor Giuseppe Bellavia per avermi preso come suo laureando e fatto scoprire Roslyn. Inoltre, Lo ringrazio per aver messo a disposizione il suo tempo seguendomi in ogni fase del lavoro.

Ringrazio poi mia madre e i miei due cari nonni per aver creduto in me e permesso di intraprendere la carriera universitaria.

Infine, vorrei ringraziare amici e colleghi per avermi accompagnato in questo percorso e aver reso ogni difficoltà incontrata assieme più facile da superare.

APPENDICE

PROGETTO 1

```
// File VarAnalyzer.cs

using System.Linq;
using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.Diagnostics;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace Analyzers
{
    [DiagnosticAnalyzer(LanguageNames.CSharp)]
    public class VarAnalyzer : DiagnosticAnalyzer
    {
        public const string DiagnosticId = "PH0001";

        private static readonly LocalizableString Title =
              "Trasformazione di var in tipo specifico";
        private static readonly LocalizableString MessageFormat =
              "Utilizzo di var invece di tipo specifico";

        private const string Category = "Types";

        public static DiagnosticDescriptor Rule =
              new DiagnosticDescriptor(DiagnosticId, Title,
                                       MessageFormat, Category,
                                       DiagnosticSeverity.Warning,
                                       isEnabledByDefault: true);

        public override ImmutableArray<DiagnosticDescriptor>
              SupportedDiagnostics
        {
            get { return ImmutableArray.Create(Rule); }
        }

        /// <summary>
        /// The 4 var use cases as stated in
        /// https://msdn.microsoft.com/en-us/library/bb384061.aspx:
        /// 1) local variables
        /// 2) for statement
        /// 3) foreach statement
        /// 4) using statement
        /// </summary>
        /// <param name="context"></param>
        public override void Initialize(AnalysisContext context)
        {
            context.RegisterSyntaxNodeAction(AnalyzeLocalDeclarationStatement,
                                             SyntaxKind.LocalDeclarationStatement);
            context.RegisterSyntaxNodeAction(AnalyzeForStatement,
```

```
                                    SyntaxKind.ForStatement);
    context.RegisterSyntaxNodeAction(AnalyzeForEachStatement,
                                    SyntaxKind.ForEachStatement);
    context.RegisterSyntaxNodeAction(AnalyzeUsingStatement,
                                    SyntaxKind.UsingStatement);
}

private static void AnalyzeLocalDeclarationStatement(
      SyntaxNodeAnalysisContext context)
{
    LocalDeclarationStatementSyntax localDeclaration =
      context.Node as LocalDeclarationStatementSyntax;
    AnalyzeTypeIdentifierName(context,
              localDeclaration?.Declaration.Type as IdentifierNameSyntax);
}

private static void AnalyzeForStatement(
      SyntaxNodeAnalysisContext context)
{
    ForStatementSyntax forStatement =
      context.Node as ForStatementSyntax;
    AnalyzeTypeIdentifierName(
      context,
      forStatement?.Declaration.Type as IdentifierNameSyntax);
}

private static void AnalyzeForEachStatement(
      SyntaxNodeAnalysisContext context)
{
    ForEachStatementSyntax forEachStatement =
      context.Node as ForEachStatementSyntax;
    AnalyzeTypeIdentifierName(
      context,
      forEachStatement?.Type as IdentifierNameSyntax);
}

private static void AnalyzeUsingStatement(
      SyntaxNodeAnalysisContext context)
{
    UsingStatementSyntax usingStatement =
      context.Node as UsingStatementSyntax;
    AnalyzeTypeIdentifierName(context,
      usingStatement?.Declaration.Type as IdentifierNameSyntax);
}

private static void AnalyzeTypeIdentifierName(SyntaxNodeAnalysisContext
      context, IdentifierNameSyntax typeIdentifierName)
{
    if (typeIdentifierName != null &&
      IsInferableTypeIdentifier(context, typeIdentifierName))
    {
        Rule = new DiagnosticDescriptor(
              DiagnosticId, Title, MessageFormat, Category,
              DiagnosticSeverity.Warning, isEnabledByDefault: true);
        Diagnostic diagnostic =
              Diagnostic.Create(Rule, typeIdentifierName.GetLocation());
        context.ReportDiagnostic(diagnostic);
    }
}

private static bool IsInferableTypeIdentifier(SyntaxNodeAnalysisContext
      context, IdentifierNameSyntax typeIdentifierName)
{
```

```
            if (!typeIdentifierName.IsVar)
            {
                return false;
            }
            // Get the inferred Type
            SemanticModel semanticModel = context.SemanticModel;
            TypeInfo typeInfo = semanticModel.GetTypeInfo(typeIdentifierName);
            ITypeSymbol inferredType = typeInfo.ConvertedType;
            if (inferredType.Name == "var")
            {
                // It may be a variable of a type named 'var'.
                // In this case we don't produce a diagnostic.
                return false;
            }
            if (inferredType.IsAnonymousType)
            {
                // var construct is considered legit for an anonymous type.
                return false;
            }
            INamedTypeSymbol inferredNamedtype =
              inferredType as INamedTypeSymbol;
            if (inferredNamedtype != null && inferredNamedtype.IsGenericType)
            {
                ImmutableArray<ITypeSymbol> typeArguments =
                      inferredNamedtype.TypeArguments;
                if (typeArguments != null &&
                      typeArguments.Any(ta => ta.IsAnonymousType))
                {
                    // In this case we have an IEnumerable<T> where T
                    // is an anonymous type.
                    // var construct is condidered legit for an anonymous list.
                    return false;
                }
            }
            return true;
        }
    }
}
```

```
// File VarCodeFix.cs

using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Composition;
using System.Collections.Immutable;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeFixes;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace Analyzers
{
    [ExportCodeFixProvider(LanguageNames.CSharp,
       Name = nameof(VarCodeFix)), Shared]
    public class VarCodeFix : CodeFixProvider
    {
        private const string CodeFixTitle = "Trasforma var nel tipo specifico";

        public sealed override ImmutableArray<string> FixableDiagnosticIds
        {
            get { return ImmutableArray.Create(VarAnalyzer.DiagnosticId); }
        }

        public sealed override async Task RegisterCodeFixesAsync(
              CodeFixContext context)
        {
            SyntaxNode root = await context.Document.GetSyntaxRootAsync(
              context.CancellationToken).ConfigureAwait(false);
            Diagnostic diagnostic = context.Diagnostics.First();
            switch (diagnostic.Id)
            {
                case VarAnalyzer.DiagnosticId:
                    RegisterVarCodeFix(context, root, diagnostic);
                    break;
            }
        }

        private static void RegisterVarCodeFix(CodeFixContext context,
              SyntaxNode root, Diagnostic diagnostic)
        {
            IdentifierNameSyntax typeIdentifierName = root.FindNode(
              context.Span) as IdentifierNameSyntax;
            CodeAction action = CodeAction.Create(
              title: CodeFixTitle,
              createChangedDocument: c => VarCodeFixAsync(
                     context.Document, typeIdentifierName, c),
              equivalenceKey: CodeFixTitle);
            context.RegisterCodeFix(action, diagnostic);
        }

        private static async Task<Document> VarCodeFixAsync(Document document,
              IdentifierNameSyntax typeIdentifierName,
                     CancellationToken cancellationToken)
        {
            SyntaxNode syntaxRoot = await document.GetSyntaxRootAsync(
              cancellationToken).ConfigureAwait(false);
            SemanticModel semanticModel = await document.GetSemanticModelAsync(
              cancellationToken).ConfigureAwait(false);
            TypeInfo typeInfo = semanticModel.GetTypeInfo(typeIdentifierName);
            ITypeSymbol inferredType = typeInfo.ConvertedType;
```

```
            TypeSyntax inferredVariableTypeName =
              SyntaxFactory.ParseTypeName(
                     inferredType.ToDisplayString(
                            SymbolDisplayFormat.MinimallyQualifiedFormat)).
                                   WithLeadingTrivia(
                                          typeIdentifierName.
                                                 GetLeadingTrivia()).
                                   WithTrailingTrivia(
                                          typeIdentifierName.
                                                 GetTrailingTrivia());
            SyntaxNode newSyntaxRoot = syntaxRoot.ReplaceNode(
              typeIdentifierName, inferredVariableTypeName);
            Document newDocument = document.WithSyntaxRoot(newSyntaxRoot);
            return newDocument;
        }
    }
}
```

PROGETTO 2

```
// File MetaTypeDefinition.cs

using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Composition;
using System.Collections.Generic;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CodeRefactorings;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;


namespace CodeRefactorings
{
    [ExportCodeRefactoringProvider(LanguageNames.CSharp,
       Name = nameof(MetaTypeDefinition)), Shared]
    internal class MetaTypeDefinition : CodeRefactoringProvider
    {
        public sealed override async Task ComputeRefactoringsAsync(
       CodeRefactoringContext context)
        {
            SyntaxNode root = await context.Document.GetSyntaxRootAsync(
                                    context.CancellationToken).ConfigureAwait(false);
            // Find the node at the selection.
            SyntaxNode node = root.FindNode(context.Span);
            PropertyDeclarationSyntax propertyDeclaration =
              node as PropertyDeclarationSyntax;
            if (propertyDeclaration == null)
            {
                return;
            }
            SemanticModel semanticModel = await context.Document.
                            GetSemanticModelAsync(context.CancellationToken).
                                    ConfigureAwait(false);
            IPropertySymbol propertySymbol = semanticModel.
              GetDeclaredSymbol(propertyDeclaration);
            // Get class/struct node where propertyDeclaration is declared
            INamedTypeSymbol namedTypeSymbol = propertySymbol.ContainingType;
            if (namedTypeSymbol.TypeKind != TypeKind.Class &&
                    namedTypeSymbol.TypeKind != TypeKind.Struct)
            {
                // If here we have an interface and we do not refactor.
                return;
            }
            MetaTypeRefactoringInfoProvider infoProvider =
              new MetaTypeRefactoringInfoProvider(
                    context.Document, propertySymbol);
            // Registrations:
            // 1) For current property
            if (!infoProvider.ExistsMetaTypeForProperty(propertySymbol.Name))
            {
                CodeAction action = CodeAction.Create(
               $"Define Default MetaType for Property '{propertySymbol.Name}'",
               c => DefineMetaTypeMethodAsync(infoProvider, new[] {
                                            propertySymbol }, c));
                context.RegisterRefactoring(action);
            }
```

```
    // 2) For all properties
    if (infoProvider.GetPropertySymbolsWithoutMetaType().Any())
    {
        CodeAction action = CodeAction.Create(
              "Define Default MetaType for all Properties",
               c => DefineMetaTypeMethodAsync(
                     infoProvider,
                     infoProvider.GetPropertySymbolsWithoutMetaType(),
              c));
        context.RegisterRefactoring(action);
    }
}

private async Task<Solution> DefineMetaTypeMethodAsync(
      MetaTypeRefactoringInfoProvider infoProvider,
      IEnumerable<IPropertySymbol> propertySymbols,
      CancellationToken cancellationToken)
{
    Document cctorDocument = infoProvider.GetCctorDocument();
    TypeDeclarationSyntax typeDeclaration =
      infoProvider.CctorTypeDeclaration;
    ConstructorDeclarationSyntax cctorDeclaration =
      infoProvider.CctorDeclaration;

    CompilationUnitSyntax compilationUnit =
      (CompilationUnitSyntax)await cctorDocument.GetSyntaxRootAsync(
      cancellationToken).ConfigureAwait(false);
    UsingDirectivesManager usingDirectivesManager =
      new UsingDirectivesManager(cctorDocument).
        AddNamespaces(
              "Phoenix.Virtuals.Model", "Phoenix.Virtuals.MetaTypes");

    SyntaxNode newTypeDeclaration = typeDeclaration;
    // If the constructor was not found, here we create it
    if (cctorDeclaration == null)
    {
        // Static Constructor creation
        cctorDeclaration =
              SyntaxFactory.ConstructorDeclaration(
                     SyntaxFactory.Identifier(typeDeclaration.
                     Identifier.Text)).
              WithModifiers(SyntaxFactory.TokenList(
                     SyntaxFactory.Token(SyntaxKind.StaticKeyword))).
                            WithBody(SyntaxFactory.Block());
        // Take the first node that is either an instance constructor,
        // method or property
        SyntaxNode firstNode = typeDeclaration.ChildNodes().
                                  FirstOrDefault(n =>
                                    n.IsKind(SyntaxKind.PropertyDeclaration));
        // New class with empty static constructor on top
        newTypeDeclaration = typeDeclaration.InsertNodesBefore(
                                    firstNode, new[] { cctorDeclaration });
        // AST for cctorDeclaration has changed
        cctorDeclaration = newTypeDeclaration.ChildNodes().
                             OfType<ConstructorDeclarationSyntax>().
                               Single(c => c.Modifiers.Any(m =>
                                        m.IsKind(SyntaxKind.StaticKeyword)));
    }
    else if (cctorDocument != infoProvider.Document)
    {
        Workspace workspace = cctorDocument.Project.Solution.Workspace;
        workspace.WorkspaceChanged -= Workspace_WorkspaceChanged;
        workspace.WorkspaceChanged += Workspace_WorkspaceChanged;
```

```
            }
            // Adding SetMetaType...
            BlockSyntax cctorBlock = cctorDeclaration.Body;
            BlockSyntax newCctorBlock = cctorBlock;
            foreach (IPropertySymbol propertySymbol in propertySymbols)
            {
                SetMetaTypeStatementBuilder builder =
                      new SetMetaTypeStatementBuilder(propertySymbol);
                newCctorBlock =
                      newCctorBlock.AddStatements(builder.BuildStatement());
            }
            newTypeDeclaration = newTypeDeclaration.ReplaceNode(
              cctorBlock, newCctorBlock);
            compilationUnit =
              compilationUnit.ReplaceNode(
                 typeDeclaration, newTypeDeclaration).
                    WithUsings(usingDirectivesManager.GetUsingDirectives());
            cctorDocument = cctorDocument.WithSyntaxRoot(compilationUnit);
            Project project = cctorDocument.Project;
            return project.Solution;
        }

        private void Workspace_WorkspaceChanged(object sender,
              WorkspaceChangeEventArgs e)
        {
            Workspace workspace = (Workspace)sender;
            workspace.WorkspaceChanged -= Workspace_WorkspaceChanged;
            workspace.OpenDocument(e.DocumentId);
        }
    }
}
```

```
// File MetaTypeRefactoringInfoProvider.cs

using System.Linq;
using System.Collections.Generic;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace CodeRefactorings
{
    class MetaTypeRefactoringInfoProvider
    {
        public MetaTypeRefactoringInfoProvider(Document document,
             IPropertySymbol propertySymbol)
        {
            Document = document;
            PropertySymbol = propertySymbol;
            NamedTypeSymbol = propertySymbol.ContainingType;
            PropertySymbols = NamedTypeSymbol.GetMembers().
                                  OfType<IPropertySymbol>().
                                     Where(s => !s.IsIndexer).ToList();
            TypeDeclarations =
              (from syntaxReference in
              NamedTypeSymbol.DeclaringSyntaxReferences
               select syntaxReference.GetSyntax() as
                    TypeDeclarationSyntax).ToList();
            CctorSymbol =
              NamedTypeSymbol.StaticConstructors.SingleOrDefault();
            CctorDeclaration =
              CctorSymbol?.DeclaringSyntaxReferences.Single().
                    GetSyntax() as ConstructorDeclarationSyntax;
        }

        public Document Document { get; }

        /// <summary>
        /// Restituisce il nodo relativo al costruttore statico del tipo
        /// NamedTypeSymbol (può essere null)
        /// </summary>
        public ConstructorDeclarationSyntax CctorDeclaration { get; }

        /// <summary>
        /// Restituisce true se il tipo NamedTypeSymbol è definito su più
        /// documenti (partial)
        /// </summary>
        public bool IsPartialType
        {
              get
              {
                    return TypeDeclarations.Count() > 1;
              }
        }

        /// <summary>
        /// Restituisce true se esiste il costruttore statico
        /// </summary>
        public bool ExistsStaticConstructor
        {
              get
              {
                    return CctorSymbol != null;
```

```
    }
}

/// <summary>
/// Restituisce il TypeDeclaration che contiene o deve contenere il
/// costruttore statico
/// Nel caso di tipi parziali, il costruttore statico potrebbe
/// essere in un documento diverso da quello che contiene la proprietà
/// coinvolta nel refactoring
/// </summary>
public TypeDeclarationSyntax CctorTypeDeclaration
{
    get
    {
        return (from typeDeclaration in TypeDeclarations
                where typeDeclaration.SyntaxTree == CctorSyntaxTree
                select typeDeclaration).Single();
    }
}

/// <summary>
/// Restituisce il SyntaxTree che contiene o deve contenere il
/// costruttore statico
/// Nel caso di tipi parziali, il costruttore statico potrebbe
/// essere in un SyntaxTree diverso da quello che contiene la proprietà
/// coinvolta nel refactoring
/// </summary>
public SyntaxTree CctorSyntaxTree
{
    get
    {
        if (ExistsStaticConstructor)
        {
            return CctorDeclaration.SyntaxTree;
        }
        else
        {
            return Document.GetSyntaxTreeAsync().Result;
        }
    }
}

/// <summary>
/// Restituisce il Document che contiene o deve contenere il
/// costruttore statico
/// Nel caso di tipi parziali, il costruttore statico potrebbe
/// essere in un Document diverso da quello che contiene la proprietà
/// coinvolta nel refactoring
/// </summary>
public Document GetCctorDocument()
{
    if (ExistsStaticConstructor && IsPartialType)
    {
        return (from document in Document.Project.Documents
                where document.SourceCodeKind ==
                   SourceCodeKind.Regular
                where document.GetSyntaxTreeAsync().Result ==
                   CctorSyntaxTree
                select document).Single();
    }
```

```
    else
    {
        return Document;
    }
}

public IEnumerable<IPropertySymbol>
     GetPropertySymbolsWithoutMetaType()
{
    if (ExistsStaticConstructor)
    {
        return from propertySymbol in PropertySymbols
               where !GetPropertiesWithMetaType().
                   Contains(propertySymbol.Name)
               select propertySymbol;
    }
    return PropertySymbols;
}

public bool ExistsMetaTypeForProperty(string propertyName)
{
    if (ExistsStaticConstructor)
    {
        return GetPropertiesWithMetaType().Contains(propertyName);
    }
    return false;
}

/// <summary>
/// Restituisce la proprietà alla quale si vuole associare il
/// MetaType
/// </summary>
private IPropertySymbol PropertySymbol { get; }

/// <summary>
/// Restituisce il tipo (class o struct) che contiene la proprietà
/// </summary>
private INamedTypeSymbol NamedTypeSymbol { get; }

/// <summary>
/// Restituisce il costruttore statico del tipo NamedTypeSymbol
/// (può essere null)
/// </summary>
private IMethodSymbol CctorSymbol { get; }

/// <summary>
/// Restituisce l'elenco di tutte le proprietà (non indexer)
/// dichiarate nel tipo NamedTypeSymbol
/// </summary>
private IEnumerable<IPropertySymbol> PropertySymbols { get; }

/// <summary>
/// Restituisce l'elenco di tutte le dichiarazioni (in 1+ documenti)
/// del tipo NamedTypeSymbol
/// </summary>
private IEnumerable<TypeDeclarationSyntax> TypeDeclarations{ get; }

private IEnumerable<string> GetPropertiesWithMetaType()
{
    if (ExistsStaticConstructor)
```

```
            {
                return from e in CctorDeclaration.Body.Statements.
                                    OfType<ExpressionStatementSyntax>()
                       where e.Expression is InvocationExpressionSyntax
                       let firstArgument =
                           (e.Expression as InvocationExpressionSyntax).
                                  ArgumentList.Arguments[0].Expression
                       where firstArgument is SimpleLambdaExpressionSyntax
                       let firstArgumentBody = (firstArgument as
                                       SimpleLambdaExpressionSyntax).Body
                       where firstArgumentBody
                           is MemberAccessExpressionSyntax
                       select (firstArgumentBody as
                           MemberAccessExpressionSyntax).Name.ToString();
            }
            return Enumerable.Empty<string>();
        }
    }
}
```

```
// File SetMetaTypeStatementBuilder.cs

using System;
using System.Text.RegularExpressions;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace CodeRefactorings
{
    class SetMetaTypeStatementBuilder
    {
        public SetMetaTypeStatementBuilder(IPropertySymbol property)
        {
            if (property == null)
                throw new ArgumentNullException(nameof(property));
            Property = property;
        }

        public IPropertySymbol Property { get; }

        public ExpressionStatementSyntax BuildStatement()
        {
            string propertyTypeName = Property.Type.Name;
            // REGOLE PER TROVARE IL NOME DEL METATYPE
            string metaTypeTypeName = propertyTypeName + "MetaType";
            string caption = GetPropertyCaption();
            return SyntaxFactory.ParseStatement($@"
VirtualsRepository.SetMetaType<{Property.ContainingType.Name}>(target =>
target.{Property.Name}, new {metaTypeTypeName}(typeof({propertyTypeName}))
            {{
                Caption = ""{caption}"",
            }});
") as ExpressionStatementSyntax;
        }

        private string GetPropertyCaption()
        {
            return Regex.Replace(Property.Name.Replace("_", " "),
                "(((?<!^)[A-Z](?=[a-z]))|((?<=[a-z])[A-Z]))", " $1");
        }
    }
}
```

```
// File UsingDirectivesManager.cs

using System;
using System.Linq;
using System.Collections.Generic;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace CodeRefactorings
{
    class UsingDirectivesManager
    {
        private readonly List<UsingDirectiveSyntax> _usingDirectives =
              new List<UsingDirectiveSyntax>();
        private readonly List<UsingDirectiveSyntax> _aliasDirectives =
              new List<UsingDirectiveSyntax>();
        private readonly List<string> _fixedNamespaces = new List<string>()
        {
            "System",
            "System.Linq",
            "System.Text",
            "System.Collections.Generic",
        };

        public UsingDirectivesManager(Document document)
        {
            Document = document;
            CompilationUnit = (CompilationUnitSyntax)document.
                                  GetSyntaxRootAsync().Result;
            SemanticModel = (SemanticModel)document.
                                  GetSemanticModelAsync().Result;
            _usingDirectives.AddRange(CompilationUnit.Usings.
                                  Where(ud => ud.Alias == null));
            _aliasDirectives.AddRange(CompilationUnit.Usings.
                                  Where(ud => ud.Alias != null));
        }

        public UsingDirectivesManager AddNamespaces(params string[] namespaces)
        {
            foreach (string @namespace in namespaces)
            {
                if (!Namespaces.Contains(@namespace))
                {
                    NameSyntax nameSyntax = SyntaxFactory.ParseName(@namespace);
                    UsingDirectiveSyntax usingDirective =
                      SyntaxFactory.UsingDirective(nameSyntax).
                             NormalizeWhitespace();
                    _usingDirectives.Add(usingDirective);
                }
            }
            return this;
        }

        public SyntaxList<UsingDirectiveSyntax> GetUsingDirectives()
        {
            //  1. Eliminazione dei namespace non utilizzati
            foreach (UsingDirectiveSyntax unusedUsingDirective in
              from d in SemanticModel.Compilation.GetDiagnostics()
              where d.Descriptor.Id == "CS8019"
              where d.Location.SourceTree == CompilationUnit.SyntaxTree
              let ud = (UsingDirectiveSyntax)CompilationUnit.
                     FindNode(d.Location.SourceSpan)
```

```
        where ud.Alias == null
        where !_fixedNamespaces.Contains(ud.Name.ToString())
        select ud)
    {
        _usingDirectives.Remove(unusedUsingDirective);
    }
    //  2. Aggiunta dei 4 namespace fissi
    AddNamespaces(_fixedNamespaces.ToArray());
    //  3. Ordinamento dei namespace
    List<UsingDirectiveSyntax> usingDirectives =
        (from ud in _usingDirectives
         let @namespace = ud.Name.ToString()
         let identifiers = @namespace.Split('.')
         orderby WeightOf(identifiers.First()), identifiers.First(),
             identifiers.ElementAtOrDefault(1)?.Length,
              identifiers.ElementAtOrDefault(1),
             identifiers.ElementAtOrDefault(2)?.Length,
              identifiers.ElementAtOrDefault(2),
             identifiers.ElementAtOrDefault(3)?.Length,
              identifiers.ElementAtOrDefault(3)
         select ud.WithoutLeadingTrivia().WithTrailingTrivia(
              SyntaxFactory.ParseTrailingTrivia(
                     Environment.NewLine))).ToList();
    //  4. Suddivisione in blocchi aggiungendo una riga vuota tra i
    //     diversi blocchi
    string lastPrimaryIdentifier = null;
    for (int k = 0; k < usingDirectives.Count; k++)
    {
        string primaryIdentifier =
             usingDirectives[k].Name.ToString().Split('.').First();
        if (primaryIdentifier != lastPrimaryIdentifier)
        {
            lastPrimaryIdentifier = primaryIdentifier;
            if (k > 0)
            {
                usingDirectives[k] =
                     usingDirectives[k].WithLeadingTrivia(
                                  SyntaxFactory.ParseTrailingTrivia(
                                         Environment.NewLine));
            }
        }
    }
    return SyntaxFactory.List(usingDirectives)
      .AddRange(_aliasDirectives);
}

private Document Document { get; }

private CompilationUnitSyntax CompilationUnit { get; }

private SemanticModel SemanticModel { get; }

private IEnumerable<string> Namespaces
{
    get
    {
        return from usingDirective in _usingDirectives
               select usingDirective.Name.ToString();
    }
}

private int WeightOf(string @namespace)
{
```

```
            switch (@namespace)
            {
                case "System":
                    return 0;
                case "Microsoft":
                    return 1;
                case "Phoenix":
                    return 3;
                case "Sifib":
                    return 4;
                default:
                    return 2;
            }
        }
    }
}
```

```
// File UsingDirectivesRefactoring.cs

using System.Composition;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CodeActions;
using Microsoft.CodeAnalysis.CodeRefactorings;
using Microsoft.CodeAnalysis.CSharp.Syntax;

namespace CodeRefactorings
{
    [ExportCodeRefactoringProvider(LanguageNames.CSharp,
       Name = nameof(UsingDirectivesRefactoring)), Shared]
    internal class UsingDirectivesRefactoring : CodeRefactoringProvider
    {
        public sealed override async Task ComputeRefactoringsAsync(
              CodeRefactoringContext context)
        {
            SyntaxNode root = await context.Document.GetSyntaxRootAsync(
                                  context.CancellationToken).ConfigureAwait(false);
            // Find the node at the selection.
            SyntaxNode node = root.FindNode(context.Span);
            UsingDirectiveSyntax usingDirective = node as UsingDirectiveSyntax;
            if (usingDirective == null)
            {
                return;
            }
            CodeAction action = CodeAction.Create(
              "Ordina gli using con lo standard di Phoenix",
               c => ReorderUsingMethodAsync(context.Document, c));
            context.RegisterRefactoring(action);
        }

        private async Task<Document> ReorderUsingMethodAsync(Document document,
              CancellationToken cancellationToken)
        {
            CompilationUnitSyntax compilationUnit =
              (CompilationUnitSyntax)await document.GetSyntaxRootAsync(
                                        cancellationToken).ConfigureAwait(false);
            UsingDirectivesManager usingDirectivesManager =
              new UsingDirectivesManager(document);
            compilationUnit = compilationUnit.WithUsings(
                                  usingDirectivesManager.GetUsingDirectives());
            document = document.WithSyntaxRoot(compilationUnit);
            return document;
        }
    }
}
```