ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA

TESI DI LAUREA

in

Ingegneria del software L-A

PROGETTO E REALIZZAZIONE DI UN COMPONENTE PER LA VISUALIZZAZIONE E LA VALUTAZIONE DI ABSTRACT SYNTAX TREE

CANDIDATO

Andrea Asta

RELATORE

Chiar.mo Prof. Giuseppe Bellavia

Anno Accademico 2009/10

Sessione II

Sommario

Introduzione

L’obiettivo di questa tesi è la realizzazione di un componente software per la visualizzazione e l’esecuzione di Expression Tree. Un Expression Tree è la specifica utilizzata da .NET per descrivere gli Abstract Syntax Tree, ossia strutture dati in grado di modellare codice compilabile e, quindi, eseguibile.

La prima parte consiste nell’analisi della struttura di un albero di espressione, nelle sue componenti principali e nei tipi di nodo più utilizzati, nonché nello studio delle classi che ne permettono la visita. Il modello di riferimento è quello delle Expression Tree versione 2 del Framework .NET 4.0. In questa versione, rispetto alla precedente (comunque compatibile), è possibile rappresentare un numero molto maggiore di nodi e, quindi, di costrutti sintattici.

Il passaggio immediatamente successivo consiste nello studio e realizzazione della parte esecutiva di una espressione: per prima cosa è stato scelto un sottoinsieme, tra tutte le possibili espressioni, per cui fornire un modulo di compilazione ed esecuzione; si è proceduto poi con l’analisi dei requisiti di compilabilità di una espressione, alla realizzazione del modulo per la raccolta dei dati di input, a quello per la compilazione e generazione del risultato e a quello per la visualizzazione dello stesso.

La seconda parte consiste nella creazione di un controllo utente per la rappresentazione dell’albero e la visualizzazione di tutte le proprietà specifiche di ogni singolo nodo. Per realizzare questa parte è stata utilizzata la tecnologia Windows Presentation Foundation (WPF), in parte integrata con controlli Windows Form, che consente una grossa flessibilità nella progettazione grafica dell’interfaccia utente. Per lo sviluppo, sono state utilizzate tecniche quali stili, data template, trigger e data binding.

Terminato lo sviluppo del componente software nella sua globalità, questo è stato incapsulato in una libreria DLL e utilizzata da un programma di test esterno per verificarne la funzionalità.

Il progetto, nella sua interezza, è stato ideato e realizzato rispettando le regole dell'ingegneria del software.

La fase di analisi è partita dalla raccolta e analisi dei requisiti, passando poi alla stesura di casi d'uso e scenari.

In fase di progettazione si è posta attenzione alle regole di progettazione object oriented e all'utilizzo di design pattern: in particolare, si è cercato di mantenere il più possibile separati la logica di rappresentazione grafica da quella di modellazione dei dati, utilizzando il pattern di progettazione Document – View, una variante del più noto Model – View – Controller in cui il controllo è distribuito tra documento e vista.

Inoltre, si è prestata attenzione alla pulizia del codice e alla riduzione al minimo indispensabile di porzioni di codice duplicate.

Il presente documento è articolato in capitoli:

- Il primo capitolo – "*Expression Tree*" – presenta una panoramica sulle espressioni e sugli alberi di espressione di Visual Studio 2010, sulla loro struttura e sulle operazioni eseguibili;
- Il secondo capitolo – "*WPF*" – presenta una panoramica sul mondo Windows Presentation Foundation e sulle tecniche utilizzate per scrivere software utilizzando questa libreria;
- Il terzo capitolo – "*Fase di analisi*" – contiene tutta l'analisi del progetto da realizzare: studio del problema della visualizzazione di espressioni, studio del problema di esecuzione di espressioni, definizione del glossario dei termini, casi d'uso e scenari;
- Il quarto capitolo – "*Modello dei dati*" – contiene la progettazione del modello dei dati per il software, ossia quella parte che descrive le espressioni, le sue componenti e le operazioni desiderate;
- Il quinto capitolo – "*Progetto grafico*" – contiene la progettazione del layout grafico e dell'interazione con l'utente e con il modello dei dati;

- Il sesto capitolo – "*Assemblaggio*" – specifica il tipo di prodotto distribuito all'utente finale;
- Il settimo capitolo – "*Test*" – contiene alcuni test effettuati con il componente software creato.
- Seguono due appendici: la prima – "*Appendice A*" – contiene dei richiami sulla teoria della progettazione software orientata agli oggetti e sui design pattern utilizzati nel progetto; la seconda – "*Appendice B*" – contiene dei richiami sui metodi e sui delegati generici, una caratteristica sintattica di C#.

CAPITOLO 1: Expression Tree

Introduzione

Un Expression Tree (ET) è una struttura dati del framework .NET in grado di rappresentare codice compilabile ed eseguibile, mediante rappresentazione ad albero. Ogni nodo di tale albero rappresenta una operazione, un operando, o una qualsiasi altra struttura dati.

Gli ET fecero la loro comparsa fin dalla versione 3.5 (Visual Studio 2008) del framework, ma avevano ridotte capacità espressive: potevano rappresentare solo pochi tipi di operazioni elementari (assegnazione e altri operatori binari) e, di fatto, la loro utilità era limitata a particolari applicazioni (legate essenzialmente a LINQ).

Le potenzialità espressive di questa struttura dati hanno invece avuto maggiore importanza con l'introduzione del framework 4 e del Dynamic Language Runtime: l'idea è quella di definire un metodo comune per rappresentare porzioni di codice – l'Expression Tree per l'appunto – e utilizzare questo sistema per far cooperare tra loro linguaggi diversi. Essenzialmente, i linguaggi dinamici di DLR (quali IronPython, IronRuby e altri) devono produrre un ET ed eventualmente alcune classi di supporto, lasciando poi al framework stesso il compito di gestire tutto il resto.

La versione di ET implementata nel framework 4 è chiamata ET 2: essa, pur mantenendo l'ossatura e tutte le caratteristiche della struttura precedente, aggiunge la possibilità di rappresentare una grande mole di tipologie di istruzioni, tra le quali blocchi, cicli e strutture di controllo, rendendo così di fatto possibile la rappresentazione di quasi ogni tipologia di insieme di istruzioni.

Struttura

Ogni ET è modellato da un albero i cui nodi sono sottoclassi della classe astratta `Expression`: ad esempio, una espressione binaria è modellata da una istanza della classe `BinaryExpression`.

Ciascun nodo espone una serie di proprietà atte ad identificare le caratteristiche specifiche del tipo di espressione rappresentata. Ci sono due proprietà, definite alla radice della gerarchia, particolarmente significative:

- Proprietà `NodeType`: definisce il tipo di operazione definita dal nodo (ad esempio, può discriminare la specifica operazione di una espressione binaria);
- Proprietà `Type`: restituisce il tipo di oggetto che l'espressione rappresenta (ad esempio, per un nodo di tipo `BinaryExpression` rappresenta il tipo restituito dall'operazione).

Esistono anche alcune classi particolari, che non discendono da `Expression`, e rappresentano parti costitutive (sotto – nodi) dell'espressione principale. Ad esempio, un nodo `SwitchExpression` può contenere una serie di `SwitchCase`, dove quest'ultima classe non discende da `Expression`, poiché di fatto non modella una espressione. Questi nodi particolari sono solitamente contenitori di altre espressioni.

Un'altra caratteristica importante degli ET è la loro immutabilità: non è possibile modificare l'albero, aggiungendo, rimuovendo o modificando i nodi. Per compiere questa operazione, è necessario costruire un nuovo albero, copiando i nodi che interessano e modificando o trascurando gli altri.

Tutte le classi inerenti agli ET sono contenute nel namespace `System.Linq.Expressions`.

Riassumendo, una espressione può essere vista come una serie di nodi collegati in una struttura solitamente ad albero (ma non necessariamente, se si pensa alle espressioni contenenti salti nel codice), in cui quasi ogni nodo, qualsiasi cosa rappresenti, ha una radice comune: la classe `Expression`.

Creazione

Esistono due modalità differenti per creare un ET:

- Creazione da una lambda expression e utilizzo della classe generica `Expression<TDelegate>`;

- Creazione mediante funzioni factory fornite da `Expression`.

Nel primo caso, è sufficiente assegnare una lambda ad una variabile di tipo `Expression<TDelegate>`, dove il tipo generico deve essere, ovviamente, lo stesso del tipo della lambda expression (solitamente si usano i delegati generici della famiglia `Func` e `Action`). Va precisato che, utilizzando questa tecnica, si possono utilizzare solo espressioni semplici, formate da una sola istruzione di calcolo e prive quindi di blocchi, cicli e istruzioni condizionali. L'istruzione seguente definisce un ET che modella una semplice somma di 2 unità ad una variabile di tipo intero.

```
Expression<Func<int, int>> expression1 = a => a + 2;
```

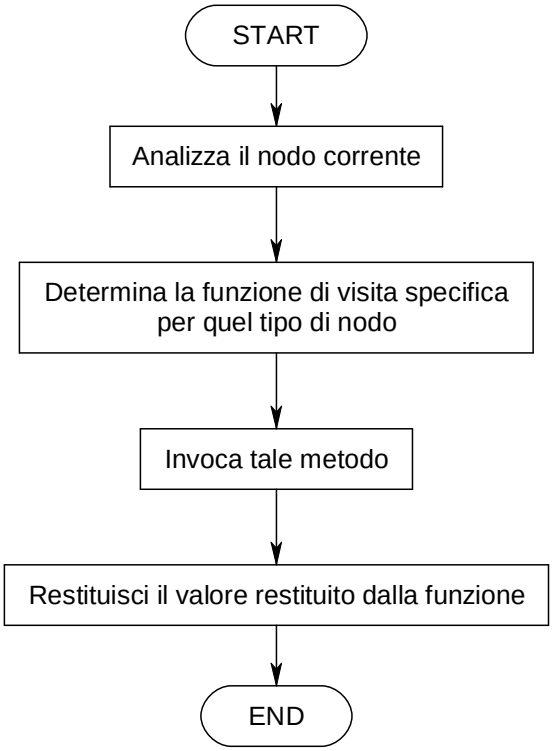
Il secondo metodo consiste nell'utilizzare una serie di funzioni factory, fornite come metodi statici della classe astratta `Expression`. Esistono una serie di funzioni per la creazione di nodi di vario tipo. Il frammento di codice seguente definisce la stessa espressione dell'esempio precedente, ma utilizzando le funzioni factory.

```
ParameterExpression parameter2 =
Expression.Parameter(typeof(int), "a");

Expression expression2 =
Expression.Lambda<Func<int, int>>(
  Expression.MakeBinary
  (
    ExpressionType.Add,
    parameter2,
    Expression.Constant(2)
  ),
  parameter2
);
```

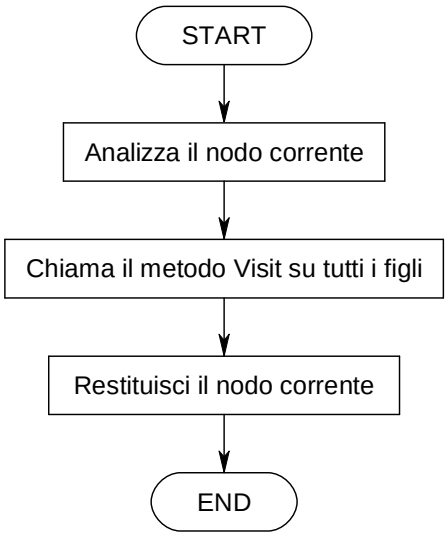
Visita dell'albero

La visita di un Expression Tree avviene mediante una struttura a pattern Visitor fornita dal namespace `System.Linq.Expressions`. La classe

`ExpressionVisitor` definisce un metodo `Visit()`, che visita un albero di espressione a partire dal nodo passatogli come parametro. Questo metodo, a sua volta, analizza il nodo corrente e tutti i figli ricorsivamente, delegando il compito della visita dei singoli tipi di nodo ad altrettanti metodi specifici (ad esempio, per i nodi di tipo `ParameterExpression` è presente un metodo `VisitParameter()`). La struttura generale della visita è quindi la seguente:



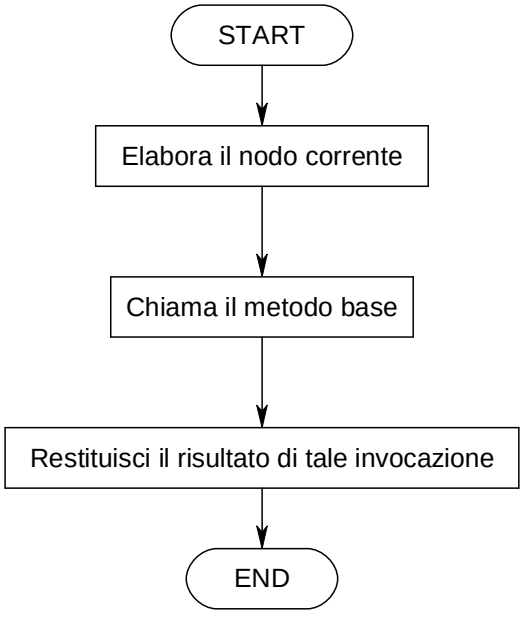
I metodi specifici, a loro volta, svolgono il loro compito in questo modo:



La classe `ExpressionVisitor`, di fatto, si limita a scorrere tutti i nodi, restituendo il nodo visitato ad ogni invocazione. Quindi, non ha alcuna effettiva utilità: per questo motivo è definita come classe astratta. Si può sfruttare il meccanismo già definito ed estendere la classe per aggiungere funzionalità. In generale, una sottoclasse di `ExpressionVisitor` ridefinisce i metodi di interesse per i singoli tipi di nodo. Ad esempio, se si desidera stampare a console l'elenco dei nodi di tipo `ParameterExpression`, sarà sufficiente ridefinire il metodo `VisitParameter()` in modo che:

```
protected override Expression VisitParameter
    (ParameterExpression node)
{
  Console.Writeln(node);
  return base.VisitParameter(node);
}
```

In generale, il processo è sempre analogo:



Rispettando questa metodologia (aggiunta di funzionalità e richiamo del metodo della classe base), il meccanismo di visita dell'albero rimane invariato, ma si aggiungono tutte le funzionalità di interesse (ovviamente, il Visitor può mantenere un proprio stato interno e svolgere così funzioni anche piuttosto complesse nella visita dell'albero).

Compilazione ed esecuzione

Ogni nodo di tipo `LambdaExpression` può essere compilato in un delegato, che a sua volta può essere eseguito in maniera statica o dinamica. Siccome è possibile costruire un nodo `LambdaExpression` partendo da un qualsiasi oggetto `Expression`, si può affermare, in maniera generale, che un qualsiasi ET sia compilabile e quindi eseguibile.

Il metodo `Compile()` della classe `LambdaExpression` fornisce, come risultato, un oggetto derivato dalla classe base di tutti i delegati, ossia `Delegate`. Se si utilizzano i delegati generici, è possibile ottenere e invocare in maniera statica il delegato ottenuto:

```
Expression<Func<int, bool>> lambda = num => num < 5;
Func<int, bool> result = lambda.Compile();
Console.WriteLine(result(4));
```

In alternativa, si può utilizzare l'invocazione dinamica:

```
Expression<Func<int, bool>> lambda = num => num < 5;
Delegate expressionDelegate = lambda.Compile();
object result =
  expressionDelegate.DynamicInvoke(4);
```

Chiaramente, è possibile che un nodo di espressione sia stato costruito con successo, ma non possa essere compilato od eseguito. In questo caso sarà lanciata una eccezione all'atto della compilazione o a quello dell'esecuzione.

Lista delle classi

Di seguito è mostrata una panoramica delle classi che modellano nodi di alberi di espressione. Tutte le classi descritte appartengono al namespace `System.Linq.Expressions`.

BinaryExpression

Modella una operazione binaria tra due espressioni (memorizzate nelle due proprietà `Left` e `Right`).

- Classe base: `Expression`

- Costruttore: `Expression.MakeBinary()` oppure costruttori specifici (sempre come metodi statici di `Expression`) per le singole operazioni

Sono modellabili tutte le operazioni binarie, in particolare di seguito è mostrata la lista dei `NodeType` ottenibili:

- `Add`: addizione;
- `AddChecked`: addizione con controllo overflow;
- `Divide`: divisione;
- `Modulo`: resto della divisione;
- `Multiply`: prodotto;
- `MultiplyChecked`: prodotto con controllo overflow;
- `Power`: elevamento a potenza;
- `Subtract`: sottrazione;
- `SubtractChecked`: sottrazione con controllo overflow;
- `And`: AND bit a bit;
- `Or`: OR bit a bit;
- `ExclusiveOr`: XOR bit a bit;
- `LeftShift`: shift a sinistra;
- `RightShift`: shift a destra;
- `AndAlso`: AND condizionale che valuta il secondo operando solo se necessario;
- `OrElse`: OR condizionale che valuta il secondo operando solo se necessario;
- `Equal`: uguaglianza;
- `NotEqual`: disuguaglianza;

- `GreaterThanOrEqual`; maggiore o uguale;
- `GraterThan`: strettamente maggiore;
- `LessThan`: strettamente minore;
- `LessThanOrEqual`: minore o uguale;
- `Coalesce`: fusione;
- `ArrayIndex`: accesso indicizzato ad un array mono – dimensionale;

BlockExpression

Modella un blocco di codice, contenente variabili (proprietà `Variables`) e un insieme di istruzioni (proprietà `Expressions`).

- Classe base: `Expression`
- Costruttore: `Expression.Block()`

CatchBlock

Rappresenta una istruzione catch in un blocco `try`. Contiene una espressione che modella il corpo del blocco (`Body`).

- Classe base: `object`
- Costruttore: `Expression.Catch()`

ConditionalExpression

Rappresenta una istruzione condizionale, del tipo `if…then…else`. Contiene una espressione per il test (`Test`), una da eseguire in caso di test positivo (`IfTrue`) e una da eseguire in caso di test negativo (`IfFalse`).

- Classe base: `Expression`
- Costruttore: `Expression.Condition()`, ma anche `Expression.IfThen()` e `Expression.IfThenElse`

ConstantExpression

Modella una costante nel codice, contenuta nella proprietà `Value`.

- Classe base: `Expression`

- Costruttore: `Expression.Constant()`

DebugInfoExpression

Imposta una serie di punti utilizzati dai debugger in fase di test del codice.

- Classe base: `Expression`
- Costruttore: `Expression.DebugInfo()`

DefaultExpression

Rappresenta il valore di default per un tipo specificato (nella proprietà `Type`, sovrascritta dalla classe base).

- Classe base: `Expression`
- Costruttore: `Expression.Default()`

DynamicExpression

Modella una operazione dinamica.

- Classe base: `Expression`
- Costruttore: `Expression.Dynamic()`

ElementInit

Rappresenta un inizializzatore per un singolo valore di una collezione che implementa l'interfaccia `IEnumerable`.

- Classe base: `object`
- Costruttore: `ExpressionElementInit()`

GotoExpression

Rappresenta una istruzione di salto non condizionato verso una label (istanza di `LabelTarget` contenuta nella proprietà `Target`), tra cui istruzioni di `return`, di `break`, di `continue` e altri tipi di salto.

- Classe base: `Expression`
- Costruttore: `Expression.Goto()`

IndexExpression

Modella l'accesso indicizzato ad un array o ad una proprietà indicizzata.

- Classe base: `Expression`
- Costruttore: `Expression.ArrayAccess()` per array; Expression`.MakeIndex()` e `Expression.Property()` per le proprietà

InvocationExpression

Rappresenta l'invocazione ad un delegato o ad una lambda expression.

- Classe base: `Expression`
- Costruttore: `Expression.Invoke()`

LabelExpression

Modella una label con valore di ritorno, associata ad una istanza di `LabelTarget` contenuta nella proprietà `Target`.

- Classe base: `Expression`
- Costruttore: `Expression.Label()`

LabelTarget

Rappresenta il target di una espressione di salto o di label.

- Classe base: `object`
- Costruttore: `Expression.Label()`

LambdaExpression

Modella una lambda expression o, più in generale, un metodo. L'espressione interna è contenuta nella proprietà `Body` e ogni istanza è compilabile mediante i metodi `Compile()`.

- Classe base: `Expression`
- Costruttore: `Expression.Lambda()`

ListInitExpression

Rappresenta un inizializzatore per un insieme di elementi. Contiene una espressione di allocazione (istanza di `NewExpression` memorizzata nella

proprietà `NewExpression`) e una lista di elementi di inizializzazione (lista di `ElementInit` memorizzata nella proprietà `Initializers`).

- Classe base: `Expression`
- Costruttore: `Expression.ListInit()`

LoopExpression

Rappresenta un ciclo infinito. Per uscire, è necessaria una istruzione di `break` (espressione istanza di `BreakExpression`). L'espressione corpo del ciclo è memorizzata nella proprietà `Body`, la destinazione di `break` in `BreakLabel` e quella di `continue` in `ContinueLabel`.

- Classe base: `Expression`
- Costruttore: `Expression.Loop()`

MemberAssignment

Rappresenta l'assegnazione di un valore ad una proprietà o campo di un oggetto.

- Classe base: `MemberBinding`
- Costruttore: `Expression.Bind()`

MemberBinding

Classe base per le classi che modellano l'associazione di un valore ad un membro di un oggetto.

- Classe base: `object`

MemberExpression

Modella l'accesso ad un campo o una proprietà di una istanza di oggetto.

- Classe base: `Expression`
- Costruttore: `Expression.PropertyOrField()`, oppure `Expression.Property()`, oppure `Expression.Field()`

MemberInitExpression

Modella la chiamata ad un costruttore (istanza di `NewEpxression` memorizzata nella proprietà `NewExpression`), più l'inizializzazione di campi e proprietà (lista di `MemberBinding` memorizzata nella proprietà `Bindings`).

- Classe base: `Expression`
- Costruttore: `Expression.MemberInit()`

MemberListBinding

Rappresenta l'inizializzazione di una collezione di oggetti in una istanza appena creata.

- Classe base: `MemberBinding`
- Costruttore: `Expression.ListBind()`

MemberMemberBinding

Rappresenta l'inizializzazione dei membri di un membro di una istanza appena creata.

- Classe base: `MemberBinding`
- Costruttore: `Expression.MemberBind()`

MethodCallExpression

Rappresenta l'invocazione ad un metodo statico o di istanza.

- Classe base: `Expression`
- Costruttore: `Expression.MethodCall()` e `Expression.ArrayIndex()`

NewArrayExpression

Rappresenta la costruzione ed eventuale inizializzazione di un array.

- Classe base: `Expression`
- Costruttore: `Expression.NewArrayBounds()` o `Expression.NewArrayInit()`

NewExpression

Rappresenta la chiamata ad un costruttore.

- Classe base: `Expression`
- Costruttore: `Expression.New()`

ParameterExpression

Rappresenta un parametro o variabile con nome. Può essere gestito per riferimento settando il flag nella proprietà `IsByRef`.

- Classe base: `Expression`
- Costruttore: `Expression.Parameter()`

RuntimeVariablesExpression

Rappresenta delle variabili create a runtime.

- Classe base: `Expression`
- Costruttore: `Expression.RuntimeVariables()`

SwitchCase

Rappresenta un blocco `case` della struttura `switch`…`case`. Il valore del `case` è contenuto nella proprietà `TestValues`, mentre il corpo nella proprietà `Body`.

- Classe base: `object`
- Costruttore: `Expression.SwitchCase()`

SwitchExpression

Rappresenta un blocco `switch`. E' possibile specificare una lista di oggetti `SwitchCase` (memorizzati nella proprietà `Cases`), nonché un caso `default` (memorizzato nella proprietà `DefaultBody`).

- Classe base: `Expression`
- Costruttore: `Expression.Switch()`

SymbolDocumentInfo

Memorizza le informazioni necessarie al debug di un particolare file sorgente in un determinato linguaggio.

- Classe base: `object`

- Costruttore: `Expression.SymbolDocument()`

TryExpression

Rappresenta un blocco `try`...`catch`...`finally`. Il corpo del blocco `try` è memorizzato nella proprietà `Body`, il corpo del blocco `finally` in `Finally`, mentre la lista di handler per le eccezioni è contenuta nella proprietà `Handlers`, definita come collezione di oggetti di tipo `CatchBlock`.

- Classe base: `Expression`
- Costruttore: `Expression.TryCatch()`

TypeBinaryExpression

Rappresenta un test per verificare se un oggetto è di un certo tipo.

- Classe base: `Expression`
- Costruttore: `Expression.TypeIs()`

UnaryExpression

Modella una espressione unaria su una espressione, memorizzata nella proprietà `Operand`.

- Classe base: `Expression`
- Costruttore: `Expression.MarkeUnary()` oppure costruttori specifici (sempre come metodi statici di `Expression`) per le singole operazioni

Sono modellabili tutte le operazioni unarie, in particolare di seguito è mostrata la lista dei `NodeType` ottenibili:

- `ArrayLenght`: lunghezza di un array;
- `Convert`: conversione numerica che non solleva eccezioni in caso di fallimento;
- `ConvertChecked`: conversione numerica che solleva eccezioni in caso di fallimento;
- `Negate`: negazione aritmetica;

- `NegateChecked`: negazione aritmetica con controllo overflow;
- `Not`: negazione logica;
- `Quote`: modella una espressione che riporta il valore costante di un'altra espressione;
- `TypeAs`: conversione di valore, con valore null restituito in caso di fallimento;
- `UnaryPlus`.

CAPITOLO 2: WPF

Introduzione

Windows Presentation Foundation (WPF) è una libreria che consente di creare interfacce utente per applicazioni .NET, introdotta dalla versione 3.0 del framework.

Una delle principali caratteristiche di WPF è che ogni elemento grafico può, in generale, contenere qualsiasi tipologia di oggetto: è possibile, ad esempio, inserire un filmato dentro ad un pulsante e il pulsante dentro ad una griglia. Utilizzando tecniche particolari (data binding, stili e data template) è addirittura possibile rappresentare in maniera molto semplice qualsiasi istanza di oggetto, sia esso definito da WPF o dall'utente.

XAML e Code Behind

Una delle principali caratteristiche di WPF è la possibilità di descrivere le interfacce grafiche mediante un dialetto XML chiamato XAML (XML for Applications Makrkup Language).

Ad ogni elemento XAML è associata una classe .NET della libreria WPF; tramite gli attributi e gli elementi figli si impostano gli attributi delle istanze e le relazioni di contenimento gerarchico. E' immediato quindi comprendere come, in ogni caso, sia possibile anche descrivere una interfaccia grafica mediante normale codice C#. In generale, comunque, ad ogni file XAML è associato un omonimo file di codice: questo codice è chiamato code behind.

Il codice XAML, per default, è compilato in codice BAML (Binary Application Markup Language) e aggiunto come risorsa all'eseguibile.

Grazie a XAML, il lavoro di grafici e sviluppatori può essere mantenuto ben differenziato e facilmente integrato: il grafico descrive l'interfaccia utente mediante il codice XAML, mentre lo sviluppatore si occupa di aggiungere funzionalità dinamiche tramite code behind.

Ogni tag XAML è associato in maniera univoca ad una classe C# con lo stesso nome.

Risorse

All'interno di ogni elemento WPF è possibile inserire delle risorse, ossia porzioni di codice che saranno accessibili dall'elemento stesso e da tutti i suoi figli.

Le risorse possono essere accedute direttamente da XAML, in due modalità:

- Ricerca statica (estensione di markup `{StaticResource nomeRisorsa}`): in questo caso le risorse sono ricercate in fase di caricamento dell'oggetto e applicate subito; questo implica che, se durante l'esecuzione del programma la risorsa subisce modifiche, queste non saranno propagate agli elementi che le utilizzano in modo statico,
- Ricerca dinamica (estensione di markup `{DynamicResource nomeRisorsa}`): in questo caso gli oggetti che utilizzano una risorsa ricevono le modifiche della risorsa stessa durante l'esecuzione;

Chiaramente, l'utilizzo di una risorsa in modo dinamico è più oneroso per il sistema, poiché questo deve caricarla ogni volta che è richiesta.

Una risorsa è anche accessibile via code behind, mediante l'invocazione del metodo `FindResource(nomeRisorsa)`, definito alla base della gerarchia WPF e disponibile quindi per tutti gli oggetti WPF.

Stili

Ogni elemento WPF può essere personalizzato nell'aspetto, mediante alcune proprietà che ne definiscono lo stile: sfondo, colore, margini eccetera. Per semplificare la scrittura e il riutilizzo di codice, è possibile memorizzare queste informazioni come risorse dell'applicazione e utilizzarle in vari punti del programma: questa pratica è chiamata "definizione di stili".

Ovviamente, gli stili possono essere definiti sia via XAML (utilizzo di tag `<Style>` all'interno di una proprietà `Resources`), sia via code behind (utilizzo della classe `Style`).

L’assegnazione di uno stile può essere automatica oppure manuale:

- L’assegnazione automatica consiste nel definire, insieme allo stile, a quali elementi dovrà essere applicato;

```
<Style TargetType="TextBlock">
  <Setter Property="Background" Value="Black"></Setter>
</Style>
```

- L’assegnazione manuale consiste nell’assegnazione di un nome identificativo allo stile e richiamarlo come risorsa statica o dinamica all’interno degli oggetti che intendono utilizzarlo.

```
<Style x:Key="headerText">
  <Setter Property="Background" Value="Black"></Setter>
</Style>
<TextBlock Style="{StaticResource headerText}" />
```

Trigger

All’interno della definizione di uno stile, è possibile definire delle modifiche all’aspetto dei controlli in base al verificarsi di eventi.

La classe `Style` ha una proprietà `Triggers`, dove è possibile definire uno o più elementi `Trigger`. La definizione di un trigger è composta da:

- Proprietà da osservare (proprietà `Property` del trigger);
- Valore per cui il trigger scatta (proprietà `Value` del trigger);
- Modifiche apportate allo stile (sequenza di elementi `Setter`).

Template

Ogni oggetto WPF ha un aspetto di default, ma questo può essere modificato a proprio piacimento mediante l’utilizzo dei template.

A seconda degli oggetti, esistono differenti tipi di template, tra cui:

- `ControlTemplate` (tag XAML `<ControlTemplate>`): ridefiniscono l’aspetto di un controllo;

- `DataTemplate` (tag XAML `<DataTemplate>`): ridefiniscono l’aspetto di oggetti di qualsiasi tipo (anche non WPF).

Come per gli stili, la definizione di un template via XAML è inserita nelle risorse di un oggetto e prevede l’associazione automatica o manuale.

Proprietà dipendenti

Lo scopo delle proprietà dipendenti consiste nel fornire un modo per calcolare il valore di una proprietà in base al valore di altri input, ad esempio proprietà del sistema, quali temi e preferenze dell'utente. Inoltre, una proprietà dipendente può essere implementata per fornire una convalida autonoma, valori predefiniti e callback per il monitoraggio delle modifiche di altre proprietà.

Le proprietà dipendenti e il sistema di proprietà WPF estendono le funzionalità della proprietà fornendo un tipo che supporta una proprietà: questo tipo è denominato `DependencyProperty`. L'altro tipo importante che definisce il sistema di proprietà WPF è `DependencyObject`, che definisce la classe base che può essere registrata ed essere proprietario di una proprietà di dipendenza.

Le proprietà dipendenti fanno capo a 3 elementi:

- Proprietà dipendente: una proprietà supportata da un oggetto `DependencyProperty`.
- Identificatore della proprietà dipendente: un'istanza dell'oggetto `DependencyProperty` ottenuta come valore restituito quando si registra una proprietà dipendente e successivamente memorizzata come membro di una classe.
- Wrapper CLR: le effettive implementazioni ottenute e impostate della proprietà. Queste implementazioni incorporano l'identificatore della proprietà di dipendenza utilizzandolo nelle chiamate ai metodi `GetValue()` e `SetValue()`.

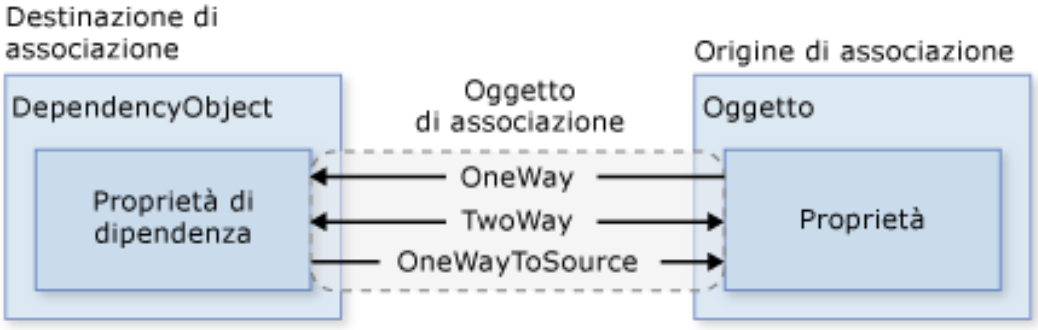
Quasi tutte le proprietà degli oggetti WPF sono dipendenti.

Data Binding

L'associazione dati (data binding) è il processo mediante il quale viene stabilita una connessione tra l'interfaccia utente dell'applicazione e il modello sottostante. Se le impostazioni dell'associazione sono corrette e i dati forniscono le notifiche appropriate, quando il valore dei dati viene modificato, gli elementi associati ai dati riflettono automaticamente le modifiche apportate. Associazione dati significa anche che, se una rappresentazione esterna dei dati in un elemento viene modificata, i dati sottostanti possono essere automaticamente aggiornati per riflettere la modifica. Ad esempio, se l'utente modifica il valore in un elemento `TextBox`, il valore dei dati sottostanti viene automaticamente aggiornato per riflettere tale modifica.

Ogni associazione ai dati segue sempre lo stesso modello: il destinatario può essere una qualsiasi proprietà dipendente di un oggetto WPF, mentre la sorgente può essere una qualsiasi proprietà di un oggetto. L'oggetto che crea l'associazione è una istanza della classe `Binding`.

Il collegamento può essere di diverso tipo, a seconda di quale tipo di interazione è desiderato tra interfaccia utente e modello sottostante.



L'associazione "One Time" prevede una relazione unidirezionale dalla sorgente alla destinazione (quindi eventuali modifiche apportate all'interfaccia utente non

vengono riflesse sul modello) e che avviene in fase di caricamento della sorgente dati: quindi, se i dati cambiano durante l'esecuzione, non si avrà alcuna modifica sull'interfaccia grafica. Questa modalità di associazione è tipica dei dati completamente statici.

L'associazione "One Way" prevede una relazione unidirezionale dalla sorgente alla destinazione, ma monitorata costantemente (una modifica al modello è riflessa sull'interfaccia utente, ma non viceversa).

L'associazione "Two Way" prevede una relazione bidirezionale tra sorgente e destinazione: la modifica di una delle due proprietà viene automaticamente riflessa sull'altra.

L'associazione "One Way to Source" prevede una relazione unidirezionale dalla destinazione alla sorgente.

Per rilevare le modifiche apportate all'origine, nel caso delle associazioni One Way e Two Way, questa deve implementare un meccanismo appropriato di notifica delle modifiche alle proprietà, quale ad esempio l'interfaccia `INotifyPropertyChanged`.

Validazione di input

Una proprietà associata ad un controllo in modalità Two Way o One Way To Source può subire modifiche in base ad azioni svolte dall'utente sull'interfaccia grafica. Per questo motivo è importante prevedere errori nell'input immesso dall'utente: questi possono essere sia di tipo sintattico che, eventualmente, di tipo semantico.

Nel caso in cui il settaggio di una proprietà collegata all'interfaccia utente generi una eccezione, questa può essere intercettata e mostrata impostando, in fase di binding, la proprietà di markup `ValidatesOnExceptions` al valore `true`. E' disponibile un template di default, modificabile specificando il nome di un template nell'attributo `Validation.ErrorTemplate` del controllo. In alternativa, si può agire sullo stile del controllo, attivando un trigger sull'evento `Validation.HasErrors`.

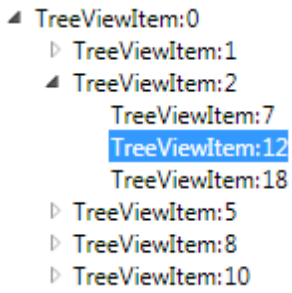
Ad esempio, per validare una semplice casella di testo e fare in modo che il suo sfondo diventi rosso in caso di errori, si può utilizzare il seguente codice:

```
<TextBox Text="{Binding Path=Value, Mode=TwoWay,
 ValidatesOnExceptions=True }">
  <TextBox.Style>
    <Style TargetType="{x:Type TextBox}">
      <Style.Triggers>
        <Trigger
         Property="Validation.HasError" Value="True">
          <Setter Property="ToolTip"
           Value="{Binding RelativeSource =
                  {x:Static RelativeSource.Self},
           Path=(Validation.Errors)[0].ErrorContent}" />
          <Setter Property="Background"
           Value="LightPink" />
        </Trigger>
      </Style.Triggers>
    </Style>
  </TextBox.Style>
</TextBox>
```

Per i controlli semantici, invece, la classe sorgente deve implementare l'interfaccia `IDataErrorInfo` e in XAML è necessario attivare, via estensione di markup, la proprietà `ValidatesOnDataErrors`.

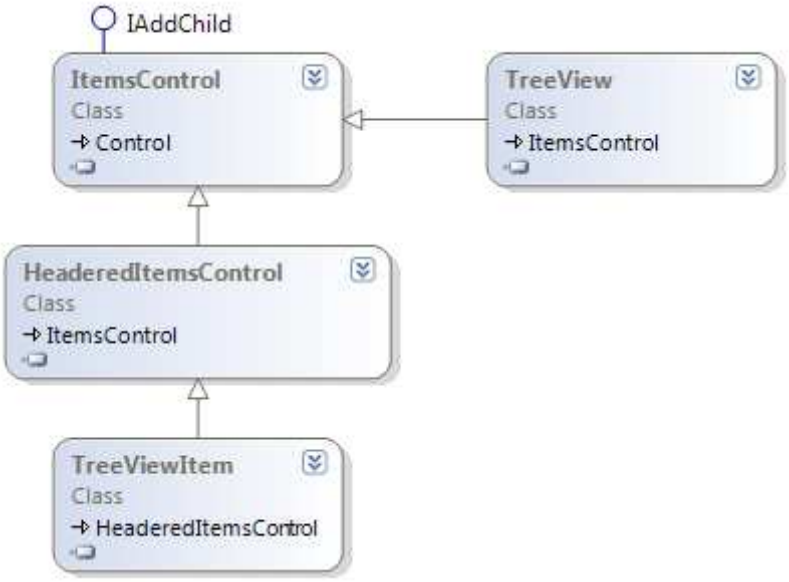
I controlli TreeView e TreeViewItem

La classe `TreeView` di WPF modella e rappresenta graficamente una struttura dati ad albero, in cui ogni nodo è espandibile separatamente.



Questa classe è derivata da `ItemsControl`, che modella un generico contenitore di elementi. Nella versione più semplice, all'interno del tag XAML `TreeView` sono inseriti una serie di elementi `TreeViewItem`. Quest'ultima classe modella un generico elemento dell'albero, capace a sua volta di contenere altri oggetti. In questo modo è possibile creare la struttura gerarchica.

La classe `TreeViewItem` fa parte di una serie di controlli chiamati controlli con testata (derivati da `HeaderedItemsControl`): questi controlli sono visualizzati a schermo proprio grazie ad una loro particolare proprietà (di tipo `object` e quindi capace di ospitare istanze di qualsiasi classe), `Header`.



Mediante la proprietà `ItemsSource` del `TreeView`, è possibile effettuare una associazione dati dell'albero ad una qualsiasi sorgente dati enumerabile (che implementi cioè l'interfaccia `IEnumerable`): in questo modo è possibile visualizzare, in automatico, una lista di qualsiasi tipologia di oggetto.

Chiaramente, per gli oggetti non nativi WPF, la visualizzazione a schermo sarà una semplice stampa della versione a stringa dell'istanza (chiamata a `ToString()`): tuttavia, è possibile definire dei `DataTemplate` specifici e ottenere la visualizzazione desiderata.

Creazione di UserControl

Un controllo utente è una serie di componenti grafici e codice per la logica sottostante, racchiusi in un pacchetto (lo UserControl, per l'appunto) utilizzabile in qualsiasi altra applicazione o modulo grafico. Per definire un controllo utente, è sufficiente creare una classe che estenda `UserControl`.

In WPF, non è possibile creare uno UserControl che ne estenda un altro, poiché è necessario che in tutto il percorso dalla classe base alla foglia sia presente un solo file XAML associato. Per emulare il comportamento, è sufficiente creare una classe derivata da `UserControl`, senza XAML associato e definire in seguito una sottoclasse di quest'ultima, con XAML associato.

CAPITOLO 3: Fase di analisi

Descrizione generale del progetto

Il software da realizzare è un componente C# .NET 4 in grado di visualizzare tutti gli ET tramite l'utilizzo della libreria WPF e di eseguirne alcune tipologie.

Le espressioni che si intende rendere eseguibili sono quelle, anche interne all'ET, tali per cui tutti i parametri di input siano di tipo "semplice", ossia tali per cui l'input sia raccoglibile mediante semplice casella di testo: fanno parte di questa categoria i tipi numerici, booleani, le stringhe e le date.

E' inoltre necessaria un'analisi sulla struttura delle espressioni, per evitare l'esecuzione di sotto – espressioni non corrette.

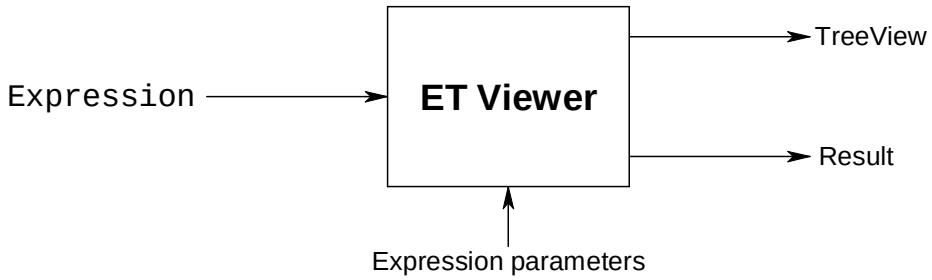
Infine, è bene valutare per quali tipi di espressione abbia effettivamente senso richiedere l'esecuzione: ad esempio, richiedere l'esecuzione di una espressione di tipo `ConstantExpression` non ha logica, poiché di fatto l'esecuzione di una costante darebbe come risultato la costante stessa (ben visibile tra le proprietà dell'espressione stessa, senza passare dalle fasi di compilazione ed esecuzione).

Analisi e specifica dei requisiti

Il progetto consiste nella realizzazione di un visualizzatore ed esecutore di Expression Tree.

L'Expression su cui lavorare:

- Sarà una qualsiasi istanza derivata dalla classe `Expression` del namespace `System.Linq.Expressions`;
- Sarà un parametro di input del sistema;
- Si suppone creata in ogni sua parte e non modificabile in alcun modo da parte dell'applicativo;
- Si suppone conforme alle specifiche della versione 2 degli ET (introdotta nel framework .NET 4.0 e compatibile con la versione precedente).



Il sistema deve essere inoltre in grado di:

- Determinare il numero e tipo di parametri richiesti dall'istanza di Expression attuale;
- Rendere possibile la raccolta di alcune tipologie di input (definite in seguito);
- Determinare il tipo di risultato della Expression.

Per quanto riguarda l'output del sistema, questo deve:

- Essere in grado di visualizzare le componenti di ogni Expression;
- Essere in grado di eseguire l'Expression radice e tutte le sue sotto – espressioni, se possibile (analisi in seguito);
- Mostrare il risultato dell'esecuzione e gli eventuali dettagli del risultato.

L'aspetto cruciale è che non solo l'espressione nella sua interezza deve essere eseguibile, ma anche una qualsiasi porzione del suo albero (sotto – espressione); chiaramente, sebbene si possa affermare che l'espressione nella sua interessa sia compilabile ed eseguibile, non può dirsi altrettanto in merito ai singoli sotto – alberi che la compongono.

Infine, il software deve essere realizzato come libreria di classi DLL, in modo da poter essere facilmente inserito in qualsiasi altra applicazione.

Analisi dei tipi di input

Ogni istanza di `LambdaExpression` dispone di una serie di parametri di input, memorizzati nel parametro dell'istanza `Parameter`. Affinché l'espressione, una

volta compilata, possa essere eseguita, è naturalmente necessario assegnare dei valori immediati ai parametri di ingresso dell'espressione.

Ai fini dello sviluppo del progetto, si è stabilito di rendere disponibile l'inserimento solo di alcune tipologie di parametro.

Il principio che si è seguito è quello di rendere disponibile solamente l'inserimento di "dati semplici", ossia il cui input sia raccoglibile mediante semplici caselle di testo. I dati scelti per l'applicazione sono qui i seguenti:

- Tutti i tipi numerici
- I tipi carattere e stringa
- I tipi booleani
- I tipi data e ora

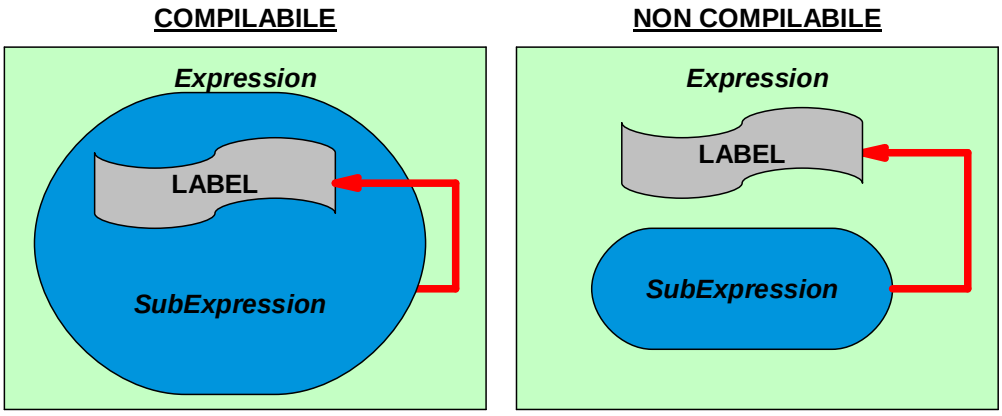
Le espressioni contenenti anche solo un parametro di altra tipologia, non devono essere compilabili ed eseguibili. Sarà ovviamente necessario garantire, in questi casi, che non sia possibile raccogliere l'input di nessun parametro dell'espressione.

Compilabilità ed eseguibilità di una Expression

Oltre al controllo del tipo dei parametri, che è una scelta di progetto, esistono altri controlli obbligatori per rendere disponibile la funzionalità di eseguibilità di una sottoespressione del nodo radice.

Il controllo più importante riguarda le espressioni che modellano salti nel flusso esecutivo del programma. Se si tenta di compilare una espressione che contiene un salto verso un punto esterno all'espressione stessa, è chiaro che questa operazione dovrebbe essere bloccata, poiché di fatto insensata e non eseguibile.

Quindi, se una espressione contiene un salto, è necessario che questo sia verso etichette altresì definite all'interno dell'espressione stessa.



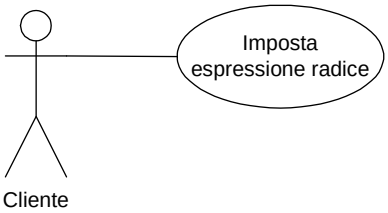
Tipi di nodo eseguibili

E' richiesto, oltre ai vari controlli già descritti, che non tutti i nodi siano resi eseguibili, all'interno del sottoalbero radice.

In particolare, attualmente, è richiesto che i seguenti tipi di nodo non siano eseguibili, poiché si è valutato che non avrebbe senso rendere disponibile questa funzionalità:

- Nodi che modellano un parametro o una variabile (istanze di `ParameterExpression`): eseguire questi nodi porterebbe al semplice risultato del valore immesso come parametro di ingresso;
- Nodi che modellano costanti (istanze di `ConstantExpression`): la loro esecuzione non farebbe altro che restituire il valore della costante, già presente, prima della compilazione, tra le proprietà del nodo stesso.

Casi d'uso "Cliente esterno"



Siccome il software sarà rilasciato come componente esterno (libreria di classi in formato DLL), l'attore principale del sistema sarà un generico cliente esterno, che dovrà inizializzare il controllo utente con una qualsiasi espressione (istanza di una sottoclasse di `Expression`).

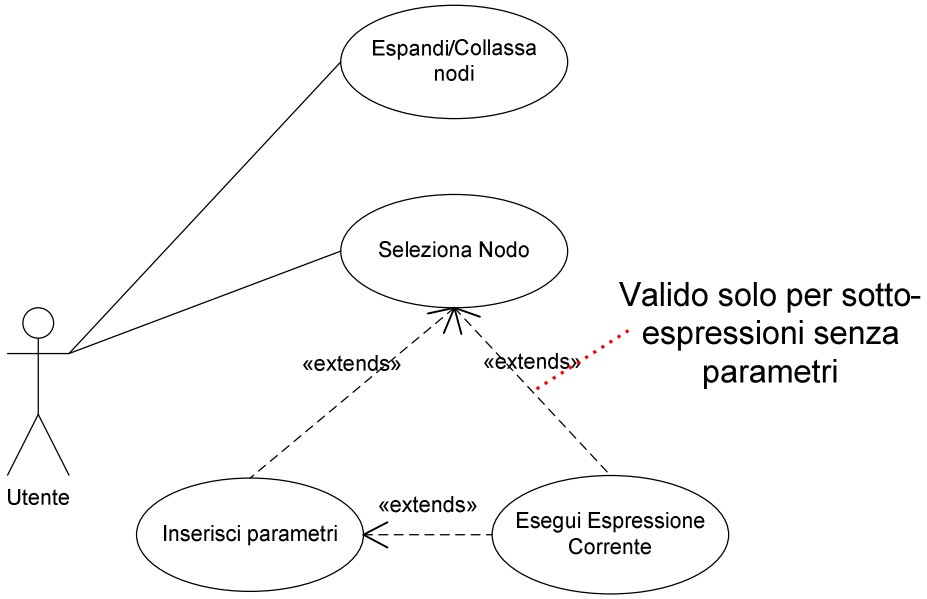
Casi d'uso "Interazione utente"

Una volta inizializzato il sistema, viene presentata all'utente l'interfaccia grafica, che permette di visualizzare tutto l'albero.

Il programma che gestisce il modulo dovrà anche essere in grado di consentire all'utente di espandere e collassare i nodi dell'albero.

Selezionato un nodo, deve essere possibile inserire i parametri e, una volta esauriti i parametri da inserire, deve essere possibile eseguire l'espressione (e quindi visualizzare il risultato), a patto che questo sia consentito dalle politiche di eseguibilità scelte.

Logicamente, per i nodi che rappresentano sotto – espressioni eseguibili senza parametri, la possibilità di richiedere l'esecuzione deve essere garantita all'atto di selezione del nodo.



Scenario "Esegui Espressione Corrente"

Precondizioni

- Aver impostato l'espressione radice al controllo utente;
- Aver selezionato un nodo;

- Aver inserito in maniera corretta tutti i parametri della sotto – espressione avente come radice il nodo selezionato.

Flusso principale

1. Compila ed esegui l'espressione corrente;
    a. **Se** l'espressione ha risultato, inseriscilo nel blocco del risultato;
    b. **Altrimenti**, inserisci nel blocco del risultato la stringa "*Void method*";
2. Rendi visibile il blocco del risultato.

Flusso alternativo

1b. **In caso di errore di compilazione o esecuzione**, inserisci nel blocco del risultato un messaggio che descriva l'errore;

1c. Rendi visibile il blocco del risultato.

Capitolo 4: Modello dei dati

Introduzione al modello dei dati

Al fine di separare la logica dell'applicazione da quella per la veste grafica, si è scelto di applicare il pattern Document - View.

Il modello deve contenere:

- L'espressione radice;
- La lista dei parametri dell'espressione;
- Un Visitor dell'espressione per costruire l'albero;
- Una lista di oggetti atti a verificare la compilabilità dell'espressione e di tutte le sue sotto – espressioni.

Visitor dell'espressione

Il Visitor dell'espressione deve costruire dinamicamente l'albero da visualizzare, restituendo quindi una istanza di `TreeViewItem`. Ogni Visitor di espressione deve discendere dalla classe `ExpressionVisitor` e ridefinire i metodi di visita specifici per i nodi di interesse.

La visita dell'albero è innescata alla richiesta del componente grafico.

```
private readonly Expression _expression;
private TreeViewItem _treeViewItem = null;

public ETTreeViewItemPrinter(Expression expression)
{
    if (expression == null)
        throw new ArgumentNullException("expression");
    _expression = expression;
}

public Expression Expression
{
```

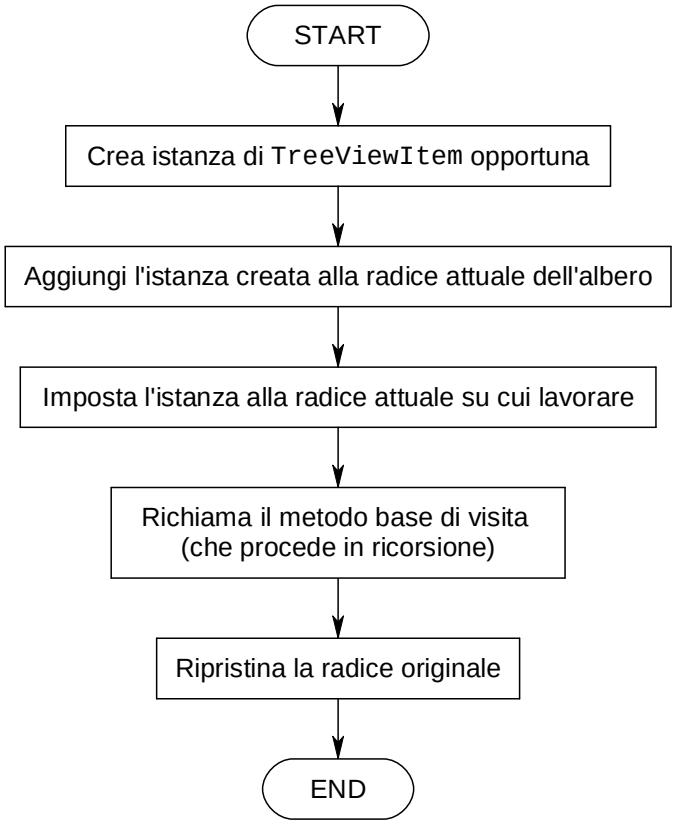
```
    get { return _expression; }
}

public TreeViewItem TreeViewItem
{
    get
    {
        if (_treeViewItem == null)
        {
            Visit(Expression);
        }
        return _treeViewItem;
    }
}
```

Siccome tutti i nodi devono essere analizzati e convertiti in nodo di albero, tutti i metodi di visita specifici devono essere ridefiniti. La presenza di nodi “speciali”, non discendenti da `Expression` ma direttamente da `object`, rende impossibile la semplice ridefinizione del metodo base `Visit()`, poiché in questo caso questi nodi particolari non verrebbero visitati e quindi rappresentati nell’albero grafico.

Si è cercato comunque di mantenere minima la duplicazione del codice, realizzando un metodo quanto più possibile generico e richiamandolo dentro ad ogni ridefinizione di metodo di visita.

Sostanzialmente, per ogni nodo dell’espressione, è necessario creare una istanza di `TreeViewItem`, inizializzando le sue varie proprietà. A questo punto si può procedere in ricorsione nella creazione dell’albero: per farlo, è necessario che i nuovi nodi siano aggiunti come figli dell’attuale nodo, non della vera radice dell’albero (in questo modo si avrebbe un appiattimento della gerarchia). Per ottenere questo effetto e non potendo modificare l’intestazione del metodo di visita (poiché sovrascritto dalla versione base), si deve utilizzare una struttura a stack, dove memorizzare le radici precedenti. Prima di procedere in ricorsione, si salva l’attuale radice nello stack e si imposta il nodo appena creato come radice,

ripristinando lo stato precedente al termine della procedura ricorsiva. In generale, quindi, l'algoritmo da seguire è il seguente:



Essendo l'algoritmo da seguire uguale per tutti i tipi di nodo visitati, è possibile creare un unico metodo, che richiede ovviamente come parametri di input il nodo attuale e il metodo di ricorsione (la versione base del metodo di visita).

Questo metodo, chiaramente, si servirà di altri servizi specifici per svolgere i vari passi dell'algoritmo.

```
private TResult RecursiveCall<T, TResult> (
  T node,
  Func<T, TResult> recursiveFunction
) where T : TResult
```

Il metodo è definite come metodo generico, avente due tipi generici in gerarchia. Il metodo di visita per un nodo di tipo T ha infatti la forma:

```
protected override Expression VisitT(T node);
```

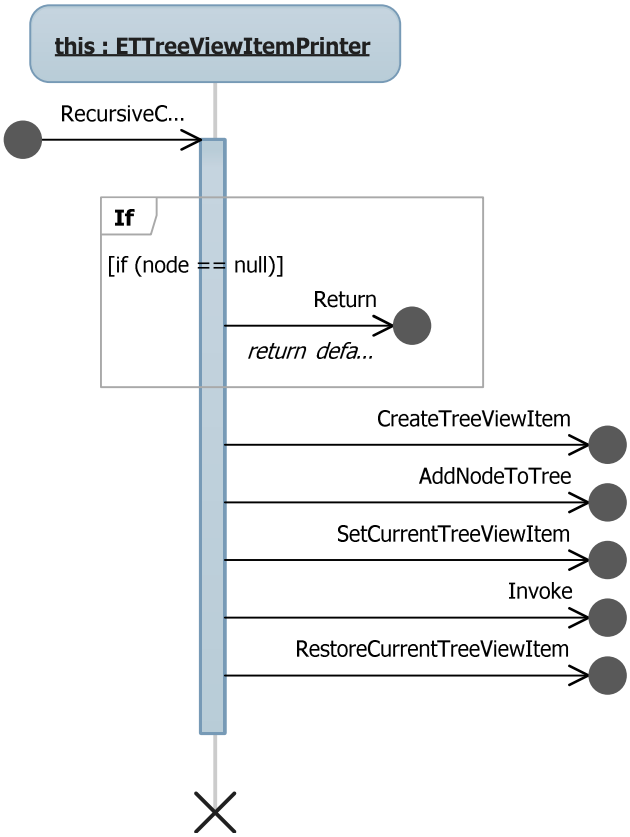
In altre parole, il metodo accetta un nodo di tipo `T` e lo visita, restituendo una istanza derivata da `Expression`, che, come detto, è radice comune della gerarchia delle espressioni.

La validità di questo metodo si estende anche ai nodi particolari, poiché in questo caso il metodo di visita (dato `T` come tipo di nodo) assume la forma:

```
protected override T VisitT(T node);
```

Il metodo riceve in input il nodo specifico, lo visita e restituisce una istanza dello stesso tipo di nodo (per default, il nodo stesso).

La definizione del metodo generico, quindi, permette l'invocazione da parte di ogni tipologia di nodo; il secondo parametro, infatti, è un delegato a cui andrà passato il metodo di visita specifico del nodo.



Una volta definito il metodo in ogni sua parte, è immediato notare che la ridefinizione di ogni metodo di vista avrà sempre lo stesso corpo, ossia la chiamata a questo metodo generico e la restituzione del suo risultato.

Per i nodi discendenti da `Expression` si avrà quindi:

```
protected override Expression VisitT(T node)
{
  return RecursiveCall(node, base.VisitT);
}
```

Per i nodi speciali si avrà invece:

```
protected override T VisitT(T node)
{
  return RecursiveCall(node, base.VisitT);
}
```

La creazione del `TreeViewItem` consiste nell'instanziare un nuovo oggetto e assegnare alla proprietà `Header` dello stesso l'espressione corrente.

```
private TreeViewItem CreateTreeViewItem(object node)
{
    TreeViewItem item = new TreeViewItem();
    item.Header = node;
    return item;
}
```

Chiaramente, il parametro di input deve essere di tipo `object`, poiché, come detto, l'albero mostrerà anche i nodi non discendenti direttamente da `Expression`.

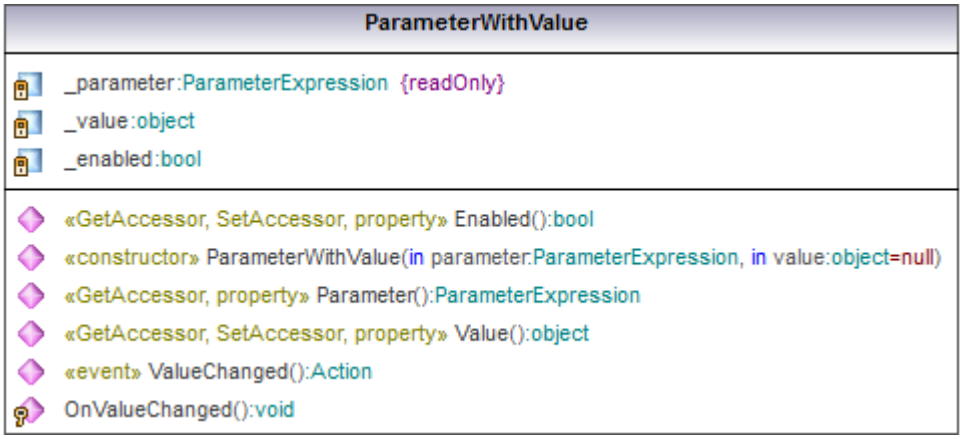
Modello dei parametri

Al fine di rendere disponibile l'esecuzione di una espressione, è necessario poter raccogliere i parametri e i loro valori; inoltre, poiché si vuole che anche singole porzioni di espressione siano eseguibili, bisogna implementare un meccanismo per l'attivazione e la disattivazione di alcuni parametri in determinati momenti dell'esecuzione del programma (se si sta lavorando su un nodo che non prevede un certo parametro, questo non deve poter essere inserito dall'utente).

Per modellare il tutto è stata creata una classe `ParameterWithValue`, che contiene una istanza di `ParameterExpression`, un eventuale valore e un flag di attivazione. Poiché un `ParameterExpression` può essere di qualsiasi tipo,

il valore sarà rappresentato come campo di tipo `object`, provvedendo poi nel metodo accessorio di `set` alla conversione di valore (e quindi all'eventuale lancio di eccezioni).

La classe espone inoltre un evento, `PropertyChanged`, che permette ai clienti esterni di reagire al cambiamento di valore del parametro.



L'aggiornamento del valore della proprietà prevede una conversione al tipo della stessa (con eventuale lancio di eccezioni) e l'invocazione del metodo `OnValueChanged()`, che scatena in sequenza tutti gli ascoltatori dell'evento `ValueChanged`.

```
set
{
  try
  {
    if (value != null)
    {
      try
      {
        value =
          Convert.ChangeType(value, _parameter.Type);
      }
      catch
      {
        value = null;
        throw;
      }
```

```
    }
  }
  finally
  {
    if (value != _value)
    {
      _value = value;
      OnValueChanged();
    }
  }
}
```

Distinzione tra parametro e variabile

Sebbene siano sempre modellati come nodi di tipo `ParameterExpression`, è importante distinguere variabili e parametri veri e propri, soprattutto in un'ottica orientata all'esecuzione di una espressione: infatti non ha senso chiedere all'utente di assegnare un valore ad una variabile, poiché deve essere il codice (e quindi l'espressione) a provvedere a ciò.

La classe `LambdaExpression` contiene una proprietà, `Parameters`, che contiene solo i parametri dell'espressione: tuttavia, siccome si vuole rendere eseguibile un qualsiasi nodo discendente da `Expression`, è necessario un meccanismo per determinare la lista dei parametri di input.

Esistono tre tipologie di nodi, descritti nelle classi `BlockExpression`, `RuntimeVariablesExpression` e `CatchBlock`, che possono contenere variabili (dentro una proprietà `Variables`).

Pertanto, visitando un albero dell'espressione, quando viene trovato un nodo `ParameterExpression`, ci sono quattro possibilità:

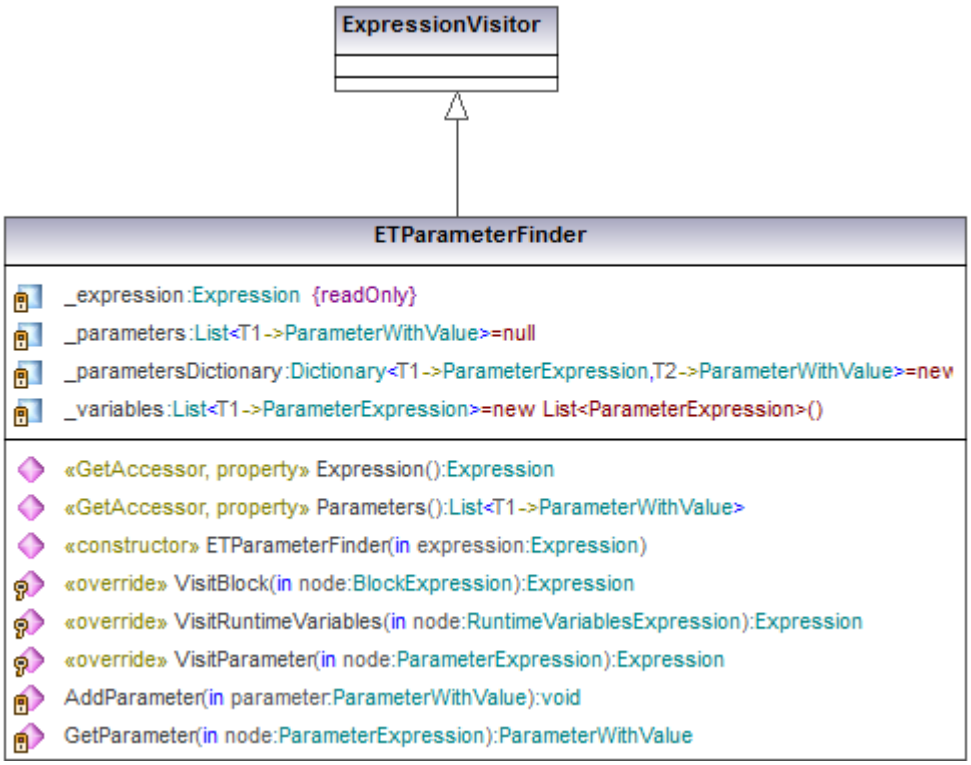
1. Si tratta di un vero e proprio parametro;
2. Si tratta di una variabile dichiarata in una `BlockExpression`;
3. Si tratta di una variabile dichiarata in una `RuntimeVariablesExpression`;

4. Si tratta di una variabile dichiarata in una istanza di `CatchBlock`.

La classe `ETParameterFinder` ha il compito, data una qualsiasi istanza di `Expression`, di ricavare la lista dei parametri di input della stessa, escludendo quindi le variabili: per farlo, implementa un meccanismo di visita (estendendo `ExpressionVisitor`) e ridefinendo i metodi di visita dei quattro nodi speciali: `VisitBlock()`, `VisitRuntimeVariables()`, `VisitCatchBlock()` e `VisitParameter()`.

Mantiene inoltre un dizionario associativo tra un nodo parametro e la rispettiva istanza di `ParameterWithValue`, nonché due liste, costruite dinamicamente durante la visita, contenenti variabili e parametri.

La visita dei blocchi, delle variabili runtime e dei blocchi `catch` prevede l'esclusione delle variabili dalla lista, mentre la visita dei parametri prevede l'aggiunta di quelli validi alla lista da restituire.

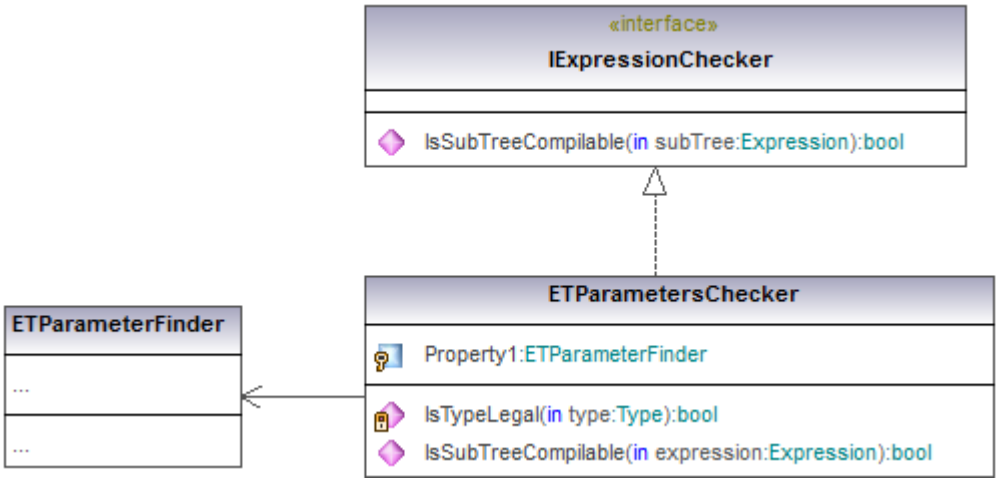


Modello per la verifica della compilabilità

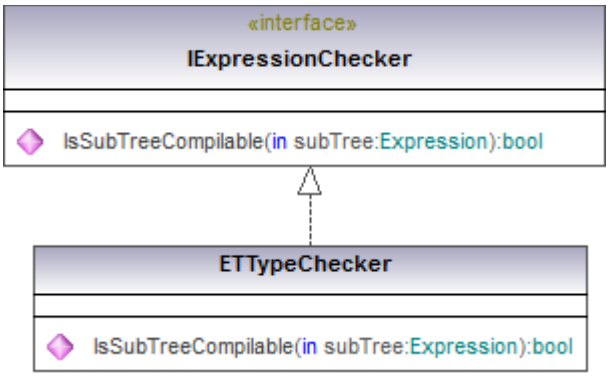
Come già spiegato in fase di analisi, è necessario provvedere ad alcuni meccanismi (intrinseci alla natura dell'espressione o definiti dall'analista) per stabilire se una particolare espressione sia o meno compilabile.

Chiaramente, il sistema è sviluppato seguendo il principio dell'Open Close: deve essere facile, in futuro, aggiungere criteri di compilabilità, dando per scontato che quelli attuali non subiranno modifiche. Per questo motivo, ci si affida ad un contratto sotto forma di interfaccia chiamata `IExpressionChecker`: questa dichiara un metodo booleano `IsSubTreeCompilable()` per la verifica della compilabilità di una espressione secondo un qualsiasi criterio.

La classe `ETParametersChecker` effettua il controllo sul tipo dei parametri, tollerando solamente quelli già discussi in precedenza: per fare questo, naturalmente, deve prima di tutto trovare i parametri associati alla particolare sotto – espressione in esame, dopodiché procederà all'esame di validità.



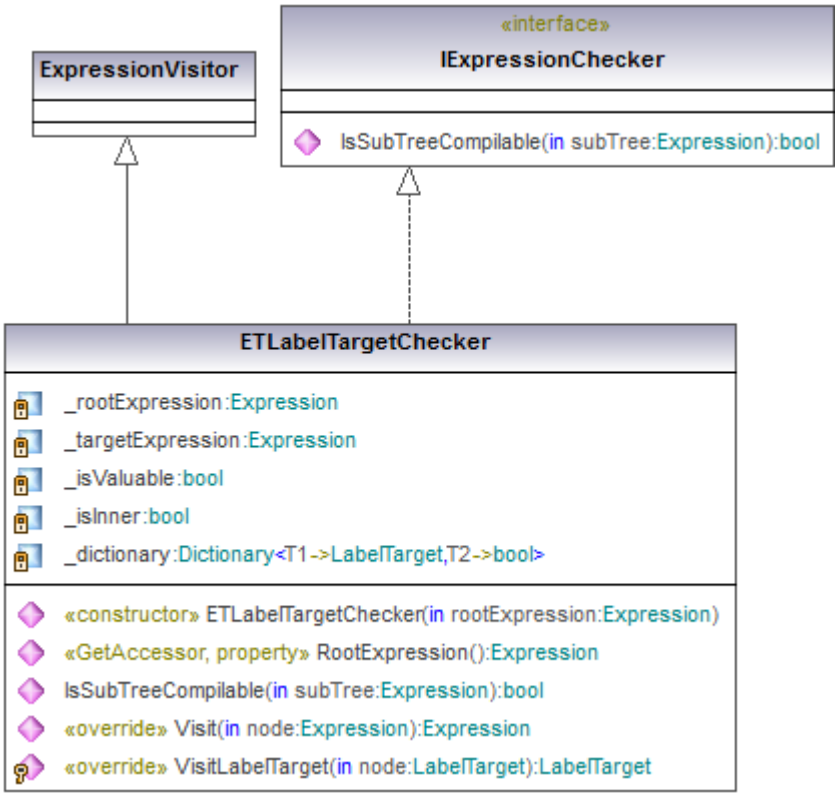
La classe `ETTypeChecker` analizza invece il tipo di nodo radice della sotto – espressione che si vuole analizzare, rendendo non compilabili i nodi di tipo `ParameterExpression` e `ConstantExpression`.



La classe `ETLabelTargetChecker` analizza i nodi di tipo `LabelTarget`, che modellano una etichetta di codice e sono inseriti in vari tipi di nodo espressione.

L'analisi compiuta in fase di visita è quella sui riferimenti alle etichette, come già spiegato in precedenza: una sotto – espressione è compilabile se tutti i riferimenti a etichetta puntano ad una istruzione interna alla sotto – espressione stessa.

Dovendo visitare tutta l'espressione, questa classe deve contenere un riferimento all'espressione radice; inoltre, deve estendere la classe `ExpressionVisitor` e ridefinire il metodo `VisitLabelTarget()`.



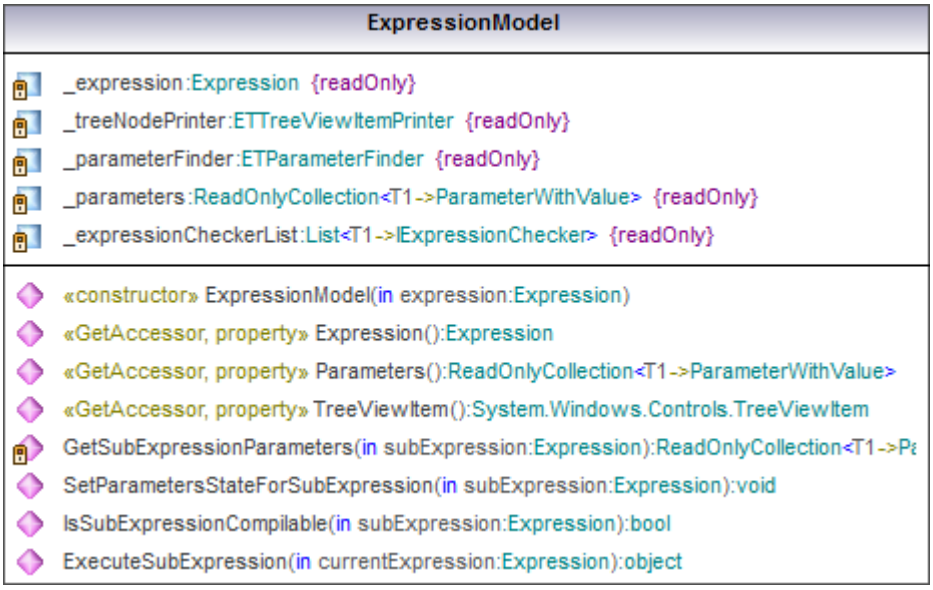
Modello completo

Il modello completo è costruito attorno all'espressione radice del controllo utente, memorizzata nel campo `_expression`.

Il modello contiene inoltre un printer (per ottenere il `TreeViewItem` radice), la lista parametri (e il rispettivo finder), nonché una lista di validatori di compilabilità (`_expressionCheckersList`).

Esso offre inoltre 3 servizi pubblici:

- Metodo `SetParametersStateForSubExpression()`: questo metodo attiva o disattiva il flag di `Enabled` dei parametri in base alla loro appartenenza alla sotto – espressione passata al metodo;
- Metodo `IsSubExpressionCompilable()`: verifica la compilabilità di una sotto – espressione, valutando in cascata la lista di validatori;
- Metodo `ExecuteSubExpression()`: esegue la sotto – espressione (eventualmente coincidente con l'espressione radice) passata come parametro.



Il metodo `SetParametersStateForSubExpression()` si occupa di scorrere la lista dei parametri dell'espressione radice, disattivando i parametri che non fanno parte anche della collezione dei metodi della sotto – espressione

corrente (ottenuti mediante chiamata al metodo protetto `GetSubExpressionParameters()`, che si serve di un `ETParameterFinder` per svolgere il suo compito).

```
public void SetParametersStateForSubExpression
  (Expression subExpression)
{
    if (subExpression == null)
        throw new ArgumentNullException("subExpression");
    ReadOnlyCollection<ParameterWithValue>
      subExpressionParameters =
        GetSubExpressionParameters(subExpression);
    foreach (
      ParameterWithValue originalParameter in Parameters
    )
    {
        originalParameter.Enabled =
          subExpressionParameters.Any(
            parameter => parameter.Parameter ==
              originalParameter.Parameter
          );
    }
}
```

Il metodo di valutazione di compilabilità scorre in sequenza la lista di `IExpressionChecker`, restituendo valore `false` al primo fallimento. In altre parole, i validatori sono, come è ovvio, valutati in AND.

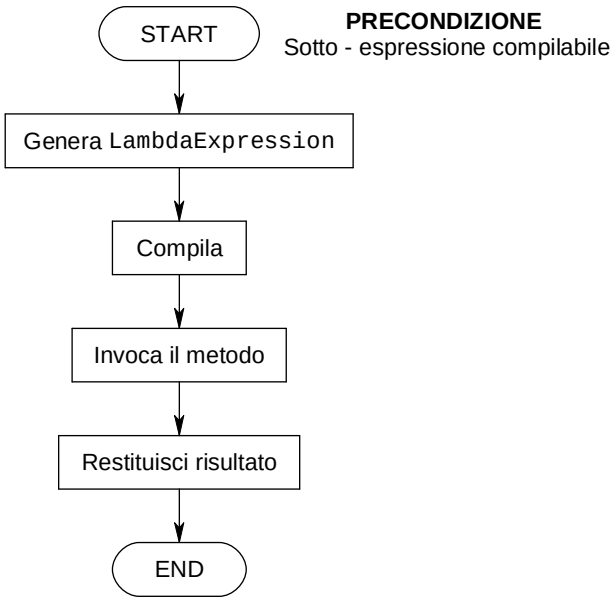
```
public bool IsSubExpressionCompilable
  (Expression subExpression)
{
    if (subExpression == null)
        throw new ArgumentNullException("subExpression");
    foreach (IExpressionChecker expressionChecker in
                                _expressionCheckerList)
    {
        if
          (!expressionChecker.IsSubTreeCompilable(
            subExpression))
              return false;
    }
    return true;
```

```
}
```

La lista di validatori è cablata all'interno del costruttore del modello:

```
_expressionCheckerList = new List<IExpressionChecker>();
_expressionCheckerList.Add(
  new ETLabelTargetChecker(expression)
);
_expressionCheckerList.Add(
  new ETTypeChecker()
);
_expressionCheckerList.Add(
  new ETParametersChecker()
);
```

La logica di esecuzione di una espressione segue questo algoritmo: per prima cosa si deve ottenere una istanza di `LambdaExpression` (unica classe ad avere un metodo di compilazione). La `LambdaExpression` deve essere creata manualmente (mediante costruttore) nel caso in cui sia richiesta l'esecuzione di un nodo che non sia una espressione lambda. La compilazione è così resa possibile mediante l'omonimo metodo della classe stessa. A questo punto si avrà a disposizione un delegato, eseguibile mediante il metodo `DynamikInvoke` della classe `Delegate`. Il risultato dell'invocazione sarà restituito dal metodo; nel caso in cui l'espressione non ritorni valore (cioè nel caso in cui rappresenti un metodo di tipo restituito `void`), sarà ottenuto un valore `null`, che sarà restituito comunque.

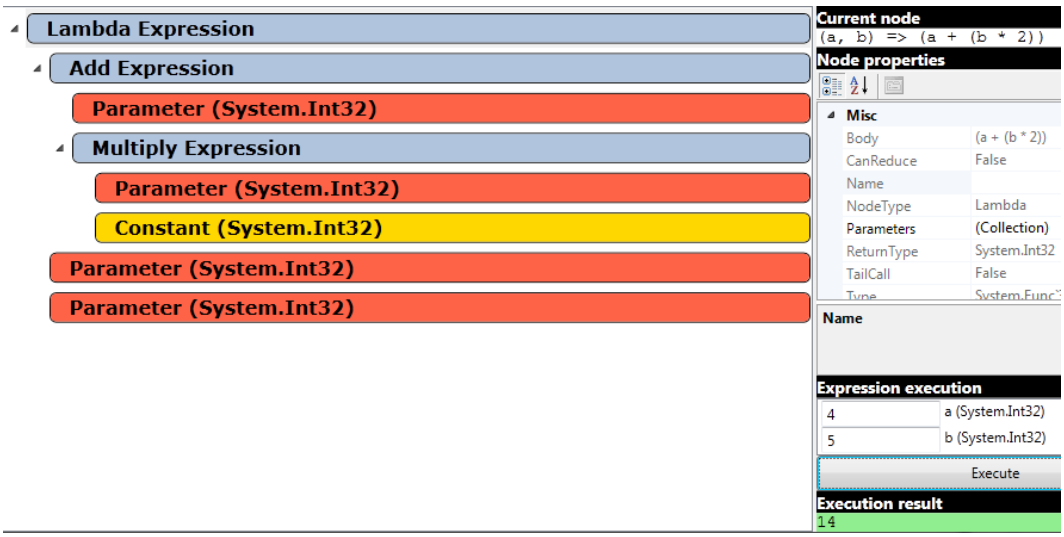


CAPITOLO 5: Progetto grafico

Definizione della veste grafica

Il controllo utente dovrà presentarsi con una struttura tabellare a due colonne:

- Nella colonna sinistra dovrà essere visibile l'albero dell'espressione, ossia un controllo `TreeView` i cui nodi sono ottenuti mediante il printer del modello;
- Nella colonna destra dovranno essere visualizzati tutti i dettagli del nodo attualmente selezionato nell'albero di sinistra; in particolare:
    - Una rappresentazione testuale del nodo corrente;
    - Una lista dei valori delle proprietà del nodo corrente;
- Per le espressioni compilabili, la colonna di destra dovrà mostrare anche:
    - La lista dei parametri di input, qualora ce ne siano, con la possibilità di assegnare loro un valore;
    - Un pulsante per richiedere l'esecuzione dell'espressione;
    - Una zona a comparsa per la visualizzazione del risultato e dei suoi dettagli (nel caso il risultato non sia un semplice valore scalare, ma una istanza di classe).



La vista base

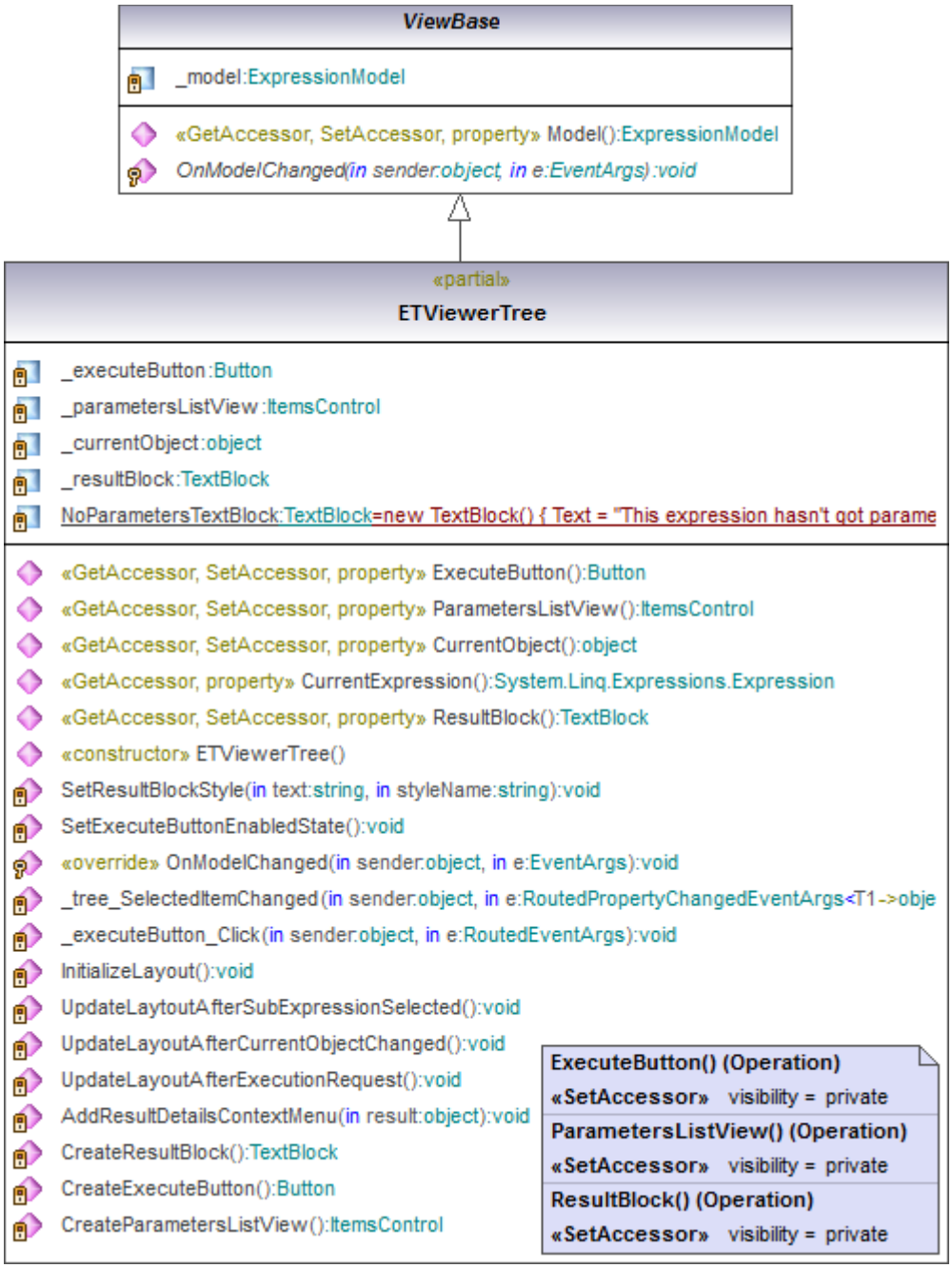
Il primo passo per l'utilizzo del pattern Document – View è la definizione della cosiddetta "Vista base", che implementa i meccanismi principali per l'interazione con il modello dei dati.

In particolare, la vista base (classe astratta `ViewBase` nel namespace `ETViewer.View`) definisce il riferimento al modello (istanza di `ExpressionModel` del namespace `ETViewer.Model`) e le azioni da scatenare al cambiamento dello stesso, utilizzando un metodo astratto `OnModelChanged()`, che sarà ridefinito dalle viste che estenderanno questo contratto.

```
public abstract class ViewBase :
   System.Windows.Controls.UserControl
{
    private ExpressionModel _model;
    public ExpressionModel Model
    {
        get { return _model; }
        set
        {
            if (value != _model)
            {
                _model = value;
                if (_model != null)
                {
                    OnModelChanged(_model,
                      EventArgs.Empty);
                }
            }
        }
    }
    protected abstract void OnModelChanged(
      object sender, EventArgs e
    );
}
```

Il controllo di visualizzazione ed esecuzione

La veste grafica di visualizzazione del modello è affidata alla classe `ETViewerTree`, che visualizza l'albero, permette la selezione ed esecuzione dei nodi e mostra gli eventuali risultati.



La vista è organizzata in due file che definiscono, come tutti i progetti WPF, la veste grafica:

- Un file XAML per la definizione della veste grafica, di stili e template;
- Un file `.cs` per la definizione dei dati associati alla classe e all'interazione con gli utenti.

La vista non espone alcun metodo all'esterno, ma permette l'interazione con l'utente mediante la definizione di due funzioni che saranno associate, via codice XAML, alla gestione di eventi:

- Il metodo `_tree_SelectedItemChanged()` viene invocato ogni volta che l'utente seleziona un nodo dall'albero dell'espressione: in questo modo sarà possibile visualizzare i dettagli del singolo nodo e procedere all'esecuzione della sotto – espressione che ha quel nodo come radice;
- Il metodo `_executeButton_Click()` viene invocato ogni volta che l'utente preme il pulsante di esecuzione della sotto – espressione corrente: la vista, in questo caso, richiederà l'esecuzione al modello sottostante, ottenendo da esso il risultato e procedendo alla visualizzazione dello stesso, fornendo inoltre la possibilità di visualizzare i dettagli in una finestra separata (che sarà definita in seguito).

Il file di code behind è organizzato in alcune sezioni:

- Campi: definiscono lo stato dell'interfaccia grafica, ossia il pulsante di esecuzione, la lista dei parametri, l'oggetto correntemente selezionato e il blocco per la visualizzazione dei risultati;
- Proprietà: mappano i campi verso l'esterno, definendo quali siano accessibili solo in lettura e quali anche in scrittura; alla modifica della selezione del nodo corrente, scatena una serie di chiamate a metodi per l'aggiornamento della veste grafica;

```
public object CurrentObject
{
    get { return _currentObject; }
    set
    {
        _currentObject = value;
```

```
        if (value != null)
        {
            UpdateLayoutAfterCurrentObjectChanged();
            if (CurrentExpression != null)
            {
               ExecuteButton.Tag = CurrentExpression;
               UpdateLaytoutAfterSubExpressionSelected();
            }
        }
    }
}
```

- Costruttore: inizializza i campi e l'interfaccia grafica (chiamata al metodo base `InitializeComponent()`);
- Metodi di modifica dello stile grafico: contiene il metodo `SetResultBlockStyle()`, per l'applicazione di uno stile (definito nel file XAML) e di un testo al blocco del risultato, e il metodo `SetExecuteButtonEnabledState()`, che imposta lo stato di attivazione al pulsante per la richiesta di esecuzione di una sotto – espressione in base al valore immesso per i parametri della stessa (andrà associato agli eventi `ValueChanged` delle singole istanze di `ParameterWithValue`);

```
void SetExecuteButtonEnabledState()
{
    if (_executeButton != null)
        _executeButton.IsEnabled =
          !Model.Parameters.Any(
            param => param.Enabled == true &&
                     param.Value == null
          );
}
```

- Metodi per la gestione degli eventi: contiene il metodo `OnModelChanged()`, ereditato dalla classe base `ViewBase`, più i metodi di reazione alla selezione di un nodo e alla pressione del tasto di esecuzione di una sotto – espressione;

- Metodi di aggiornamento del layout: questi metodi definiscono una serie di modifiche alla veste grafica e saranno richiamati in seguito all'avvenimento di certi eventi (essenzialmente, sono richiamati dai metodi della sezione di gestione eventi);
- Metodi factory: metodi che istanziano e inizializzano i campi della classe.

Visualizzazione dell'albero

Il `TreeView` è creato attraverso codice XAML e associato ad un handler per l'evento di cambio selezione del nodo correntemente selezionato.

```
<TreeView
  Name="_tree"
  SelectedItemChanged="_tree_SelectedItemChanged"
>
```

L'associazione all'albero dell'espressione è definita via code behind, al momento di assegnazione del modello (che sarà quindi associato in maniera univoca al controllo utente).

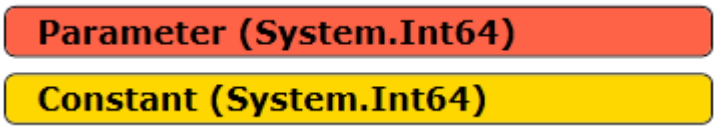
Il file XAML definisce inoltre una serie di `DataTemplate`, interni alle risorse dell'albero, per la visualizzazione diversificata di diversi tipi di nodo. In particolare si definiscono:

- Un template generico per i nodi discendenti da `Expression`:


```
<DataTemplate DataType="{x:Type expressions:Expression}">
    <DockPanel>
        <Border
           Margin="0,4"
           DockPanel.Dock="Top"
           BorderThickness="1"
           CornerRadius="5"
           Padding="10,2,10,2"
           Background="LightSteelBlue"
           BorderBrush="Black">
            <TextBlock
               FontFamily="Verdana"
               FontSize="16"
               FontWeight="Bold">
```

```
                <TextBlock.Text>
                    <MultiBinding
                       StringFormat=" {0} Expression">
                        <Binding Path="NodeType"/>
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </Border>
    </DockPanel>
</DataTemplate>
```

- Due template specifici per i nodi di tipo `ParameterExpression` e `ConstantExpression`, che variano dal primo per il testo visualizzato e lo sfondo:



- Un template con sfondo diverso per i tipi di nodo rappresentabili, ma non discendenti da `Expression`; è utilizzata una tecnica di template base con template specifici che fanno riferimento a quello base, modificandone solo il colore di sfondo:


```
<ControlTemplate x:Key="generalObjectTemplate">
    <DockPanel>
        <Border
           Margin="0,4"
           DockPanel.Dock="Top"
           BorderThickness="1"
           CornerRadius="5"
           Padding="10,2,10,2"
           Background="{TemplateBinding Background}"
           BorderBrush="Black">
            <TextBlock
               FontFamily="Verdana"
               FontSize="16"
               FontWeight="Bold"
               Text="{TemplateBinding Tag}">
            </TextBlock>
```

```
        </Border>
    </DockPanel>
</ControlTemplate>


<DataTemplate DataType="{x:Type expressions:CatchBlock}">
    <ContentControl
       Content="{Binding}"
       Template="{StaticResource generalObjectTemplate}"
       Background="LimeGreen"
       Tag="CatchBlock Node" />
</DataTemplate>


<!-- Uguale per gli altri nodi -->
```

Visualizzazione dei dettagli sui nodi

La parte destra del layout è formata da 4 blocchi, tutti riferiti al nodo attualmente selezionato:

- Rappresentazione testuale del nodo;
- Dettagli del nodo;
- Eventuale lista per raccogliere i parametri di input e tasto di richiesta esecuzione, solo per sotto – espressioni compilabili;
- Spazio di visualizzazione del risultato, solo per espressioni di cui è richiesta l'esecuzione.

Ogni blocco è preceduto da un testo che lo introduca, implementato come `TextBlock` a cui è assegnato uno stile.


```
<Style x:Key="headerText" TargetType="TextBlock">
    <Setter Property="Background" Value="Black"></Setter>
    <Setter Property="Foreground" Value="White"></Setter>
    <Setter Property="FontSize" Value="14"></Setter>
    <Setter Property="FontWeight" Value="Bold"></Setter>
</Style>
```

Il blocco di visualizzazione testuale prevede una semplice stampa della versione stringa del nodo corrente (quindi banalmente la chiamata al metodo `ToString()` della classe).

Il blocco di visualizzazione dei dettagli del nodo è implementato mediante utilizzo di un controllo di Windows Form, una `PropertyGrid`. Per utilizzare un controllo Windows Form dentro ad un progetto WPF è necessario:

- Aggiungere al progetto un riferimento all'assembly `WindowsFormIntegration`, contenuto nel file WindowsFormIntegration.dll;
- Aggiungere al progetto un riferimento all'assembly `WindowsForm`, contenuto nel file System.Windows.Forms.dll;
- Aggiungere nella radice del file XAML la definizione del namespace per Windows Form, aggiungendo un attributo di tipo `xmlns`:

```
xmlns:wf="clr-namespace:
System.Windows.Forms;assembly=System.Windows.Forms"
```

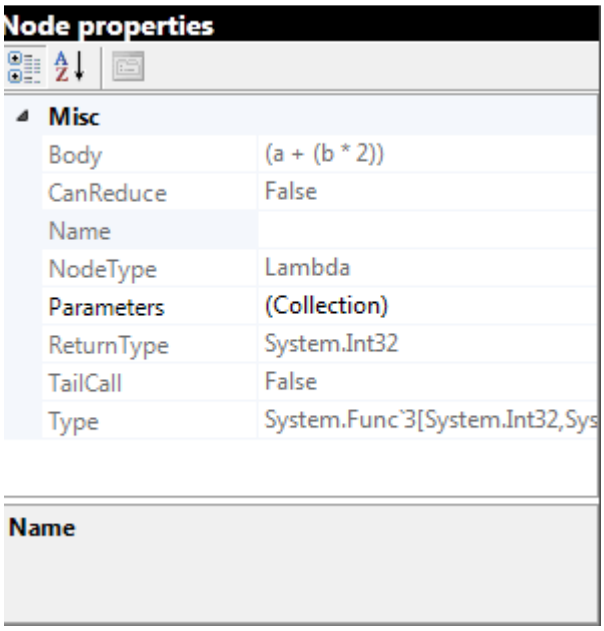
- Racchiudere i controlli Windows Form, utilizzabili con dei tag XAML omonimi (proprio come si fa con i controlli nativi WPF), dentro ad un controllo chiamato `WindowsFormHost`:

```
<WindowsFormsHost>
    <wf:PropertyGrid x:Name="_propertyGrid" />
</WindowsFormsHost>
```

L'utilizzo del controllo `PropertyGrid` è molto semplice, poiché è sufficiente assegnare alla sua proprietà `SelectedObject` un qualsiasi oggetto per avere in automatico la lista delle sue proprietà pubbliche e dei rispettivi valori (il tutto ottenuto, ovviamente, mediante Reflection e quindi in maniera dinamica).

Come già detto, la struttura dell'Expression Tree è immutabile per sua definizione: non è possibile aggiungere o rimuovere nodi, e neanche modificare quelli esistenti; l'applicazione non prevede alcun meccanismo per la modifica dei nodi (che dovrebbe prevedere la ricostruzione da zero dell'albero), il suo scopo è

unicamente quello di visualizzare gli alberi e rendere eseguibili alcune tipologie di espressione.



Sono definiti inoltre altri due blocchi, che avranno lo scopo di mostrare parametri e risultati: ovviamente, all'avvio dell'applicazione, questi dovranno essere invisibili. Ciascun blocco è definito sia come insieme di testata e contenuto (`_executionPanel` e `_resultPanel`), sia come vero e proprio blocco per i dati (`_executionInnerPanel` e `_resultInnerPanel`), ovvero la zona dove compariranno, rispettivamente, i parametri con il tasto di esecuzione e il risultato.

```
<DockPanel Name="_executionPanel" Visibility="Collapsed">
    <TextBlock
       DockPanel.Dock="Top"
       Text="Expression execution"
       Style="{StaticResource headerText}" />
        <DockPanel
           DockPanel.Dock="Top"
           Name="_executionInnerPanel" />
</DockPanel>
<DockPanel Name="_resultPanel"  Visibility="Collapsed">
    <TextBlock
       DockPanel.Dock="Top"
       Text="Execution result"
       Style="{StaticResource headerText}" />
```

```
        <DockPanel
           DockPanel.Dock="Top"
           Name="_resultInnerPanel" />
</DockPanel>
```

Evento di selezione di un nodo

All'atto di selezione di un nodo dell'albero da parte dell'utente, il sistema deve:

1. Ottenere l'oggetto attualmente selezionato;
2. Mostrare il risultato della chiamata al metodo `ToString()` sull'oggetto corrente nell'apposito pannello;
3. Impostare l'oggetto corrente come valore della proprietà `SelectedObject` della `PropertyGrid`;
4. Rendere invisibili i pannelli di esecuzione (`_executionPanel`) e di risultato (`_resultPanel`).

Inoltre, per i nodi discendenti da `Expression` e che modella sotto – espressioni eseguibili (verificabile mediante chiamata a `IsSubExpressionCompilable()` del modello), il sistema deve:

5. Mostrare, per ogni parametro dell'espressione, una casella di input e il nome e tipo del parametro corrente (o notificare all'utente che la sotto – espressione selezionata non ha parametri);
6. Mostrare un pulsante per l'esecuzione della sotto – espressione corrente;
7. Rendere visibile il blocco di raccolta input (`_executionPanel`).

Per fornire questo servizio, è sufficiente modificare il valore della proprietà `CurrentObject` della classe `ETViewerTree`:

```
private void _tree_SelectedItemChanged(
  object sender,
  RoutedPropertyChangedEventArgs<object> e)
{
    TreeView tree =
      sender as System.Windows.Controls.TreeView;
    TreeViewItem item =
      tree.SelectedItem as TreeViewItem;
```

```
    CurrentObject = item != null ? item.Header : null;
}
```

Il setter della proprietà `CurrentObject` si occuperà quindi di modificare il layout, delegando il compito ad altri servizi della classe:

```
set
{
    _currentObject = value;
    if (value != null)
    {
        UpdateLayoutAfterCurrentObjectChanged();
        if (CurrentExpression != null)
        {
            ExecuteButton.Tag = CurrentExpression;
            UpdateLaytoutAfterSubExpressionSelected();
        }
    }
}
```

Il metodo funziona in questo modo:

- In ogni caso, viene aggiornato il layout per la selezione di un nuovo nodo, visualizzando nei pannelli i dettagli sul nodo corrente:

```
private void UpdateLayoutAfterCurrentObjectChanged()
{
    _propertyGrid.SelectedObject = CurrentObject;
    _toStringTextBlock.Text = CurrentObject.ToString();
    _executionInnerPanel.Children.Clear();
    _executionPanel.Visibility =
      System.Windows.Visibility.Collapsed;
    _resultPanel.Visibility =
      System.Windows.Visibility.Collapsed;
}
```

- Nel caso il nodo sia discendente da `Expression` (la proprietà `CurrentExpression` è in realtà un wrapper verso una versione di `_currentObject` dopo casting a `Expression`), viene assegnata la

sotto – espressione corrente alla proprietà `Tag` del pulsante di esecuzione e si aggiorna nuovamente il layout:

```
private void UpdateLaytoutAfterSubExpressionSelected()
{
    if (Model.IsSubExpressionCompilable(
      CurrentExpression))
    {
        if (Model.Parameters.Count > 0)
        {
            DockPanel.SetDock(
              ParametersListView, Dock.Top);
            _executionInnerPanel.Children.Add(
              ParametersListView);
            ParametersListView =
              CreateParametersListView();
            Model.SetParametersStateForSubExpression(
              CurrentExpression);
            SetExecuteButtonEnabledState();
        }
        else
        {
            DockPanel.SetDock(
              NoParametersTextBlock, Dock.Top);
            _executionInnerPanel.Children.Add(
              NoParametersTextBlock);
        }

        DockPanel.SetDock(ExecuteButton, Dock.Top);
        _executionInnerPanel.Children.Add(ExecuteButton);

        _executionPanel.Visibility =
          System.Windows.Visibility.Visible;
        _resultPanel.Visibility =
          System.Windows.Visibility.Collapsed;
    }
}
```

Il modello dei dati fornisce un metodo per l'attivazione e disattivazione dei parametri, `SetParametersStateForSubExpression()`, in grado di modificare lo stato della proprietà `Enabled` dell'istanza di `ParameterWithValue` in modo da renderla conforme all'istanza attualmente selezionata.

Raccolta input

Il metodo di creazione della lista si occupa di istanziare un oggetto di tipo `ItemsControl` (controllo utente WPF che consente il binding ad una qualsiasi istanza di classe che implementi l'interfaccia `IEnumerable`) e di effettuare il data binding alla lista dei parametri ottenuta dal modello, nonché di assegnargli un `DataTemplate` appropriato.

```
private ItemsControl CreateParametersListView()
{
    return new ItemsControl()
    {
        ItemsSource = Model.Parameters,
        ItemTemplate =
         (DataTemplate)
           FindResource("listViewItemTemplate")
    };
}
```

Come si evince dal frammento di codice, questo codice non risolve due situazioni importanti:

1. Vengono mostrati i parametri dell'espressione radice, anche se non direttamente coinvolti nella sotto – espressione corrente;
2. Non è specificato in alcun modo di inserire una casella di testo per la raccolta dell'input.

Entrambi i problemi sono risolti dal file XAML associato, mediante la definizione, come risorsa del controllo, del template specificato nel code behind, ossia un `DataTemplate` di nome "listViewItemTemplate".

Il template definisce che ogni oggetto sia visualizzato in questo modo:

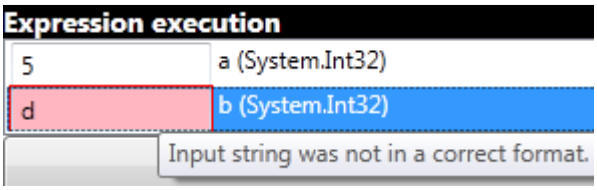
- Una casella di testo per l'input utente, con controlli di validazione e stile modificato in presenza di errori (sfondo rosso):

```
<TextBox
   IsEnabled="{Binding Enabled}"
   Text="{Binding Path=Value,
                  Mode=TwoWay,
                  ValidatesOnExceptions=True}">
    <TextBox.Style>
        <Style TargetType="{x:Type TextBox}">
            <Style.Triggers>
                <Trigger
                   Property="Validation.HasError"
                   Value="True">
                    <Setter
                       Property="ToolTip"
                       Value="{Binding
                               RelativeSource={x:Static
                                 RelativeSource.Self},
                       Path=(Validation.Errors)[0].
                             ErrorContent}" />
                    <Setter
                       Property="Background"
                       Value="LightPink" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </TextBox.Style>
</TextBox>
```

- Un blocco di testo indicante nome e tipo della proprietà:

```
<TextBlock Grid.Column="1">
    <TextBlock.Text>
        <MultiBinding StringFormat=" {0} ({1})">
            <Binding Path="Parameter.Name"/>
            <Binding Path="Parameter.Type"/>
        </MultiBinding>
    </TextBlock.Text>
</TextBlock>
```

La disattivazione della casella di input è effettuata sulla base del flag di `Enabled` del parametro associato al controllo stesso.



Evento di richiesta di esecuzione

La richiesta di esecuzione di una sotto – espressione può avvenire solo nel caso in cui siano stati inseriti tutti i parametri della stessa.

All'atto della pressione del pulsante di esecuzione, all'utente viene mostrato il risultato della compilazione ed esecuzione della sotto – espressione corrente.

La compilazione ed esecuzione è ovviamente delegata al modello, mediante chiamata al metodo `ExecuteSubExpression()`, che si serve della lista parametri con i rispettivi valori per calcolare e restituire il risultato.

Nel caso in cui si verifichi qualche problema, verrà visualizzato un messaggio di errore. Chiaramente, l'esecuzione di una espressione che non fornisce alcun risultato deve essere lecita ed abilitata, poiché questa, di fatto, modella semplicemente un metodo con valore di ritorno `void`.

```
void _executeButton_Click(
  object sender, RoutedEventArgs e)
{
    try
    {
        object result =
          Model.ExecuteSubExpression(CurrentExpression);
        if (result != null)
        {
            SetResultBlockStyle(
              result.ToString(),"correctResultBlock");
            AddResultDetailsContextMenu(result);
        }
        else
        {
            SetResultBlockStyle(
              "Void method", "voidResultBlock");
```

```
        }
    }
    catch (Exception exception)
    {
        SetResultBlockStyle(
          exception.Message,"exceptionResultBlock");
    }


    UpdateLayoutAfterExecutionRequest();
}
```

Sono quindi possibili tre situazioni:

1. Compilazione ed esecuzione della sotto – espressione correttamente eseguite e risultato disponibile;
2. Compilazione ed esecuzione della sotto – espressione correttamente eseguite ma nessun risultato restituito;
3. Errore di compilazione e/o di esecuzione.

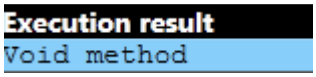
Il pannello del risultato (`_resultPanel`) viene quindi reso visibile (in ogni caso) e al suo interno viene creato un box di testo che sarà popolato con il risultato (versione `ToString()`) o con il messaggio di errore ottenuto.

Sono quindi applicati tre stili differenti, a seconda dei casi:

- Uno stile di default (`correctResultBlock`) per risultati corretti:

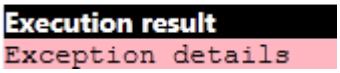

```
<Style x:Key="correctResultBlock"
   TargetType="TextBlock">
    <Setter
       Property="Background"
       Value="LightGreen" />
</Style>
```

- Uno stile (`voidResultBlock`) per metodi `void`:



```
<Style x:Key="voidResultBlock"
   TargetType="TextBlock">
    <Setter
       Property="Background"
       Value="LightSkyBlue" />
</Style>
```

- Uno stile (`exceptionResultBlock`) per gli errori:



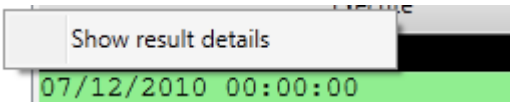

```
<Style x:Key="exceptionResultBlock"
   TargetType="TextBlock">
    <Setter
       Property="Background"
       Value="LightPink" />
</Style>
```

Il metodo di aggiornamento layout deve occuparsi solo di rendere visibile il blocco per la visualizzazione del risultato:

```
private void UpdateLayoutAfterExecutionRequest()
{
    _resultPanel.Visibility =
      System.Windows.Visibility.Visible;
}
```

Dettagli del risultato

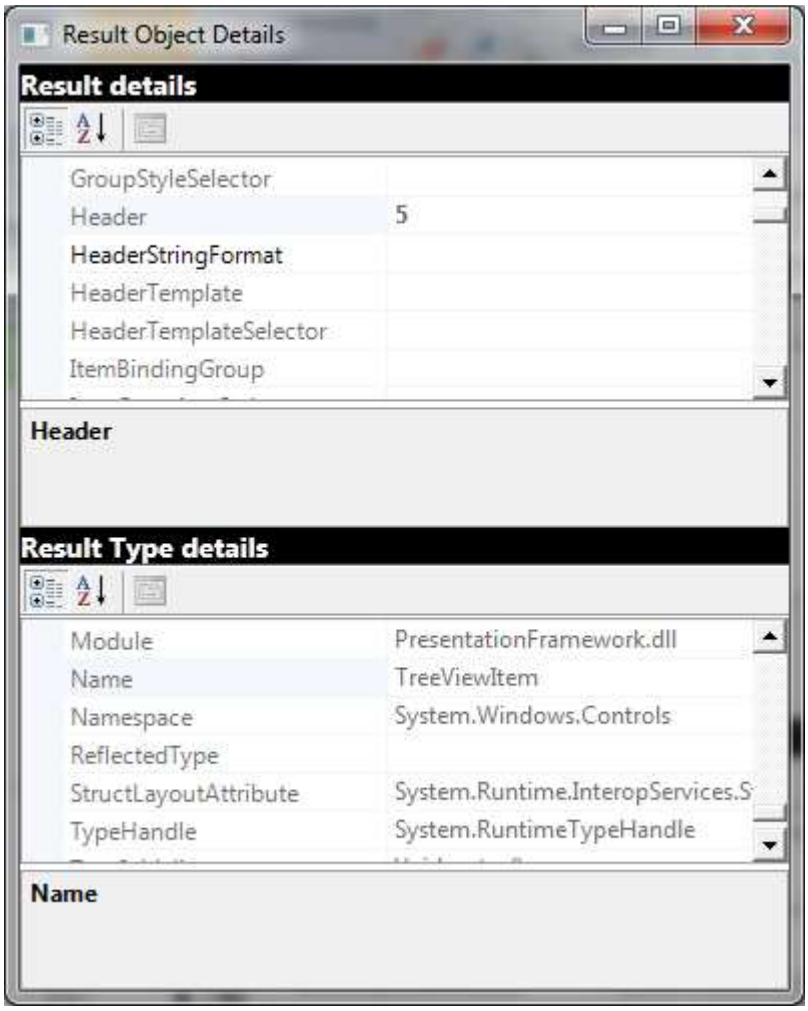
Se il valore restituito dopo l'esecuzione della sotto – espressione corrente non è un valore semplice, ma complesso, viene aggiunto un menu contestuale al box di visualizzazione del risultato. Tale menù contiene una sola voce, in grado di far visualizzare all'utente tutti i dettagli dello stesso.



La visualizzazione dei dettagli del risultato si ha attraverso una nuova finestra, mostrata in modalità `Dialog`. Questa finestra, implementata in una classe chiamata `PropertyGridWindow`, mostra due `PropertyGrid`, per mostrare due tipi di informazioni:

- Le proprietà dell'oggetto risultato;
- Le proprietà del tipo dell'oggetto risultato.

In questo modo si garantisce una informazione molto completa.



Il metodo per l'esecuzione della sotto – espressione corrente delega al metodo `AddResultDetailsContextMenu()` il compito di aggiungere il menù contestuale ed associargli un handler per l'apertura della finestra.

Chiaramente, per i tipi scalari non ha alcun senso mostrare la finestra dei dettagli: quindi il menù contestuale sarà attivo solamente per quei risultati la cui classe ha almeno una proprietà pubblica visualizzabile (le proprietà pubbliche sono ottenute via Reflection grazie alla chiamata al metodo `GetProperties()` della classe `Type`).

```
private void AddResultDetailsContextMenu(
  TextBlock resultBlock, object result)
{
    if (
      result != null &&
      result.GetType().GetProperties().Count() > 0)
    {
        resultBlock.ContextMenu = new ContextMenu();
        MenuItem menuItem = new MenuItem()
        { Header = "Show result details" };
        menuItem.Click +=
          delegate(object sender, RoutedEventArgs e)
        {
            PropertyGridWindow window =
              new PropertyGridWindow()
              { SelectedObject = result };
            window.ShowDialog();
        };
        resultBlock.ContextMenu.Items.Add(menuItem);
    }
}
```

Associazione del modello

L'associazione del modello avviene nel momento in cui è settato un nuovo valore alla proprietà `Model` della classe `ETViewerTree` (proprietà definita nella classe base `ViewBase`): questo evento scatena la chiamata al metodo `OnModelChanged()`, astratto nella classe base e ridefinito nella classe derivata.

```
protected override void OnModelChanged(
  object sender, EventArgs e)
{
    InitializeLayout();
    ParametersListView = CreateParametersListView();
    foreach (
      ParameterWithValue parameter in Model.Parameters)
        parameter.ValueChanged +=
          new Action(SetExecuteButtonEnabledState);
}
```

All’atto di associazione con un modello dati, quindi:

- Viene inizializzato il layout, che prevede lo svuotamento dell’albero, la scomparsa dei blocchi di raccolta input e visualizzazione del risultato e la pulizia del pannello di visualizzazione dettagli su un nodo;
- Viene creata la lista dei parametri e associata ad un controllo grafico e al rispettivo template:

```
private ItemsControl CreateParametersListView()
{
    return new ItemsControl()
    {
        ItemsSource = Model.Parameters,
        ItemTemplate =
          (DataTemplate)
            FindResource("listViewItemTemplate")
    };
}
```

- Viene registrato all’evento di cambio di ciascun parametro un metodo per l’aggiornamento del pulsante di esecuzione:

```
void SetExecuteButtonEnabledState()
{
    if (_executeButton != null)
        _executeButton.IsEnabled =
          !Model.Parameters.Any(
            param => param.Enabled == true &&
                     param.Value == null
          );
}
```

Come già detto, quindi, l’esecuzione di una sotto – espressione è ammessa solo una volta che tutti i parametri coinvolti (quelli attivi) hanno un valore associato.

CAPITOLO 6: Assemblaggio

Creazione del modulo DLL

Il software sarà distribuito come modulo DLL, ossia una libreria di classi utilizzabile da altri applicativi e caricata a runtime. Per creare una libreria di questo tipo, con Visual Studio 2010, è sufficiente creare un progetto C# di tipo "Class Library".

All'interno del progetto si inseriscono tutti i file di codice, procedendo inoltre all'aggiunta dei reference di progetto necessari (di base, sono inclusi solo i riferimenti essenziali, ma vanno aggiunti tutti quelli per la veste grafica e la modellazione di espressioni).

Effettuando il "build" di un progetto "Class Library" si avrà in output, anziché il classico file `.exe`, un file `.dll`.

Utilizzo della libreria

Per utilizzare una libreria DLL in un progetto Visual Studio 2010, è sufficiente aggiungere l'assembly ai reference del progetto: utilizzando la reflection, la suite Visual Studio è in grado di fornire indicazioni (intelliSense) sui metodi contenuti nell'assembly.

Una volta realizzata la libreria DLL, quindi, si può procedere con la creazione di un programma di prova, contenente un menù per la selezione di una espressione (tra una lista predefinita) e il controllo utente precedentemente creato.

Per utilizzare questa libreria, in particolare, è sufficiente:

- Aggiungere un riferimento al namespace `ETViewer.View` ad un file XAML di progetto;

```
xmlns:viewBase=
  "clr-namespace:ETViewer.View;assembly=ETViewerDLL"
```

- Aggiungere un controllo di tipo `ETViewerTree` a tale file XAML;

```
<viewBase:ETViewerTree x:Name="_treeView" />
```

- Inizializzare il modello con una particolare espressione.

```
private void SetNewExpression(Expression newExpression)
{
  _treeView.Model = new ExpressionModel(newExpression);
}
```

CAPITOLO 7: Test

Modalità di test

Per effettuare alcuni test sulla libreria creata, si costruisce un programma di prova che utilizzi la DLL stessa. Questo programma contiene una serie di espressioni da utilizzare per verificare la funzionalità delle principali strutture dati implementate.

Espressioni di prova

Espressione 1

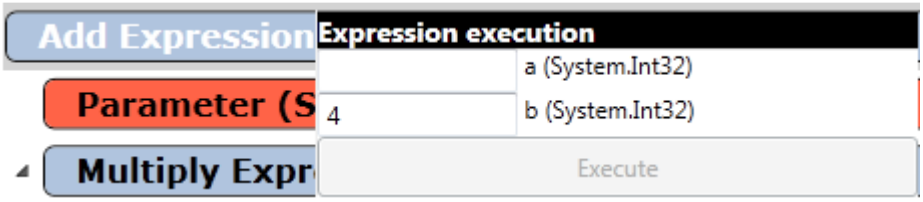
```
Expression<Func<Int64, Int64>> expression1 = a => a + 2;
```

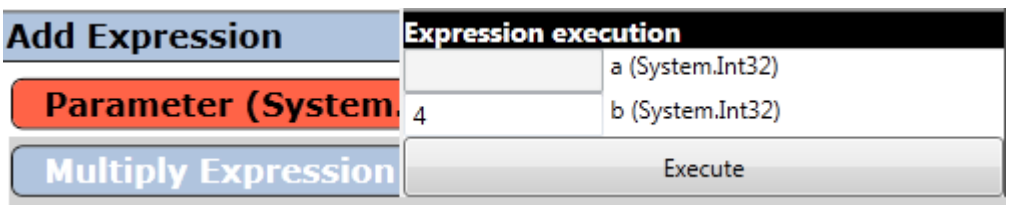
Questa espressione è una banale somma di 2 unità ad un numero intero da 64 bit. Permette di verificare i template per parametri e costanti, la raccolta di un tipo intero e la visualizzazione di un risultato intero (quindi non deve essere presente il menù contestuale per i dettagli sul risultato, poiché un intero non dispone di parametri pubblici).

Espressione 2

```
Expression<Func<int, int, int>> expression2 =
  (a, b) => a + b * 2;
```

Questa espressione permette di testare l'applicazione quando i parametri di input sono più di uno. In particolare, se viene selezionata la sotto – espressione "b * 2", il sistema deve verificare solo la validità del parametro "b" e non permettere la modifica di "a".





Espressione 3

```
Expression<Func<int, int>> expression3 = a => -a;
```

Questa espressione permette di analizzare l'esecuzione di una espressione unaria.

Espressione 4

```
ParameterExpression parameter4 =
  Expression.Parameter(typeof(int), "b");
Expression expression4 =
  Expression.Lambda<Func<int, int>>(
    Expression.MakeBinary
      (
        ExpressionType.Add,
        parameter4,
        Expression.Constant(1)
      ),
parameter4);
```

Questa espressione, simile alla prima, è utilizzata per verificare la costruzione di istanze di `Expression` mediante costruttori e non da una lambda.

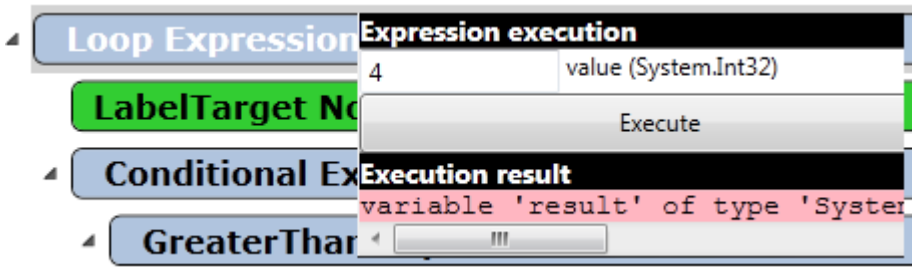
Espressione 5

```
ParameterExpression value =
  Expression.Parameter(typeof(int), "value");
ParameterExpression result =
  Expression.Parameter(typeof(int), "result");
LabelTarget label =
  Expression.Label(typeof(int), "_labelProva");
BlockExpression block = Expression.Block(
  new[] { result },
  Expression.Assign(result, Expression.Constant(1)),
  Expression.Loop(
    Expression.IfThenElse(
      Expression.GreaterThan(
        value, Expression.Constant(1)),
      Expression.MultiplyAssign(
        result,
        Expression.PostDecrementAssign(value)),
      Expression.Break(label, result)
    ),
```

```
    label
  )
);
```

Questa espressione, che calcola il fattoriale di un intero passato come argomento (parametro `value`), permette di verificare diversi elementi:

- Modellazione di un ciclo;
- Rappresentazione di nodi non derivati da `Expression` (in questo caso, nodi di tipo `LabelTarget`);
- Compilabilità in base ai riferimenti delle etichette: in particolare, l'espressione nel suo complesso deve essere eseguibile, ma non può esserlo, ad esempio, la struttura condizionale interna, poiché il fallimento del test comporta un salto ad una etichetta esterna alla struttura stessa;
- Verifica di altri fallimenti: chiedendo l'esecuzione della sola espressione di ciclo, verrà restituita una eccezione, poiché all'interno del ciclo è utilizzata una variabile (`result`), dichiarata all'esterno dello stesso e quindi sconosciuta in quel singolo ambito.



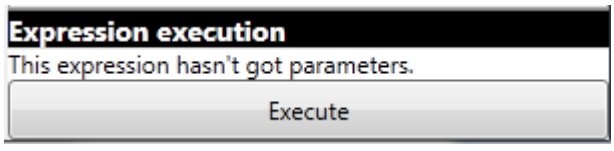
Espressione 6

```
ConstantExpression switchValue = Expression.Constant(2);
SwitchExpression switchExpr =
  Expression.Switch(
    switchValue,
    new SwitchCase[]
    {
        Expression.SwitchCase(
            Expression.Call(
                null,
```

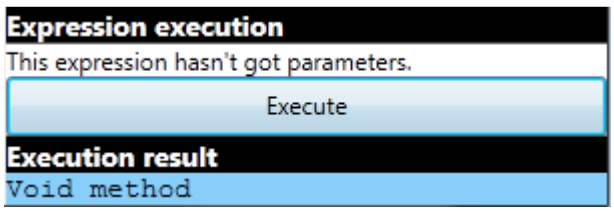
```
                typeof(Console).GetMethod(
                  "WriteLine",
                  new Type[] { typeof(String) }),
                Expression.Constant("First")
            ),
            Expression.Constant(1)
        ),
        Expression.SwitchCase(
            Expression.Call(
                null,
                typeof(Console).GetMethod(
                  "WriteLine",
                  new Type[] { typeof(String) }),
                Expression.Constant("Second")
            ),
            Expression.Constant(2)
        )
    }
);
```

Questa espressione modella una struttura di selezione multipla senza caso di default, in cui il parametro di scelta è pre – cablato nel codice. Permette di testare:

- Strutture a scelta multipla;
- Rappresentazione di nodi non derivati da `Expression` (in questo caso, nodi di tipo `SwitchCase`);
- Attivazione del pulsante di esecuzione per metodi senza parametri;



- Visualizzazione del risultato di metodi `void`.



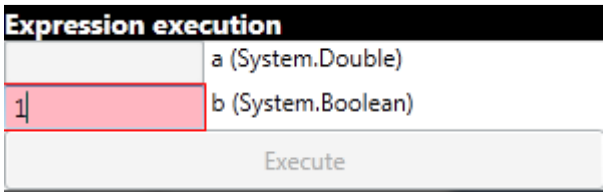
Espressione 7

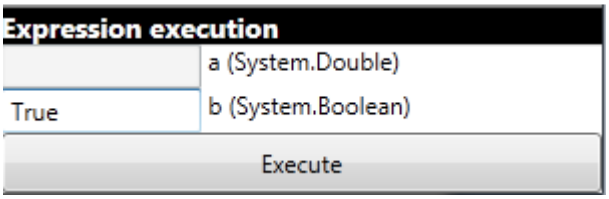
```
Expression<Func<double, bool, string>> expression7 =
  (a, b) => String.Format("{0} - {1}", a, b);
```

Questa espressione stampa a schermo due parametri, uno numerico reale e uno booleano, separandoli da spazi e un trattino.

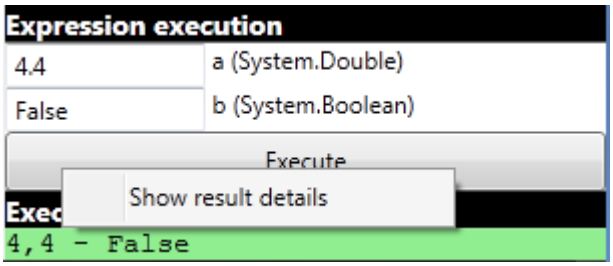
Permette di verificare:

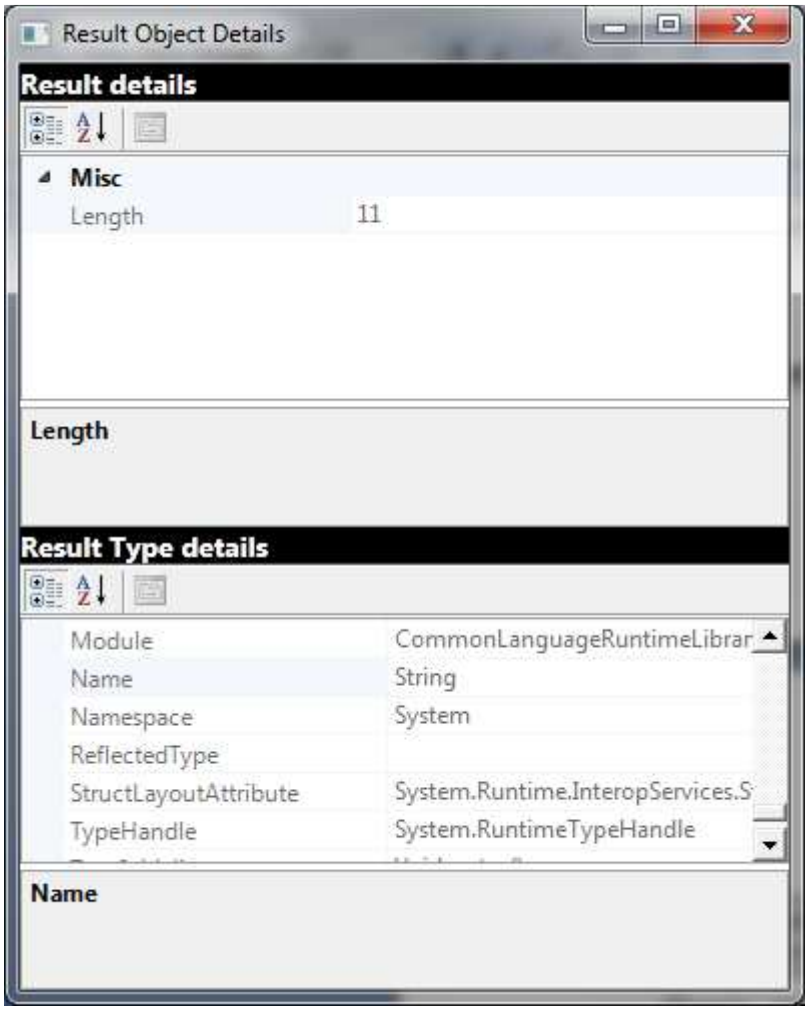
- La raccolta di dati booleani;





- La raccolta di dati numerici reali (il dato è inserito secondo cultura locale, quindi nel caso di italiano con separatore tra interi e decimali rappresentato dalla virgola e non dal punto);
- La conversione di un oggetto ad `object` (l'espressione lambda è tradotta in una struttura contenente due espressioni di tipo `ConvertExpression`, per la conversione dei parametri ad `object`);
- La visualizzazione di un output stringa (e quindi possibilità di visualizzarne i dettagli in una finestra separata, contenente la lista dei valori delle proprietà pubbliche dell'istanza del risultato e del suo tipo).



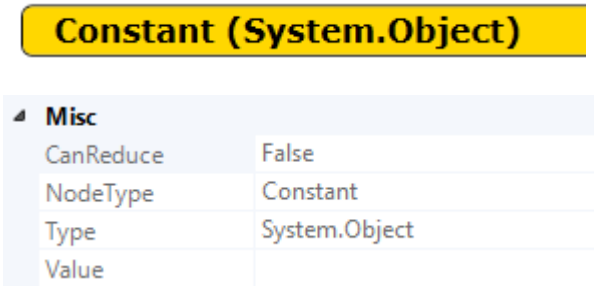


Espressione 8

```
Expression<Func<Expression, bool>> expression8 =
  a => a == null;
```

Questa espressione modella una struttura non eseguibile, poiché il parametro di input non è un tipo semplice, ma una istanza di `Expression` (input non raccoglibile).

Permette inoltre di visualizzare la modellazione del valore speciale `null`, rappresentato come `ConstantExpression`.



Espressione 9

```
Expression<Func<int, TreeViewItem>> expression9 =
  a => new TreeViewItem() { Header = a };
```

Questa espressione, eseguibile, costruisce una istanza di `TreeViewItem` a partire da un intero, che viene salvato nella proprietà `Header` del controllo. Questa espressione deve essere eseguibile e permette di:

- Visualizzare costruzione e inizializzazione di istanze di classe (espressioni di tipo `NewExpression` e `MemberInitExpression`;
- Visualizzare nodi non discendenti da `Expression`, in particolare nodi `MemberInit`;
- Visualizzare risultati complessi (istanze di classe `TreeViewItem`), con menù contestuale per la visualizzazione dei dettagli dell'istanza e del tipo.

Espressione 10

```
Expression<Func<int, string>> expression10 =
  a => (a + 3) + "Ciao";
```

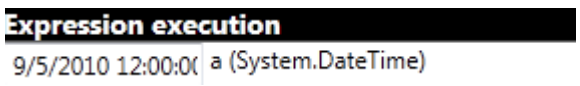
Questa espressione è utilizzata per testare l'eseguibilità e il tipo di risultato fornito dalle singole sotto – espressioni.

Espressione 11

```
Expression<Func<DateTime, int, TreeViewItem, bool>>
  expression11 =
    (a, b, c) =>
      (
        a.AddDays(4).ToString() +
        String.Format("{0} {1}", b, c.Header)
      ).Length > 0;
```

Questa espressione non è eseguibile nella sua interezza (poiché utilizza un parametro complesso, di tipo `TreeViewItem`), ma può essere eseguita a blocchi.

Permette inoltre di verificare la raccolta di input di tipo `DateTime`, conformi alla cultura locale nell'input (ma non nell'output, nel senso che la data è comunque visualizzata nel formato `MM/GG/AAAA hh:mm:ss`).



Espressione 12

```
Expression<Func<int, int>> expression12 =
  a => a + (int)Convert.ChangeType("2", typeof(int));
```

Questa espressione modella una conversione da stringa ad intero. Se si modifica il valore della costante stringa ad un valore non convertibile in intero (ad esempio: “2x”), il risultato dell’espressione sarà sempre il lancio di una eccezione.

Inoltre si tratta di una espressione con un parametro, contenente una sotto – espressione senza parametri (la conversione).

APPENDICE A: Design Pattern

Panoramica

I Design Pattern sono schemi di progettazione utilizzati per risolvere in maniera affidabile e rapida problemi specifici frequenti. Ogni design pattern permette di sfruttare l'esperienza acquisita nell'affrontare e risolvere uno specifico problema progettuale, rendendo inoltre possibile il riutilizzo rapido in futuro.

Nel mondo object oriented, i design pattern devono rispettare i principali principi di progettazione, che garantiscono l'efficienza e vera riusabilità del modulo creato:

1. Principio della singola responsabilità: ogni classe ha una sola responsabilità, che affronta in maniera completa, corretta ed esclusiva;
2. Principio dell'open/close: le entità (classi, funzioni, moduli) devono essere facilmente estendibili, ma difficilmente modificabili;
3. Principio di inversione delle dipendenze: bisogna dipendere da astrazioni, non da oggetti concreti; moduli di alto livello non devono dipendere da moduli di basso livello, ma entrambi devono dipendere da astrazioni; le astrazioni non devono dipendere da moduli concreti, ma da altre astrazioni;
4. Principio di sostituzione di Liskov: ogni sottoclasse deve essere una possibile sostituta della propria classe base, onorandone i suoi contratti;
5. Principio di segregazione delle interfacce: i client non devono essere obbligati a dipendere da interfacce che non usano.

Ogni design pattern ha diverse caratteristiche, tra cui:

- Un nome identificativo;
- La descrizione di un problema che sono in grado di risolvere;
- La descrizione della soluzione al problema;
- Le conseguenze (positive e negative) scaturite dall'utilizzo del pattern.

Pattern Visitor

Il pattern Visitor è utilizzato per consentire di aggiungere operazioni su una struttura dati, senza dover modificare le classi degli elementi coinvolti.

E' molto utile per aggiungere operazioni che lavorano con collezioni di dati eterogenei non conosciuti a priori.

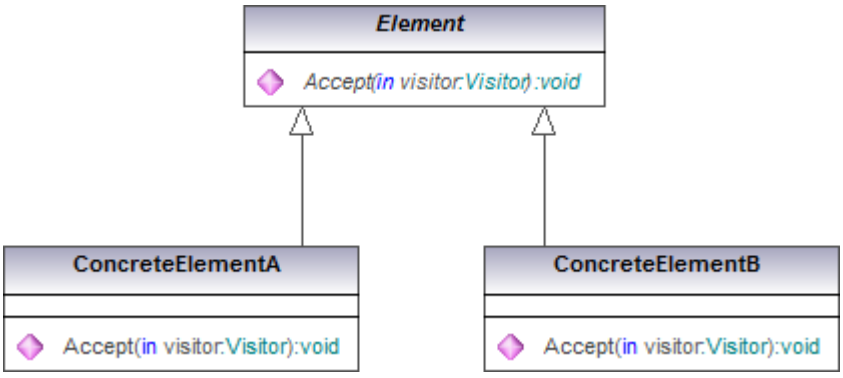
L'approccio più immediato potrebbe essere quello di implementare il codice delle operazioni dentro alle singole classi, ma questo comporterebbe problemi nell'aggiunta di nuove operazioni (ogni nuova operazione comporterebbe la ricompilazione di tutte le classi della struttura).

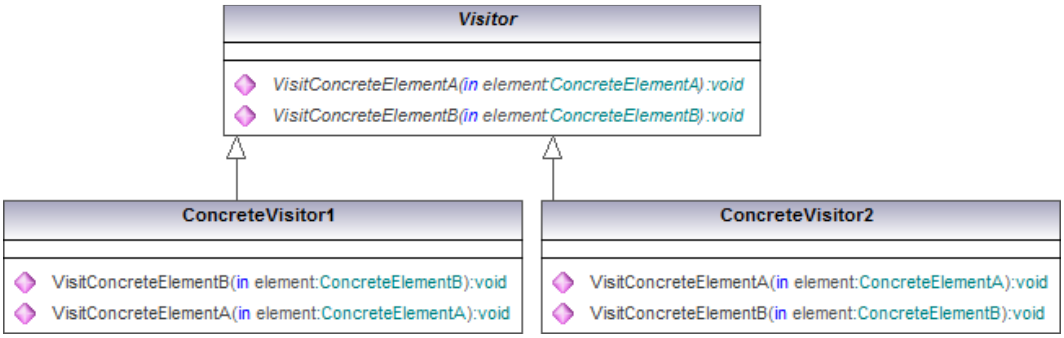
Gli obiettivi del pattern Visitor sono quindi due:

- Rappresentare una operazione eseguibile su una struttura dati;
- Rendere possibile l'aggiunta di nuove operazioni senza modificare tale struttura dati e le classi coinvolte.

La gerarchia di elementi è solitamente modellata con una radice comune: questa può essere una classe (astratta o meno), ma anche una semplice interfaccia. In altre parole, non è necessario che gli elementi da visitare abbia una vera radice semantica comune.

L'unico requisito è che questa gerarchia di elementi sia altamente stabile, ossia è necessario che non sia probabile l'aggiunta o cancellazione di tipi di elementi, poiché questo, di fatto, renderebbe necessaria la sistemazione di tutti i visitor.





Ogni visitor modella una particolare operazione, eseguibile su tutti i tipi di elemento. Da questo si evince come sia necessaria la stabilità della gerarchia di elementi.

Ogni visitor può naturalmente contenere uno stato, aggiornato mentre si esegue l'operazione sui dati.

Infine, se necessario, è possibile aggiungere un generico metodo `Visit()`, che accetti in input un generico elemento di tipo `Element`, provvedendo al suo interno ad effettuare il dispatch sulla base del tipo dell'elemento, delegando poi l'esecuzione dell'operazione al metodo appropriato: questa strategia è quella utilizzata dalla classe `ExpressionVisitor`.

All'interno del metodo `Accept()` dei singoli elementi, si dovrà semplicemente invocare il metodo di visita specifico per quel tipo di istanza:

```
public void Accept(Visitor visitor)
{
  visitor.VisitConcreteEementA(this);
}
```

Per aggiungere una nuova operazione sulla struttura dati, è sufficiente aggiungere un nuovo tipo di visitor.

Per invocare una data operazione, è necessario:

- Istanziare l'oggetto discendente da `Element`
- Istanziare un oggetto discendente da `Visitor`
- Chiamare il metodo `Accept()` dell'elemento, passando come parametro immediato il visitor.

In altre parole, l’operazione che deve essere effettuata dipende dal tipo di due oggetti: quello dell’elemento e quello del visitor. Si parla quindi di operazione “double dispatch”.

Nella gerarchia di `Expression`, i metodi `Accept()` sono protetti e, come tali, non invocabili dall’esterno. Lo stesso vale per i metodi di visita specifici per i singoli nodi. L’unica operazione consentita all’esterno è la chiamata al metodo (del tutto generale, visto che accetta in input una generica istanza di `Expression`) `Visit()` di un generico visitor.

APPENDICE B: Delegati generici

Metodi generici

Un metodo generico è un metodo in cui un parametro è mantenuto generico, ossia specificabile in fase di invocazione.

```
void GenericMethod<T> (T input)
{
  ...
}
```

L'invocazione del metodo può avvenire specificando il tipo generico:

```
int x;
GenericMethod<int>(x);
```

Tuttavia, poiché il compilatore C# può prendere il tipo dei parametri invocando il metodo, non è necessario specificare il tipo generico in fase di chiamata: il metodo può essere invocato in maniera del tutto analoga a quelli non generici.

```
int x;
GenericMethod(x);
```

Delegati generici

Allo stesso modo dei metodi generici, è possibile definire un delegato generico, con uno o più tipi generici.

```
public delegate U GenericDelegate<T,U>(T input);
```

Ad un delegato generico si possono assegnare metodi specifici o anche espressioni lambda.

```
GenericDelegate<int, bool> method = a => a == 0;
```

La libreria standard di C# prevede una serie di delegati generici per vari scopi. Ad esempio, i delegati generici della famiglia `Action` modellano un metodo senza valore di ritorno, mentre quelli `Func` modellano un metodo con un valore di ritorno. Ad esempio:

- Un delegato `Action<int>` modella un metodo con un input intero e senza output;
- Un delegato `Func<int, bool>` modella un metodo con un input intero e un output booleano.

APPENDICE C: Glossario

Glossario dei termini

Abstract Syntax Tree	Struttura dati ad albero che modella una porzione di codice.
Albero (dell'espressione)	Struttura dati ad albero che modella una particolare espressione .NET 4.
Compilabilità (dell'espressione)	Insieme di regole che rendono una particolare espressione compilabile ed eseguibile.
Compilazione (dell'espressione)	Atto con cui una espressione viene trasformata in un delegato eseguibile.
Controllo utente	Serie di componenti grafici e codice per la logica sottostante, racchiusi in un pacchetto utilizzabile in qualsiasi altra applicazione o modulo grafico.
Delegato	Dato che modella un puntatore a funzione.
DLL	Libreria di classi, caricabile dinamicamente in fase di esecuzione di un programma.
Eseguibilità (dell'espressione)	*Vedere "Compilabilità (dell'espressione)"*
Espressione	Struttura dati che modella un frammento di codice.
Espressione radice	Espressione principale su cui sta lavorando l'applicazione. Rappresentata con un albero dell'espressione.
Etichetta	Istruzione di codice che rappresenta un punto con un

	nome, in modo che sia raggiungibile da altre istruzioni.
Expression	*Vedere "Espressione"*
Expression Tree	*Vedere "Albero (dell'espressione)"*
Label	*Vedere "Etichetta"*
Libreria di classi	*Vedere "DLL"*
Parametri di input (dell'espressione)	Variabili che devono essere assegnate all'espressione per ottenere un risultato.
Risultato (dell'espressione)	Valore di ritorno ottenuto al termine dell'esecuzione dell'espressione.
Sotto – albero	Porzione dell'albero dell'espressione che modella una sotto – espressione.
Sotto – espressione	Porzione dell'espressione radice.
User Control	*Vedere "Controllo utente"*
Validatore (di compilabilità)	Componente che giudica compilabile o meno una particolare espressione o sotto – espressione, sulla base di uno o più criteri.
Windows Presentation Foundation	Libreria Windows per la creazione di interfacce utente.
WPF	*Vedere "Windows Presentation Foundation"*

Conclusioni

Il software sviluppato permette, in conclusione, la visualizzazione di tutti i tipi di nodo e l'esecuzione di alcune particolari espressioni. I test eseguiti con il componente finale hanno dato esito positivo.

In eventuali future versioni della libreria, potrebbero essere aggiunti nuovi controlli per stabilire la compilabilità di una sotto – espressione (è sufficiente aggiungere un nuovo validatore ed aggiungerlo alla lista di quelli installati nel modello dei dati), ponendo attenzione ad altri particolari (ad esempio, si potrebbe stabilire di non rendere eseguibili espressioni che contengono riferimenti a variabili dichiarate all'esterno, situazione che in questa versione genera eccezione di esecuzione e un messaggio di errore mostrato all'utente).

L'eventuale aggiunta di nuovi tipi di espressione da parte degli sviluppatori di C# potrebbe richiedere, in futuro, la ricompilazione di alcune parti di codice: in particolare il visitor dell'albero ridefinisce necessariamente tutti i metodi di visita specifici e questo comporterebbe la ridefinizione anche dei nuovi metodi eventualmente aggiunti.

Dal punto di vista grafico, il cambio di stile di visualizzazione dei nodi richiede solo la modifica del file XAML e non del codice alle spalle. L'aggiunta di nuovi tipi di nodo, purché discendenti dalla classe base `Expression`, non richiede modifiche da questo punto di vista; sono invece necessarie modifiche in caso di aggiunta di tipi di nodo non discendenti dalla suddetta classe base.

Fonti

Materiale sulle Expression

- Materiale specifico sugli Expression Tree v2 sul sito CodePlex (www.codeplex.com), nella sezione sul Dynamic Language Runtime: http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Home
- Materiale ufficiale sul sito Microsoft (www.msdn.it), in particolare la sezione sugli alberi di espressione: http://msdn.microsoft.com/it-it/library/bb397951.aspx
- Libro "*C# 2008*" di Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson e Morgan Skinner (edizione italiana Hoepli)

Materiale su Windows Presentation Foundation

- Libro "*C# 2008*" di Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson e Morgan Skinner (edizione italiana Hoepli)
- Materiale ufficiale sul sito Microsoft (www.msdn.it), in particolare la sezione su WPF: http://msdn.microsoft.com/it-it/library/ms754130.aspx

Materiale sulla progettazione software

- Libro "*Ingegneria del software*" di I. Sommerville (settima edizione, editore Addison Wesley, 2005)
- Materiale fornito dal professor Bellavia Giuseppe, per il corso di "Ingegneria del software L-A" dell'anno accademico 2009/2010 dell'Università di Bologna: http://lia.deis.unibo.it/Courses/IngSwA0910/