ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DISI

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in
Fondamenti di Informatica T2

Generatori di codice in Visual Studio: modelli di funzionamento e applicazioni pratiche

CANDIDATO
Michele Francesco Di Lella

RELATORE:
Chiar.mo Prof. Enrico Denti

CORRELATORE
Roberta Calegari

Anno Accademico 2014/15

Sessione II

Indice

Introduzione

Un generatore di codice è un componente software realizzato con lo scopo di generare codice sorgente a partire da alcune informazioni chiave fornite dallo sviluppatore. In Visual Studio, l'ambiente di sviluppo di riferimento del framework .NET sviluppato da Microsoft, un componente di questo genere è realizzabile mediante il meccanismo delle estensioni.

Nell'upgrade avvenuto tra le versioni 2010 e 2013 di Visual Studio, Microsoft ha apportato alcune modifiche nel modello di funzionamento delle estensioni. Queste modifiche hanno conseguentemente intaccato anche il meccanismo di funzionamento dei generatori di codice in Visual Studio.

Primo obiettivo di questa tesi sarà l'analisi approfondita delle caratteristiche delle estensioni per Visual Studio e del processo di implementazione di un generatore di codice nell'IDE di Microsoft. L'analisi sarà dapprima incentrata sugli aspetti implementativi generali, per soffermarsi, in seguito, sulle differenze principali presenti tra la versione 2010 e 2013 di Visual Studio e sulle tecniche da adottare per compiere il porting di un generatore di codice da Visual Studio 2010 verso Visual Studio 2013.

Il secondo obiettivo della tesi riguarderà l'applicazione dei risultati dell'analisi precedente su di un vero generatore di codice progettato per Visual Studio 2010, con lo scopo di effettuare il porting di questo verso Visual Studio 2013. Il generatore di codice preso in analisi per raggiungere l'obiettivo sopracitato è stato sviluppato in un precedente lavoro di tesi[17], con l'obiettivo di rendere pienamente funzionante il framework P@J, parte di tuProlog, anche nella versione .NET di tale sistema.

La tesi è dunque strutturata come segue. Nel capitolo 1 sarà esplorato il framework .NET e il suo ambiente di sviluppo di riferimento Visual Studio. Nel capitolo 2 saranno analizzate le principali caratteristiche dei generatori di codice in Visual Studio, insieme al relativo processo implementativo specifico per Visual Studio 2010. In seguito, nel capitolo 3 saranno analizzate le novità introdotte in Visual Studio 2013 e saranno descritte le tecniche da adottare per

compiere il porting di un generatore di codice da Visual Studio 2010 verso Visual Studio 2013.

Nel capitolo 4 saranno illustrati alcuni aspetti di background relativi a tuProlog, insieme al processo di implementazione del generatore di codice, citato precedentemente, realizzato da Fabio Gravina.

Infine, il capitolo 5 affronterà il porting verso Visual Studio 2013 del generatore di codice di cui sopra, insieme all'implementazione di un'apposita applicazione di test che ne verifichi l'effettivo funzionamento.

Capitolo 1
Ambiente di sviluppo .NET

1.1 .NET Framework

.NET Framework[3] è una piattaforma di sviluppo per applicazioni e servizi web basati sulla implementazione della Common Language Infrastructure (CLI[4]) realizzata da Microsoft. La CLI è la specifica aperta, regolamentata dall'ISO, in cui sono descritte la struttura, il funzionamento e le caratteristiche base di questa piattaforma di sviluppo. Esistono altre implementazioni di questa specifica tra cui i progetti MONO[5] e DotGNU[6], entrambi concentrati principalmente sulla piattaforma Linux.

Seguendo le indicazioni della CLI, .NET mette a disposizione del progettista una suite completa di linguaggi in grado di essere compilati in un linguaggio intermedio chiamato Common Intermediate Language (CIL[4]) ed eseguiti su una macchina virtuale nota come Common Language Runtime (CLR[7]).

.NET condivide alcune caratteristiche con Java: entrambi gli ambienti sono basati sulle rispettive macchine virtuali che eseguono il codice intermedio, consentendo così la separazione tra il sistema operativo e le applicazioni, rendendo queste ultime portabili. Ciò rende quindi possibile eseguire il CIL ottenuto dalla compilazione con la piattaforma .NET su un calcolatore con in esecuzione la macchina virtuale CLR MONO[5], in quanto entrambe si rifanno alla stessa specifica.

CLI è concepita per essere compatibile con molti linguaggi di programmazione object oriented. Questo aspetto è stato fondamentale per Microsoft nello sviluppo di .NET, il quale ha goduto di ampia diffusione sin dal suo rilascio grazie al fatto che tra i linguaggi compatibili ci sono anche Visual Basic[8] e Visual C++[9], linguaggi di riferimento nella programmazione in ambiente Windows. Ha inoltre introdotto un nuovo linguaggio: C#[10], derivato da Delphi[11], C++[12] e Java

1.2 Visual Studio

L'ambiente di sviluppo (Integrated Development Environment, IDE) Visual Studio[14], sviluppato da Microsoft, fornisce l'accesso a tutti gli strumenti necessari per lo sviluppo di una applicazione per il framework .NET, tra cui: i compilatori, gli editor e i debugger per i diversi linguaggi della piattaforma. Visual Studio è stato progettato e sviluppato, similmente ad altri IDE (Eclipse, Netbeans), come un contenitore di funzionalità alle quali fornisce tutti i necessari meccanismi di accesso ai dati e la possibilità di scambiarsi informazioni. Queste funzionalità possono essere implementate e installate in due forme diverse: gli *add-in* e le *estensioni*.

Queste due categorie di componenti si differenziano per le API a cui hanno accesso, e quindi hanno differenti peculiarità. Gli add-in sono principalmente utilizzati per personalizzare e automatizzare l'IDE. Le estensioni, invece, sono focalizzate sulla business logic, quindi hanno come target una determinata classe di progetti (C#, VB, Console, Windows Form, etc...) o classe di file (xml, Resource, etc...).

In questo modo anche componenti come i compilatori e gli editor sono visti come funzionalità aggiuntive che vengono caricate dentro l'IDE solo quando necessarie. Ciò significa che il compilatore e l'editor per codice C# saranno caricati solo se l'utente aprirà o creerà un progetto di tipo C#-

1.3 Estensioni per Visual Studio

Per gli scopi di questa tesi, non si è ritenuto necessario analizzare i processi per personalizzare l'IDE, ma bensì quelli relativi all'aggiunta di una unità di business logic. Per questa ragione, come si vedrà nei capitoli successivi, si è optato per l'analisi della realizzazione di una estensione. Tale argomento verrà perciò brevemente approfondito.

La creazione di un'estensione per Visual Studio ha come unico prerequisito l'installazione del Visual Studio Software Development Kit (SDK)[15]. Questo kit viene distribuito separatamente rispetto alla installazione base di Visual Studio e fornisce anche alcuni esempi e la documentazione sul processo di sviluppo

1.3.1 Creare l'involucro di una nuova estensione

La creazione di una nuova estensione si basa su una breve procedura guidata messa a disposizione dal Visual Studio SDK.

Le estensioni di Visual Studio sono distribuite attraverso i file VSIX (Visual Studio Installer eXtension), i quali incapsulano l'estensione stessa e alcune informazioni aggiuntive atte a verificare la compatibilità dell'estensione con la versione di Visual Studio installata nel sistema.

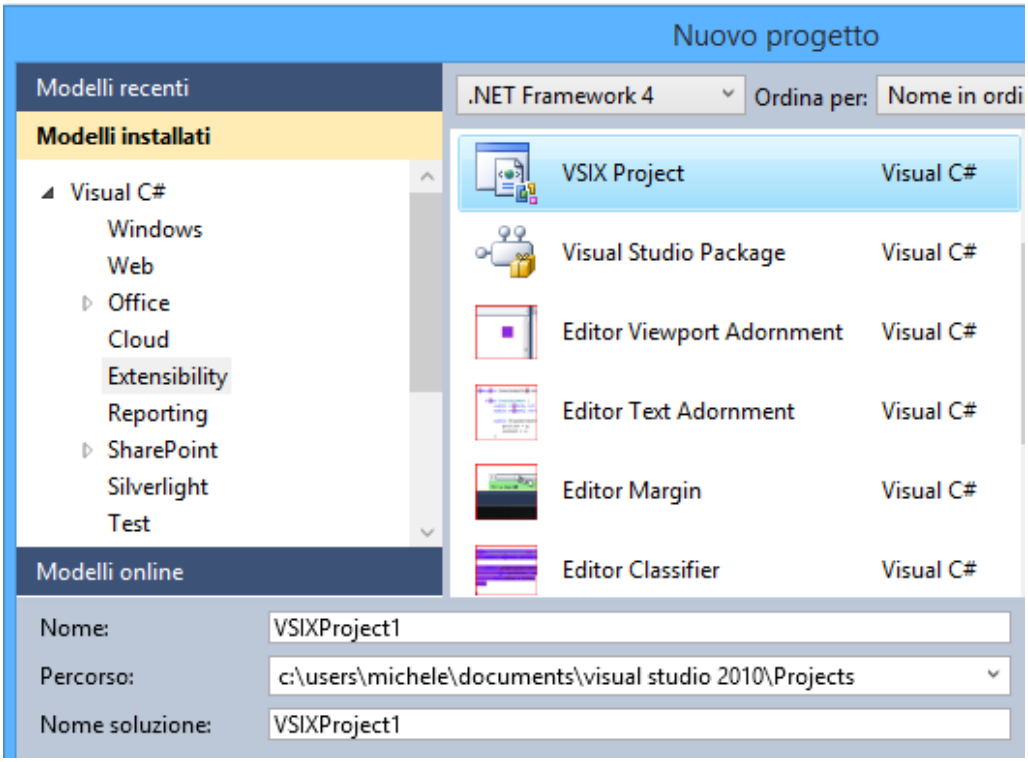

Figura 1.1: reazione di una nuova estensione come VSIX Project

1.3.2 Analisi del Manifest File

Al termine della elaborazione Visual Studio genera un nuovo progetto contenente il solo file "`source.extension.vsixmanifest`", chiamato "manifest file" dell'estensione, internamente strutturato come un file XML e contenente le informazioni di seguito riportate, le quali sono facilmente individuabili nella Tabella 1.2:

- Nome della estensione;
- Stringa identificativa;
- Autore;
- Versione;
- Descrizione;
- Versioni supportate di Visual Studio;
- Versioni supportate del Framework .NET.

Oltre a queste informazioni è presente un prologo con la specifica dello schema xml da utilizzare per interpretare correttamente tutti i parametri.

```
<?xml version="1.0" encoding="utf-8"?>
<Vsix
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  Version="1.0.0"
  xmlns="http://schemas.microsoft.com/developer/vsx-schema/2010"
>
  <Identifier Id="VSIXProject1..ebbfa21c-54e4-484c-a305-bf6b3cfaece5">
    <Name>VSIXProject1</Name>
    <Author>Michele Di Lella</Author>
    <Version>1.0</Version>
    <Description xml:space="preserve">Empty VSIX Project.</Description>
    <Locale>1033</Locale>
    <SupportedProducts>
      <VisualStudio Version="10.0">
        <Edition>Ultimate</Edition>
        <Edition>Pro</Edition>
      </VisualStudio>
    </SupportedProducts>
    <SupportedFrameworkRuntimeEdition MinVersion="4.0" MaxVersion="4.0"
/>
  </Identifier>
  <References />
  <Content />
</Vsix>
```

Tabella 1.2: Manifest File di una Estensione

1.4 Template

Il template rappresenta il modello di creazione di un nuovo file in funzione della sua tipologia. All'interno del template trovano posto il contenuto predefinito del file e un insieme di proprietà accessorie che permettono all'ambiente di sviluppo di gestire al meglio i file creati usando il template.

Nel caso specifico dell'IDE Visual Studio questi modelli di file prendono il nome di Template Item.

1.4.1 Creare un nuovo Template Item

Per realizzare un nuovo Template Item è sufficiente creare un nuovo progetto di tipo "Item Template". Al termine della procedura guidata Visual Studio genera all'interno del progetto tre file:

- Modello;
- Icona da associare;
- Manifest.

All'interno del modello si trova il contenuto predefinito di ogni file creato usando quel particolare template, a titolo di esempio una nuova classe C# viene creata seguendo il seguente modello:

```
using System;
using System.Collections.Generic;
$if$ ($targetframeworkversion$ >= 3.5)
using System.Linq;
$endif
$using System.Text;

namespace $rootnamespace$
{
      class $safeitemrootname$
      {
      }
}
```

Tabella 1.3: Modello di una Classe in linguaggio C#

Il manifest è un file XML che contiene tutte le informazioni necessarie a registrare il nuovo template nella galleria di Visual Studio, cioè nella collezione di tutti i possibili template selezionabili all'atto della creazione di un nuovo Item. Esso è strutturato in due parti: *TemplateData* e *TemplateContent*. Dalla Tabella 1.4 si evince che nella prima sezione vi trovano posto:

- Nome, associato al Template Item;
- Identificativo alfanumerico univoco, con il quale è possibile identificare il Template Item;
- Descrizione, con la quale è possibile aggiungere ulteriori informazioni relative al template;
- Icona, associata al Template Item nel menu di Visual Studio;
- Nome di Default del nuovo File.

Nella seconda sezione, cioè TemplateContent, viene riportata la definizione vera e propria del nuovo file da aggiungere, che contiene il nome del file e il CustomTool da associare.

```
<?xml version="1.0" encoding="utf-8"?>
<VSTemplate Version="3.0.0" Type="Item"
   xmlns="http://schemas.microsoft.com/developer/vstemplate/2005">
  <TemplateData>
    <Name>ItemTemplate1</Name>
    <Description>&lt;No description available&gt;</Description>
    <Icon>ItemTemplate1.ico</Icon>
    <TemplateID>c0881b82-4447-4b03-8a68-5c92e505d5a8</TemplateID>
    <ProjectType>CSharp</ProjectType>
    <RequiredFrameworkVersion>2.0</RequiredFrameworkVersion>
    <NumberOfParentCategoriesToRollUp>
      1</NumberOfParentCategoriesToRollUp>
    <DefaultName>Class.cs</DefaultName>
  </TemplateData>
  <TemplateContent>
    <References>
      <Reference>
         <Assembly>System</Assembly>
      </Reference>
    </References>
    <ProjectItem ReplaceParameters="true" CustomTool="" >
    Class.cs</ProjectItem>
  </TemplateContent>
</VSTemplate>
```

Tabella 1.4: Manifest File di un Template Item

Capitolo 2
Generatori di codice in Visual Studio

Un generatore di codice è un componente software realizzato con lo scopo di generare codice sorgente a partire da alcune informazioni chiave fornite dallo sviluppatore. In Visual Studio, un componente di questo genere è realizzabile mediante il meccanismo delle estensioni.

In questo capitolo saranno inizialmente analizzate alcune caratteristiche generali dei generatori di codice, come le tipologie di generazione, per poi concentrarsi su aspetti caratteristici dell'implementazione in Visual Studio, come le interfacce principali di supporto e la registrazione del COM relativo al generatore. Tale analisi riguarderà la versione 2010 di Visual Studio. In seguito, nel capitolo 3, saranno illustrate le novità apportate in Visual Studio 2013.

2.1 Tipologie

Esistono diverse tipologie di generatori di codice tra cui:

- Generatori di tipo Text Templating, basati cioè su normali file di testo contenenti il modello della classe e il cui funzionamento si basa sulla sostituzione di place holder presenti nel modello con i valori finali;
- Generatori basati sul diagramma delle classi: negli IDE avanzati è possibile trasformare un diagramma UML realizzato tramite tool grafici in una classe;
- Generatori basati su xml, in cui la struttura della classe è specificata secondo uno schema xml, che poi viene tradotto nella classe vera e propria.

2.2 Generatori di tipo Text Templating

Al contrario dei generatori basati su diagrammi UML o su XML, i quali tendono a specificare solo le proprietà della classe che potrà poi quindi essere implementata in linguaggi diversi, il meccanismo dei generatori di tipo Text Templating è language oriented, nel senso che è valido solo per un linguaggio, in quanto esplicita la struttura della classe seguendo le regole sintattiche del linguaggio scelto.

Visual Studio supporta tutte e tre queste tipologie di generatori. In questa tesi verrà però analizzato il meccanismo del Text Templating in quanto adottato nel caso di studio illustrato nel capitolo 4.7.

In tabella 2.1 è stato riportato un semplice esempio in cui è illustrato come implementare una delle più comuni funzioni dei più diffusi ambienti di sviluppo: la creazione di una nuova classe con costruttore vuoto.

Tutte le informazioni personalizzabili della classe sono presenti nel template sotto forma di place holder, cioè stringhe segnaposto caratterizzate, in questo caso, dal carattere iniziale `%`. Dopo che saranno stati inseriti i valori desiderati, ad esempio attraverso una procedura guidata, i place holder saranno sostituiti con il valore corrispondente.

Template	Generated Code

`%accessType class %className{`
`  Public %className(){`
`  }`
`}`
	`Public class Person{`
	`  Public Person(){`
	`  }`
	`}`

Tabella 2.1: Esempio di generazione di codice in linguaggio C#

2.3 Implementazione in Visual Studio 2010

Nei paragrafi che seguono, saranno illustrate le principali caratteristiche riguardanti l'implementazione di un generatore di codice come estensione di Visual Studio 2010.

In quanto estensione di Visual Studio, ogni generatore di codice sarà inoltre provvisto di un manifest file, caratteristica già trattata in precedenza nel paragrafo 1.3.2. Nello specifico, in Tabella 1.2 è mostrato un esempio completo di manifest file di un'estensione relativa a Visual Studio 2010, in cui è possibile notare la versione del manifest file: `Version="1.0.0"`.

2.3.1 Interfacce di supporto

Per la realizzazione del componente software "generatore di codice" della tipologia Text Templating, Visual Studio mette a disposizione dello sviluppatore alcune interfacce di supporto. Le più rilevanti tra queste sono le interfacce IVsSingleFileGenerator e IObjectWithSite.

2.3.1.1 Interfaccia IVsSingleFileGenerator

Questa interfaccia permette di ricevere in input un singolo file, portare a termine le dovute elaborazioni e quindi generare un singolo file in output. Tale file potrà poi essere aggiunto automaticamente al progetto. Qualsiasi oggetto COM che implementi questa interfaccia è detto *custom tool*.

L'interfaccia IVsSingleFileGenerator è adatta all'implementazione del meccanismo del Text Templating, in quanto il file di input citato in precedenza può essere inteso come il file di template contenente i vari place holder da sostituire, mentre il file di output diventa il file generato sostituendo i place holder nel template secondo le direttive dettate dallo sviluppatore.

L'interfaccia dichiara due metodi: `DefaultExtension` e `Generate`. Il metodo `DefaultExtension` serve a Visual Studio per ottenere l'estensione del file di output. Il metodo `Generate` trasforma il file di input secondo quanto specificato dallo sviluppatore, generando il file di output. Questo metodo è invocato da Visual Studio ogni qualvolta il custom tool associato viene caricato, o nel momento in cui il file di input viene salvato all'interno del progetto. È inoltre possibile forzare l'invocazione del metodo eseguendo il comando "RunCustomTool" su di un elemento del progetto.

2.3.1.2 Interfaccia IObjectWithSite

L'interfaccia IObjectWithSite permette al custom tool l'accesso a varie informazioni riguardanti l'ambiente in cui sta eseguendo. In particolare, permette di ottenere informazioni inerenti al progetto in cui è contenuto il custom tool, mediante il metodo `GetProject`. Questo metodo restituisce, difatti, un riferimento al progetto contenitore, grazie al quale è possibile, ad esempio, accedere a informazioni quali il nome, il percorso o il tipo del progetto.

Informazioni più dettagliate sulle interfacce IVsSingleFileGenerator e IObjectWithSite e sui relativi metodi possono essere trovate nell'esempio completo illustrato nel capitolo 4.7, e in particolare nel paragrafo 4.7.3.2.

2.3.2 Registrazione del COM

Una delle operazioni fondamentali da compiere affinché un custom tool (e quindi un generatore di codice) funzioni a dovere è la registrazione dell'oggetto COM nel registro di sistema di Windows. Quest'operazione è necessaria perché permette a Visual Studio di individuare correttamente il componente per poterlo poi mettere in esecuzione quando serve.

In Visual Studio 2010, la registrazione dell'oggetto COM di un generatore di codice si effettua attraverso la marcatura della classe che implementa il custom tool relativo mediante degli appositi attributi, che servono a definire le seguenti informazioni:

- Visibilità del COM, tramite attributo `ComVisibleAttribute`, da impostare al valore `true`;
- Guid, tramite attributo `GuidAttribute`, da impostare con l'opportuno codice `GUID`;
- Registrazione del COM per un determinato tipo di progetto, tramite attributo `CodeGeneratorRegistrationAttribute`;
- Indicazione del tipo di oggetto che fornisce le funzionalità, tramite attributo `ProvideObjectAttribute`, da impostare con il tipo della classe che implementa il custom tool.

2.3.2.1 CodeGeneratorRegistrationAttribute

Il CodeGeneratorRegistrationAttribute è uno degli attributi principali in gioco nella registrazione di un generatore di codice in Visual Studio. Esso permette di rendere il tool accessibile nei progetti del tipo specificato nella creazione dell'attributo.

Il suo costruttore presenta tre parametri d'ingresso:

- `generatorType`, di tipo `Type`: rappresenta il tipo della classe che implementa il custom tool del generatore di codice;

- `generatorName`, di tipo `string`: semplice stringa che rappresenta il nome che verrà associato al generatore all'interno del progetto in cui esso sarà utilizzato;
- `contextGuid`, di tipo `string`: stringa che rappresenta il codice GUID del tipo di progetto relativo alla registrazione.

```
public CodeGeneratorRegistrationAttribute(
       Type generatorType,
       string generatorName,
       string contextGuid
)
```

L'attributo CodeGeneratorRegistrationAttribute può essere utilizzato più di una volta nell'ambito della registrazione dello stesso generatore di codice, in modo da renderlo visibile nei vari tipi di progetto di Visual Studio secondo i requisiti imposti.

Al momento dell'installazione del generatore di codice cui è associato l'attributo, questo fa in modo che nel registro di sistema della macchina su cui si sta installando il custom tool venga inserita una nuova chiave di registro contenente le informazioni passate al suo costruttore. Nella Tabella 2.2 è possibile vedere la struttura della chiave di registro creata. Si noti inoltre il valore "`generatorGuid`", ottenuto dall'attributo `GuidAttribute` citato nel paragrafo precedente.

```
[$RootKey$\Generators\{contextGuid}\generatorType.Name]
@="generatorName"
"CLSID"="{generatorGuid}"
"GeneratesDesignTimeSource"=dword:00000001
```

Tabella 2.2: Chiave di registro generata da CodeGeneratorRegistrationAttribute

2.3.2.2 ProvideObjectAttribute

Un altro attributo molto importante per la registrazione del COM del custom tool è ProvideObjectAttribute. Quest'attributo permette di indicare l'oggetto COM che potrà essere poi caricato da Visual Studio come custom tool.

Il suo costruttore presenta un solo parametro d'ingresso: objectType, di tipo Type, indicante il tipo della classe che implementa il custom tool del generatore di codice.

```
public ProvideObjectAttribute(
       Type objectType
)
```

Come il CodeGeneratorRegistrationAttribute, anche il ProvideObjectAttribute, al momento dell'installazione del generatore di codice, crea una chiave nel registro di sistema della macchina su cui si sta eseguendo l'installazione, utilizzando le informazioni passate in precedenza al costruttore. La chiave di registro creata è strutturata come mostrato nella Tabella 2.3. Come nel caso precedente, anche qui il valore "`generatorGuid`" è ottenuto dall'attributo GuidAttribute.

```
[$RootKey$\CLSID\{generatorGuid}]
@="objectType.FullName"
"InprocServer32"="$WinDir$\SYSTEM32\MSCOREE.DLL"
"Class"="objectType.FullName "
"Assembly"="objectType.Name, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null"
"ThreadingModel"="Both"
```

Tabella 2.3: Chiave di registro generata da ProvideObjectAttribute

Un esempio completo di registrazione dell'oggetto COM di un generatore di codice in Visual Studio 2010 può essere trovato nella Tabella 4.15.

Capitolo 3
Porting di un generatore di codice da VS 2010 verso VS 2013

In questo capitolo saranno analizzate alcune delle novità presenti in Visual Studio 2013 rispetto alla versione del 2010 riguardanti il funzionamento dei custom tool. In seguito, alla luce dell'analisi di cui sopra, saranno illustrate alcune strategie per compiere il porting verso Visual Studio 2013 di un generatore di codice originariamente progettato per Visual Studio 2010.

In seguito, nel capitolo 5, verrà scelta e messa in pratica una delle strategie descritte alla fine di questo capitolo per compiere il porting verso Visual Studio 2013 del generatore di codice descritto nel capitolo 4.7.

3.1 Novità introdotte in Visual Studio 2013

Nella nuova versione del suo IDE, Microsoft ha apportato alcune modifiche riguardanti la gestione dei custom tool, e di conseguenza anche dei generatori di codice.

Gli aspetti principali che sono stati alterati da queste modifiche sono il manifest file delle estensioni, che è passato alla versione 2, e la gestione della reperibilità dell'oggetto COM del generatore di codice. Per quanto riguarda il meccanismo di generazione vera e propria del codice, questo non ha subito ripercussioni dalle modifiche apportate da Microsoft.

Di seguito verranno perciò illustrate, nello specifico, le novità riguardanti il manifest file e il reperimento dell'oggetto COM.

3.1.1 Manifest File v2

Con Visual Studio 2013, Microsoft ha introdotto la versione 2 del manifest file per le estensioni di Visual Studio.

Nella Tabella 3.1 è mostrata la struttura del nuovo manifest file.

```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest Version="2.0.0"
xmlns="http://schemas.microsoft.com/developer/vsx-schema/2011"
xmlns:d="http://schemas.microsoft.com/developer/vsx-schema-design/2011">
  <Metadata>
    <Identity Id="GeneratorId" Version="1.0" Language="en-US"
     Publisher="Michele Di Lella" />
    <DisplayName>Generator Name</DisplayName>
    <Description xml:space="preserve">GeneratorDescription</Description>
    <Icon>VSIXProject_icon.png</Icon>
    <PreviewImage>VSIXProject_previewImage.png</PreviewImage>
  </Metadata>
  <Installation>
    <InstallationTarget Id="Microsoft.VisualStudio.Pro"
                        Version="[12.0,]" />
    </Installation>
  <Dependencies>
    <Dependency Id="Microsoft.Framework.NDP"
                DisplayName="Microsoft .NET Framework" d:Source="Manual"
                Version="[4.0,)" />
  </Dependencies>
  <Assets>
  </Assets>
</PackageManifest>
```

Tabella 3.1: Manifest File di un'estensione, versione 2

Rispetto al manifest file versione 1 (Tabella 1.2), oltre la variazione poco rilevante dei nomi di alcune proprietà XML, come `Identity` o `DisplayName`, la più grossa e importante novità può essere notata nella sezione `Installation`. Sostanzialmente, Microsoft ha deciso di variare la gestione delle versioni di Visual Studio compatibili con la specifica estensione. Infatti, se fino a Visual Studio 2010, per rendere installabile un'estensione sulle versioni volute di Visual Studio era necessario elencarle tutte, una per una, adesso è possibile indicare direttamente il range di versioni compatibili, attraverso un meccanismo di pattern matching che fa uso di parentesi tonde e

quadre e di virgole. Nel particolare esempio della Tabella 3.1, è indicato come range di versioni compatibili l'insieme che comprende Visual Studio 12 (cioè Visual Studio 2013) e tutte le versioni successive a questo. Lo stesso meccanismo è utilizzato anche per indicare la versione minima e massima supportata del framework .NET.

Bisogna tener presente che queste modifiche nel manifest file hanno avuto impatto solo sul processo di installazione del generatore di codice, e non sul suo effettivo funzionamento. Nel paragrafo successivo, sarà invece mostrato come il corretto funzionamento del generatore di codice è stato minato da una variazione del meccanismo con cui Visual Studio accede ad un determinato custom tool.

3.1.2 Reperibilità dell'oggetto COM

Come accennato nel paragrafo precedente, se le novità apportate al manifest file non hanno influito sul corretto funzionamento del generatore di codice, lo stesso non si può dire delle modifiche riguardanti il meccanismo di accesso ai custom tool in Visual Studio. Infatti, dopo aver portato a termine alcune prove sperimentali su Visual Studio 2013 adottando le stesse tecniche di registrazione COM utilizzate nella versione del 2010 e descritte nel paragrafo 2.3.2, si è osservato un malfunzionamento nel sistema: Visual Studio 2013 non era più in grado di rintracciare il generatore di codice fra i custom tool installati.

Alla luce di questi risultati, è stata portata avanti un'attività di ricerca relativa al processo di registrazione COM dei custom tool in Visual Studio 2013 atta a evidenziare la presenza di modifiche sostanziali nel meccanismo con cui Visual Studio 2013 riesce ad accedere ai custom tool installati nel sistema.

Tale attività di ricerca ha portato alla conclusione che Visual Studio 2013 necessita della creazione di una nuova chiave di registro di sistema che gli permetta di individuare l'oggetto COM del custom tool all'interno dell'assembly in cui è definito. Per creare questa nuova chiave di registro di sistema, è stato introdotto un nuovo attributo, ProvideBindingPathAttribute, che sarà descritto nel paragrafo che segue.

3.1.2.1 ProvideBindingPathAttribute

L'attributo ProvideBindingPathAttribute, come accennato nel paragrafo precedente, è stato introdotto per gestire la creazione di una nuova chiave di registro di sistema che permetta a Visual Studio 2013 di individuare un custom tool all'interno dell'assembly in cui è definito.

Nello specifico, la nuova chiave di registro è strutturata come in Tabella 3.2, in cui `generatorGuid` rappresenta il codice GUID del generatore di codice e `packageFolder` rappresenta la directory in cui è installato il package del generatore di codice.

```
[$RootKey$\BindingPaths\{generatorGuid}]
"$PackageFolder$"="packageFolder"
```

Tabella 3.2: Chiave di registro generata da ProvideBindingPathAttribute

La directory d'installazione del package viene inoltre aggiunta alla "Visual Studio Probing List", che viene utilizzata da Visual Studio per risolvere i riferimenti agli assembly.

Alla luce delle informazioni ottenute dall'analisi descritta nel capitolo, nei paragrafi successivi, saranno illustrate alcune strategie per compiere il porting verso Visual Studio 2013 di un generatore di codice inizialmente progettato per Visual Studio 2010.

3.2 Strategie di porting

In questo e nei successivi paragrafi saranno descritte le possibili strategie per rendere compatibile con Visual Studio 2013 un generatore di codice originariamente progettato per Visual Studio 2010. Il motivo della presenza di più di una strategia è da attribuire alle novità presenti nel manifest file versione 2, descritte nel paragrafo 3.1.1. Tali modifiche hanno reso il nuovo manifest file incompatibile con Visual Studio 2010, e perciò il relativo generatore di codice non è installabile su questa versione di Visual Studio. Per questo motivo, sarà necessario decidere come comportarsi per non perdere comunque la compatibilità del generatore di codice anche con Visual Studio 2010.

Le strategie possibili, descritte in dettaglio nei paragrafi 3.2.1 e 3.2.2, saranno due:

- strategia con unica versione del generatore, che sfrutta il manifest file versione 2;
- strategia con versioni del generatore differenziate tra Visual Studio 2010 e 2013, che sfrutta invece entrambe le versioni del manifest file.

Per quel che riguarda, invece, la corretta funzionalità del generatore di codice una volta installato, le considerazioni fatte nel paragrafo 3.1.2, relative alla registrazione dell'oggetto COM del generatore di codice, sono da intendersi necessarie in ogni caso, a prescindere dalla strategia di porting scelta. Senza di esse, infatti, anche se rimarrebbe possibile installare il generatore di codice su Visual Studio 2013, esso risulterebbe comunque inutilizzabile, perché questa versione di Visual Studio non riuscirebbe a reperire il generatore fra i custom tool installati.

3.2.1 Unica versione del generatore

Questa strategia si basa sulla possibilità di implementare il generatore di codice in Visual Studio 2013 utilizzando, però, un manifest file versione 1.

Tale strategia è attuabile perché, mentre Visual Studio 2010 non è compatibile con il manifest file versione 2, è invece vero che Visual Studio 2013 mantiene la retro compatibilità con il manifest file versione 1. Il vantaggio di questa strategia è che il generatore di codice resta installabile anche su Visual Studio 2010 senza la necessità di mantenere due versioni differenti per Visual Studio 2010 e 2013. Lo svantaggio è che, a causa della struttura del manifest file versione 1 (Tabella 1.2), ogni versione di Visual Studio che si vuole rendere compatibile dovrà essere elencata nel manifest file e, soprattutto, ogni qualvolta si vorrà rendere il generatore di codice compatibile con una nuova versione di Visual Studio sarà necessario modificare il manifest file per includere la nuova versione, e quindi ricompilare l'intero progetto.

3.2.2 Versioni differenziate per VS 2010 e 2013

Questa strategia, contrapposta alla precedente, propone di mantenere la retro compatibilità con Visual Studio 2010 semplicemente mantenendo la versione originale del generatore di codice, per sua natura compatibile con questa versione di Visual Studio.

In questo modo, la nuova versione del generatore di codice verrà normalmente implementata in Visual Studio 2013 utilizzando il manifest file versione 2. Quest'ultimo sarà opportunamente modellato in modo da rendere il generatore di codice compatibile, oltre che con Visual Studio 2013, anche con versioni future. Per fare ciò, basterà indicare l'apposito range di versioni di Visual Studio compatibili, come descritto nel paragrafo 3.1.1 (Tabella 3.1).

L'unico svantaggio di questo approccio è il fatto di dover mantenere due versioni diverse del generatore di codice per conservare la retro compatibilità con Visual Studio 2010. Questa strategia, però, presenta l'importante vantaggio di mettere a disposizione una versione del generatore di codice che, pur non essendo compatibile con Visual Studio 2010, rimane compatibile con Visual Studio 2013 e con le versioni successive senza alcuna necessità di modifiche e di ricompilazioni.

Capitolo 4
L'interprete tuProlog

4.1 Caratteristiche

tuProlog[1] è un interprete Prolog interamente scritto in Java, minimale, dinamicamente configurabile ed estensibile tramite librerie. Esso è progettato in modo da supportare pienamente la programmazione multi-paradigma tramite una piena integrazione fra il paradigma logico dato dal linguaggio Prolog e il paradigma a oggetti, rappresentato primariamente da Java, ma anche nella versione .NET dai linguaggi disponibili su tale piattaforma.

tuProlog è distribuito in un unico archivio di ridotte dimensioni e ha come unico prerequisito la presenza della Java Virtual Machine (JVM) o del corrispondente strato Common Language Runtime[7].

tuProlog è stato sviluppato con un meccanismo di estensione basato sul caricamento statico o dinamico di librerie, così da poter facilmente aggiungere funzionalità non presenti nel motore. In particolare, grazie alla libreria JavaLibrary è stato possibile integrare il tuProlog con Java. La libreria offre un'integrazione bidirezionale che permette di utilizzare entità Java all'interno di tuProlog il quale le gestisce come normali termini. In questo modo è possibile estendere le funzionalità di tuProlog accedendo a tutte quelle di Java. Allo stesso modo è possibile rappresentare, lato Java, il motore tuProlog come un semplice oggetto che sarà utilizzato secondo il paradigma object oriented, cioè tramite l'invocazione di metodi come ad esempio la `setTheory` per caricare una base di conoscenza all'interno del motore, oppure la `solve` per la risoluzione di un goal.

4.2 Interoperabilità con linguaggi imperativi

Prolog non è stato pensato per essere utilizzato nella realizzazione di applicazioni complete di interfaccia utente, connessioni di rete o gestione di basi di dati. Per poter sfruttare le potenzialità dei linguaggi di logica molte implementazioni di Prolog mettono a disposizione alcuni meccanismi che gli permettono di dialogare con altri linguaggi di programmazione, dando vita ad un ambiente di sviluppo completo. Anche tuProlog è stato esteso per offrire questo tipo di interoperabilità, in particolare con Java e con la piattaforma .NET.

La prima, e più naturale integrazione che è stata sviluppata, è quella con il linguaggio Java. Grazie a questa integrazione è possibile, ad esempio, creare applicazioni con interfaccia grafica swing, connesse in rete o che utilizzino database come fonte di dati, o avere accesso a qualunque classe o API di Java. Tutti i dati acceduti attraverso Java potranno inoltre essere elaborati da uno o più engine tuProlog.

Un diverso approccio di integrazione, atto ad accrescere le funzionalità di Java tramite tuProlog, è il framework P@J. Quest'ultimo permette di specificare codice tuProlog come implementazione di metodi Java, attraverso il meccanismo delle Java Annotation. Ciò significa che, ogni qualvolta un metodo Java annotato verrà chiamato, il sistema eseguirà il codice tuProlog specificato nella annotazione.

In Tabella 4.1 è riportato un esempio completo di utilizzo del framework P@J tratto dal manuale di tuProlog[1]. In questo caso il metodo astratto `Perm.permutation` è stato decorato con l'annotazione `@PrologMethod`. L'effetto di questa annotazione è che a runtime verrà creata una implementazione del metodo `permutation` capace di risolvere il problema delle permutazioni usando la teoria espressa nella annotazione.

```
//A Java class augmented via Prolog
abstract class Perm{
  @PrologMethod ( clauses = {
    "permutation([],[])." ,
    "permutation(U,[X|V]):-remove(U,X,Z),permutation(Z,V)." ,
    "remove([X|T],X,T)." ,
    "remove([X|U],E,[X|V]):-remove(U,E,V)."
    }
  )
  public abstract < $X extends List<Int>, $Y extends List<Int> >
  Iterable<$Y> permutation($X list);
}

//A sample client class
public class PJexample {
  public static void main(String[] args) throws Exception {
    java.util.Collection<Integer> v = java.util.Arrays.asList(1,2,3);
    Perm p=PJ.newInstance(Perm.class);
    for (List<Int> list : p.permutation(new List<Int>(v))) {
      System.out.println(list.toJava());
    }
  }
}

// Output printed:
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Tabella 4.1: Esempio completo di utilizzo del framework P@J

L'interoperabilità con la piattaforma .NET non è implementata in modo diretto, cioè attraverso lo sviluppo di librerie ad hoc, come la JavaLibrary, o con strumenti come P@J, ma passa per un processo di traduzione automatica del codice di tuProlog in .NET. Questa scelta è dovuta alla necessità di mantenere allineate le versioni per Java e per .NET senza introdurre l'overhead relativo ad una riscrittura manuale del codice.

4.3 tuProlog.NET

Lo sviluppo degli strumenti per la programmazione multi paradigma tra tuProlog e Java è stato facilitato dall'origine stessa del tuProlog, cioè di un interprete Prolog scritto in Java. Ciò significa che in fondo il tuProlog è codice Java e quindi affinché l'interazione tra i due linguaggi avvenga con successo è necessario affrontare e risolvere i problemi di accesso relativi alle funzionalità e alle risorse, tenendo presente che una volta risolti le risorse saranno direttamente utilizzabili.

Il discorso è molto diverso per ciò che riguarda l'interazione tra tuProlog e i linguaggi del framework .NET. In questo caso anche la più elementare delle strutture dati, o la definizione delle classi avviene secondo approcci diversi e il codice intermedio generato non è compatibile tra i due diversi linguaggi in quanto: nel caso del tuProlog viene generato il bytecode Java, mentre nel caso del .NET viene generato il bytecode CIL.

A seguito di queste profonde diversità sono state portate avanti due differenti versioni del progetto tuProlog: quella di riferimento sviluppata in Java e quella alternativa/secondaria, conosciuta come tuProlog.NET, basata sulla traduzione del codice Java in .NET in modo automatico, appoggiandosi al software IKVM.NET[16].

4.4 IKVM

IKVM è una implementazione di Java per il framework .NET. Può essere scomposto nelle tre seguenti componenti base:

- Una Java Virtual Machine sviluppata in .NET;
- Una libreria Java, derivata dalla riscrittura in .NET della libreria OpenJDK;
- Alcuni strumenti aggiuntivi tra cui ikvmc per la traduzione del codice Java in .NET.

IKVM dispone di due modalità di traduzione del codice: statica e dinamica. La modalità statica prevede l'utilizzo del tool ikvmc che genera un assembly incapsulato in file dll o exe; ciò dipende dal fatto che ciò che si sta traducendo sia una libreria o una applicazione. La modalità dinamica, implementata con il tool ikvm, esegue una traduzione a runtime del bytecode Java in .NET che viene immediatamente eseguito.

Utilizzando IKVM, il progetto tuProlog scritto in Java, quando maturo/pronto per un nuovo rilascio, viene semplicemente tradotto in .NET in modo statico, rendendo immediatamente disponibile la nuova versione per entrambe le piattaforme.

4.5 Approcci alla programmazione multi-paradigma

L'interoperabilità tra tuProlog e .NET rispecchia i filoni già analizzati nel Capitolo 4.2. Infatti, si hanno i seguenti possibili scenari:

- Utilizzare .NET da tuProlog: OOLibrary;
- Utilizzare tuProlog da .NET: tuProlog API;
- Accrescere tuProlog via .NET: nuove librerie;
- Accrescere .NET via tuProlog: P@J via .NET

4.5.1 Usare .NET da tuProlog: OOLibrary

La tecnica di generazione automatica della versione .NET di tuProlog, ottenuta grazie ad IKVM, fa si che in questa versione sia disponibile anche la JavaLibrary, cioè la libreria che ci permette di accedere al mondo Java a partire dal codice tuProlog. La JavaLibrary utilizzata da tuProlog.NET permette di accedere agli oggetti nativi .NET ma purtroppo solo in modo parziale, perché, essendo progettata e realizzata appositamente per Java, non da accesso a tutte le peculiarità proprie di .NET, come ad esempio le proprietà di una classe.

Per ovviare a questi problemi, che di fatto limiterebbero le possibilità di utilizzo, è stata sviluppata la libreria OOLibrary che, estendendo la JavaLibrary, la rende pienamente compatibile con .NET.

Infine, per gestire le diverse convenzioni sui nomi dei metodi o delle proprietà nei diversi linguaggi, sono state introdotte le *Conventions*. È naturale aspettarsi che a questo punto sia possibile utilizzare i diversi linguaggi imperativi nel modo più simile/trasparente possibile. Nella Tabella 4.2 (tratta dal manuale di tuProlog[1]) si può vedere come cambia l'uso della OOLibrary con e senza le convention.

```
visualbasicWithoutConvention :-
  new_object('VBStudent.Student',[123456, john, smith], Obj),
  Obj <- 'PrintStudent' returns Student,
  Obj <- get_Id returns StudentNumber,
  class('VBStudent.Student,VBStudent') <- get_StaticProperty returns
  Value,
  new_object('VBStudent.Student, VBStudent[]',[10], Array).
```

```
visualbasicWithConvention :-
  load_convention('VBConvention.dll','VBConvention.VBDotNet',Conv),
  new_object(Conv,'VBStudent.Student, VBStudent', [123456, john,
  smith], Obj),
  Obj <- printStudent returns Student,
  Obj.id <- get(StudentNumber),
  class('VBStudent.Student, VBStudent').staticProperty <- get(Value),
  new_object(Conv, 'VBStudent.Student, VBStudent()',[10], Array).
```

Tabella 4.2: OOLibrary senza e con Convention

4.5.2 Usare tuProlog da .NET: tuProlog API

Come nel caso precedente, anche in questa occasione tuProlog.NET è prodotto della generazione automatica del codice tramite IKVM che, oltre al motore tuProlog e alla JavaLibrary, traduce anche le API che permettono di accedere al tuProlog da Java e che, a loro volta, dopo la traduzione, permettono di accedere da .NET al motore tuProlog. In tabella 4.3 (tratta dalla tesi di Fabio Gravina[17]) ne è riportato un esempio applicativo.

```
string th = @"
    any([X|Xs],X,Xs).
    any([X|Xs],E,[X|Ys]) :- any(Xs,E,Ys).
    perm([],[]).
    perm(Xs,[X|Ys]) :- any(Xs,X,Zs), perm(Zs,Ys).
";
Prolog engine = new Prolog();
Theory t = new Theory(th);
Term goal = new Struct("perm",Term.createTerm(InputTextBox.Text),new
Var("R"));
SolveInfo solveInfo = engine.solve(goal);
string result;
if (!solveInfo.isSuccess())
   result = "no";
while (solveInfo.isSuccess()){
   result = solveInfo.getTerm("R").toString() + "\t";
   if (prolog.hasOpenAlternatives())
     SI = prolog.solveNext();
   else
     break;
}
```

Tabella 4.3: Esempio di utilizzo di tuProlog da .NET

4.5.3 Accrescere tuProlog via .NET: nuove librerie

Lo sviluppo di nuove librerie per tuProlog scritte in .NET ha lo scopo di accrescere le potenzialità del linguaggio tuProlog mettendo a disposizione nuovi predicati utilizzabili nel codice tuProlog, i quali quando vengono richiamati dal motore inferenziale vanno in realtà ad eseguire metodi .NET. Questo approccio permette di usufruire dei vantaggi della programmazione imperativa in ambito tuProlog. Nella Tabella 4.4 (tratta dal manuale di

tuProlog[1]) è stata realizzata una libreria .NET che estende Library, con lo scopo di fornire al tuProlog due nuovi predicati: sum e println.

```
using alice.tuprolog;
public class TestLibrary: Library {
  // functor sum(A,B)
  public Term sum_2(Number arg0, Number arg1){
    float n0 = arg0.floatValue();
    float n1 = arg1.floatValue();
    return new Float(n0+n1);
  }
  // predicate println(Message)
  public boolean println_1(Term arg){
    Console.WriteLine(arg);
    return true;
  }
}
```

Tabella 4.4: Accrescere tuProlog via .NET

4.5.4 Accrescere .NET via tuProlog: P@J framework

Per ciò che concerne l'utilizzo del framework P@J dall'ambiente .NET è necessario differenziare due scenari:

- Le applicazioni Java che utilizzano P@J funzionano anche dopo essere state tradotte in .NET;
- Lo sviluppo di nuove applicazioni .NET, a causa di una eccezione nell'uso del tool Javaassist, non può essere portato a termine correttamente.

Il supporto al framework P@J in .NET sarà ulteriormente investigato nei prossimi paragrafi.

4.6 Approccio basato sull'utilizzo di P@J su .NET

Attualmente, il framework P@J fornisce solo una parte delle sue funzionalità quando utilizzato nell'ambiente .NET. Infatti,

- Le applicazioni Java che utilizzano P@J continuano a funzionare anche dopo esser state tradotte in .NET, mentre
- Le applicazioni .NET che tentino di utilizzare il framework falliscono. La ragione di fondo per questo comportamento risiede nel diverso modo con cui Java e .NET gestiscono (a compile time, ma soprattutto a run time) i tipi generici, pesantemente utilizzati dal framework P@J per il proprio funzionamento, in particolare riguardo alla type inference. Ciò rende impossibile per la .NET virtual machine di interoperare con le DLL tradotte da IKVM partendo da sorgente Java, in quanto le signature dei metodi non corrispondono a livello di bytecode.

Per questo motivo, verranno analizzate di seguito le differenze di implementazione dei *Generici* tra Java e .NET, per comprendere le cause ultime del problema e valutare le eventuali possibilità di soluzione.

Entrambe le piattaforme Java e .Net supportano i tipi generici, con le stesse finalità d'uso e con una sintassi molto simile, ma con notevoli differenze implementative poiché nel caso di Java sussisteva il vincolo di non modificare la macchina virtuale, definita in origine senza tenere conto di tale concetto.

Entrambe le piattaforme Java e .NET supportano i tipi generici, con le stesse finalità d'uso e con una sintassi molto simile, ma con notevoli differenze implementative poiché nel caso di Java sussisteva il vincolo di non modificare la macchina virtuale, definita in origine senza tenere conto di tale concetto.

4.6.1 Implementazione dei Tipi Generici in Java e .NET

In ambito Java si è scelto un approccio basato sul meccanismo della *Type Erasure*, che ha permesso di non modificare la macchina virtuale Java. La type erasure si applica a compile time e ciò significa che, quando il codice di classi o metodi generici viene compilato, i tipi generici vengono rimossi e sostituiti con il tipo generale Object.

L'implementazione dei tipi generici in .NET è passata per un approccio diverso. In questa piattaforma, i generici sono stati implementati a livello di intermediate language. Questo approccio, che quindi fornisce un supporto nativo ai generici, fa sì che questi restino reificati a runtime.

Nella Tabella 4.5 è possibile osservare le differenze nel codice sorgente "equivalente" al compilato tra l'approccio Java e quello .NET.

Sorgente	Sorgente “equivalente” al compilato

// Java
class GenericPrint
<T, U extends Number>{
private T x;
private U u;
public void print(T y){
System.out.print(y);
}
Public void demo(){
T k = new T (); // error
T[] ar = new T [0]; // error
Class c = T.class; // error
}
}
	class GenericPrint{
	private Object x;
	private Number u;
	public print(Object y){
	System.out.println(y);
	}
	}

// C#
class GenericClass<T> where T : new(){
private T x;
public void print(T y){
Console.Write(y);
}
Public void demo(){
T k = new T(); // OK with
// new() constraint.
T[] ar = new T [0]; // OK
Type type = typeof(T); // OK
}
}
	internal class GenericClass<T> where T :
	new(){
	private T x;
	public GenericClass(){}
	public void Print(T y){
	Console.Write(y);
	}
	public void demo(){
	T t;
	T t1 = default(T);
	if (t1 == null){
	t = Activator.CreateInstance<T>();
	}
	else{
	t1 = default(T);
	t = t1;
	}
	T k= t;
	T[] ar = new T[0];
	Type type = typeof(T);
	}
	}

Tabella 4.5: Implementazione Java e .NET dei generici

4.6.2 Considerazioni su P@J in .NET

Dopo aver analizzato le differenze implementative dei generici tra .NET e Java si può passare all'analisi dei due casi specifici inerenti P@J.

Le applicazioni scritte in Java che fanno uso di P@J continuano a funzionare anche dopo la traduzione del codice, perché in tal caso il meccanismo dei generici è utilizzato da due entità Java e quindi i generici sono gestiti allo stesso modo. Quando il bytecode java viene tradotto in .NET da IKVM i generici sono stati già correttamente gestiti e quindi non si ha nessun errore.

Differente è la situazione per le applicazioni scritte in .NET nelle quali il meccanismo dei generici viene usato cross-language; ovvero vengono richiamati in .NET e risolti in Java. In Tabella 4.6 (tratta dalla tesi di Fabio Gravina[17]) si può osservare come, se funzionasse, dovrebbe essere definito un PrologMethod P@J lato .NET e come lo si dovrebbe poter utilizzare.

```
using List = alice.tuprologx.pj.model.List;
public abstract class Perm{
  [PrologMethodAttribute(clauses = new string[]{
    "permutation([],[]).",
    "permutation(U,[X|V]):-remove(U,X,Z),permutation(Z,V).",
    "remove([X|T],X,T).",
    "remove([X|U],E,[X|V]):-remove(U,E,V)."
  })]
  public abstract Y permutation<X,Y>(<X> l)
    where X : List
    where Y : List;
}

public class Program{
  static void Main(string[] args){
    ...
    List list = new List(l);
    java.lang.Class c =
        java.lang.Class.forName("javaprolog.Perm, javaprolog");
    Perm perm = (Perm)PJ.newInstance(c);
    List retList = perm.permutation<List,List>(list);
    Console.WriteLine(retList.toJava());
  }
}
```

Tabella 4.6: Esempio di codice .NET che usa P@J

Tenendo in considerazione le differenze implementative dei generici in Java e .NET, ci si aspetta che il codice appena illustrato venga compilato correttamente ma che riporti degli errori a runtime, in quanto il meccanismo di Type Erasure fa sì che a runtime i tipi di ritorno non coincidano con quanto atteso da .NET.

In effetti, l'esecuzione del codice di esempio restituisce la seguente eccezione:

```
AbstractMethodError was unhandled

Method javaprolog.Perm.permutation is unsupported by
IKVM
```

L'eccezione, generata da IKVM, appare confermare l'ipotesi precedente. L'impossibilità di recuperare il metodo "annotato da Java" è dovuta non tanto a una limitazione di IKVM in sé, ma, appunto, alla diversa gestione dei tipi generici lato Java, che cancella a livello di bytecode informazioni cruciali per il successivo lookup del metodo in .NET.

Scemano così le possibilità di far funzionare il framework P@J rispettando uno degli obiettivi prefissati, cioè non modificare l'attuale implementazione java del framework: la sola possibilità teorica di proseguire per questa strada, infatti, implicherebbe di rivedere profondamente il funzionamento del framework P@J, riprogettando le parti relative alla gestione dei tipi. Considerata però la lunga attività di ricerca necessaria in passato per giungere all'attuale impostazione, tale via non appare percorribile nell'immediato.

Pertanto, nei paragrafi successivi verrà illustrato un caso di studio completo relativo alla creazione di un generatore di codice per Visual Studio che riguarderà l'interfacciamento tra P@J e .NET e che porrà rimedio ai problemi analizzati precedentemente, rispettando al tempo stesso il vincolo di non modificare l'attuale progetto (e codice) di P@J .

4.7 Generatore di codice per P@.NET

Nei paragrafi che seguono verrà illustrato il processo completo di creazione di un generatore di codice per Visual Studio. Tale generatore di codice venne in origine realizzato come lavoro di tesi da Fabio Gravina[17].

Nello specifico, il generatore di codice in questione è stato creato per definire una nuova metodologia di integrazione tra tuProlog e .NET, basata sulla generazione di codice sorgente .NET a partire da una teoria tuProlog, che ha permesso di superare i problemi relativi ai tipi generici analizzati nei precedenti paragrafi. È stato così possibile sfruttare le potenzialità di tuProlog utilizzando la classe .NET generata.

4.7.1 Requisiti del generatore di codice

1. Il generatore di codice deve poter funzionare a partire da sorgenti tuProlog standard, cioè senza che sia necessario modificarli per renderli compatibili con il generatore stesso;
2. Il generatore di codice deve consentire la personalizzazione del nome della classe generata e del namespace;
3. Il generatore di codice deve offrire il supporto al linking statico e dinamico del codice tuProlog. Deve quindi offrire le due seguenti modalità di funzionamento:
   - 3.1. Statica: il codice tuProlog è incapsulato all'interno del codice .NET all'atto della generazione della classe;
   - 3.2. Dinamica: il codice tuProlog viene caricato nella classe .NET a runtime consentendo successivamente la modifica del codice tuProlog senza la necessità di rigenerare la classe.

4.7.2 Analisi dei requisiti

Per soddisfare i requisiti appena descritti è necessario effettuare alcune scelte di progetto che saranno di seguito illustrate.

I linguaggi .NET ai quali si vuole fornire supporto implementativo sono C# e VisualBasic, in quanto i più diffusi di questo framework. Dato che i due linguaggi sono entrambi object oriented e differiscono solo per la notazione sintattica, nel resto del capitolo ci si riferirà solo a C#, fermo restando che, fatte le dovute analogie sintattiche, tutto ciò che sarà discusso resta valido anche per VisualBasic.

4.7.2.1 Modello della classe generata

Affinché sia possibile generare una classe C# fully qualified è necessario specificare nel file generato il namespace, cioè lo spazio dei nomi a cui afferirà la classe, e il nome della classe stessa. Per soddisfare questa necessità e al contempo il requisito 1 si è scelto di:

- Assegnare al namespace lo stesso nome del progetto .NET all'interno del quale si trova il file tuProlog in cui si troverà la classe generata;
- Assegnare alla classe lo stesso nome del file tuProlog, ovviamente sprovvisto di estensione.

Nella Tabella 4.7 (tratta dalla tesi di Fabio Gravina[17]) viene illustrato tramite un esempio quello che sarà il codice generato come comportamento di default del sistema.

Perm.pl	Perm.cs

any([X|Xs],X,Xs).
any([X|Xs],E,[X|Ys]) :-
any(Xs,E,Ys).
permutation([],[]).
permutation(Xs,[X|Ys]) :-
any(Xs,X,Zs),
permutation(Zs,Ys).
	using System;
	using System.Collections.Generic;
	using System.Linq;
	using System.Text;
	using alice.tuprolog;
	namespace PrologTestSolution {
	public class Perm : Prolog {
	private string th = @"
	any([X|Xs],X,Xs).
	any([X|Xs],E,[X|Ys]) :- any(Xs,E,Ys).
	perm([],[]).
	perm(Xs,[X|Ys]) :- any(Xs,X,Zs),
	perm(Zs,Ys).
	";
	public Perm(){
	setTheory(new Theory(th));
	}
	}
	}

Tabella 4.7: Generazione di una classe C# non personalizzata

4.7.2.2 Personalizzazione della classe generata

Il requisito 2 richiede di fornire allo sviluppatore la possibilità di personalizzare i due parametri precedenti. Si rende necessario annotare il codice tuProlog con meta-informazioni, che saranno usate dal generatore di codice:

```
%NameSpace nomeDelNameSpace
%ClassName nomeDellaClasse
```

le meta-info per il generatore devono ovviamente essere viste come commenti dal motore Prolog per non interferire con esso, quindi l'aggiunta di queste keyword non modifica in nessun modo la logica del codice tuProlog.

4.7.2.3 Modalità di generazione

Per dare una maggiore flessibilità al programmatore, il requisito 2 richiede due modalità di generazione del codice attraverso le quali è possibile scegliere se incorporare la teoria all'interno della classe, oppure se mantenere la teoria in un file esterno accessibile al software. Le due modalità prendono rispettivamente il nome di modalità Statica e Dinamica.

Il comportamento di default del generatore segue le regole della modalità statica. Per attivare la modalità dinamica è necessario annotare il sorgente tuProlog con la seguente meta-informazione:

```
%ExtFile nomeFileProlog.pl
```

4.7.2.3.1 Modalità statica

Questa modalità si basa sull'idea di portare dentro il codice .NET la teoria tuProlog in quanto questa è semplicemente un stringa e quindi facile da gestire. Il risultato di questo incapsulamento è un codice chiuso, cioè dà meno possibilità allo sviluppatore o utilizzatore di andare ad agire sulla teoria dopo il processo di generazione. Questa rigidità porta dei vantaggi in termini di sicurezza e robustezza del software, in quanto siamo sicuri che il comportamento della classe generata sarà sempre lo stesso. Di contro vengono limitate: le personalizzazioni post-generazione, che dovranno passare attraverso un processo di estensione della classe generata; le correzioni della teoria tuProlog, che richiederanno la ricompilazione dell'intero progetto. Il codice generato utilizzando questa modalità è lo stesso di quello visto in Tabella 4.7.

4.7.2.3.2 Modalità dinamica

Per ovviare ai limiti della modalità statica, accettando al contempo un compromesso sulla intrinseca sicurezza derivante da un codice chiuso e sulle prestazioni, è stata introdotta la modalità dinamica. Utilizzando questa modalità nel file tuProlog non specificheremo più il sorgente tuProlog ma bensì il file in cui andare a recuperarlo e il cui contenuto sarà caricato a runtime dal software. In questo modo sarà possibile modificare la teoria senza ricompilare

il software ma semplicemente aggiornando il file della teoria. Gli svantaggi di questa tecnica sono principalmente legati a due possibili situazioni: la cancellazione accidentale o volontaria del file tuProlog, che rende il software inutilizzabile; la volontaria manomissione dello stesso da parte di male intenzionati. Si veda a corredo l'esempio in Tabella 4.8 (tratta dalla tesi di Fabio Gravina[17]) inerente la risoluzione simbolica delle derivate, il cui codice tuProlog si suppone sia contenuto nel file Der.pl. Per il contenuto di questo file si veda l'esempio completo nel capitolo 5.3.

%ClassName PrologDerivate
%ExtFile Der.pl
	using System;
	using System.Collections.Generic;
	using System.Linq;
	using System.Text;
	using alice.tuprolog;
	using java.io;
	
	namespace PrologTestSolution {
	  public class PrologDerivate : Prolog {
	  public PrologDerivate(){
	  setTheory(
	  new Theory(
	  new FileInputStream("Der.pl")
	  )
	  );
	  }
	  }
	}

Tabella 4.8: Generazione di una classe C# in modalità dinamica

4.7.2.3.3 Confronto Prestazionale

Dal punto di vista prestazionale le differenze sono minime in quanto la differente modalità di caricamento della teoria influenza soltanto la creazione dell'oggetto. Una volta portato a termine il costruttore i due oggetti saranno equivalenti e quindi anche le prestazioni si equivarranno. Questa affermazione è vera a patto che non ci sia la necessità di creare molte istanze di oggetti della stessa classe, caso poco comune dato che tutti gli oggetti fornirebbero la medesima funzionalità.

4.7.3 Progetto del generatore di codice

Per realizzare il generatore di codice descritto nei requisiti si è scelto di realizzare il progetto sotto forma di estensione per Visual Studio piuttosto che come tool esterno.

Questa scelta è giustificata dal fatto che l'integrazione con Visual Studio rende tutto il processo di sviluppo dell'applicazione più lineare e nasconde all'utilizzatore la presenza del generatore di codice. Un ulteriore vantaggio di questa metodologia risiede nella possibilità di utilizzare un canale di comunicazione, messo a disposizione dal meccanismo delle estensioni, attraverso il quale il generatore di codice può inviare messaggi inerenti l'esito della generazione del codice a Visual Studio, che si occuperà di visualizzarli opportunamente.

4.7.3.1 Vincoli Progettuali

In riferimento a quelle che saranno le scelte progettuali e implementative è doveroso specificare che alcune di queste derivano direttamente da vincoli imposti dal meccanismo di gestione delle estensioni, dalla interfaccia IVsSingleFileGenerator e dalle classi astratte BaseCodeGenerator e BaseCodeGeneratorWithSite.

4.7.3.2 Diagramma delle classi

Nel diagramma delle classi sottostante (Figura 4.9, tratta dalla tesi di Fabio Gravina[17]) è possibile vedere come il PrologGenerator sia correlato al resto del sistema.

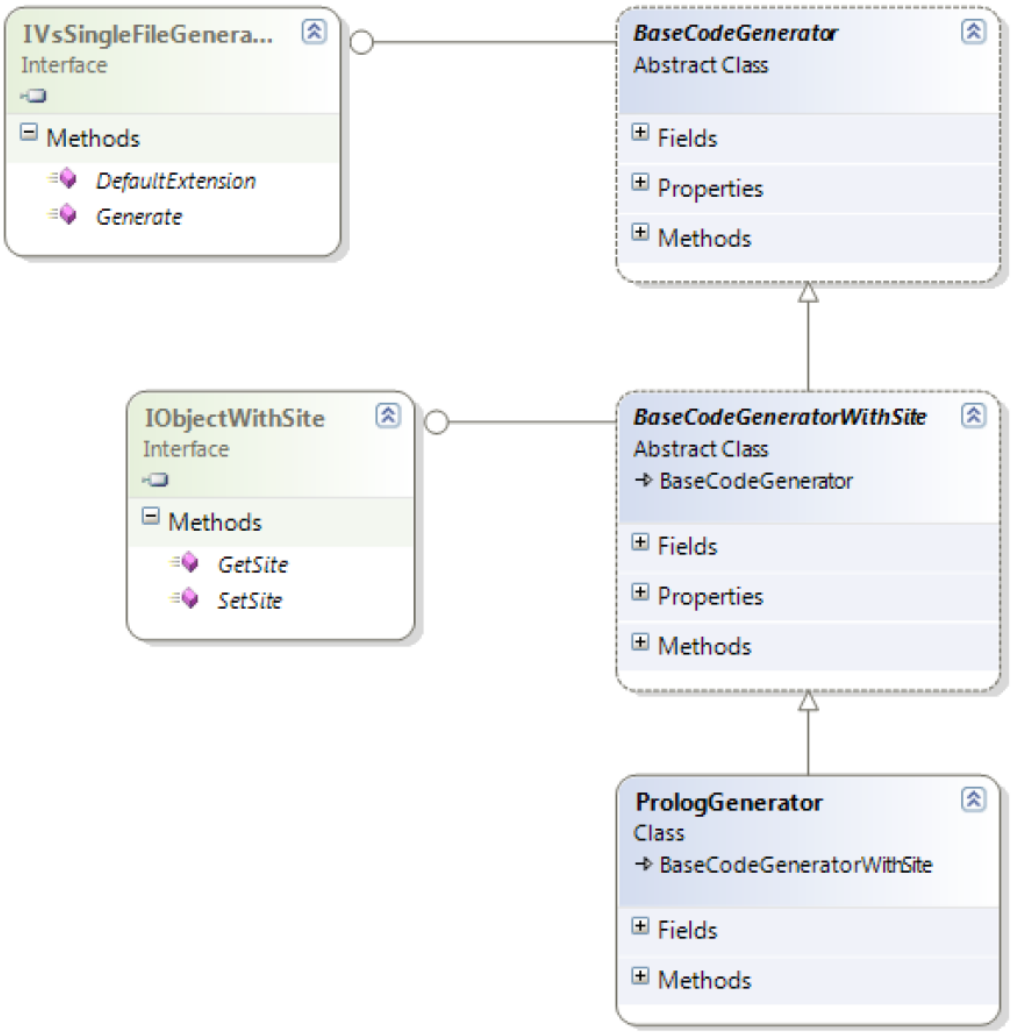

Figura 4.9: Diagramma delle Classi

4.7.3.2.1 IVsSingleFileGenerator

Questa interfaccia, appartenente al Visual Studio Development Kit, definisce i metodi che un generatore di file deve implementare.

Il metodo `DefaultExtension` restituisce, attraverso il parametro di out, l'estensione che dovrà essere assegnata al file generato (il nome del file sarà lo stesso di quello di partenza); il metodo stesso restituirà come valore di ritorno `VSConstant.S_OK` in caso di successo oppure `VSConstant.E_FAIL` in caso di errore.

```
int IVsSingleFileGenerator.DefaultExtension(out string
pbstrDefaultExtension)
```

Il metodo *Generate* accetta come parametri di input:

- `wszInputFilePath`: il percorso del file originale;
- `bstrInputFileContents`: il contenuto del file originale;
- `wszDefaultNameSpace`: il NameSpace in cui sarà salvato il file;
- `pGenerateProgress`: un riferimento alla interfaccia IVsGeneretorProgress per gestire l'avanzamento della generazione.

E restituisce:

- `rgbOutputFileContents`: il contenuto del file generato;
- `pcbOutput`: il numero di byte contenuti in `rgbOutputFileContents`;
- come valore di ritorno: `VSConstant.S_OK` in caso di successo oppure `VSConstant.E_FAIL` in caso di errore.

```
int IVsSingleFileGenerator.Generate(
       string wszInputFilePath, string bstrInputFileContents,
       string wszDefaultNamespace, IntPtr[] rgbOutputFileContents,
       out uint pcbOutput, IVsGeneratorProgress pGenerateProgress )
```

4.7.3.2.2 BaseCodeGenerator

Questa classe astratta implementa l'interfaccia IVsSingleFileGenerator e quindi i due metodi descritti nel paragrafo precedente, aggiunge inoltre i seguenti:

- metodi astratti:
    - `byte[] GenerateCode(string inputFileContent)`: metodo che quando implementato provvederà a generare il contenuto del file di destinazione;
    - `string GetDefaultExtension()`: metodo che quando implementato restituirà l'estensione del file di destinazione;

- metodi implementati:
    - `GenerateError(uint level, string message, uint line, uint column):` questo metodo permette di segnalare a VisualStudio che si è verificato un errore durante l'esecuzione, inviando: il livello di errore, il messaggio, la linea e la colonna in cui l'errore si è verificato;
    - `GenerateWarning(uint level, string message, uint line, uint column):` questo metodo permette di segnalare a VisualStudio la presenza di un warning generato durante l'esecuzione, inviando: il livello di warning, il messaggio, la linea e la colonna relativi al warning stesso;
- proprietà:
    - FileNameSpace, InputFilePath, HasError.

4.7.3.2.3 BaseCodeGeneratorWithSite

La classe astratta BaseCodeGeneratorWithSite estende la classe BaseCodeGenerator e ne implementa solo il metodo `GetDefaultExtension`. Implementa anche l'interfaccia IObjectWithSite, grazie alla quale sarà possibile accedere a molte informazioni inerenti il progetto VisualStudio all'interno del quale sta eseguendo il generatore. In particolare, con il metodo `GetProject` viene restituito un riferimento al progetto dal quale è possibile estrarre informazioni tra cui: il nome, il percorso e il tipo (VB o C#).

4.7.4 Implementazione di PrologVSExtension

L'estensione realizzata, progettata per Visual Studio 2010, è stata suddivisa in due parti: il PrologGenerator e il PrologCSharpItemTemplate. Entrambe queste porzioni di codice sono incapsulate secondo le metodologie appena viste in due progetti, i quali sono a loro volta incapsulati in una Visual Studio Solution.

- PrologVSExtension (Solution)
    - PrologItemTemplate (ItemTemplate Project)
        - PrologItemTemplate.ico
        - PrologItemTemplate.vstemplate
        - PrologSource.pl
    - PrologGenerator (VSIX Project)
        - PrologGenerator C# Files
        - Source.extension.vsixmanifest
        - Resources Files

4.7.4.1 PrologItemTemplate

Come già accennato nel Capitolo 1.4 inerente i Template Item, il PrologItemTemplate implementato mediante un ItemTemplate Project, contiene i seguenti file:

- `PrologItemTemplate.ico` : contenente l'icona che verrà associata ai file prolog;
- `PrologItemTemplate.vstemplate`: come già visto nel Capitolo 1.4.1, questo file contiene tutti i parametri dell'ItemTemplate tra cui: nome, descrizione, tipo del progetto compatibile;

```
<?xml version="1.0" encoding="utf-8"?>
<VSTemplate Version="3.0.0" Type="Item"
xmlns="http://schemas.microsoft.com/developer/vstemplate/2005"
xmlns:sdk="http://schemas.microsoft.com/developer/vstemplate-
sdkextension/2010">
  <TemplateData>
    <Name>PrologItemTemplate</Name>
    <Description>PrologItemTemplate</Description>
    <Icon>PrologTemplateIcon.ico</Icon>
    <TemplateID>4625ff2f-3521-40b8-b9b4-4f62a2603ba2</TemplateID>
    <ProjectType>CSharp</ProjectType>
    <RequiredFrameworkVersion>2.0</RequiredFrameworkVersion>
    <NumberOfParentCategoriesToRollUp>1
    </NumberOfParentCategoriesToRollUp>
    <DefaultName>PrologSource.pl
    </DefaultName>
  </TemplateData>
  <TemplateContent>
    <ProjectItem ReplaceParameters="true"
       CustomTool="PrologGenerator">PrologSource.pl
    </ProjectItem>
  </TemplateContent>
</VSTemplate>
```

Tabella 4.10: Manifest File del PrologItemTemplate

- `PrologSource.pl`: il contenuto di default del sorgente tuProlog contiene solo la definizione dei due parametri opzionali NameSpace e ClassName già dettagliati nel paragrafo 4.7.2.1. Si noti l'assenza della keyword `%ExtFile`, che indirizza l'utente verso un utilizzo statico della estensione.

```
%NameSpace MyNameSpace
%ClassName MyClassName
```

Tabella 4.11: Contenuto del file PrologSource.pl

4.7.4.2 PrologGenerator

Questo progetto contiene i file che realizzano il generatore di codice. Il PrologGenerator, una volta attivato da Visual Studio, procederà alla generazione del codice seguendo tre passi di elaborazione:

- individuazione del tipo di progetto (C# o VB);
- analisi del sorgente tuProlog al fine di individuare eventuali keyword, con particolare attenzione alla keyword `%ExtFile` che stabilirà quale modello di classe sarà utilizzato;
- generazione della classe personalizzata in funzione delle keyword.

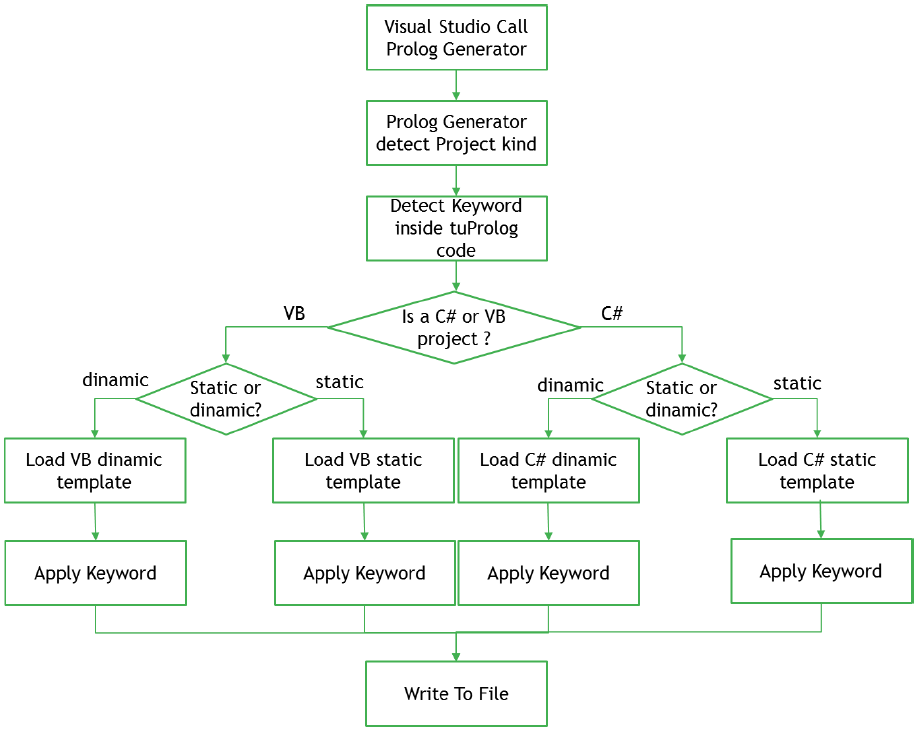

Figura 4.12: Diagramma di Flusso del PrologGenerator

4.7.4.2.1 Modelli di classe statico e dinamico

In funzione della modalità di generazione della classe che sarà attivata il generatore utilizzerà un diverso modello di classe. Questi modelli sono specificati nei file: CSharpTemplate.txt (Tabella 4.13) per il linking statico e CSharpExternalTemplate.txt (Tabella 4.14) per il link dinamico (tabelle tratte dalla tesi di Fabio Gravina[17]).

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

namespace #NameSpace {
   public class #ClassName : Prolog {
      private string th = @"
         #Theory";
      public #ClassName(){
         setTheory(new Theory(th));
      }
   }
}
```

Tabella 4.13: Template per generare una classe C# in modalità statica

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using alice.tuprolog;
using java.io;

namespace #NameSpace {
   public class #ClassName : Prolog {
      public #ClassName(){
         setTheory(new Theory(new FileInputStream("#ExtFile")));
      }
   }
}
```

Tabella 4.14: Template per generare una classe C# in modalità dinamica

Verrà adesso analizzato il file principale del progetto, `PrologGenerator.cs`, suddiviso per sezioni funzionali.

4.7.4.2.2 Registrazione del COM

La registrazione dell'oggetto COM nel registro di sistema viene realizzata attraverso la specifica dei seguenti attributi della classe:

- Visibilità del COM;
- Guid;
- Registrazione del COM per i progetti C#;
- Registrazione del COM per i progetti VB;
- Indicazione del tipo di oggetto che fornisce le funzionalità, in questo caso PrologGenerator.

```
[ComVisible(true)]
[Guid("52B316AA-1997-4c81-9969-83604C09EEB4")]
[CodeGeneratorRegistration(typeof(PrologGenerator),
"C# Prolog Class Generator",
vsContextGuids.vsContextGuidVCSProject,
GeneratesDesignTimeSource=true)]

[CodeGeneratorRegistration(typeof(PrologGenerator),
"VB Prolog Class Generator",
vsContextGuids.vsContextGuidVBProject,
GeneratesDesignTimeSource = true)]
[ProvideObject(typeof(PrologGenerator))]
```

Tabella 4.15: Registrazione del COM nel registro di sistema

4.7.4.2.3 Generazione del codice differenziata per tipo di progetto

Il metodo `GenerateCode`, richiamato da Visual Studio quando quest'ultimo attiva il CustomTool, analizza il tipo di progetto e di conseguenza diversifica la generazione di codice C# da codice VB. Si veda la Tabella 4.16 (tratta dalla tesi di Fabio Gravina[17]) per la sua implementazione.

```
protected override byte[] GenerateCode(string inputFileContent)
{
    Project project = this.GetProject();
    if (project.FullName.EndsWith("csproj"))
       return GenerateCSharpCode(inputFileContent);
    else
       return GenerateVBCode(inputFileContent);
}
```

Tabella 4.16: Identificazione del tipo di progetto

4.7.4.2.4 Generazione di Codice Specifico

La generazione del codice specifico per un determinato linguaggio è eseguita rispettivamente dai metodi `GenerateCSharpCode` e `GenerateVBCode`. Di seguito, in Tabella 4.17 (tratta dalla tesi di Fabio Gravina[17]), è riportato il codice solo del metodo `GenerateCSharpCode` in quanto il codice del metodo `GenerateVBCode` risulta speculare.

```
private byte[] GenerateCSharpCode(string inputFileContent){
  StringReader stringReader = new StringReader(inputFileContent);
  List<string> rows = new List<string>();
  List<string> potentialParam = new List<string>();
  string tmp;
  while ((tmp = stringReader.ReadLine()) != null){
    if(tmp.StartsWith("%"))
      potentialParam.Add(tmp.Trim());
    rows.Add(tmp);
  }

  try{
    this.HasError = false;
    // Set optional params if not found set default Value
    string 54amespace, className, extFile;
    extFile = GetValue("%extfile", potentialParam, "");
    54amespace = GetValue("%namespace", potentialParam,
                base.GetProject().Name);
    className = GetValue("%classname", potentialParam,
                Path.GetFileNameWithoutExtension(base.InputFilePath));
    string template;
    // if no extfile defined
    if(extFile == ""){
      template = FileResource.CsharpTemplate;
      template = template.Replace("#NameSpace", 54amespace.Trim());
      template = template.Replace("#ClassName", className.Trim());
      // copy and set theory inside template row by row
      string theory = rows.Aggregate("", (current, row) =>
                      current + (row + "\r\n"));
      template = template.Replace("#Theory", theory);
    }
    else // extFile defined
    {
      template = FileResource.CsharpExternalTemplate;
      template = template.Replace("#NameSpace", 54amespace.Trim());
      template = template.Replace("#ClassName", className.Trim());
      template = template.Replace("#ExtFile", extFile.Trim());
    }
    MemoryStream ms = new MemoryStream();
    StreamWriter writer = new StreamWriter(ms, new UTF8Encoding());
    writer.Write(template);
    writer.Flush();
    ms.Position = 0;

    return this.HasError ? null : ms.ToArray();
  }
  catch (Exception e){
    this.GeneratorError(4, e.ToString(), 1, 1);
    //Returning null signifies that generation has failed);
    return null;
  }
}
```

Tabella 4.17: Riconoscimento e applicazione delle keyword

Capitolo 5
Porting di PrologVSExtension verso VS 2013

In questo capitolo sarà illustrato il processo di porting verso Visual Studio 2013 di PrologVSExtension, secondo le direttive illustrate nel capitolo 3.

L'intero processo è basato, in primis, sulla riscrittura in Visual Studio 2013 del progetto del generatore di codice PrologVSExtension. In seguito, verranno apportate le modifiche per la registrazione dell'oggetto COM descritte nel paragrafo 3.1.2 e verrà scelta una delle strategie di porting descritte nel capitolo 3.2.

5.1 Registrazione del COM

Per quanto concerne la registrazione dell'oggetto COM del generatore di codice, l'unica modifica da apportare per il corretto funzionamento del generatore di codice su Visual Studio 2013 è l'aggiunta dell'attributo ProvideBindingPathAttribute alla classe che implementa il custom tool del generatore, secondo quanto dedotto dall'analisi descritta nel paragrafo 3.1.2. Nella Tabella 5.1 è possibile vedere la struttura definitiva della registrazione del COM per il nuovo generatore di codice, comprendente il ProvideBindingPathAttribute.

```
[ComVisible(true)]
[Guid("C5DCB497-F6A5-40F8-9604-D3BF29E43DDA")]
[CodeGeneratorRegistration(typeof(PrologGenerator),
"C# Prolog Class Generator",
vsContextGuids.vsContextGuidVCSProject,
GeneratesDesignTimeSource=true)]

[CodeGeneratorRegistration(typeof(PrologGenerator),
"VB Prolog Class Generator",
vsContextGuids.vsContextGuidVBProject,
GeneratesDesignTimeSource = true)]
[ProvideObject(typeof(PrologGenerator))]
[ProvideBindingPath]
```

Tabella 5.1: Struttura definitiva della registrazione dell'oggetto COM

5.2 Strategia di porting

Ultimo passo per il porting della PrologVSExtension è la scelta di una tra le strategie di porting descritte nel capitolo 3.2.

Considerati i vantaggi e gli svantaggi, si decide di adottare la seconda strategia (paragrafo 3.2.2), poiché si ritiene che i vantaggi siano superiori sia agli svantaggi della stessa strategia che ai vantaggi dell'altra strategia (paragrafo 3.2.1).

Infatti, per lo scopo finale di includere il nuovo generatore di codice nella nuova release di tuProlog.NET, si ritiene che il vantaggio di avere un componente software che non necessita di modifiche e ricompilazioni per rimanere compatibile con versioni future di Visual Studio sia nettamente superiore allo svantaggio di dover portarsi dietro anche la vecchia estensione progettata per Visual Studio 2010, in modo da mantenere la retro compatibilità con questa versione di Visual Studio.

In accordo a questa strategia, perciò, si utilizza il manifest file versione 2 per l'implementazione del generatore di codice. Nella Tabella 5.2 è mostrata la versione integrale di tale manifest file, in cui si può notare l'indicazione del range delle versioni compatibili di Visual Studio (`Version="[12.0,]"`), che rende il generatore di codice automaticamente installabile anche su versioni future di Visual Studio senza necessità di modifiche.

```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest Version="2.0.0"
xmlns="http://schemas.microsoft.com/developer/vsx-schema/2011"
xmlns:d="http://schemas.microsoft.com/developer/vsx-schema-design/2011">
  <Metadata>
    <Identity Id="PrologGenerator" Version="1.0" Language="en-US"
              Publisher="Michele Di Lella" />
    <DisplayName>PrologGenerator</DisplayName>
    <Description xml:space="preserve">PrologGenerator</Description>
    <Icon>VSIXProject_small.png</Icon>
    <PreviewImage>VSIXProject_large.png</PreviewImage>
  </Metadata>
  <Installation>
    <InstallationTarget Id="Microsoft.VisualStudio.Pro"
                        Version="[12.0,]" />
  </Installation>
  <Dependencies>
    <Dependency Id="Microsoft.Framework.NDP"
                DisplayName="Microsoft .NET Framework"
                d:Source="Manual"
                Version="[4.0,)" />
  </Dependencies>
  <Assets>
    <Asset Type="Microsoft.VisualStudio.ItemTemplate"
           d:Source="Project" d:ProjectName="PrologItemTemplate"
           d:TargetPath="|PrologItemTemplate;TemplateProjectOutputGroup|"
           Path="ItemTemplates"
           d:VsixSubPath="ItemTemplates" />
    <Asset Type="Microsoft.VisualStudio.ItemTemplate"
           d:Source="Project"
           d:ProjectName="VBPrologItemTemplate"
           d:TargetPath="|VBPrologItemTemplate;TemplateProjectOutputGroup|"
           Path="ItemTemplates"
           d:VsixSubPath="ItemTemplates" />
  </Assets>
</PackageManifest>
```

Tabella 5.2: Manifest file definitivo del nuovo generatore di codice

Nei paragrafi che seguono, sarà illustrata un applicazione di test che proverà l'effettivo funzionamento del nuovo generatore di codice in Visual Studio 2013, e che perciò verificherà la fondatezza delle ipotesi portate avanti in seguito all'analisi compiuta in precedenza nel capitolo 3 e messe in pratica in questo capitolo..

5.3 Collaudo della tecnologia in Visual Studio 2013

In questo capitolo sarà analizzata una applicazione di test utilizzata per verificare il funzionamento del generatore di codice realizzato e per confrontare l'utilizzo delle due diverse modalità di linking (statica e dinamica) tra loro. Questa applicazione, sviluppata in origine da Fabio Gravina[17] per testare il funzionamento del generatore di codice su Visual Studio 2010, è stata portata su Visual Studio 2013 come lavoro di questa tesi, in modo tale da testare il pieno funzionamento del generatore di codice su Visual Studio 2013 dopo il porting illustrato nel in precedenza nel capitolo.

Per realizzare l'applicazione è stato scelto di utilizzare il linguaggio C#, anche se è possibile ripetere il procedimento in maniera analoga in VB.

L'applicazione risulterà perciò strutturata (Figura 5.3, tratta dalla tesi di Fabio Gravina[17]) in due parti: una logica, che si occupa della interazione con il motore tuProlog, e una imperativa, che invece si occupa della gestione della interfaccia grafica.

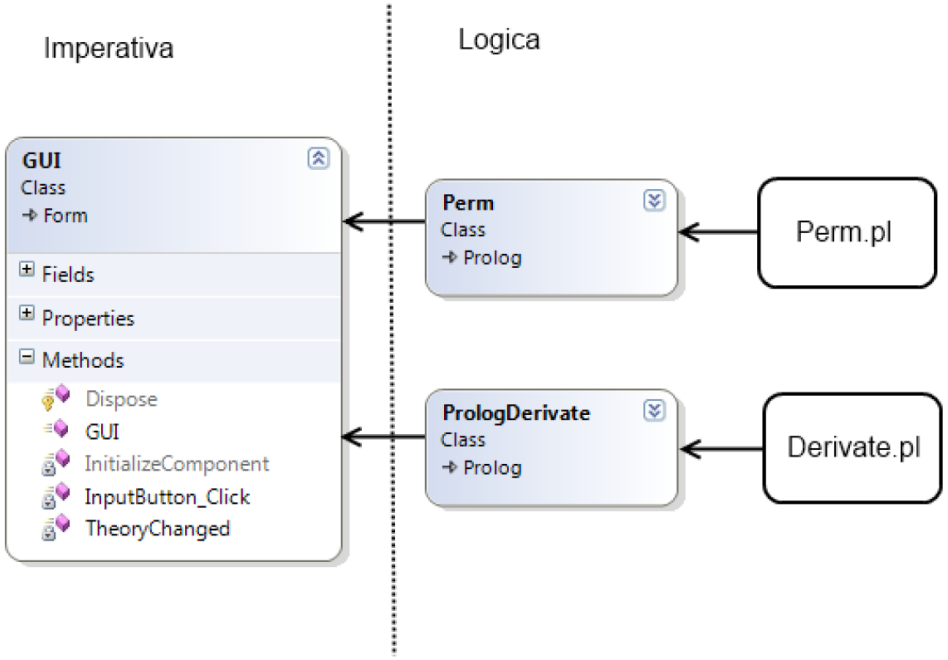

Figura 5.3: Struttura della applicazione

5.3.1 Sezione logica

Questa sezione sarà analizzata in due sotto parti afferenti ai due diversi esempi di generazione del codice. Al fine di mostrare entrambe le modalità, statica e dinamica, in questo esempio scegliamo di risolvere il problema del calcolo delle permutazioni con la modalità statica e il problema della risoluzione simbolica delle derivate in modalità dinamica. Questa è solo una delle possibilità, dato che entrambi i problemi sono risolvibili con entrambe le modalità.

5.3.1.1 Permutazioni

In questo caso l'obbiettivo è al risoluzione del calcolo delle permutazioni utilizzando l'algoritmo tuProlog contenuto nel file Perm.pl. Come detto sopra, si è scelta per questo esempio la modalità statica, quindi si può notare nel sorgente tuProlog l'assenza della `%ExtFile`.

Perm.pl

```
any([X|Xs],X,Xs).
any([X|Xs],E,[X|Ys]) :- any(Xs,E,Ys).
perm([],[]).
perm(Xs,[X|Ys]) :- any(Xs,X,Zs), perm(Zs,Ys).
```

Tabella 5.4: Sorgente tuProlog per la risoluzione delle permutazioni

Inoltre non sono state applicate neanche le altre keyword progettate nel sistema. A seguito di ciò, l'esecuzione del generatore di codice produce un file C# in cui vengono assegnati valori di default per il NameSpace e il ClassName. Nella classe generata `Perm.cs` (Tabella 5.5, tratta dalla tesi di Fabio Gravina[17]) si può infatti notare:

- il nome della classe corrispondente al nome del file;
- il namespace uguale al nome del progetto;
- l'utilizzo della modalità statica che ha incapsulato la teoria dentro il file C#.

Perm.cs

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

namespace PrologTestSolution {
   public class Perm : Prolog {
      private string th = @"
         any([X|Xs],X,Xs).
         any([X|Xs],E,[X|Ys]) :- any(Xs,E,Ys).
         perm([],[]).
         perm(Xs,[X|Ys]) :- any(Xs,X,Zs), perm(Zs,Ys).
      ";

      public Perm(){
             setTheory(new Theory(th));
      }
   }
}
```

Tabella 5.5: Codice C# generato in modalita statica

5.3.1.2 Derivate

Il questo secondo esempio lo scopo è la risoluzione simbolica delle derivate, utilizzando l'algoritmo tuProlog contenuto nel file Der.pl. Volendo testare il funzionamento della modalità dinamica è stato creato il file di "linking" `Derivate.pl`.

Derivate.pl

```
%NameSpace DerivateNamespace
%ClassName PrologDerivate
%ExtFile Der.pl
```

Tabella 5.6: File di collegamento a Der.pl per il linking dinamico

In questo caso sono state utilizzate tutte e tre le keyword progettate e di conseguenza è possibile vedere i loro effetti nel file generato `Derivate.cs` (Tabella 5.7, tratta dalla tesi di Fabio Gravina[17]):

- NameSpace e il nome della classe sono quelli specificati nel file tuProlog e non i valori di default;
- la keyword `ExtFile` ha attivato la modalità dinamica portando all'implementazione di un costruttore che non include la teoria ma che la andrà a leggere a runtime.

Derivate.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using alice.tuprolog;
using java.io;

namespace DerivateNamespace {

   public class PrologDerivate : Prolog {

      public PrologDerivate(){

         setTheory(new Theory(new FileInputStream("Der.pl")));

      }
   }
}
```

Tabella 5.7: Classe C# generata in modalità dinamica

La teoria Prolog sarà contenuta nel file `Der.pl` che si riporta di seguito per completezza (Tabella 5.8, tratta dalla tesi di Fabio Gravina[17]).

```
dExpr(T,DT) :- dTerm(T,DT).
dExpr(E+T,[DE+DT]) :- dExpr(E,DE), dTerm(T,DT).
dExpr(E-T,[DE-DT]) :- dExpr(E,DE), dTerm(T,DT).
dTerm(F,DF) :- dFactor(F,DF).
dTerm(T*F,[[DT*F]+[T*DF]]) :- dTerm(T,DT), dFactor(F,DF).
dTerm(T/F,[[F*DT]-[T*DF]]/[F*F]) :- dTerm(T,DT), dFactor(F,DF).
dFactor(x,1).
dFactor(N,0) :- number(N).
dFactor([E],DE) :- dExpr(E,DE).
dFactor(-E,-DE) :- dExpr(E,DE).
dFactor(sin(E), [cos(E)*DE] ) :- dExpr(E,DE).
dFactor(cos(E), [-sin(E)*DE] ):- dExpr(E,DE).
dFactor(E^N, [N*E^(Nm1)*(DE)] ) :- dExpr(E,DE), Nm1 is N-1.
dExpr(T,DT) :- dTerm(T,DT).
dExpr(E+T,[DE+DT]) :- dExpr(E,DE), dTerm(T,DT).
dExpr(E-T,[DE-DT]) :- dExpr(E,DE), dTerm(T,DT).
dTerm(F,DF) :- dFactor(F,DF).
dTerm(T*F,[[DT*F]+[T*DF]]) :- dTerm(T,DT), dFactor(F,DF).
dTerm(T/F,[[F*DT]-[T*DF]]/[F*F]) :- dTerm(T,DT), dFactor(F,DF).
dFactor(x,1).
dFactor(N,0) :- number(N).
dFactor([E],DE) :- dExpr(E,DE).
dFactor(-E,-DE) :- dExpr(E,DE).
dFactor(sin(x), cos(x)) :- !.
dFactor(cos(x), -sin(x)) :- !.
dFactor(sin(E), [cos(E)*DE] ) :- dExpr(E,DE).
dFactor(cos(E), [-sin(E)*DE] ):- dExpr(E,DE).
dFactor(x^N, [N*x^(N1)] ) :- N1 is N-1, !.
dFactor(E^N, [N*E^(Nm1)*(DE)] ) :- dExpr(E,DE), Nm1 is N-1.
```

Tabella 5.8: Algoritmo tuProlog per la risoluzione simbolica delle derivate

5.3.2 Interfaccia grafica

Tramite i tool di design di Visual Studio è stata realizzata una semplice interfaccia grafica (Figura 5.9) che permette di:

- selezionare la teoria da testare;
- inserire la query tuProlog;
- avviare l'esecuzione.
- Mostrare i risultati

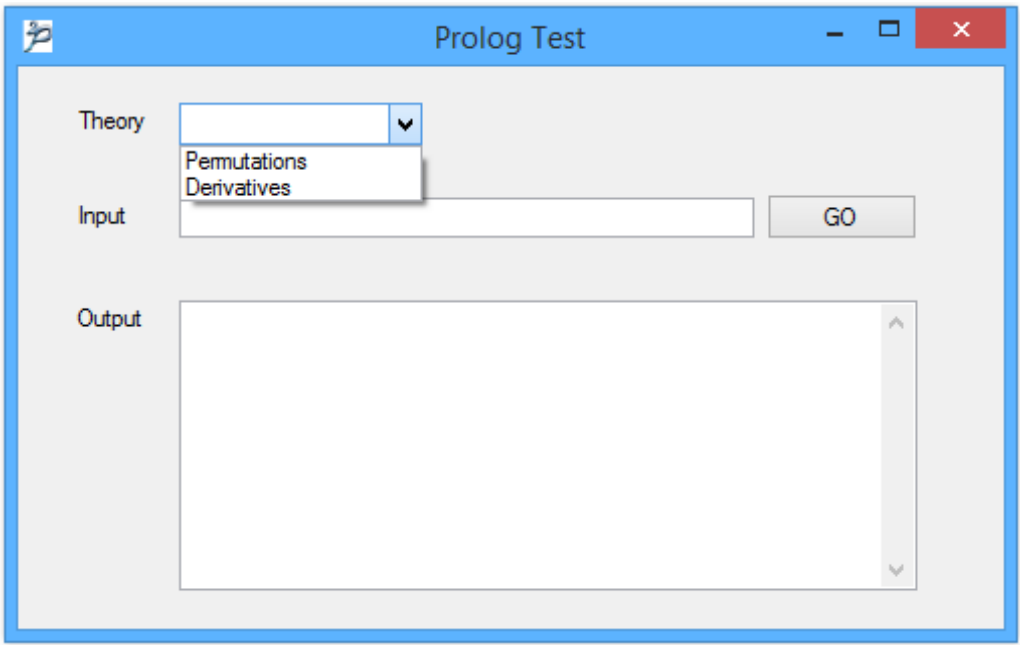

Figura 5.9: Interfaccia utente della applicazione di collaudo

Infine sono state collegate:

- la selezione della teoria con il metodo handler `TheoryChanged` nel codice in basso. Si può notare come la creazione dell'oggetto PrologDerivate necessiti di un costrutto try-catch per gestire correttamente un eventuale eccezione dovuta ad un errato riferimento al file `der.pl`;

```
private void TheoryChanged(object sender, EventArgs e)
{
    switch (((ComboBox)sender).SelectedItem.ToString())
    {
        case "Permutations":
            prolog = new Perm();
            labelError.Text = "Usage: [x,y,z, ... ]";
            break;
        case "Derivatives":
            try
            {
                prolog = new PrologDerivatives();
                labelError.Text = "";
            }catch{
                labelError.Text = "Cannot find prolog source";
            }
            break;
    }
}
```

Tabella 5.10: Gestione dell'evento a seguito del cambio di teoria

- la pressione del tasto “GO” con il metodo handler `InputButton_Click`;

```
private void InputButton_Click(object sender, EventArgs e)
{
    try
    {
        if (prolog == null)
        {
            labelError.Text = "Please select a Theory";
            return;
        }
        if (InputTextBox.Text.Length == 0)
        {
            OutputTextBox.Text = "Please insert a Goal";
            return;
        }

        OutputTextBox.Clear();
        if (prolog is Perm)
        {
            Term goal = new Struct("perm",Term.createTerm(InputTextBox.Text),new
                                                                        Var("R"));
            SolveInfo SI = prolog.solve(goal);

            if (!SI.isSuccess())
                OutputTextBox.AppendText("no");

            while (SI.isSuccess())
            {
                OutputTextBox.AppendText(SI.getTerm("R").toString() + "\n");
                if (prolog.hasOpenAlternatives())
                    SI = prolog.solveNext();
                else
                    break;
            }
        }
        else if (prolog is PrologDerivatives)
        {
            Term goal = new Struct("dExpr", Term.createTerm(InputTextBox.Text), new
                                                                    Var("DT"));
            SolveInfo SI = prolog.solve(goal);

            if (!SI.isSuccess())
                OutputTextBox.AppendText("no");
            else
                OutputTextBox.AppendText(SI.getTerm("DT").toString() + "\n");

        }

    }
    catch (MalformedGoalException ex) {
        OutputTextBox.AppendText("Malformed Goal\n");
    }
    catch (InvalidTermException ex) {
        OutputTextBox.AppendText("Malformed Goal\n");
    }
}
```

Tabella 5.11: interazione con il motore tuProlog

Nelle figure 5.12 e 5.13 sono illustrati, rispettivamente, un esempio del funzionamento delle permutazioni e uno delle derivate.

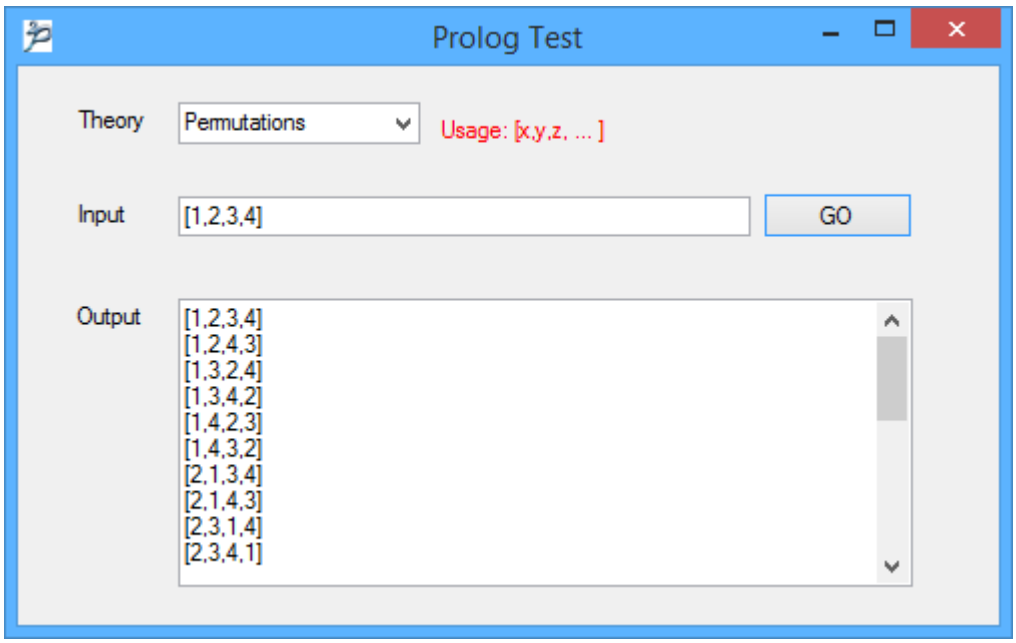

Figura 5.12: Esempio del calcolo delle permutazioni

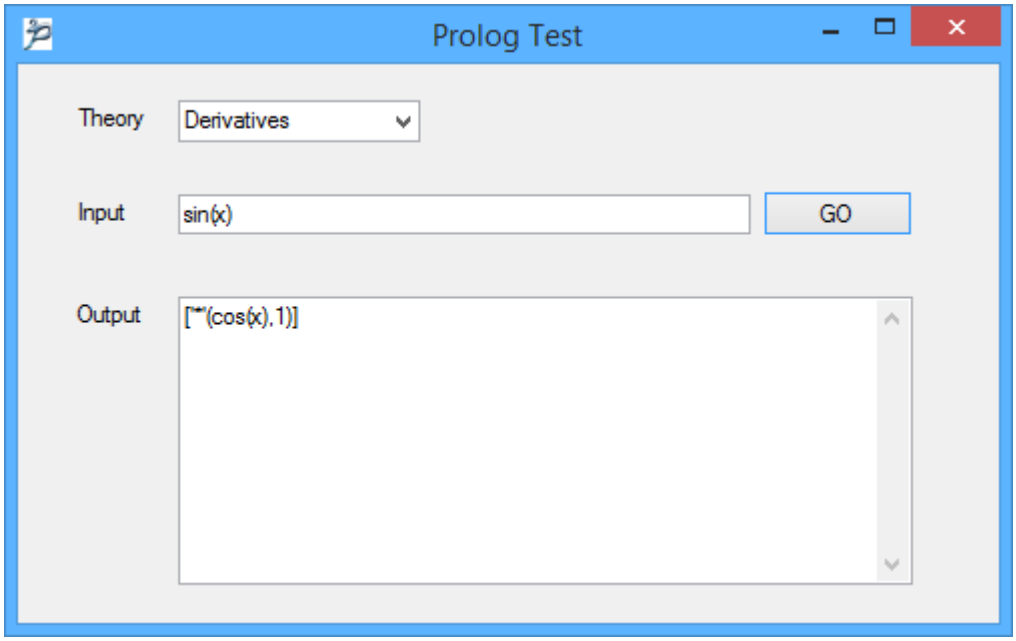

Figura 5.13: Esempio del calcolo delle derivate

5.3.3 Osservazioni

Il test dell'applicazione di collaudo ha dimostrato la piena funzionalità su Visual Studio 2013 del generatore di codice successivamente al porting illustrato in precedenza nel capitolo, rispettando tutti requisiti preposti.

Capitolo 6
Conclusioni

In questa tesi ci si è posto l'obiettivo di analizzare in dettaglio il modello di funzionamento dei generatori di codice il Visual Studio, con lo scopo di rendere funzionante in Visual Studio 2013 un generatore di codice originariamente progettato per Visual Studio 2010.

Come si è visto nel capitolo 3, le novità apportate da Microsoft nella versione 2013 di Visual Studio hanno inficiato l'installazione e l'effettivo funzionamento in Visual Studio 2013 di un generatore di codice progettato per la precedente versione del 2010, a causa di modifiche al manifest file e al meccanismo di registrazione e reperimento dell'oggetto COM del generatore di codice.

Oltre alla piena compatibilità con Visual Studio 2013, un altro dei requisiti che sono stati posti è stata la necessità di mantenere, allo stesso tempo, la retro compatibilità con Visual Studio 2010. Come si è visto nella parte finale del capitolo 3, sono state studiate due tecniche di porting per rispettare al meglio i requisiti posti.

Alla luce dei risultati di tale studio di porting, sono state perciò prese alcune decisioni per portare a termine l'effettivo porting del generatore di codice PrologVSExtension descritto nel capitolo 4.

Per quanto riguarda la retro compatibilità con Visual Studio 2010, è stata scelta la strategia meno conservativa, che prevede il mantenimento di due versioni del generatore di codice: quella già esistente, originariamente progettata per Visual Studio 2010, e una nuova, progettata per Visual Studio 2013 e, inoltre, compatibile anche con versioni successive.

Il nuovo generatore di codice, risultato del processo di porting, è stato infine testato nell'implementazione di un'applicazione di test multi-paradigma che sfrutta il framework .NET e tuProlog per il calcolo delle permutazioni e delle derivate simboliche. Tale applicazione è risultata perfettamente funzionante,

confermando il pieno raggiungimento degli obiettivi posti inizialmente, nel rispetto di tutti i requisiti preposti.

Bibliografia

1 **tuProlog**
http://alice.unibo.it/xwiki/bin/view/Tuprolog/

2 **Java**
http://java.sun.com

3 **.NET Framework**
http://www.microsoft.com/net

4 **Common Language Infrastructure (CLI)**
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-335.pdf

5 **Mono .NET Framework**
http://www.mono-project.com/Main_Page

6 **DotGNU**
http://www.gnu.org/software/dotgnu/

7 **Common Language Runtime**
http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx

8 **Visual Basic**
http://msdn.microsoft.com/en-us/vstudio/hh388573

9 **Visual C++**
http://msdn.microsoft.com/en-us/vstudio/hh386302.aspx

10 **C#**

http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx

11 **Delphi**

http://www.delphibasics.co.uk/

12 **C++**

http://www.open-std.org/jtc1/sc22/wg21/

13 **Visual J#**

http://msdn.microsoft.com/en-us/library/7xsxf8e2(v=vs.80).aspx

14 **Visual Studio**

http://www.microsoft.com/visualstudio/

15 **Visual Studio SDK**

http://msdn.microsoft.com/en-us/vstudio/vextend.aspx

16 **IKVM.NET**

http://www.ikvm.NET/

17 **Fabio Gravina**

Integrazione di codice tuProlog in linguaggi .NET: approcci a confronto. Tesi di Laurea, AA 2011/2012.