Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI SCIENZE
CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

REALIZZAZIONE DI UN GENERATORE DI CODICE .NET ESTENDIBILE

Tesi di Laurea in:
PROGRAMMAZIONE AD OGGETTI

Relatore:
Chiar.mo Prof. MIRKO VIROLI

Presentata da:
LUCA TREMAMUNNO

Correlatore:
Dott. GIOVANNI CIATTO

Prima Sessione di Laurea
Anno Accademico 2017-2018

Abstract

Parole chiave: CodeDOM, Generatore di codice, .Net, Modulare, COBOL

Cobol2Net è uno strumento per la traduzione automatica di programmi scritti in *COBOL* verso linguaggi dell'ambiente *.Net*. Questo progetto è stato realizzato dall'Università di Bologna sotto richiesta di Harvard Group s.r.l., un'azienda del territorio.

Uno dei principali motivi dietro alla volontà di tradurre programmi *COBOL* è dovuto a questioni di manutenibilità del codice: tuttora sono ancora utilizzati molti programmi scritti in questo linguaggio, ma il numero di programmatori che lo conoscono è limitato, e per questo si desidera il passaggio a linguaggi moderni. Allo stesso tempo, data la dimensione di questi programmi, è impossibile effettuarne la traduzione manuale.

Cobol2Net, al momento di inizio di questa tesi, è in grado di tradurre in maniera completamente automatica la maggior parte dei programmi forniti dall'azienda commissionante e, per produrre il codice del programma tradotto, fa uso della libreria CodeDOM, che consente di generare codice scritto in un linguaggio dell'ambiente *.Net* partendo dall'*Abstract Syntax Tree* rappresentante il programma target, ma non fornisce alcun controllo sulla generazione del codice.

Di conseguenza non risulta possibile soddisfare alcune richieste dell'azienda relative alla forma dei programmi tradotti. Inoltre il codice prodotto da CodeDOM, a causa di alcune limitazioni della libreria, risulta spesso di scarsa qualità, riducendo di fatto l'impatto positivo che ha la traduzione sulla manutenibilità dei programmi.

L'obiettivo di questa tesi, quindi, è la realizzazione di un nuovo generatore, in modo da acquisire il totale controllo sul codice prodotto. Questo dovrà inoltre fornire un semplice metodo per modificare il modo in cui i singoli costrutti sono generati senza dover ridefinire le regole di gestione degli altri, in modo da agevolare ulteriori modifiche future.

Indice

Introduzione

Cobol2Net è un progetto che si occupa di effettuare la traduzione automatica di software scritto in *COBOL* verso linguaggi dell'ambiente *.Net*. Il progetto è stato realizzato dall'Università di Bologna per conto di Harvard Group s.r.l., un'azienda informatica del territorio; uno dei principali obiettivi dietro alla realizzazione di questo progetto è l'aumento della manutenibilità dei programmi tradotti rispetto agli originali, in quanto il numero di programmatori che conosce il linguaggio *COBOL* è molto limitato.

Nell'effettuare la traduzione, in particolare per produrre il codice finale, Cobol2Net fa uso di CodeDOM, una libreria che permette di generare codice per linguaggi dell'ambiente *.Net* a partire da un grafo rappresentante il programma da realizzare. Questa libreria però non consente di controllare il modo in cui il codice deve essere generato; di conseguenza non è possibile soddisfare alcune richieste dell'azienda relative alla forma del codice. Inoltre, a causa di alcune limitazioni di CodeDOM, il codice risultante, seppur funzionante, presenta problemi relativi alla qualità, che impattano negativamente sulla sua manutenibilità.

L'obiettivo di questa tesi è la realizzazione di un nuovo generatore di codice, che andrà sostituito a quello di CodeDOM, in modo da poter ottenere il controllo sulla forma del codice prodotto e, di conseguenza, operare per migliorare la qualità di quest'ultimo.

Per evitare che un simile problema si riproponga in futuro, una caratteristica fondamentale che deve avere il nuovo generatore è fornire un semplice modo per poter ridefinire come alcuni costrutti vengono generati, ma senza costringere l'utente a dover reimplementare anche la gestione di costrutti che invece non si vogliono modificare. Per realizzare questo obiettivo si è deciso di costruire il generatore in maniera modulare permettendo di specificare a *run-time* il modo in cui il generatore debba produrre determinati costrutti.

Questo generatore è stato realizzato come una libreria separata da Cobol2Net, denominata CodeDomExt, in quanto la modularità del generatore potrebbe risultare interessante e utile per ulteriori progetti.

Di seguito viene indicata la struttura della tesi, fornendo una breve descrizione del contenuto dei capitoli.

- **CodeDOM**: descrizione della struttura di CodeDOM, il generatore di codice utilizzato da Cobol2Net al momento di inizio di questa tesi.
- **Cobol2Net**: descrizione della struttura di Cobol2Net e dei motivi che hanno portato alla sua realizzazione.
- **Requisiti**: motivazioni dietro alla realizzazione di questa tesi e obiettivi dettagliati.
- **Progettazione**: architettura generale del nuovo generatore e motivazione delle scelte effettuate.
- **Implementazione**: dettagli di basso livello e presentazione di alcuni frammenti di codice del nuovo generatore ritenuti interessanti.
- **Validazione e conclusioni**: criteri e strumenti utilizzati per determinare la correttezza dell'implementazione e riassunto dei risultati ottenuti dalla tesi citando anche possibili sviluppi futuri.

Capitolo 1

CodeDOM

CodeDOM[1], acronimo di *Code Document Object Model*, è una libreria sviluppata da *Microsoft*, presente sin dalle prime versioni del *.Net framework*. Questa fornisce meccanismi per la generazione di codice sorgente in più linguaggi di programmazione, a partire da un grafo rappresentante la struttura logica del programma da generare (in seguito indicato come *grafo CodeDOM*).

Un'altra funzionalità offerta da CodeDOM è la compilazione di codice a *run-time*; il modo in cui questa viene realizzata è tramite un *wrapping* del compilatore del linguaggio target. CodeDOM quindi in questo caso si occupa di fornire un accesso semplificato a quest'ultimo.

Tipicamente la libreria viene utilizzata in situazioni in cui occorre generare codice per più linguaggi; è il caso per esempio di strumenti che generano codice basato su dei *template*, quali i designer *WinForms*.

La libreria si compone di due parti principali, ciascuna contenuta in un proprio *namespace*:

- `System.CodeDom`: raccoglie le definizioni dei nodi utilizzabili per la costruzione del grafo
- `System.CodeDom.Compiler`: fornisce gli strumenti necessari per la generazione del codice sorgente a partire dal grafo CodeDOM

Le classi facenti parte di questa libreria sono in genere facilmente individuabili anche grazie al fatto che la quasi totalità di queste ha nel nome il prefisso `Code`

1.1 Nodi

Un grafo CodeDOM è formato da vari tipi di nodi; in genere il tipo di nodo definisce delle proprietà[1], che conterranno i dettagli dell'elemento rappresentato dal nodo, o i suoi figli, ma esistono anche nodi senza alcuna proprietà.

Una caratteristica comune a tutti i nodi è quella di essere considerati dei semplici contenitori, pertanto essi sono privi di logica; gli unici vincoli presenti in fase di costruzione del grafo sono, quindi, dati dalla struttura dei nodi. Inoltre sono tutti derivati dalla classe `CodeObject`.

Di seguito sono illustrate le classi che definiscono i principali tipi di nodi, indicandone le proprietà più importanti, in modo da fornire una panoramica di come potrebbe essere strutturato un grafo CodeDOM.

- `CodeCompileUnit`: è il nodo radice dell'albero, rappresenta un file del programma. Può contenere dei *namespace*.

- `CodeNamespace`: rappresenta un *namespace*. Ha una proprietà che ne indica il nome, e può contenere degli *import* e delle *dichiarazioni di tipi*.

- `CodeNamespaceImport`: rappresenta un *import*. Ha una proprietà che indica il nome del *namespace* importato.

- `CodeTypeDeclaration`: rappresenta una *dichiarazione di tipo*. Ha delle proprietà per indicarne nome, eventuali tipi di base, *membri* e attributi; quest'ultimi indicano il livello di accessibilità e se la classe è *sealed*, *abstract* o nessuna delle due.

  Sono inoltre presenti delle proprietà flag che indicano cosa sia il tipo rappresentato, ossia se questo è *class*, *struct*, *enum* o *interface*, mentre per poter rappresentare il tipo *delegate* si utilizza la sottoclasse `CodeTypeDelegate`. Quest'ultima contiene ulteriori proprietà, per indicare il tipo di ritorno ed eventuali parametri del *delegate*.

- `CodeTypeMember`: è una classe astratta che rappresenta un *membro* di un tipo. Ha delle proprietà che ne specificano il nome e attributi, ossia il livello di accessibilità e altre informazioni, che indicano per esempio se il membro è *static*, *abstract* o altro.

[1] Le proprietà[2] in ambito *.Net* sono membri che possono essere utilizzati come normali campi, ma richiamano in realtà degli speciali metodi detti *accessor* che funzionano in maniera analoga a metodi *getter* e *setter*

Questa classe non rappresenta nessun *membro* specifico di per sè, ma esistono varie sottoclassi, ciascuna rappresentante un tipo di membro distinto:

  - `CodeMemberEvent`: rappresenta un *evento*, contiene un'ulteriore proprietà per specificarne il tipo.
  - `CodeMemberField`: rappresenta un *campo*, contiene ulteriori proprietà per specificarne il tipo ed una eventuale espressione di inizializzazione.
  - `CodeMemberMethod`: rappresenta un *metodo*, contiene ulteriori proprietà per specificarne il tipo di ritorno, eventuali parametri ed istruzioni. Esistono ulteriori sottoclassi di questa, per rappresentare metodi *main*, *costruttori* o *inizializzatori di tipo*.
  - `CodeMemberProperty`: rappresenta una *proprietà*, contiene ulteriori proprietà per specificarne il tipo e le istruzioni dei suoi *getter* e *setter*.
  - `CodeTypeDeclaration`: illustrata in precedenza, deriva da `CodeTypeMember` per permettere di rappresentare il *nesting* tra i tipi.

- `CodeStatement`: è una classe astratta che rappresenta un'*istruzione*. Esistono varie sottoclassi di questa, ciascuna rappresentante un'istruzione specifica. In genere le sottoclassi definiscono proprietà che possono contenere altre *istruzioni* e/o *espressioni*.

  Di seguito sono illustrati alcuni esempi:

  - `CodeVariableDeclarationStatement`: rappresenta un'istruzione di *dichiarazione di variabile*, ha delle proprietà che ne contengono il tipo, il nome e l'eventuale espressione di inizializzazione.
  - `CodeConditionStatement`: rappresenta un'istruzione *if-else*, ha delle proprietà per indicare l'espressione da valutare, e le istruzioni da eseguire nel caso quest'ultima sia vera e quelle da eseguire nel caso sia falsa.
  - `CodeExpressionStatement`: è un *wrapper* che permette di utilizzare un'espressione come istruzione.

- `CodeExpression`: è una classe astratta che rappresenta un'*espressione*. Esistono varie sottoclassi di questa, ciascuna rappresentante un'espressione specifica. In genere le sottoclassi definiscono proprietà che possono contenere altre *espressioni*.

  Di seguito sono illustrati alcuni esempi:

  - `CodeFieldReferenceExpression`: rappresenta un *riferimento ad un campo*, ha delle proprietà per indicare il nome del campo e l'eventuale *espressione* rappresentante l'oggetto target.
  - `CodeMethodInvokeExpression`: rappresenta una *invocazione di metodo*, ha delle proprietà che contengono espressioni rappresentanti il metodo da invocare ed eventuali parametri.
  - `CodeBinaryOperatorExpression`: rappresenta una *operazione binaria*, ha delle proprietà che indicano l'operatore e le espressioni rappresentanti gli operandi.
  - `CodeThisReferenceExpression`: rappresenta il *riferimento all'istanza corrente* della classe in cui è usata (parola chiave `this` in *C#*), non contiene alcuna proprietà.
  - `CodePrimitiveExpression`: rappresenta un *valore* di un tipo primitivo; questo può essere *null*, un numero (intero o a virgola mobile), una stringa, un carattere o un booleano.

- `CodeTypeReference`: rappresenta un *tipo*. Ha delle proprietà per indicarne, oltre al nome, l'eventuale rango, se questo è un *array*, e gli eventuali valori per i parametri di tipo, rappresentati da altre `CodeTypeReference`, se generico.

- `CodeTypeParameter`: rappresenta un *parametro di tipo*, viene utilizzato nelle dichiarazioni di tipi e metodi generici. Ha delle proprietà per indicarne il nome ed eventuali vincoli.

- `CodeAttributeDeclaration`: rappresenta un *attributo*[2]. `CodeTypeDeclaration`, `CodeTypeMember` e `CodeParameterDeclarationExpression` definiscono una proprietà per poter contenere eventuali *attributi* relativi all'oggetto generato.

- `CodeDirective`: rappresenta una *direttiva al preprocessore*. `CodeCompileUnit`, `CodeTypeDeclaration`, `CodeTypeMember` e `CodeStatement` definiscono delle proprietà che permettono di specificare eventuali *direttive* da generare immediatamente prima o dopo l'oggetto in questione.

- `CodeComment`: rappresenta un *commento*. `CodeNamespace`, `CodeTypeDeclaration`, `CodeTypeMember` e `CodeStatement` definiscono una proprietà che permette di specificare eventuali *commenti* da generare immediatamente prima dell'oggetto in questione.

[2]Gli attributi[2] sono analoghi alle *annotazioni* in *java* e sono utilizzati per associare al codice delle informazioni, che verranno poi utilizzate dal compilatore. Un esempio è l'attributo `obsolete`, che indica l'obsolescenza di un tipo o di un suo membro

- `CodeSnippet`: contiene una stringa che rappresenta un frammento di codice, che verrà generata in output senza modifiche. Può essere utilizzato per generare costrutti non previsti da *CodeDOM*, ma il codice generato dall'albero potrebbe risultare non più valido per alcuni linguaggi. Esistono varie classi di questo tipo, che ereditano da `CodeStatement`, `CodeExpression` e `CodeTypeMember`; in questo modo si possono inserire in vari punti dell'albero.

1.2 Struttura di un grafo CodeDOM

Il grafo CodeDOM è in genere strutturato come un albero, in cui i nodi, ciascuno rappresentante uno specifico elemento del programma, sono dei contenitori collegati l'un l'altro tramite proprietà degli stessi. Nella figura 1.1 è rappresentato un esempio di grafo CodeDOM rappresentante un programma "Hello World!", costruito tramite il codice rappresentato nel blocco 1.1

Diversamente da quanto avviene negli alberi, è però possibile, per i nodi di un grafo CodeDOM valido, avere più di un singolo padre; in presenza di tale situazione il nodo verrà visitato più volte dal generatore, che produrrà il relativo codice in output ad ogni visita. Il risultato finale è, quindi, equivalente a quello che si otterrebbe generando il codice sulla base di un nuovo albero, prodotto sostituendo ricorsivamente ai nodi con più padri nel grafo originale, un nuovo nodo, identico all'originale, per ogni padre.

Rimane comunque impossibile avere dei cicli; sebbene questo sia permesso in fase di costruzione del grafo, causerà errori in fase di generazione.

1.3 Generazione

La generazione del codice a partire da un grafo CodeDOM avviene per mezzo di un `CodeDomProvider`, che rappresenta un generatore di codice per uno specifico linguaggio. CodeDOM consente anche di specificare delle opzioni, che modificano lo stile del codice generato; queste vanno passate al provider per mezzo di una classe `CodeGeneratorOptions`. Alcuni esempi di opzioni sono:

- `IndentString`: specifica quale stringa usare per l'indentazione.
- `BracingStyle`: permette di specificare se la parentesi di inizio blocco deve essere generata sulla stessa linea dell'istruzione o dichiarazione a cui è associata, o la successiva.

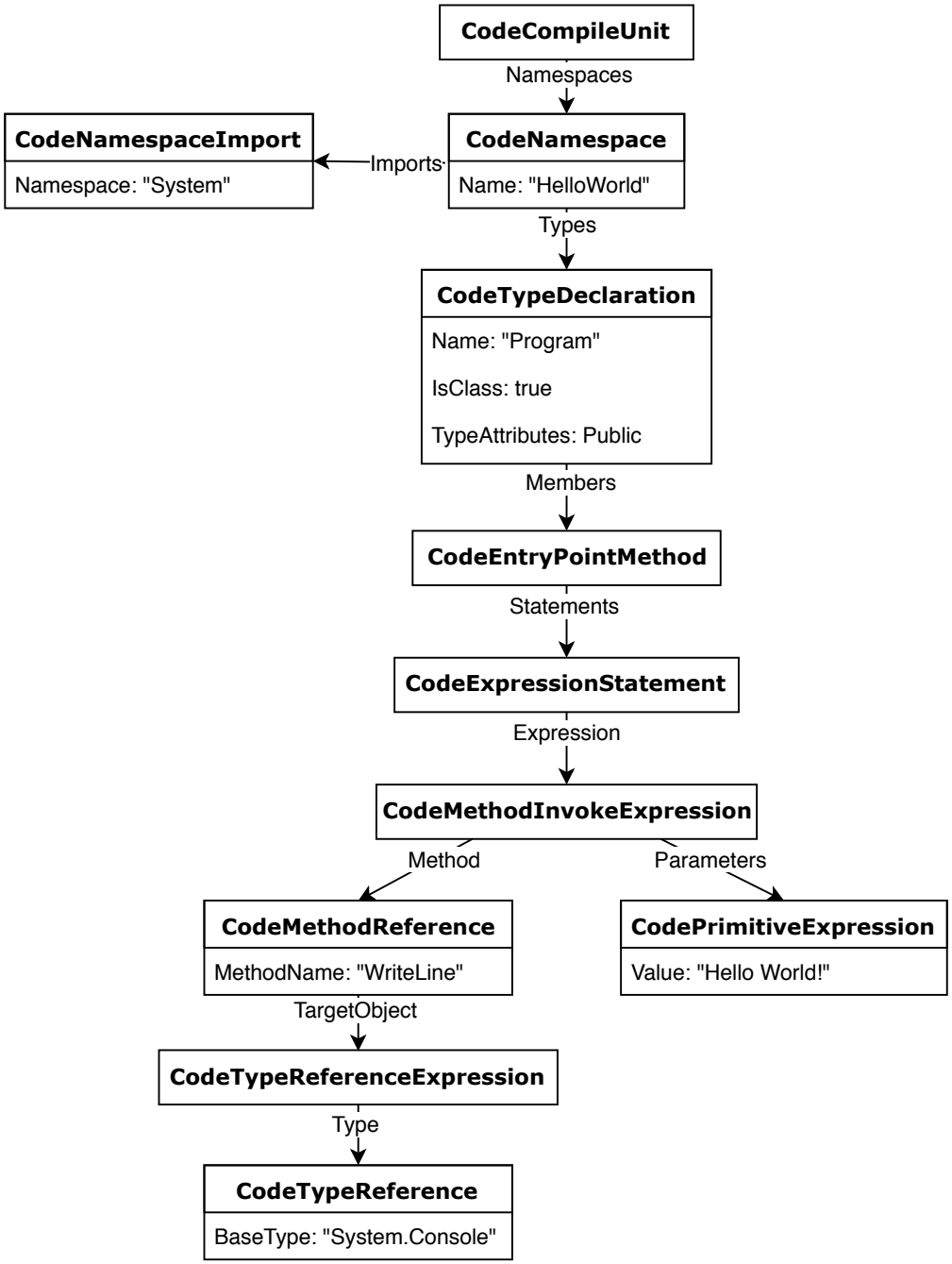

Figura 1.1: Rappresentazione grafica di un albero CodeDOM per un programma "Hello World!". Le frecce rappresentano proprietà che referenziano altri oggetti di CodeDOM.

```
1  CodeCompileUnit compileUnit = new CodeCompileUnit();
2  CodeNamespace example = new CodeNamespace("HelloWorld");
3  example.Imports.Add(new CodeNamespaceImport("System"));
4  compileUnit.Namespaces.Add(example);
5  CodeTypeDeclaration program = new CodeTypeDeclaration("Program");
6  example.Types.Add(program);
7  CodeEntryPointMethod main = new CodeEntryPointMethod();
8  main.Statements.Add(new CodeMethodInvokeExpression(
9      new CodeTypeReferenceExpression(typeof(System.Console)), //target
           object
10     "WriteLine", //method
11    new CodePrimitiveExpression("Hello World!")); //arguments
12 program.Members.Add(main);
```

Codice 1.1: Esempio di codice per la costruzione di un grafo CodeDOM rappresentante un programma "Hello World!"

- `BlankLinesBetweenMembers`: permette di specificare se lasciare una linea vuota tra i membri di un tipo.

Il codice viene generato effettuando una visita dell'albero, scrivendo man mano in output il codice relativo al nodo visitato. L'ordine con cui vengono visitati eventuali figli o proprietà di un nodo dipende sia dal suo tipo, che dal linguaggio del provider; si pensi ad esempio alla dichiarazione di una variabile in *Visual Basic* e *C#*:

```
Dim i As Integer = 0
```

```
int i = 0;
```

Appare evidente che, quando si genera codice *Visual Basic*, le proprietà del nodo `CodeVariableDeclarationStatement` vengono visitate nell'ordine nome, tipo e espressione di inizializzazione; quando invece si genera codice *C#* l'ordine di visita è tipo, nome e espressione di inizializzazione.

Capitolo 2

Cobol2Net

Cobol2Net è uno strumento per effettuare automaticamente la traduzione di programmi scritti in *COBOL* verso linguaggi dell'ambiente *.Net*, in particolare *VB.Net*.

Questo è stato realizzato per l'azienda informatica *Harvard Group s.r.l.* (in seguito abbreviato come *HG*), interessata ad effettuare il porting di uno dei suoi prodotti principali, costituito per lo più da software scritto in *COBOL*.

La necessita di tradurre il sistema deriva dal fatto che rimane tuttora utilizzato, e richiede quindi manutenzione, ma il numero di programmatori in grado di comprendere questo linguaggio è limitato. Allo stesso tempo però, date le grandi dimensioni del sistema, risulta improbabile riuscire a tradurlo manualmente, lasciando come unica opzione disponibile la traduzione automatica.

Cobol2Net è composto da tre parti principali:

- **Libreria di supporto**: fornisce supporto ai programmi tradotti. La principale funzione svolta da questa consiste nell'esporre una *API* (Application Programming Interface), che permette di simulare la gestione della memoria e l'utilizzo dei verbi *COBOL*[1]; questo ha permesso di semplificare sia la realizzazione del traduttore che la struttura dei programmi tradotti, che, altrimenti, sarebbero risultati estremamente complicati, rendendone difficile la comprensione. Molte delle interfacce esposte da questa libreria aderiscono al pattern *Fluent*[2].

[1] I verbi COBOL sono parole chiave utilizzate per richiedere l'esecuzione di una istruzione (*statement*). Alcuni esempi sono il verbo `ADD`, per aggiungere un valore ad una variabile e salvarne il risultato, o il verbo `PERFORM` che permette di esprimere un ciclo.

[2] Le interfacce fluenti[3] definiscono metodi pensati per essere concatenati, ed utilizza-

- **Modulo grafico**: si tratta del *porting*, effettuato manualmente, di *Visa*, il modulo grafico utilizzato dal sistema di *HG*. Fornisce le stesse funzionalità del modulo originale, ma è realizzato utilizzando i *Windows Forms*, producendo una resa grafica più accattivante, rispetto all'originale interfaccia a riga di comando.

- **Traduttore**: è la parte che si occupa di effettuare la traduzione dei sorgenti *COBOL*, descritto in dettaglio più avanti.

Il progetto è principalmente scritto in linguaggio *C#*, a parte per il modulo grafico realizzato utilizzando *VB.Net*

2.1 Traduttore

La traduzione di un programma *COBOL* avviene in due fasi: durante la prima fase viene effettuato il *parsing* del programma e ne viene costruito il relativo *Abstract Syntax Tree*[3], utilizzando CodeDOM per rappresentarne la struttura, che verrà poi utilizzato nella seconda fase per generare il codice nel linguaggio *.Net* desiderato.

La costruzione dell'*AST*, durante la prima fase, avviene utilizzando il pattern *Visitor*[4]: il *parser* genera un albero di oggetti *context* di vario tipo, noto come *parse tree*. Ogni oggetto *context* contiene informazioni riguardo ad uno specifico costrutto del *COBOL*; ognuno di questi oggetti sarà poi gestito da uno specifico *Visitor*, in maniera dipendente dal suo tipo e secondo un ordine specificato dal *Visitor* stesso, che si occupa di costruire l'*AST*.

In modo da consentire un semplice utilizzo del traduttore è anche presente una interfaccia a riga di comando.

2.2 Struttura dei programmi tradotti

I programmi tradotti hanno tutti una struttura simile: sono rappresentati da una singola classe, che eredita dalla classe astratta `AProgram` o, nel caso in cui un altro programma *COBOL* può richiederne l'esecuzione tramite il verbo `CALL`, da `ACallableProgram`.

---

no nomi tali che la narrazione ottenuta dalla concatenazione di questi risulti simile alla normale lingua parlata.

[3] Albero rappresentante la struttura del programma senza però esporre i dettagli della sintassi

La classe risulta poi divisa in due sezioni: una di dichiarazione, definizione ed eventuale inizializzazione delle variabili, e l'altra contenente la logica del programma.

La dichiarazione delle variabili è effettuata semplicemente definendo un campo per ciascuna, mentre la loro definizione ed eventuale inizializzazione è realizzata tramite chiamate di metodi ed accessi a proprietà concatenati (fluenti). Le istruzioni del programma invece sono tradotte quando possibile come costrutti del linguaggio *.Net* target (è il caso ad esempio di cicli), o altrimenti come invocazioni di metodi della libreria di supporto citata in precedenza.

Nel blocco di codice 2.1 è rappresentato un esempio di programma *COBOL* tradotto in *VB.Net*

[4]Nella `PROCEDURE DIVISION` sono presenti le istruzioni di un programma *COBOL*. Queste sono divise in sezioni e quest'ultime sono ulteriormente divise in paragrafi; questi sono analoghi a funzioni in *C*, in quanto forniscono un meccanismi per la separazione di un problema in più sotto-problemi e riuso del codice.

```
Namespace Generated
   <CobolProgramName("Barcode")>  _
   Public Class Barcode
      Inherits AProgram
      Implements IProgram

      Private bufferVisa As Reference(Of IAlphaNumeric)
      Private wInrec11 As Reference(Of IComposed)
      Private wi11SiglaBarcode As Reference(Of IAlphaNumeric)
      [...]

      Protected Overrides Sub WorkingStorage()
         NewAlphaNumeric(bufferVisa, 4000).Build
         NewCompound(wInrec11).Begin.NewCompound.Begin.[...]
         NewAlphaNumeric(wncSiglaBarcode, 4).Value("SIGL").Build
         [...]
      End Sub

      Private Sub EseguibileSection__MainModulo()
      begin:
         EseguibileSection__ApriVisa
         EseguibileSection__LoopProgramma
         [Call]("CLSVISA").Do
      End Sub

      Private Sub EseguibileSection__ApriVisa()
      begin:
         Move(1).To(inizioFin()).Do
         Move(23).To(righeFin()).Do
         Move("I").To(wAbilMkPrec()).Do
         [...]
      End Sub

      Private Sub EseguibileSection__LoopProgramma()
      begin:
         EseguibileSection__InizializzaFunzione

         Do While (fineFunzione().GetBoolean = false)
            EseguibileSection__LoopInput

         Loop
      End Sub

      [...]
```

Codice 2.1: Esempio di programma tradotto. Nel metodo `WorkingStorage` si possono vedere esempi di definizione e inizializzazione delle variabili. Negli altri metodi, corrispondenti a paragrafi della `PROCEDURE DIVISION`[4]nel programma originale, si può anche notare come alcuni verbi sono tradotti tramite invocazioni di metodo (nell'esempio `MOVE` e `CALL`), mentre altri tramite costrutti del linguaggio (il ciclo `While`, in origine `PERFORM`).

Capitolo 3

Requisiti

La principale motivazione dietro alla realizzazione di questa tesi è data dalla necessità di avere controllo sulla forma del codice. L'utilizzo di CodeDOM per la generazione del codice tradotto ha consentito di velocizzare lo sviluppo di Cobol2Net, ma allo stesso tempo il limitato livello di controllo fornito da questa libreria ha portato a delle problematiche che di fatto riducono l'impatto dei vantaggi offerti dalla traduzione.

Allo stato attuale, ad esempio, Cobol2Net è in grado di tradurre la maggior parte dei sorgenti *COBOL* consegnati da *HG*, ma sono frequenti casi in cui il codice generato, seppur funzionante, risulta di difficile lettura. Questo è dovuto al fatto che l'obiettivo principale di CodeDOM è fornire la possibilità di generare codice sorgente per il maggior numero possibile di linguaggi parte del *.Net Framework* partendo dallo stesso albero e per questo motivo il numero di costrutti supportati dalla libreria è molto limitato. Ad esempio non viene fornito supporto per rappresentare l'operatore booleano *not* e *cicli con controllo in coda*, in quanto è possibile ottenere comportamenti equivalenti sfruttando una combinazione di nodi già presenti, ma questo impatta negativamente sulla leggibilità del codice prodotto.

Avendo quindi il controllo sul codice generato si potrebbe migliorare la forma dei programmi tradotti, migliorando ulteriormente la loro manutenibilità, che, come detto in precedenza, è uno dei principali motivi dietro alla realizzazione di Cobol2Net.

Per poter stabilire al meglio i requisiti è stato necessario effettuare una fase iniziale di analisi della struttura e difetti di CodeDOM, i cui risultati sono stati riassunti nel capitolo 1. Da questa è anche emerso che CodeDOM non permette in alcun modo di estendere o modificare i generatori esistenti, rendendo quindi necessario realizzare un generatore completamente nuovo.

Di seguito sono evidenziati i requisiti emersi dall'analisi del sistema esistente e delle sue problematiche. Si è deciso prima di evidenziare i requisiti che impatteranno direttamente sul codice generato, e successivamente quelli relativi alla struttura del nuovo generatore.

3.1 Leggibilità del codice

Uno degli obiettivi principali è il miglioramento della leggibilità del codice tradotto. Definire in maniera oggettiva quando un sorgente sia più leggibile di un altro può risultare difficile, ma si sono individuati alcuni possibili cambiamenti che si può affermare, con buon margine di certezza, saranno considerati miglioramenti dai programmatori che lavoreranno sul codice tradotto.

Nel blocco di codice 3.1 è rappresentato un esempio di programma tradotto automaticamente, seguito dal risultato desiderato, in cui si può notare l'applicazione di alcuni dei miglioramenti citati.

Di seguito si descrivono nel dettaglio i miglioramenti individuati:

- **Definizione delle variabili *COBOL***: la definizione delle variabili *COBOL* nel programma tradotto avviene tutta su una riga, per mezzo di lunghe chiamate fluenti. Nel programma originale invece la definizione delle variabili più complesse avviene su più righe e su più livelli di indentazione; queste informazioni visive aiutano molto a capire la struttura delle variabili, ma nel caso dei programmi tradotti vengono perse completamente. Occorre quindi riuscire a generare il codice relativo alla definizione di variabile su più livelli, emulando l'indentazione presente nel programma originale.

- **Else if**: catene di `if` ... `else if` vengono generate su più livelli di indentazione; sarebbe invece opportuno generarle sullo stesso livello, eventualmente utilizzando l'apposita parola chiave `ElseIf`, nella generazione di codice *VB.Net*.

- **Do-while**: CodeDOM non supporta il costrutto *do-while*; per ovviare al problema, in fase di traduzione, eventuali cicli di questo tipo vengono gestiti come dei normali *while*, generando due volte le relative istruzioni, una prima del ciclo e una al suo interno. Sarebbe invece opportuno generare il codice utilizzando il costrutto *do-while*.

- **Not booleano**: CodeDOM non supporta l'operatore unario di *negazione booleana*. In fase di traduzione questo viene rappresentato come

```
1   If (mascheraInterrotta().GetBoolean = False) Then
2   If wol1gSiglaOp().IsDifferentFrom("AN") Then
3   If esisteRollback().GetBoolean Then
4   Move(0).To(wStatoPgm(), wStepPgm()).Do
5   Else
6   Move(1).To(wStepPgm()).Do
7   End If
8   Else
9   Move(0).To(wStatoPgm(), wStepPgm()).Do
10  End If
11  Else
12  If chiaveMenu().GetBoolean Then
13  Move(1).To(wFineFunzione()).Do
14  Else
15  If chiaveMkSucc().GetBoolean Then
16  Move(1).To(wStepPgm()).Do
17  Else
18  If chiaveRkPrec().GetBoolean Then
19  Move(0).To(wStatoPgm(), wStepPgm()).Do
20  Else
21  Move(-1).To(wStepPgm()).Do
22  End If
23  End If
24  End If
25  End If
```

```
1   If Not mascheraInterrotta().GetBoolean() Then
2   If wol1gSiglaOp().IsDifferentFrom("AN") Then
3   If esisteRollback().GetBoolean() Then
4   Move(0).To(wStatoPgm(), wStepPgm()).Do()
5   Else
6   Move(1).To(wStepPgm()).Do()
7   End If
8   Else
9   Move(0).To(wStatoPgm(), wStepPgm()).Do()
10  End If
11  ElseIf chiaveMenu().GetBoolean() Then
12  Move(1).To(wFineFunzione()).Do()
13  ElseIf chiaveMkSucc().GetBoolean() Then
14  Move(1).To(wStepPgm()).Do()
15  ElseIf chiaveRkPrec().GetBoolean() Then
16  Move(0).To(wStatoPgm(), wStepPgm()).Do()
17  Else
18  Move(-1).To(wStepPgm()).Do()
19  End If
```

Codice 3.1: Confronto tra il codice generato da CodeDOM (sopra) e il risultato desiderato (sotto), tratto da un programma *COBOL* tradotto. Si notino, nel blocco generato, le parentesi ridondanti e il confronto con `False` in riga 1, la catena di `Else If` su più livelli e l'assenza di parentesi nelle chiamate ai metodi *GetBoolean* e *Do*. La presenza delle parentesi in *mascheraInterrotta, wol1gSiglaOp, ecc.* è invece dovuta al fatto che queste non sono invocazioni di metodo, ma di delegati, e l'utilizzo di queste risulta necessario, in quanto cambia il significato dell'espressione.

un'operazione di confronto sull'uguaglianza tra la condizione da negare e il valore *false*. Sarebbe invece opportuno generare il codice utilizzando l'apposito operatore.

- **Parentesi ridondanti**: vengono generate molte parentesi ridondanti, soprattutto quando si generano operazioni binarie; queste potrebbero risultare utili quando si ha un'espressione lunga o complicata, ma sono spesso presenti casi in cui la superfluità di queste risulta molto evidente; sarebbe quindi opportuno evitare la generazione delle parentesi in queste situazioni.

- **Metodi senza argomenti in VB.Net**: in *VB.Net* risulta possibile chiamare metodi privi di argomenti senza includere una coppia di parentesi vuote. CodeDOM evita di includerle quando possibile, rendendo difficile comprendere se un'espressione rappresenta un accesso a proprietà o invocazione di metodo, si preferisce quindi aggiungere sempre le parentesi in caso di invocazione di metodo.

3.2 Configurabilità dello stile del codice

Sono stati individuati altri possibili cambiamenti, che, a differenza di quelli relativi alla leggibilità del codice, non possono essere classificabili come miglioramenti, in quanto si tratta di scelte stilistiche soggettive. Sarà quindi necessario aggiungere delle opzioni, che permettano di modificare lo stile del codice finale. Queste saranno:

- **Stringa di indentazione**: permette di specificare quale stringa utilizzare per l'indentazione

- **Utilizzo del nome completo di tipi**: permette di specificare se utilizzare sempre il nome completo dei tipi, ossia includendone il *namespace*, oppure se analizzare le informazioni sugli *import* e *namespace* corrente per dedurre quando è invece possibile utilizzarne solo il nome.

- **Scelte specifiche per VB.Net**: visual basic fornisce spesso più modi equivalenti di scrivere la stessa istruzione, consentendo al programmatore di decidere quale utilizzare. Occorre quindi consentire all'utilizzatore del traduttore di specificare come queste vadano generate. Di

seguito sono riportati alcuni esempi:

`While` *`condition`*		`Do While` *`condition`*
...	*risulta equivalente a*	...
`End While`		`Loop`
`If` *`condition`* `Then`		`If` *`condition`*
...	*risulta equivalente a*	...
`End If`		`End If`
`While Not` *`condition`*		`Do Until` *`condition`*
...	*risulta equivalente a*	...
`End While`		`Loop`

3.3 Maggior numero di sorgenti traducibili

Esistono alcuni casi in cui i sorgenti vengono tradotti, ma il codice risultante richiede piccoli interventi manuali prima di poter essere compilato. Occorre verificare se è possibile evitare questi problemi durante la fase di generazione

Un esempio è dato dal caso in cui un'espressione risulta troppo lunga, e non può essere compilata. Come detto in precedenza, la definizione delle variabili *COBOL* è realizzata tramite chiamate concatenate di metodi. Esistono casi, seppur rari, in cui la definizione di una variabile composta risulta talmente lunga da causare il suddetto problema; si può risolvere in maniera semplice spezzando la catena di chiamate, memorizzando il risultato della prima parte in una variabile d'appoggio, ed eseguendo la seconda parte di chiamate su questa variabile. Effettuare tale operazione in fase costruzione dell'albero però risulta particolarmente complicato, ma effettuarla durante la generazione potrebbe risultare più semplice.

3.4 Compatibilità con CodeDOM

CodeDOM è una libreria molto stabile, inclusa nella libreria standard del *.Net framework* sin dalle prime versioni e pertanto non fa uso dei costrutti più nuovi dei linguaggi *.Net*, cosa che lo rende spesso verboso da utilizzare. Si è quindi valutato se fosse opportuno realizzare il nuovo generatore di codice in maniera completamente indipendente da CodeDOM, reimplementando quindi i nodi, con l'obiettivo di semplificarne l'utilizzo.

Questo però avrebbe richiesto di effettuare molti cambiamenti in Cobol2Net, mentre si potrebbero ottenere vantaggi simili a quelli precedentemente citati realizzando una libreria di supporto, che fornisca per esempio

metodi di estensione[1] o *factory*. Si è quindi deciso che il nuovo generatore dovrà essere compatibile con CodeDOM; questo significa che dovrà poter generare il codice relativo ad un qualunque tipo di grafo CodeDOM valido, producendo un risultato equivalente a quello prodotto dalla libreria originale.

3.5 Generazione personalizzabile

Il codice generato da CodeDOM è per la maggior parte non solo funzionante, ma anche ben leggibile; le parti che si vorrebbero modificare sono solo una piccola percentuale dell'intero generatore. Come detto in precedenza però la libreria non fornisce alcun metodo per modificarne parzialmente il comportamento, o, più semplicemente, per definire un nuovo nodo, specificando come poi debba essere gestito da un determinato generatore. Questo porterà quindi a spendere tempo per reimplementare feature già esistenti, in quanto il nuovo generatore avrà un comportamento identico a quello del vecchio nella maggior parte dei casi.

Se si realizzasse la nuova libreria in maniera analoga ai generatori di CodeDOM, il problema si potrebbe riproporre in futuro; sarà quindi opportuno che questa fornisca un metodo per modificare in maniera semplice la gestione di un singolo nodo, senza costringere l'utilizzatore a ridefinire anche il modo in cui altri nodi siano gestiti nè costringendo ad operare direttamente sul codice sorgente del generatore.

[1] I metodi di estensione[2] forniscono un modo di "aggiungere" metodi ad una classe senza modificarne la definizione. Si tratta in realtà di metodi statici che prendono in input un'istanza della classe estesa, ma possono anche essere direttamente invocati su di questa.

Capitolo 4

Progettazione

Dall'analisi del sistema, discussa nel capitolo precedente, è emerso che sarà necessario realizzare un generatore di codice personalizzabile; tale strumento potrebbe risultare utile in ulteriori progetti, non necessariamente collegati a Cobol2Net. Si è quindi deciso di realizzare una libreria, denominata *CodeDomExt*, in cui saranno implementati tutti gli aspetti non direttamente collegati a Cobol2Net. Il linguaggio di programmazione che verrà utilizzato è C#.

In seguito vengono discusse le scelte progettuali effettuate relativamente a *CodeDomExt*

4.1 Struttura generale

Il nucleo di CodeDomExt, discusso nel dettaglio più avanti, è formato dalle classi che permettono di definire un generatore di codice personalizzabile, definite nel *namespace* `CodeDomExt.Generators`. All'interno di questo sono anche presenti le implementazioni di generatori per *VB.Net* e *C#*, in quanto si tratta dei possibili linguaggi target per il traduttore di Cobol2Net.

Sono stati anche realizzati nuovi nodi, contenuti nel namespace `CodeDomExt.Nodes`, con l'obiettivo di rimediare alla limitata espressività di CodeDOM fornendo una più ampia disponibilità di costrutti; in particolare si sono realizzati nodi per esprimere cicli con controllo in coda e l'operatore di negazione booleana.

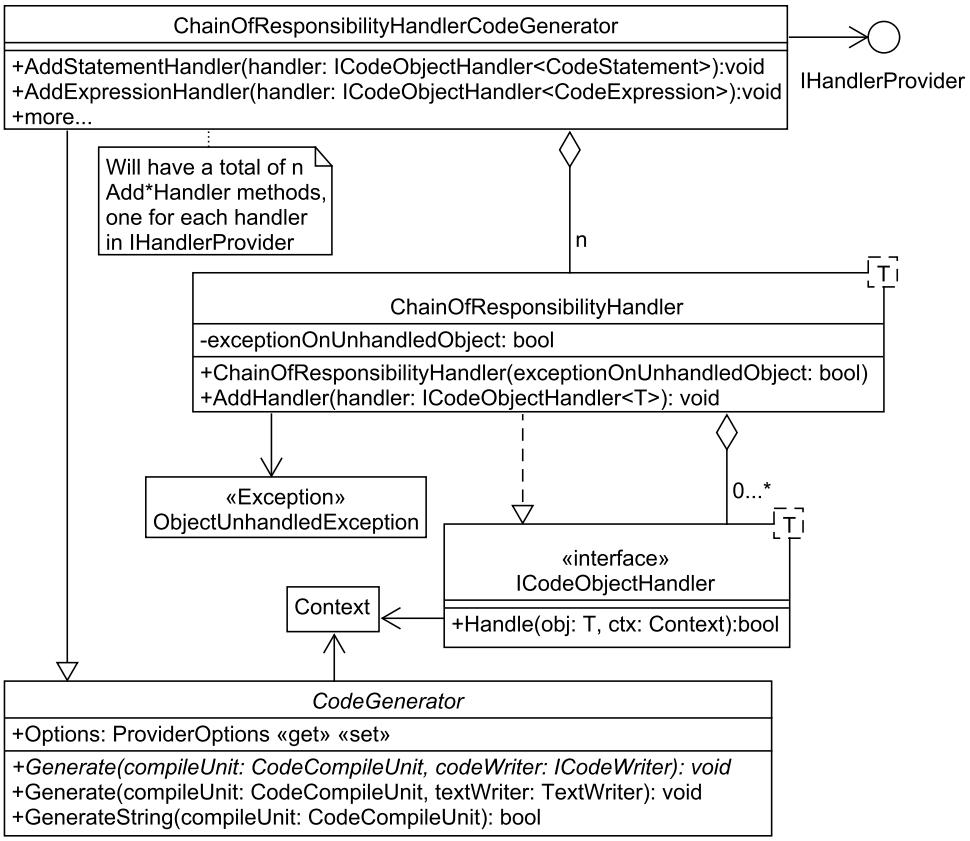

Figura 4.1: Diagramma UML raffigurante le principali classi della libreria. `IHandlerProvider` e `Context` sono meglio definite più avanti

4.2 Generatori

Di seguito sono evidenziati i principali elementi del *namespace* `CodeDomExt.Generators`, fornendo dettagli sul loro ruolo nella libreria e le motivazioni che hanno portato alla loro realizzazione.

4.2.1 Interfaccia `ICodeObjectHandler<T>`

L'elemento principale della libreria è l'interfaccia generica `ICodeObjectHandler<T>`[1], in cui `T`, in genere, rappresenta un nodo del grafo CodeDOM. Le classi che implementano questa interfaccia sono in grado di *gestire* oggetti

[1] Le naming convention del *.Net framework* richiedono che i nomi delle interfacce inizino con una `I`

di tipo `T` o suoi sottotipi, ossia possono generare il codice relativo ad una specifica istanza di questo, tramite il metodo `Handle`, restituendo infine un valore *booleano* che indica se l'oggetto ricevuto è stato gestito con successo. Il metodo `Handle` richiede in input, oltre all'oggetto da gestire, un oggetto di tipo `Context`. Questo contiene informazioni che vanno condivise tra i vari *handler*, quali ad esempio il livello di indentazione corrente o opzioni del generatore, meglio definite in fase di implementazione; tali classi sono rappresentate più dettagliatamente in figura 4.2.

Durante l'*handling* di un nodo capita spesso di dover anche gestire eventuali nodi figli, il cui tipo è conosciuto e risulta spesso diverso dal tipo del nodo padre; un esempio è il nodo `CodeContitionStatement` il cui figlio `TestExpression` sarà di tipo `CodeExpression`. L'utilizzo di `ICodeObject-Handler<T>` ha permesso di creare più *handler*, uno per ogni tipo principale di nodo, così se un *handler* dovesse gestire un nodo figlio di tipo diverso può delegarne la gestione ad un altro *handler*; per agevolare questo il `Context` conterrà anche dei riferimenti ai vari *handler*.

Il risultato di questo è che, scegliendo in maniera appropriata i parametri `T`, il codice risulterà ben separato e ciascun *handler* concreto gestirà in maniera simile eventuali sottotipi di `T` al suo interno, ma al tempo stesso la gestione di questi risulterà molto diversa rispetto a quella di altri tipi, effettuata in *handler* differenti. Inoltre questa separazione permette di ridefinire in maniera semplice come gestire determinati nodi; se, per esempio, si volesse ridefinire la gestione di un nodo `CodeNamespaceImport` basterà creare un nuovo *handler* per questo tipo e sostituirlo a quello presente nel `Context`. Di fatto l'utilizzo dell'interfaccia `ICodeObjectHandler` consente di definire a tempo di esecuzione la *strategia* per la gestione dei nodi.

4.2.2 Classe `ChainOfResponsibilityHandler<T>`

Il grado di controllo fornito dalla separazione in `ICodeObjectHandler<T>` non risulta però abbastanza fine da poter considerare il requisito relativo al "Generatore di codice personalizzabile" soddisfatto. Nel caso, per esempio, si volesse modificare l'*handling* di un nodo sottotipo di `CodeExpression`, se si utilizzasse un singolo *handler* per tutte queste, bisognerebbe ridefinire anche la gestione delle altre *espressioni*.

Una possibile soluzione per questo sarebbe la realizzazione un *handler* per ogni sottotipo di `CodeExpression`, ma, considerando che spesso di un determinato nodo si sa solo il gruppo di appartenenza e non il tipo specifico, occorrerebbe anche avere un *handler* per una generica `CodeExpression`, che ne analizza il tipo effettivo e ne delega la *gestione* all'*handler* appropriato. Questa soluzione però risulta problematica nel caso in cui si volesse utiliz-

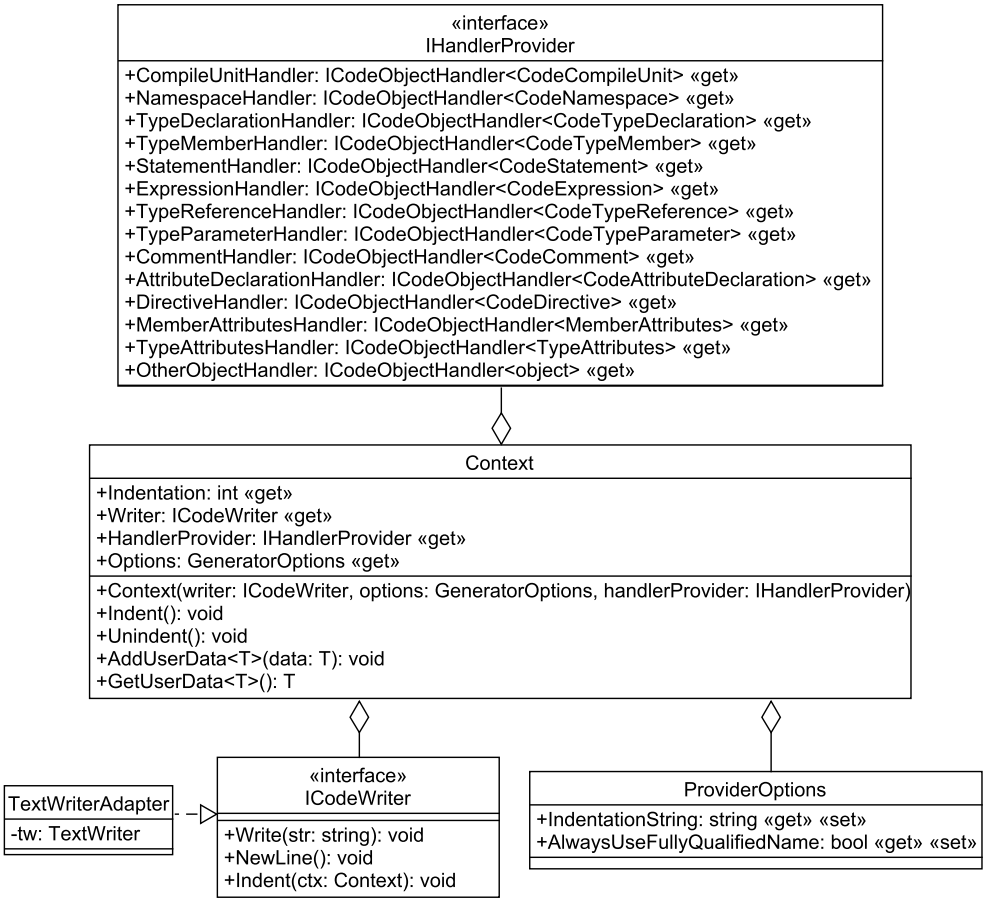

Figura 4.2: Diagramma UML raffigurante la classe `Context` e i suoi elementi individuati in fase di progettazione. Altre eventuali informazioni da condividere tra gli handler saranno individuate e aggiunte al context a seconda delle necessità in fase di implementazione.
È anche rappresentata la classe `TextWriterAdapter`, che fa da *wrapper* ad un `TextWriter` per consentirne l'utilizzo come `ICodeWriter`.
Si sono inoltre realizzati metodi per gestire dati utente, in modo da fornire all'utilizzatore della libreria la possibilità di definire ulteriori informazioni da condividere tra le proprie implementazioni di *handler*

zare il nuovo *handler* solo se le proprietà dell'oggetto da gestire rispettano determinate condizioni, o anche se si volesse aggiungere un nuovo tipo di nodo, in quanto risulterebbe necessario modificare anche l'*handler* relativo alla generica `CodeExpression`.

Si sono quindi analizzate varie possibili soluzioni, e quella che si è ritenuta più appropriata è stata la realizzazione della classe generica `ChainOfResponsibilityHandler<T>`, che implementa l'interfaccia `ICodeObjectHandler<T>`. Questa classe, come suggerito dal nome, fa uso del pattern *Chain-of-Responsibility*[4] , sebbene siano presenti alcune variazioni rispetto alle implementazioni tradizionali, realizzate con l'obiettivo di aumentarne la flessibilità.

Tipicamente questo pattern viene utilizzato in situazioni in cui non si conosce a priori come debba venire gestita una determinata richiesta, sia nel caso in cui il gestore non è effettivamente disponibile a tempo di compilazione che nel caso in cui questo debba cambiare durante l'esecuzione. Viene quindi utilizzata una *catena* di oggetti *handler* che, in ordine, valutano sulla base di determinate condizioni se gestire la richiesta o inoltrarla al successivo *handler* (in alcuni casi la richiesta viene inoltrata anche se è stata gestita). Queste condizioni, nel caso specifico di *CodeDomExt*, potrebbero per esempio essere basate su quale sia l'effettivo tipo di `T`, nel caso questo fosse un gruppo di nodi (ad esempio `CodeStatement`), o il valore di determinate proprietà dell'istanza corrente.

Una tipica implementazione di questo pattern fa uso di una classe astratta, contenente un campo che indica l'*handler successore*, dalla quale derivano tutte le classi *handler* concrete. Un esempio di questo è rappresentato dalla figura 4.3 e nel blocco di codice 4.1.

Il `ChainOfResponsibilityHandler<T>`, a differenza delle tipiche implementazioni, fa invece uso di una collezione di `ICodeObjectHandler<T>` e delega l'esecuzione del suo metodo `Handle` a questi, richiamandoli uno per uno in ordine, a partire dall'ultimo aggiunto, finché l'oggetto ricevuto non viene gestito, ossia fino a che il metodo `Handle` dell'`ICodeObjectHandler<T>` corrente non restituisce `true`; questo processo è illustrato in dettaglio nella figura 4.4. Inoltre, per poter aderire all'interfaccia implementata, il metodo `Handle` deve restituire un valore che indica se l'oggetto è stato gestito con successo, cosa che tipicamente non avviene nel pattern *Chain-Of-Responsibility*. Il vantaggio principale derivato da questa variante è che gli *handler* dovranno semplicemente implementare l'interfaccia `ICodeObjectHandler<T>`, senza dover ereditare da una specifica classe astratta, lasciando più libertà nella loro realizzazione; inoltre risulta più semplice gestire l'aggiunta di nuovi *handler*.

I `ChainOfResponsibilityHandler<T>` forniscono anche una opzione per specificare se debbano lanciare una eccezione qualora non siano stati in grado di gestire l'oggetto richiesto. Senza di questa eventuali nodi sconosciuti

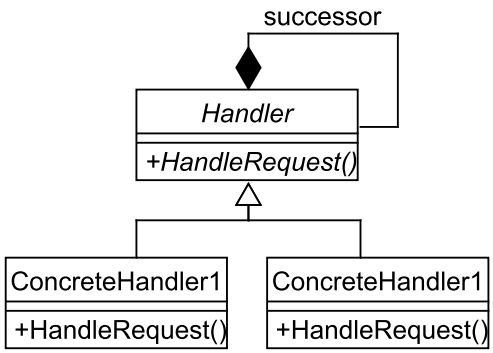

Figura 4.3: Esempio di diagramma delle classi per il pattern Chain-Of-Responsibility

```
1  public abstract class ChainOfResponsibilityHandler<T> : ICodeObjectHandler<
       T>
2  {
3     public ChainOfResponsibilityHandler<T> Successor { get; set; }
4
5     protected abstract bool ConcreteHandle(T obj, Context ctx);
6
7     public bool Handle(T obj, Context ctx)
8     {
9        if (ConcreteHandle(obj, ctx))
10       {
11          return true;
12       }
13       if (Successor != null)
14       {
15          return Successor.Handle(obj, ctx);
16       }
17       return false;
18    }
19 }
```

Codice 4.1: Codice di esempio per la classe `ChainOfResponsibilityHandler<T>` nel caso si fosse deciso di seguire un approccio più simile a quello tipicamente utilizzato per il pattern *Chain-Of-Responsibility*

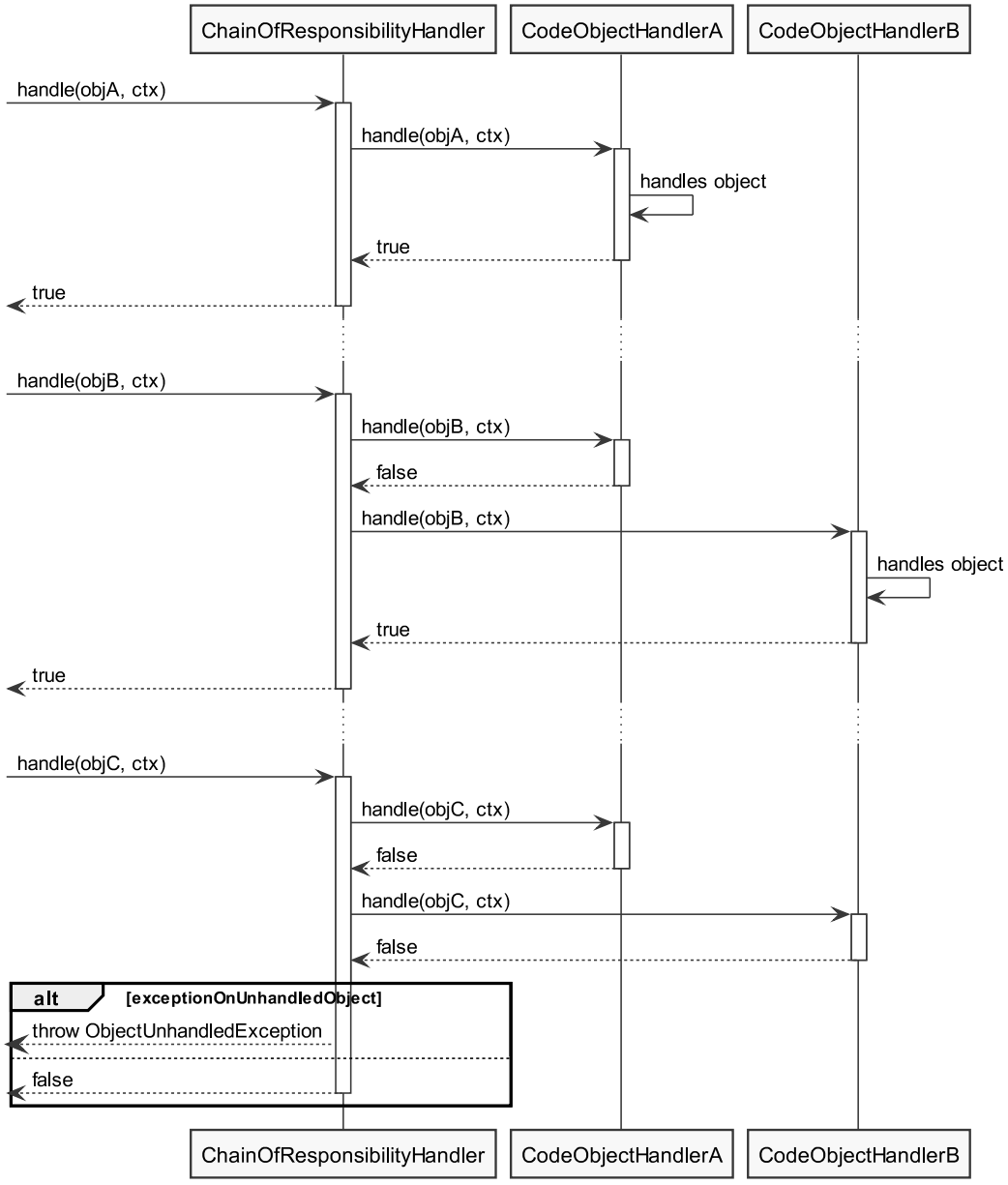

Figura 4.4: Diagramma di sequenza rappresentante l'esecuzione del metodo `Handle` in un `ChainOfResponsibilityHandler`, contenente un `CodeObjectHandlerA` in grado di gestire oggetti di tipo `A`, ed un `CodeObjectHandlerB` in grado di gestire oggetti di tipo `A` o `B`.

verrebbero silenziosamente ignorati in fase di generazione, dando la falsa illusione che il codice sia stato generato con successo. Allo stesso tempo si è deciso di non forzare questo comportamento, in modo da consentire di annidare più *handler* di questo tipo.

Sfruttando questa classe risulterà semplice modificare l'*handling* dei nodi anche nei casi citati in precedenza. Se si volesse, per esempio, modificare il modo in cui un determinato sottotipo di `CodeExpression` viene gestito basta creare un nuovo `ICodeObjectHandler<CodeExpression>` in grado di gestire solamente tale nodo ed aggiungerlo al `ChainOfResponsibilityHandler<CodeExpression>`. In questo modo il nodo del tipo desiderato verrà gestito dal nuovo *handler*, mentre altri tipi di nodi continuano ad essere gestiti dai precedenti.

4.2.3 Classe `CodeGenerator`

La classe astratta `CodeGenerator` fornisce la base per la realizzazione dell'effettivo generatore di codice. Questa fornisce lo scheletro di un metodo `Generate` per l'avvio della generazione a partire da una `CodeCompileUnit` e un'istanza di un `ICodeWriter`, che specifica come scrivere in *output* il codice generato. Implementazioni concrete di questa dovranno occuparsi di preparare un'istanza `Context` appropriata e avviare la generazione del codice.

Una implementazione di `CodeGenerator` è data dalla classe `ChainOfResponsibilityHandlerCodeGenerator` che fa uso di `ChainOfResponsibilityHandler` per realizzare un generatore personalizzabile. I generatori di codice per i linguaggi *C#* e *VB.Net* sono realizzati come semplici istanze di questa classe a cui vengono poi aggiunti appropriati *handler*, e sono ottenibili tramite la classe statica[2] `CodeGeneratorFactory`.

4.2.4 Interfaccia `IHandlerProvider`

L'interfaccia `IHandlerProvider` fornisce un punto d'accesso agli *handler* di un generatore, ed è utilizzata da questi quando devono delegare la gestione di un nodo. Questa definisce quali *handler* dovranno essere realizzati da un'implementazione completa di un generatore di codice per uno specifico linguaggio. Tali *handler* sono accessibili tramite proprietà dell'interfaccia, e sono elencati in figura 4.2.

Gli *handler* presenti, ad esclusione degli ultimi tre, sono relativi a nodi del grafo CodeDOM.

[2] In C# una classe statica è una classe non istanziabile contenente solo membri statici

`TypeAttributes` e `MemberAttributes`, invece, sono presenti nel grafo come proprietà di `CodeTypeDeclaration` e `CodeTypeMember` rispettivamente. Si è deciso di inserirli in quanto la loro gestione non è immediata, e la loro presenza agevola eventuali modifiche alla gestione di attributi di *tipi* o *membri*.

È inoltre presente un *handler* di *object*, per consentire l'aggiunta in futuro di nuovi *handler* per tipi realizzati dall'utente non derivati da quelli precedentemente elencati.

4.2.5 Handlers per gruppi di nodi

A questo punto della progettazione si sono potute individuare due categorie principali di *handler*, suddivisi tra quelli che dovranno gestire singoli nodi, come ad esempio `CodeCompileUnit` o `CodeNamespace`, oppure interi gruppi, come nel caso di `CodeStatement` o `CodeExpression`.

L'implementazione di *handler* appartenenti alla prima categoria non richiede l'utilizzo di strategie particolari; al contrario per la gestione dei gruppi di nodi occorre decidere quale tattica utilizzare per determinare come vada *gestito* uno specifico nodo sulla base del suo sottotipo.

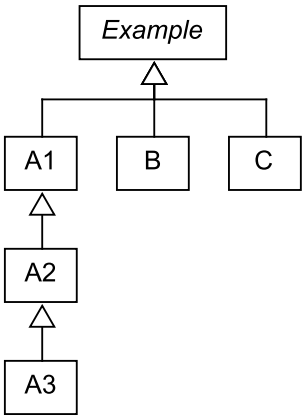

Figura 4.5: Classi d'esempio per un handler di gruppo

Una possibilità è data dall'utilizzo di una catena di *if-else* valutando condizioni sul tipo del nodo da generare utilizzando l'operatore `is`[3]. Questo metodo è apparentemente semplice da realizzare, ma richiede particolare attenzione all'ordine con cui vengono valutate le condizioni, in quanto potrebbe cambiare il risultato dell'operazione.

[3] *`expression`* `is` *`type`*: valuta se *expression* può essere convertita nel tipo *type*

Si considerino per esempio le classi in figura 4.5 nel caso in cui si voglia costruire un *handler* per `Example`; se si volesse gestire la classe `A3` in maniera diversa da `A1` bisogna prestare attenzione a valutare la condizione `obj is A3` prima di `obj is A1`, in quanto quest'ultima risulta vera anche qualora `obj` sia un sottotipo di `A1`.

Una possibile alternativa consiste nel valutare il tipo utilizzando l'operatore di uguaglianza, ma verrebbero così introdotti problemi diversi: se per esempio si vuole che `A2` venga gestito come `A1` bisogna introdurre una condizione anche per questo; inoltre non si sarebbe in grado di gestire eventuali sottotipi di `A1` definiti dall'utilizzatore della libreria.

Una soluzione di questo tipo quindi risulterebbe, seppur funzionante, molto fragile, in quanto eventuali cambiamenti alla gerarchia delle classi da gestire richiederebbe interventi sull'*handler*.

La soluzione che si è deciso di adottare invece è l'utilizzo del pattern *Visitor*, già citato in precedenza. Tale pattern utilizza un'interfaccia `Visitor`, in questo caso corrispondente all'*handler*, che dichiara vari metodi per effettuare la *visita* di istanze del tipo `Element`, corrispondenti ai *nodi*, e fornisce un semplice modo per richiamare il più *appropriato* di questi metodi, ossia il metodo più specifico per il tipo effettivo di `Element`. Se per esempio (considerando le classi in figura 4.5) si avessero solamente metodi per la visita oggetti di tipo `A1` e `A3`, istanze di `A2` verrebbero visitate dal primo di questi metodi, mentre istanze di `A3` dal secondo.

Tipiche implementazioni di questo pattern richiedono che gli `Element` implementino il metodo `Accept`, quindi non sono applicabili a *CodeDomExt*, in quanto non è possibile modificare le implementazioni dei nodi; inoltre la necessità di esprimere i metodi per la visita nell'interfaccia del `Visitor` limita l'aggiunta di nuovi nodi.

In *C#* però si può sfruttare il tipo `dynamic` per realizzare un'implementazione del pattern che non pone alcuna restrizione sugli elementi visitati né, di conseguenza, richiede l'esposizione dei metodi per la visita; un'esempio di applicazione di questo pattern è rappresentato nel blocco di codice 4.3. Questo può essere realizzato in quanto l'invocazione di un metodo passando come argomento un oggetto di tipo `dynamic` anziché il tipo richiesto compila indipendentemente da quale sia quest'ultimo; a tempo di esecuzione poi viene richiamato il metodo più appropriato per il tipo effettivo di questo, secondo i criteri specificati in precedenza, e nel caso non sia presente nessun metodo valido viene generata una eccezione di tipo `RuntimeBinderException`. Riprendendo l'esempio precedente, in cui sono presenti solo metodi per la visita di `A1` e `A3`, il tentativo di visitare un elemento di tipo `B` risulterà nel lancio di tale eccezione.

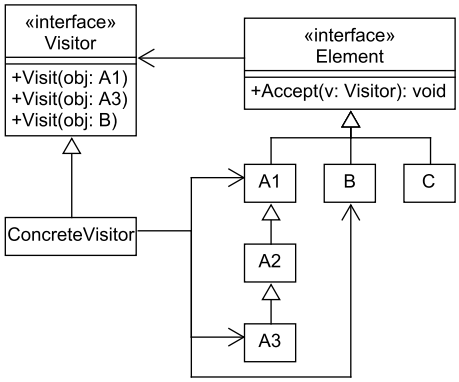

Figura 4.6: Esempio di diagramma delle classi per il pattern Visitor

Codice 4.2: Implementazione dei metodi `Accept`; i commenti indicano il metodo di `Visitor` effettivamente chiamato

```
1  public class A1 : Element
2  {
3      public void Accept(Visitor v)
4      {
5          v.Visit(this); //visit A1
6      }
7  }
8  public class A2 : A1
9  {
10     public void Accept(Visitor v)
11     {
12         v.Visit(this); //visit A1
13     }
14 }
15 public class A3 : A2
16 {
17     public void Accept(Visitor v)
18     {
19         v.Visit(this); //visit A3
20     }
21 }
22 public class B : Element
23 {
24     public void Accept(Visitor v)
25     {
26         v.Visit(this); //visit B
27     }
28 }
29 public class C : Element
30 {
31     public void Accept(Visitor v)
32     {
33         v.Visit(this); //compile
               error
34     }
35 }
```

```
1  public class ExampleHandler : ICodeObjectHandler<Example>
2  {
3     public bool Handle(Example obj, Context ctx)
4     {
5        try
6        {
7           //Invokes visit method, throws RuntimeBinderException when obj is
                C
8           return DynamicHandle(obj as dynamic, ctx);
9        }
10       catch (RuntimeBinderException)
11       {
12          return false;
13       }
14    }
15    //Visit methods
16    //Handles A1 and A2
17    private bool DynamicHandle(A1 obj, Context ctx) { [...] }
18    //Handles A3
19    private bool DynamicHandle(A3 obj, Context ctx) { [...] }
20    //Handles B
21    private bool DynamicHandle(B obj, Context ctx) { [...] }
22 }
```

Codice 4.3: Esempio di possibile implementazione per un *handler* delle classi rappresentate in figura 4.5, realizzato come dynamic visitor

4.3 Costrutti

Il soddisfacimento di alcuni dei requisiti, in particolare quelli relativi alla leggibilità del codice, richiede che CodeDomExt sia in grado di generare nuovi costrutti. Per fare questo si sono individuate più possibili strategie, spiegate di seguito; ciascun costrutto verrà poi implementato utilizzando una di queste.

4.3.1 Nuovi nodi

A differenza di CodeDOM la nuova libreria mira a generare codice per *C#* e *VB.Net*, e non l'intero set di linguaggi del *.Net framework*; per questo motivo un'opzione valida è l'aggiunta di nuovi tipi di nodi, a patto che questi possano essere usati per generare costrutti equivalenti in entrambi i linguaggi sopracitati. Un'altra caratteristica che devono rispettare i nodi realizzati è di essere semplici contenitori privi di logica, in modo da replicare struttura e funzionamento dei nodi già esistenti; inoltre, per permetterne l'immediato riconoscimento come nodi CodeDOM, il loro nome inizierà con il prefisso *Code*, come avviene nella libreria originale.

Questa strategia può essere utilizzata per implementare la generazione di costrutti completamente assenti in CodeDOM, come per esempio la negazione

booleana.

4.3.2 Estensione di nodi esistenti

In alcuni casi il supporto fornito da CodeDOM per la generazione di alcuni costrutti è presente, ma non consente di esprimere tutte le possibili variazioni di questi. Un esempio sono gli *import statici*: è presente il nodo `CodeNamespaceImport` per poter rappresentare appunto un *import*, ma non permette di rappresentarne la versione *statica*.

Si è quindi valutata l'idea di aggiungere il supporto sfruttando la proprietà *UserData*, presente in tutti i nodi, in quanto definita da `CodeObject`, che fornisce un dizionario in cui l'utente può salvare informazioni sui nodi non espresse dalle loro proprietà. Proseguendo per questa strada però, l'unico modo di indicare la presenza di tali cambiamenti sarebbe stato tramite documentazione esterna, rischiando che questi passino inosservati; inoltre l'utilizzo delle nuove funzionalità risulterebbe poco intuitivo.

Per questo motivo si è deciso di estendere i nodi originali, implementando il supporto per varianti non esprimibili con le versioni originali nelle classi derivate. Per distinguere queste ultime dalla versione presente in CodeDOM si utilizza il suffisso *Ext*.

4.3.3 Valutazioni in fase di generazione

L'ultima strategia individuata per la generazione di nuovi costrutti non richiede la realizzazione di nuove classi, ma consiste nell'analizzare nodi esistenti per valutare se possono essere generati in modo diverso.

Un esempio è dato dal costrutto *if-else*, rappresentato dalla classe `CodeConditionStatement`: se la proprietà `ElseStatements` di questo nodo, contenente le istruzioni da eseguire nel blocco *else*, contiene un singolo nodo e questo è un altro `CodeConditionStatement`, si può generare il blocco come un *elseif-else* anziché come un nuovo *if-else* contenuto all'interno del blocco else appartenente al nodo padre.

Capitolo 5

Implementazione

Data la natura del progetto, la sua implementazione è avvenuta in due fasi: una prima fase di realizzazione della libreria CodeDomExt e una seconda fase in cui questa è stata sostituita a CodeDOM nel progetto Cobol2Net. Ciascuna di queste può essere poi considerata suddivisa in due sotto-fasi risultando nelle seguenti:

- Implementazione delle classi base di CodeDomExt
- Implementazione handlers per C# e VB.Net
- Sostituzione di CodeDOM in Cobol2Net
- Implementazione di funzionalità del generatore specifiche per Cobol2Net

In seguito sono evidenziate le scelte di basso livello effettuate relativamente a queste fasi.

5.1 Classi base di CodeDomExt

In questa fase si sono implementate le classi relative ai generatori individuate in fase di progettazione ed i nuovi nodi per il grafo CodeDOM. Questa fase è stata breve, in quanto la logica di funzionamento delle classi da implementate è molto semplice, e i dettagli di basso livello da definire per queste sono pochi. In seguito si evidenziano i dettagli di alcune delle implementazioni citate.

5.1.1 Dati utente di `Context`

Per l'implementazione dei dati utente nella classe `Context` si è deciso di utilizzare un `Dictionary`[1] che utilizza come *chiave* il nome completo del tipo di

[1] Implementazione di *array associativo* nella libreria standard del *.Net framework*

dato aggiunto dall'utente. Un utente che volesse aggiungere nuove informazioni al `Context` dovrà preparare una classe che le contenga, ed aggiungere questa al context, anziché i singoli dati. Si è deciso di utilizzare questa strategia, piuttosto che l'assegnamento di un nome ad ogni informazione, per limitare il numero di possibili collisioni nel caso in cui l'aggiunta di dati venga effettuata da più utenti diversi.

Per semplificare l'aggiunta di dati utente si è in seguito deciso che il generatore dovrà poter accettare ulteriori oggetti, nei suoi metodi `Generate`, che rappresentano appunto dati utente da aggiungere al `Context` prima di iniziare la generazione.

5.1.2 Generatore

Il generatore di codice si deve occupare di preparare un'istanza di `Context` da passare all'*handler* di `CodeCompileUnit` per avviare la generazione; per costruire un'istanza di tale classe deve fornire oggetti di tipo `GeneratorOptions`, `ICodeWriter` e `IHandlerProvider`. I primi due sono forniti al generatore dall'esterno, ma il terzo deve essere specificato dal generatore stesso.

Nel caso del `ChainOfResponsibilityHandlerCodeGenerator` si è deciso per semplicità di implementare direttamente in questo l'interfaccia `IHandlerProvider`, e quindi passerà un riferimento a sè stesso per creare l'istanza di `Context`.

5.1.3 Nodi

Come deciso in fase di progettazione i nodi sono stati realizzati come dei contenitori privi di logica; la loro implementazione quindi è risultata molto semplice, trattandosi principalmente di definire proprietà ed eventuali costruttori per semplificarne l'utilizzo.

Anche in questo caso è stato però necessario effettuare alcune scelte, in particolare si è dovuto scegliere come rappresentare *collezioni* di nodi: CodeDOM non fa uso di classi generiche, in quanto realizzato prima dell'implementazione di queste nel *.Net framework*, ma utilizza classi `Collection` definite ad-hoc; si è quindi dovuto scegliere se, quando possibile, continuare ad utilizzare queste anche nelle nuove implementazioni di nodi, o sfruttare le collezioni generiche. Dopo aver valutato vantaggi e svantaggi di entrambe si è deciso di proseguire con la prima opzione, in modo da consentire di sfruttare eventuali funzioni di utilità realizzate per le collezioni di CodeDOM

```
1  public class CodePostTestIterationStatement : CodeStatement
2  {
3      public CodeStatementCollection Statements { get; }
4      public CodeExpression TestExpression { get; set; }
5
6      public CodePostTestIterationStatement()
7      {
8          Statements = new CodeStatementCollection();
9      }
10     public CodePostTestIterationStatement(CodeExpression testExpression,
           params CodeStatement[] statements)
11     {
12         TestExpression = testExpression;
13         Statements = new CodeStatementCollection(statements);
14     }
15 }
```

Codice 5.1: Implementazione del nodo utilizzato per rappresentare il costrutto *do-while*

anche con i nuovi nodi; inoltre effettuare la conversione da una di queste a collezione generica risulta più semplice del procedimento inverso.[2]

5.2 Handlers per VB.Net e C#

In questa fase si sono effettivamente implementati i generatori di codice per *C#* e *VB.Net*, in quanto nella fase precedente si è solo implementata la struttura per un generico generatore. Le scelte e considerazioni evidenziate di seguito sono state effettuate nell'interesse di questi specifici linguaggi, e potrebbero non essere valide nel caso si volesse realizzare un generatore per un linguaggio diverso.

Gli *handlers* per *VB.Net* e *C#* sono contenuti rispettivamente nei namespace `CodeDomExt.Generators.VisualBasic` e `CodeDomExt.Generators.CSharp`

5.2.1 Formattazione del codice

Per far sì che il codice risultante venga formattato correttamente su più linee, occorre definire il modo in cui gli *handlers* devono gestire indentazione e nuove righe. La strategia generale che si è deciso di seguire è di lasciarne la gestione al livello superiore; questo significa gli *handler* non dovranno

[2] È possibile utilizzare il metodo generico `Cast<T>` per ottenere un `IEnumerable<T>` dalle collezioni non generiche, che potrà essere utilizzato per costruire il tipo di collezione desiderato.

occuparsi di effettuare l'indentazione prima della gestione del nodo, nè di generare una nuova riga dopo di questa, in quanto sarà l'*handler* che ha effettuato la richiesta ad occuparsene.

5.2.2 Context

Durante l'implementazione di alcuni *handler* si è individuata la necessità di condividere ulteriori informazioni, oltre a quelle già considerate in fase di progettazione, che sono state aggiunte alla classe `Context`.

- `TypeDeclarationStack`: in alcuni casi durante l'*handling* dei membri di un tipo occorrono informazioni sulla dichiarazione di tipo che si sta generando; ad esempio nel caso venga generato il membro di un'interfaccia bisogna ometterne il livello di visibilità, o nel caso si stia generando un costruttore in *C#* bisogna conoscere il nome del tipo. Per contenere queste informazioni si è deciso di utilizzare una *stack*, in quanto è possibile avere più dichiarazioni di tipo annidate e occorre tenerne traccia. Si sono inoltre definite proprietà per utilizzare la stack in maniera semplificata, come ad esempio `CurrentTypeDeclaration` che restituisce l'elemento in cima a questa.

- `TypeMemberStack`: come per le dichiarazioni di tipi occorre tener traccia del tipo di membro che si sta generando, utilizzato durante la gestione dei `MemberAttributes` per determinare quali parole chiave generare.

- `CurrentNamespace` e `ImportedNamespaces`: queste informazioni vengono utilizzate durante la generazione di `CodeTypeReference`, per determinare se occorre utilizzare il nome completo del tipo o se si può invece usarne una versione ridotta.

- `StatementShouldTerminate`: valore booleano che indica se gli *statement* devono essere gestiti scrivendo il carattere di terminazione riga (';' in *C#*, assente in *VB.Net*). È necessario per la generazione degli *statement* di inizializzazione e incremento del ciclo `for` in *C#*, e sebbene sia di fatto inutilizzato nel caso del generatore per *VB.Net* è comunque presente nel context generale in quanto viene utilizzato da un'implementazione di *handler* astratta utilizzata da entrambi i generatori.

- `VisualBasicContext`: utilizzato per contenere informazioni necessarie per la generazione di codice *VB.Net*, fornisce solamente la proprietà

`BlockTypeStack`, che contiene appunto una *stack* dei tipi di blocco che stanno venendo generati, come ad esempio `Class`, `Sub` o `While`. Queste informazioni sono principalmente utilizzate da alcune funzioni di utilità, ma sono anche necessarie per la generazione dei costrutti `Exit` (equivalente a `break`) e `Continue`, in quanto in *VB.Net* occorre specificare il blocco alla quale si vogliono applicare.

- `CSharpContext`: utilizzato per contenere informazioni necessarie per la generazione di codice *C#*, fornisce solamente la proprietà `TypeParameterHandlerRequestedOperation`, utilizzata per indicare se l'handler di `CodeTypeParameter` deve generare la dichiarazione del parametro di tipo o i suoi vincoli, in quanto questi vanno generati in posizioni del codice diverse.

5.2.3 Handler astratti

Spesso la generazione del codice per *VB.Net* e *C#* risulta simile; in molti casi l'unica differenza presente è data dalla parola chiave utilizzata, in altri invece parte dell'*handling* risulta equivalente, in quanto si occupa per esempio di gestire i dati nel *context* o richiamare altri *handler*. Si sono quindi realizzate delle versioni di *handler* astratte nel namespace `CodeDomExt.Generators.Common`, che si occupano appunto di gestire questi casi.

Nel secondo caso citato, per il quale si può vedere un esempio nel blocco di codice 5.3, è stato necessario introdurre un metodo per consentire di riutilizzarne l'implementazione nel caso in cui si volesse ridefinire la gestione di qualche nodo. Questo è stato realizzando introducendo le proprietà `CanHandle*` che indicando se uno specifico nodo è gestito dall'implementazione; si fa notare che se si decidesse in alternativa di utilizzare un valore di ritorno direttamente nel metodo `Handle*` si otterrebbero risultati errati, in quanto parte della gestione comune viene effettuata prima di richiamare tale metodo.

5.2.4 Gestione dei nodi

In generale i metodi per l'effettiva gestione dei nodi risultano molto semplici, in quanto consistono principalmente nella scrittura di codice tramite il *writer* presente nel *context* e nella delegazione della gestione di nodi figli ad altri *handlers*. Si fa notare che per delegare la gestione di nodi occorre utilizzare sempre l'*handler* fornito dalla proprietà `HandlerProvider` del *context*, anche nel caso in cui si debba gestire un nodo figlio dello stesso tipo del padre e che quindi potrebbe essere gestito dallo stesso *handler* del padre, in modo da

```
1  public abstract class DefaultMemberAttributesHandler
2      : ICodeObjectHandler<MemberAttributes>
3  {
4      public bool Handle(MemberAttributes obj, Context ctx)
5      {
6          [...]
7          if (ctx.CurrentTypeMember != MemberTypes.Constructor)
8          {
9              if ((obj & MemberAttributes.VTableMask) == MemberAttributes.New &&
10                 !string.IsNullOrEmpty(GetNewKeyword(ctx)))
11             {
12                 ctx.Writer.Write(GetNewKeyword(ctx));
13                 ctx.Writer.Write(" ");
14             }
15         }
16         [...]
17         return true;
18     }
19     [...]
20     protected abstract string GetNewKeyword(Context ctx);
21     [...]
22 }
```

Codice 5.2: Esempio di *handler* astratto per `MemberAttributes` in cui l'implementazione si limita a definire le parole chiave da utilizzare

```
1  [...]
2  protected abstract bool CanHandleField { get; }
3  protected abstract void HandleField(CodeMemberField obj, Context ctx);
4  private bool HandleDynamic(CodeMemberField obj, Context ctx)
5  {
6      return HandleIfTrue(() =>
7      {
8          HandleField(obj, ctx);
9      }, obj, ctx, CanHandleField, MemberTypes.Field);
10 }
11 private bool HandleIfTrue(Action handle, CodeTypeMember obj, Context ctx,
12     bool condition, MemberTypes memberType)
13 {
14     if (condition)
15     {
16         //handles member stack, start directives, comments and member
               attributes
17         [...]
18         //handles member
19         handle();
20         //handles end directives and member stack
21         [...]
22     }
23     return condition;
24 }
25 [...]
```

Codice 5.3: Esempio di *handler* astratto che si occupa di effettuare la parte di gestione comune tra gli *handler* per *C#* e *VB.Net*, nel caso di nodi di tipo `CodeTypeMember`.

```
protected override bool HandleDynamic(CodeMethodInvokeExpression obj,
    Context ctx)
{
   ctx.HandlerProvider.ExpressionHandler.Handle(obj.Method, ctx);
   ctx.Writer.Write("(");
   GeneralUtils.HandleCollectionCommaSeparated(obj.Parameters.Cast<
       CodeExpression>(),
      ctx.HandlerProvider.ExpressionHandler, ctx);
   ctx.Writer.Write(")");
   return true;
}
```

Codice 5.4: Esempio di metodo per la gestione di un nodo `CodeMethodInvokeExpression`; le proprietà `Method` e `Parameters` sono di tipo `CodeMethodReferenceExpression` e `CodeExpressionCollection` e indicano rispettivamente il metodo da invocare e gli argomenti da passare per l'invocazione. `HandleCollectionCommaSeparated` utilizza il metodo `HandleCollection` citato in precedenza, definendo le azioni da svolgere per generare gli elementi della collezione specificata separandoli tramite virgole.

assicurarsi che eventuali modifiche effettuate alla gestione di tale nodo siano sempre applicate.

La gestione dei nodi è ulteriormente semplificata dall'utilizzo di funzioni di utilità realizzate nella classe statica `GeneralUtils`. Il metodo più utilizzato di questa è `HandleCollection<T>`, che permette di gestire una collezione di nodi `T` specificando quale *handler* utilizzare ed eventuali azioni da eseguire prima e dopo l'handling del nodo e se l'azione eseguita in seguito all'handling del nodo sia da eseguire anche qualora questo sia l'ultimo della collezione; tale metodo ad esempio semplifica di molto l'implementazione dei metodi per l'handling di nodi quali `CodeMethodInvokeExpression` che devono gestire collezioni di espressioni separandole tramite delle virgole.

Non tutti i nodi però risultano semplici da gestire; un esempio è dato dalla gestione del nodo `CodePrimitiveExpression`, che rappresenta valori *literal*. Questo ha richiesto particolari attenzioni nel caso della gestione di valori numerici; la generazione di un *long* per esempio richiede prima di valutare se il suo valore può essere anche espresso da un *int*, tipo di default per i *literal* numerici interi, e in caso affermativo occorre generarne il valore utilizzando l'eventuale il suffisso previsto dal linguaggio per forzarne l'interpretazione come *long* o, qualora tale suffisso non sia definito, utilizzare un'operazione di cast.

5.2.5 Gestione di errori nel grafo

I nodi di CodeDOM sono privi di logica, e per questo motivo il grafo potrebbe rappresentare un programma non valido; ad esempio si potrebbero avere interfacce contenenti membri marcati come *private*, o dichiarazioni di tipo non innestate con livelli di accesso diversi da *public* o *internal*.

In questi casi si è deciso di ignorare l'errore a patto che sia possibile determinare con certezza quale fosse l'effettivo valore desiderato per poter generare codice funzionante; ad esempio nel primo caso citato si genera il membro con il livello di accesso di default, in quanto è l'unico che non genererebbe errore. Qualora invece non si possa essere certi del valore effettivamente desiderato, come ad esempio nel secondo caso citato, si è deciso di generare il codice come indicato nel nodo, in modo da causare un errore di compilazione che evidenzierà il problema.

5.2.6 Miglioramenti sul codice generato

Il codice prodotto dai generatori di CodeDomExt, nel caso in cui non si utilizzino nuovi tipi di nodi, risulta in genere uguale a quello generato da CodeDOM, ma in alcuni casi sono presenti variazioni dovute a cambiamenti introdotti per poter soddisfare i requisiti sulla leggibilità del codice.

In particolare la generazione dei nodi `CodeConditionStatement` valuta anche le istruzioni presenti nel gruppo *else* per determinare se è possibile utilizzare il costrutto *elseif*; analogamente per la generazione del nodo `CodeIterationStatement` si valutano se i valori delle proprietà `InitStatement` e `IncrementStatement` sono nulli o espressioni vuote per evitare di lasciare righe vuote nel caso della generazione di codice *VB.Net*, o per determinare quale costrutto utilizzare tra *for* e *while* nel caso della generazione di codice *C#*.

Per quanto riguarda invece la riduzione del numero di parentesi ridondanti, si è deciso di utilizzare l'approccio opposto a quello utilizzato da CodeDOM: durante la generazione di espressioni che potrebbero richiedere parentesi per assicurarne il corretto ordine di valutazione, CodeDOM genera sempre una coppia di parentesi. Al contrario invece in CodeDomExt non si generano mai le parentesi durante l'handling di questi costrutti, ma queste vengono generate dal nodo padre qualora ritenga le parentesi potrebbero essere necessarie, sulla base del tipo di nodo figlio da generare. Questo approccio non elimina tutte le parentesi ridondanti, ma consente di rimuovere quelle che risultano più evidenti. Di seguito se ne può vedere un esempio.

```
Method(((a * b) + c)); //CodeDOM
```

```
Method((a * b) + c); //CodeDomExt
```

5.3 Utilizzo di CodeDomExt in Cobol2Net

Una volta realizzati i generatori di codice si è potuto iniziare ad operare effettivamente su Cobol2Net; in questa prima sotto-fase si è sostituito CodeDOM con CodeDomExt. Siccome si è deciso di mantenere la compatibilità con CodeDOM, come indicato dal requisito descritto nella sezione 3.4, questa fase è stata molto breve: è stato sufficiente cambiare il generatore utilizzato e introdurre, dove necessario l'utilizzo, dei nuovi nodi definiti da CodeDomExt.

5.3.1 Sostituzione del generatore

La sostituzione del generatore è avvenuta nella `CobolTranslationFacade`, il punto di accesso del traduttore che espone i metodi per effettuare la traduzione. Questa operazione già permette di sfruttare i miglioramenti introdotti da CodeDomExt che non richiedono l'utilizzo di nuovi nodi, come ad esempio la generazione di *if-else*, l'utilizzo del nome di tipo parziale quando possibile e la riduzione del numero di parentesi ridondanti.

5.3.2 Introduzione dei nuovi nodi

In seguito alla sostituzione del generatore si è introdotto l'utilizzo dei nuovi nodi per la generazione di costrutti *do-while* e dell'operatore *not*. Il primo di questi è stato introdotto direttamente nel codice del *visitor* relativo ai verbi *COBOL*. La negazione di una condizione è invece un'operazione effettuata in molti *visitor*, e per queste è presente una funzione di utilità `NegateCondition`; il nodo per l'operatore *not* è stato quindi introdotto all'interno di questa.

5.4 Implementazione di handler per Cobol2Net

Molti dei requisiti di questa tesi, sono già stati soddisfatti nelle implementazioni dei generatori di CodeDomExt, in quanto si è ritenuto che questi non fossero relativi a problemi specifici di Cobol2Net, ma potrebbero risultare utili anche in progetti differenti. In questa fase, invece, si sono implementati *handlers* per risolvere i problemi che si sono ritenuti specifici di Cobol2Net, in modo da soddisfare i rimanenti requisiti.

In particolare gli *handlers* realizzati si occupano di soddisfare i requisiti relativi alla configurabilità dello stile del codice, chiamate fluenti troppo lunghe e indentazione del codice relativo alla definizione delle variabili *COBOL*. Questi saranno poi aggiunti al generatore per il linguaggio target tramite gli appositi metodi `AddHandler`.

Per realizzare questi *handler* è stato necessario condividere ulteriori informazioni oltre a quelle già presenti nel *context*; queste sono state raccolte in una classe `CobolTranslationContext` (in seguito indicato come *context cobol*) che è stata poi aggiunta al *context* originale come dati utente. Le informazioni aggiunte sono indicate e descritte nel dettaglio nelle sezioni relative agli *handler* che le utilizzano.

5.4.1 Configurabilità dello stile di codice

Si è realizzata una classe `GenerationOptions` per consentire di impostare delle preferenze sullo stile del codice generato, in modo da soddisfare il corrispondente requisito.

Le impostazioni relative alla stringa di indentazione e l'utilizzo del nome completo dei tipi sono già implementate in CodeDomExt, quindi non sono stati necessari ulteriori interventi per la loro implementazione.

Per realizzare invece le opzioni specifiche per *VB.Net* si è costruito un nuovo *handler*, che ridefinisce la gestione degli *statement* interessati. Questo genera il codice sulla base dei valori presenti in una classe `VisualBasicOptions`, presente nel *context cobol*, che contiene le preferenze dell'utente relative alla generazione di alcuni specifici costrutti di *VB.Net*; ad esempio è presente un'opzione per indicare se utilizzare, quando possibile, `Until` *`condition`* al posto di `Not` *`condition`*.

5.4.2 Separazione di chiamate fluenti

Per risolvere il problema di compilazione causato da lunghe sequenze di invocazioni di metodo concatenate, evidenziato nella sezione 3.3, si è fatto uso di un nuovo *handler* di `CodeStatement`.

Questo analizza *statement* di tipo `CodeExpressionStatement` e `CodeVariableDeclarationStatement`, e valuta se l'espressione contenuta rispettivamente nelle proprietà `Expression` e `InitExpression` rappresenta invocazioni di metodo concatenate e in caso affermativo conta il numero di queste; se quest'ultimo valore supera una determinata soglia allora l'espressione viene separata e vengono creati due nuovi *statement* appropriati da gestire al posto dell'originale, in cui il tipo del primo sarà sempre `CodeVa-`

```
public bool Handle(CodeStatement obj, Context ctx)
{
   CobolTranslationContext cobolCtx =
      ctx.GetUserData<CobolTranslationContext>();
   if (obj is CodeExpressionStatement expressionStatement &&
      IsFluentExpression(expressionStatement.Expression))
   {
      //if number of chained expressions is more than THRESHOLD splits it
          into 2 expressions; Item2 will have the rightmost expressions,
          for a total number of THRESHOLD expressions.
      Tuple<CodeExpression,CodeExpression> exps =
         SeparateFluentExpression(expressionStatement.Expression);
      if (exps != null)
      {
         CodeVariableDeclarationStatement varDecl =
            new CodeVariableDeclarationStatement("",
               cobolCtx.SupportVariableName)
            {
               InitExpression = exps.Item1
            };
         varDecl.StartDirectives.AddRange(
            expressionStatement.StartDirectives);
         cobolCtx.SupportVariablesCount++;
         ctx.HandlerProvider.StatementHandler.Handle(varDecl, ctx);
         ctx.Writer.NewLine();
         ctx.Writer.Indent(ctx);
         //sets the TargetObject of the leftmost expression
         exps.Item2.GetFluentRoot().SetFluentParent(
            new CodeVariableReferenceExpression(varDecl.Name));
         CodeExpressionStatement expr2 =
            new CodeExpressionStatement(exps.Item2);
         expr2.EndDirectives.AddRange(expressionStatement.EndDirectives);
         ctx.HandlerProvider.StatementHandler.Handle(expr2, ctx);
         return true;
      }
   }
   else if (obj is CodeVariableDeclarationStatement declarationStatement &&
         (declarationStatement.InitExpression))
   {
      [...] //similar to CodeExpressionStatement handling
   }
   return false;
}
```

Codice 5.5: Implementazione del metodo `Handle` per l'*handler* che si occupa di separare le chiamate che risultano essere troppo lunghe per il compilatore.

riableDeclarationStatement, mentre quello del secondo dipende dal tipo dello *statement* originale.

Il risultato di questo è che l'istruzione originale sarà suddivisa in due o più istruzioni; le prime saranno di tipo `CodeVariableDeclarationStatement`, e salvano il risultato della loro esecuzione in delle variabili di supporto, e l'ultima sarà invece di tipo `CodeExpressionStatement`.

I nomi delle variabili di supporto sono stabiliti sulla base di una variabile `SupportVariableCount` presente nel *context cobol*, che indica il numero di variabili di supporto utilizzate. Per poter riutilizzare i nomi di queste variabili si è deciso di aggiungere un ulteriore *handler* per `CodeTypeMember` che si limita a resettare il contatore presente nel context se il membro ricevuto in input è un metodo, senza però occuparsi di gestirlo.

Nel blocco di codice 5.5 è rappresentata parte dell'implementazione dell' *handler* che si occupa di separare statement contenenti espressioni fluenti. Questi vengono gestiti solo quando il numero di espressioni concatenate supera una certa soglia; si fa notare però che l'*handler* non si occupa mai di generare effettivamente il codice, a parte per l'output di una nuova linea, ma delega la gestione degli statement creati all'*handler* presente nel *context*, che in questo caso specifico è uno `ChainOfResponsibilityHandler` che tenterà di delegarne la gestione prima a questo stesso *handler*, e successivamente a quello di *default* per il linguaggio target. Di conseguenza gli statement ottenuti verranno divisi ulteriormente se necessario, altrimenti saranno gestiti normalmente. In questo modo non è stato necessario ridefinire il modo in cui gli statement di questo tipo vanno gestiti e l'*handler* realizzato funziona correttamente per la generazione di codice *VB.Net* e *C#*

5.4.3 Indentazione della definizione di variabili COBOL

Per quanto riguarda la gestione dell'indentazione per la definizione di variabili *COBOL* nei programmi tradotti, è stato necessario analizzare la struttura che hanno le corrispondenti espressioni fluenti. Si è notato che queste sono suddivisibili in blocchi e ciascuna componente dell'espressione fluente può essere considerata parte di uno dei seguenti gruppi, sulla base del nome della proprietà o metodo a cui fa riferimento:

- **Apertura**: iniziano un *blocco*; fanno parte di questo gruppo espressioni che iniziano la definizione di una variabile.

- **Chiusura**: terminano un *blocco*.

- **Fine dichiarazione**: rappresenta la fine dell'espressione fluente. Questo gruppo è costituito da una singola espressione, che è utilizzata al posto delle espressioni di *chiusura* di *blocco* per terminare quello più esterno della dichiarazione.

- **Altro**: permettono di definire alcune caratteristiche di un *blocco*, come ad esempio il valore con cui deve essere inizializzata la variabile corrispondente.

Ciascun *blocco* rappresenta una variabile *COBOL*; all'interno di questi è possibile avere espressioni parte del gruppo *altro*, che definiscono le caratteristiche della variabile rappresentata, eventualmente seguite da ulteriori blocchi annidati, nel caso in cui la variabile rappresentata sia *composta*.

Si è quindi realizzato un *handler* che gestisce le espressioni fluenti relative alla definizione di variabili *COBOL*. Questo gestisce le singole componenti dell'espressione fluente sulla base del gruppo di appartenenza. Nell'effettuare ciò si fa uso di nuove variabili condivise contenute nel *context cobol*; in particolare si è definita una variabile intera `ExtraIndentation`, che rappresenta l'ulteriore livello di indentazione da utilizzare durante l'*handling* di queste espressioni.

L'*handler* utilizza la seguente strategia: se l'espressione valutata fa parte del gruppo di *apertura* la genera su una nuova linea, ad un livello di indentazione pari a quello base contenuto nel *context* più `ExtraIndentation` ed incrementa il valore di quest'ultima variabile. Le espressioni di *chiusura* e *fine dichiarazione*, invece, decrementano il valore di `ExtraIndentation` e vengono generate su una nuova linea, ma solo se l'espressione precedentemente valutata era anch'essa parte del gruppo di *chiusura*; in questo modo espressioni che non contengono ulteriori blocchi annidati sono generate su una singola linea. Infine le espressioni parte del gruppo *altro* sono gestite normalmente sulla linea corrente.

Capitolo 6

Validazione e conclusioni

Per assicurare la correttezza della libreria realizzata e delle modifiche effettuate a Cobol2Net, si sono effettuati dei test di vario tipo, ciascuno pensato per verificare validità di una specifica parte del sistema.

In seguito sono evidenziati i test effettuati, fornendo infine un riassunto dei risultati ottenuti dalla tesi.

6.1 Validazione degli handler

Per verificare il funzionamento degli *handler* si sono realizzati piccoli programmi equivalenti in *C#* e *VB.Net* dai quali si è costruito l'albero CodeDOM; si sono poi preparati dei test automatici che confrontano il codice originale con quello generato dall'albero. Si sono costruiti test di questo tipo per validare sia gli *handler* di CodeDomExt, che quelli realizzati appositamente per Cobol2Net.

Per poter considerare il codice prodotto dal generatore corretto, si è deciso che questo deve essere uguale al programma target e non solo equivalente. Il motivo principale dietro a tale scelta è il fatto che parte dei requisiti di questa tesi è legata alla forma del codice, e se si fosse deciso di valutare l'equivalenza dei programmi alcuni test avrebbero perso il loro significato.

I programmi costruiti per effettuare questi test sono di due tipi principali: la maggior parte di questi è stata realizzata per testare la generazione di un singolo costrutto, quindi verificano la validità di un singolo *handler* o parte di questo, ma sono anche presenti programmi pensati per testare il funzionamento del generatore quando sono utilizzati più tipi di costrutti contemporaneamente. I programmi del primo tipo citato in genere si occupano di verificare la corretta generazione di varie versioni del nodo utilizzato

```
1  namespace Test.Namespace
2  {
3      public class TestClass
4      {
5          [...]
6          private protected TestClass(int a, double b)
7              : base(a, b)
8          {
9          }
10
11         public static int Main(string[] args)
12         {
13         }
14
15         protected abstract void Method1();
16
17         public virtual void Method2()
18         {
19         }
20         [...]
21     }
22 }
```

Codice 6.1: Esempio di un programma di test target. Questo viene confrontato con il codice prodotto da un grafo CodeDOM equivalente, per poter determinare la corretta generazione delle *signature* dei metodi da parte del generatore per codice *C#* di CodeDomExt.

per rappresentare il costrutto testato e per questo motivo risultano molto semplici ma lunghi; se ne può vedere un esempio nel blocco di codice 6.1.

Questi test sono stati realizzati durante l'implementazione: man mano che si è implementata una funzionalità se ne è realizzato anche il test corrispondente. In questo modo si è riusciti ad individuare immediatamente eventuali bug presenti e, inoltre, si sono agevolate le modifiche al codice, in quanto si è potuto verificare immediatamente che queste non "rompessero" il sistema esistente.

6.2 Validazione del traduttore di Cobol2Net

I test relativi agli *handler* hanno consentito di verificare il corretto funzionamento del generatore nel caso di piccoli programmi d'esempio. Questi test non sono però sufficienti per poter affermare il funzionamento del generatore anche nel caso di applicazioni reali, come Cobol2Net; è inoltre necessario verificare che i cambiamenti effettuati nel traduttore per l'aggiunta di CodeDomExt a Cobol2Net non abbiano compromesso le capacità di quest'ultimo di tradurre sorgenti *COBOL*.

Per questo motivo si sono effettuati dei test nei quali si sono verificate le capacità del sistema aggiornato nell'effettuare la traduzione di sorgenti *COBOL* di vario tipo; tutti i programmi testati risultano traducibili dalla versione di Cobol2Net precedente all'introduzione di CodeDomExt e, a parte per errori dovuti ad espressioni troppo lunghe, le versioni tradotte risultano compilabili. Alcuni di questi sono molto semplici, realizzati appositamente per testare la traduzione di uno specifico costrutto; altri invece sono stati forniti da *HG* e sono effettivi programmi parte del sistema da tradurre, di dimensioni variabili comprese tra alcune centinaia ed alcune decine di migliaia di righe di codice. Nella tabella 6.1 sono indicate nel dettaglio le dimensioni e quantità dei programmi testati e correttamente funzionanti.

Numero di righe programmi originali	**Quantità**
<200 (Programmi d'esempio)	25
200 - 1000	5
1000 - 3000	13
3000 - 5000	9
5000 - 10000	2
>10000	2

Tabella 6.1: Tabella contenente i dettagli relativi alle dimensioni dei programmi utilizzati per validare il sistema.

Per poter considerare validi i cambiamenti effettuati occorre che Cobol2Net sia in grado di tradurre tutti i sorgenti forniti e che il codice tradotto sia compilabile senza interventi manuali. Inoltre occorre anche verificare che i miglioramenti attesi dall'introduzione di CodeDomExt e l'utilizzo di *handler* realizzati specificatamente per Cobol2Net siano effettivamente presenti nei programmi tradotti.

Alcuni dei programmi *COBOL* utilizzati in questi test hanno delle caratteristiche che consentono di verificare il corretto funzionamento del sistema in situazioni critiche e per questo motivo risultano essere dei casi di studio interessanti.

Uno di questi, ad esempio, definisce una variabile *composta* formata da un gran numero di variabili, che nel caso del traduttore originale veniva tradotta in una espressione fluente estremamente lunga, costituita da oltre mille chiamate concatenate, e per questo motivo non risultava compilabile; si è potuto verificare quindi che il nuovo generatore si occupa effettivamente di spezzare la chiamata fluente in più espressioni, in modo da consentirne la compilazione, senza però cambiarne il significato.

```
01  W-ICAST-PARAM.
    03 WICAST-SIGLA-OP              PIC X(03).
    03 WICAST-RIPRISTINA            PIC X.
    03 WICAST-RIGA-INIZIO           PIC 99.
    03 WICAST-TAB-CAT-STAT.
       05 WICAST-EL-CAT-STAT        OCCURS 10.
          07 WICAST-COD-CAT-STAT    PIC X(6).
    03 FILLER                       PIC X(1).
    03 WICAST-TIPO-TAB              PIC X.
       88 WICAST-CAT-TBPARMAGAZ     VALUE ' ' '0'.
    03 WICAST-SE-GEST-GRUPPO        PIC X.
       88 WICAST-SET-IGNO-GRUPPO    VALUE '1'.
       88 WICAST-RESET-IGNO-GRUPPO  VALUE '0'.
    03 FILLER                       PIC X(198).
    03 WICAST-CHIAVE-CHIUSURA       PIC 9(2) COMP.
    03 WICAST-STATUS-BIT9           PIC 9.
    03 WICAST-FLAG-ERR              PIC 9.
```

```
NewCompound(wIcastParam).Begin _
   .NewAlphaNumeric(wicastSiglaOp, 3).End _
   .NewAlphaNumeric(wicastRipristina, 1).End _
   .NewIntegral(wicastRigaInizio, 2, SignFormats.UNSIGNED).End _
   .NewCompound().Begin _
      .NewTableOfCompound().Occurrs(10) _
         .Item _
            .NewAlphaNumeric(wicastCodCatStat, 6).End _
         .End _
      .End _
   .End _
   .NewAlphaNumeric(1).End _
   .NewAlphaNumeric(1) _
      .NewCondition().Id(wicastCatTbparmagaz).Values(ValueSet(" ").And("0")
          ).End _
   .End _
   .NewAlphaNumeric(1) _
      .NewCondition().Id(wicastSetIgnoGruppo).Values(ValueSet("1")).End _
      .NewCondition().Id(wicastResetIgnoGruppo).Values(ValueSet("0")).End _
   .End _
   .NewAlphaNumeric(198).End _
   .NewIntegral(wicastChiaveChiusura, 2, SignFormats.UNSIGNED, Usages.
       COMPUTATIONAL_4).End _
   .NewIntegral(1, SignFormats.UNSIGNED).End _
   .NewIntegral(1, SignFormats.UNSIGNED).End _
.Build()
```

Codice 6.2: Esempio di indentazione automatica del codice nella rappresentazione delle variabili composte. Nel blocco superiore si può vedere la definizione della variabile nel programma *COBOL* originale; nel blocco inferiore invece si vede la definizione della stessa variabile nel programma tradotto. Prima dell'introduzione di CodeDomExt la definizione di una variabile veniva generata tutta sulla stessa riga.

Un'altra situazione analoga è data da programmi che definiscono variabili *composte* complicate, contenenti per esempio multiple condizioni, ridefinizioni e ulteriori variabili composte innestate; in questi casi si è potuta verificare la corretta generazione dell'indentazione per le variabili *composte*, realizzata in modo da emulare la formattazione di tali variabili nel codice originale. Un esempio di questo tipo è rappresentato nel blocco di codice 6.2

6.3 Conclusioni

Il principale vantaggio introdotto in Cobol2Net da questa tesi è dato dalla possibilità di controllare la generazione del codice sorgente a seguito dell'introduzione di CodeDomExt; grazie a questo si sono potute applicare delle modifiche al codice generato introducendo miglioramenti di vario tipo alcuni dei quali sono elencati di seguito. Inoltre, grazie alla natura modulare di CodeDomExt, sarà possibile in futuro introdurre facilmente ulteriori cambiamenti nella generazione del codice.

Il miglioramento più evidente nei programmi generati risulta essere l'indentazione della parte di codice relativa alla definizione delle variabili *COBOL*, in modo da emulare la struttura che queste hanno nei programmi originali.

Un ulteriore miglioramento è dato dallo sfruttamento di più costrutti del linguaggio target che ha permesso di evitare l'utilizzazione di tecniche artificiose per generare costrutti non supportati da CodeDOM; di conseguenza il codice risulta di più semplice comprensione e l'impatto positivo che la traduzione ha sulla manutenibilità del programma risulta aumentato.

Inoltre si sono introdotte opzioni per il controllo della forma del codice tradotto utilizzabili dall'utente di Cobol2Net, rendendo quindi possibile produrre codice più conforme alle convenzioni adottate dai programmatori che dovranno mantenere il codice tradotto.

Oltre ai miglioramenti relativi alla forma del codice, si è anche riusciti ad aumentare il numero di sorgenti traducibili in maniera completamente automatica dallo strumento.

Infine questa tesi non porta vantaggio solo a Cobol2Net; la realizzazione di CodeDomExt come libreria separata ne permetterà l'utilizzo in sistemi differenti, e potrebbe quindi avere ulteriori applicazioni industriali.

6.4 Sviluppi futuri

La realizzazione di CodeDomExt e l'introduzione di questa in Cobol2Net, fornendo controllo sul codice generato, ha aperto la strada alla realizzazione di ulteriori cambiamenti, ed è possibile che in futuro si individueranno ulteriori possibili miglioramenti al codice generato che potranno essere facilmente implementati.

In futuro si potrebbe anche lavorare ulteriormente su CodeDomExt, qualora diventasse interessante per un altro progetto, implementando alcune nuove funzionalità che non sono state necessarie a Cobol2Net. Queste ad esempio potrebbero essere il supporto ad ulteriori linguaggi, o anche l'implementazione di meccanismi che avvisano l'utente in caso di anomalie nel grafo, in modo da facilitare l'individuazione di errori nella costruzione di questo.

Bibliografia

[1] Microsoft Docs. *Using the CodeDOM*. URL: `https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/using-the-codedom` (visitato il 03/05/2018).

[2] *C# Language Specification*. Ecma International, 2017.

[3] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[4] "Gang of Four". *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Elenco delle figure

Lista dei blocchi di codice