Programmazione a Livelli

di Andrea Guerrieri

Descrizione

L'idea è di codificare (descrivere) la propria applicazione usando diversi livelli di astrazione, fino ad arrivare al codice sorgente. Ogni livello di astrazione usa un proprio linguaggio, e ha associato uno o più "compilatori" che lo traducono in un livello di astrazione più basso (meno astratto)

- Per definire un linguaggio è sufficiente descriverne la grammatica / lessico (es. in notazione EBNF)
- Il compilatore può essere codificato con un linguaggio di manipolazione alberi (es. XSLT)

Durante il passaggio di livello non vi è perdita, ovvero viene mantenuto un collegamento tra il codice meno astratto e il codice più astratto in modo che:

- Ogni modifica nel linguaggio meno astratto venga riportata in quello più astratto
- Ogni modifica del linguaggio più astratto venga riportata in quello meno astratto, senza che vengano perse le eventuali modifiche fatte.

Lo stesso programma può essere composto da più blocchi, ognuno descritto nel livello di astrazione più adatto, anche mischiando livelli diversi all'interno dello stesso sorgente.
Il punto di forza di questa sistema è che può facilmente esprimere il "cosa si fa" (e questo è immutabile a meno di variazione nelle specifiche) mentre i linguaggi di programmazione odierni esprimono "il come si fa", rendendo i software vincolati alla tecnologia.

Applicazioni

Applicazione 1:

Poter descrivere algoritmi in un linguaggio (linguaggio B) più "comodo" e "compatto" di quello a disposizione (linguaggio A). Esempi:

- Regular Expression
- XSLT
- Grammatiche

Soluzioni adottate:

- Creazione di una classe o funzione che prende in input una stringa nel linguaggio B, lo interpreta, e fornisce i risultati.
- Generazione di codice compilato runtime che esegue quella particolare espressione (soluzione adottata da .net).

Svantaggi:

- Necessità di "compilare" ogni volta un qualcosa che è staticamente definito (es. una regular expression che valida un indirizzo e-mail)
- Necessità di interpretare con il linguaggio A il linguaggio B, con notevole perdita di efficienza

Vantaggi dei livelli:
Con i livelli è possibile mantenere l'astrazione del linguaggio B, generando direttamente il codice nel linguaggio A che esegue B. Ogni modifica in B genererà il nuovo codice in A che potrà essere compilato.

Applicazione 2:

Creazione di generatori di codice a partire da un particolare pattern o template:

- Applicazione di Design pattern
- Autocomposizione interfacce utente / classi
- Creazione automatica di stub /skeleton per remoting (sun rpc, java rmi)

Soluzioni adottate:

- Tool grafici che tramite "wizard" creano in automatico il codice
- Applicazioni specifiche che permettono di definire e manipolare template (es. Macromedia Dreamweaver per HTML)
- Tramite paradigmi del linguaggio (Generici C#, Java, c++)
- Tool a riga di comando che dato in input un linguaggio B generano il sorgente per il linguaggio A (es da XML tramite XSLT a HTML, oppure da XDR tramite RPCGEN a sorgenti C)
- Creazione di programmi ad-hoc per generare un particolare template.

Svantaggi:

- Necessità di usare tool diversi da re-invocare ogni volta che si fa una piccola modifica
- Se si genera del codice con un tool automatico, una modifica ad un parametro del generatore, comporta la rigenerazione di tutto quanto: se nel frattempo sono state effettuate modifiche al codice auto-generato difficoltà di riportarle nella nuova versione. (Possibile ausilio di tool di merge, ma costoso)
- Bisogna costruire per intero una nuova applicazione per ogni problema di auto-generazione, molto spesso il costo è maggiore dei benefici

Vantaggi dei livelli:

Con i livelli si esprime il generatore con un linguaggio ad un livello di astrazione più alto. E' possibile facilmente definirsi un nuovo linguaggio in grado di rappresentare al meglio il "generatore", semplicemente formalizzando la sua grammatica. In molti casi si può fare ausilio di un linguaggio esistente generico (es. xml). Ogni modifica al generatore viene riportata al livello inferiore, senza intaccare eventuale aggiunte o modifiche fatte.

Applicazione 3:

Trascodifica automatica da un linguaggio all'altro

Vantaggi dei livelli:

Con i livelli è possibile partire da un linguaggio meno astratto e portarlo ad u livello di astrazione più alto, per poi essere convertito di nuovo ad un livello più basso, ma in un linguaggio diverso da quello di partenza. Durante questo passaggio è possibile che ci sia perdita di informazioni.

Applicazione 4:

Creazione di un codice riutilizzabile ma al contempo efficiente. Rendere più facile la descrizione della semantica dell'applicazione, divedendo in componenti / blocchi, a sua volta divisi in sotto-componenti e sotto-blocchi, e cosi via

Soluzioni adottate:

- Dividere il proprio progetto nelle entità "riutilizzabili" fornite dal linguaggio (es. funzioni/classi/moduli)

- Creazione di componenti pre-compilati riutilizzabili (es. DLL)
- Uso di linguaggio di descrizione diversi da quelli di codifica (es. UML)
- Uso delle regioni (.net), blocchi di codice al quale si può associare un descrizione

Svantaggi:

- La sovrastrutturazione rende si il codice riutilizzabile ma di gran lunga meno efficiente, sia a cause delle innumerevoli chiamate tra componenti, sia perché si costruiscono oggetti che risolvono classi di problemi, e non quello specifico
- La scrittura di un codice monolitico ad-hoc rende molto difficile la leggibilità/riusabilità. Necessità di trovare un compromesso tra efficienza e struttura.
- Difficoltà di portabilità in piattaforme diverse

Vantaggi dei livelli:
Tramite i livelli partendo da una descrizione generica dell'applicazione, si può diventare sempre più specifici, fino ad arrivare al codice sorgente. La stessa descrizione "astratta" può essere facilmente trasformata in diversi linguaggi di programmazione a livello base, nel modo che si ritiene più opportuno. Se si vuole creare codice efficiente, a partire da componenti "astratti" si può addirittura generare un programma che stia tutto su una funzione, oppure se quello che interessa è avere programmi di piccole dimensioni, è possibile associare ogni componente astratto in un componente del linguaggio base. L'unità di "riutilizzo" non diventa più quella fornita dal linguaggio base (es. funzione, classe) ma il blocco che si ha nel linguaggio più astratto.

Applicazione 5:

Applicazioni distribuite, rendere trasparante al programmatore la comunicazione tra componenti che vengono eseguiti in macchine diverse.

Soluzioni adottate:

- Soluzioni adottate a livello di linguaggio come corba, java rmi, sun rpc, remoting .net. Uso di tool a riga di comando che generare le infrastrutture di comunicazione al fine della traspararenza.
- Uso di protocolli e servizi standard per la trasmissione e la comunicazione al fine della portabilità in diverse piattaforme (es. soap)

Svantaggi:

- I protocolli general-purpose non sono spesso in grado di trasmettere al meglio certe tipologia di informazioni (es. html / soap per trasmettere contenuti non testuali)
- Dipendenza dalla tecnologia, e dalla tipologia di rete. Necessità di un nuovo progetto per cambiare tipologica di rete o protocollo di comunicazione

Vantaggi dei livelli:
Tramite i livelli è possibile partendo dalla descrizione astratta delle entità coinvolte nella comunicazione (oggetti, messaggi, etc), e dalla tipologia di rete (peer-to-peer, client-server, etc), generare un insieme di classi che realizzano la comunicazione

Applicazione 6:

Estraopolazione di dati a partire da testi non strutturati / formattati.

Esempi

Qui riportati una serie di scenari in cui la programmazione a livelli può essere applicata:

1. Portare le espressioni regolari ad un livello di astrazione più alto in modo da generare codice ad-hoc che valuta quella espressione

2. Portare la descrizione delle interfacce utente ad un livello di astrazione più alto in modo da poter riusare la stessa descrizione per generare interfacce in linguaggi / tecnologie diverse (form: java, c# - html)
3. Portare la descrizione dei dati ad un livello di astrazione superiore, in modo da poter generare in automatico:
    a. Query sql di creazione tabella per rappresentare quella struttura in un db relazionale
    b. Classi che rappresentano quei dati
    c. Maschere (interfacce utente) per poter modificare/inserire quei dati
4. Traduzione di un algoritmo dal linguaggio A al linguaggio B. Ad esempio trasformare un set di classi in "funzioni – strutture dati" per un linguaggio che non supporta gli oggetti (es. c++ -> c)
5. Descrizione di algoritmi in pseudo-codice che poi verrà specializzato in base al tipo di dati / piattaforma in cui verrà eseguito
6. Creazione di interfacce utenti: molte applicazioni e finestre presentano strutture standard. Ad esempio, le form di configurazione raggruppano una serie di opzioni in categorie, ognuna delle quali viene inserita in un pannello. I pannelli possono essere strutturati a tab, accessibili tramite menu, o icone. La finestra presenta tre pulsanti (ok, annulla, applica), alla quale pressione bisogna effettuare sempre le solite operazioni.
    a. Livello 4: (Dati-Pannelli) A partire dalla descrizione delle opzioni e delle categorie, generazione di un'insieme di pannelli, ognuno per categoria, che contengono i controlli (textbox, combobox, etc) per quelle opzioni
    b. Livello 3: (Pannelli-Form) A partire dai pannelli, e dalla descrizione dello stile (a tab, a icone, a menu), generazione di una finestra di dialogo e tutti suoi componenti.
    c. Livello 2: (Form-PseudoLinguaggio) A partire dalla descrizione del form e di tutti i suoi componenti, generazione in pseudo-codice della classe che realizza quel form.
    d. Livello 1: (Pseudocodice-Sorgente): a partire dalla descrizione nello pseudo- codice, generazione del sorgente nel linguaggio di destinazione (c#, java, etc)

Implementazione:

Creazione di una IDE o di un pannello che si integra a Microsoft Visual Studio .NET 2005 che permetta di usare la programmazione a livelli. Caratteristiche:

- Possibilità di definire e salvare linguaggi
- Possibilità di definire e salvare compilatori
- Possibilità di poter vedere il proprio sorgente al livello di astrazione desiderato. Il passaggio da un livello all'altro è fatto con pulsanti o menu
- Possibilità di espandere ad albero un solo blocco di codice ad un livello di astrazione più basso, ricorsivamente

Esempio:

Linguaggio:

```
<program>::=[{[_](<method>|<not-method-line>)[_]}];
<method>::=<method-declaration><block>;
<method-declaration>::=[{<method-modifier>_}]<type-name>_<method-name>\([<method-params>]\);
<method-params>::=<method-param>[{[_],[_]<method-param>}];
<method-param>::=[{<param-modifier>_}]<type-name>_<param-name>;
<block>::=[_]\{[<block-content>]\}[_];
<block-content>::={<block>|<block-char>};
<type-name>::=<identifier>[\[\]];
<method-name>::=<identifier>;
<param-name>::=<identifier>;
<param-modifier>::=ref|in|out;
<method-modifier>::=static|public|private|protected|internal|virtual|override|new|sealed|wrapped|
                    checked;
<block-char>::=@\[^\{\}\];
<identifier>::=@\[a-zA-Z0-9\]+;
<separators>::=@\(\\n\|\\b\)+;
<not-method-line>::=@\[^\r\n\]*;

#language “csharp-ext”
{
        scope: program;
        grammar: ;
        lexic: ;
}
```

Compilatore:

```
#compiler “csharp-ext-to-csharp”
{
        source: csharp-ext
        target: csharp
        nodes: program, method, method-declaration, method-modifier, method-declaration
}

#begin-file $(outfile)

foreach [/program/*]
{
        switch [.]
        {
                case [./method//method-modifier = “wrapped”]
                        out [./method-declaration/* exept method-modifier = “wrapped”] linked
                        out “{“
                        out “//Codice di inizio metodo.” readonly
                        out [./block-content] linked
                        out “//Codice di fine metodo.” readonly
                        out “}”
                case [./method//method-modifier = “checked”]
                        out [./method-declaration/* exept method-modifier = “checked”] linked
                        out “{“
                        foreach [./method-param]
                        {
                                if [./type.name != “int”]
                                {
                                        out “if “ [./param-name] “ == null)\n throw
                                            new NullPointerExceptiin()” readonly
                                }

                        }
                        out [./block-content] linked
                        out “}”
                case else
                        out [.] linked
        }
}

#end-file $(outfile)
```

Programma:

```
#var outfile = “out.cs”
#language csharp-ext -> csharp

using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Text;
using Ebnf.Language;

namespace Ebnf
{
    public enum SymbolListType
    {
        Inclusive,
        Esclusive
    }

    public class LanguageSymbolList : Collection<string>
    {
         public LanguageSymbolList()
         {
         }

        public wrapped void AddRange(EbnfSymbolCollection items)
        {
            foreach (EbnfSymbol symbol in items)
                Add(symbol.Name);
        }

        public void AddRange(IList<string> items)
        {
            foreach (string symbol in items)
                Add(symbol);
        }

        protected checked override void InsertItem(int index, string item)
        {
            if (Contains(item))
                return;
            base.InsertItem(index, item);
        }

        public SymbolListType ListType;
    }
}

#end-language
```

Albero derivazione:

```
program(0):
  not-method-line(0): using System;
  not-method-line(0): using System.Collections.Generic;
  not-method-line(0): using System.Collections.ObjectModel;
  not-method-line(0): using System.Text;
  not-method-line(0): using Ebnf.Language;
  not-method-line(0): namespace Ebnf
  not-method-line(0): {
  not-method-line(0): public enum SymbolListType
  not-method-line(0): {
  not-method-line(0): Inclusive,
  not-method-line(0): Esclusive
  not-method-line(0): }
  not-method-line(0): public class LanguageSymbolList : Collection<string>
  not-method-line(0): {
  not-method-line(0): public LanguageSymbolList()
  not-method-line(0): {
  not-method-line(0): }
  method(0): public wrapped void AddRange(EbnfSymbolCollection items)
        {
            foreach (EbnfSymbol symbol in items)
                Add(symbol.Name);
        }


    method-declaration(0): public wrapped void AddRange(EbnfSymbolCollection items)
      method-modifier(1): public
      method-modifier(9): wrapped
      type-name(0): void
      method-name(0): AddRange
      method-params(0): EbnfSymbolCollection items
        method-param(0): EbnfSymbolCollection items
          type-name(0): EbnfSymbolCollection
          param-name(0): items
    block(0):
        {
            foreach (EbnfSymbol symbol in items)
                Add(symbol.Name);
        }


  not-method-line(0): public void AddRange(IList<string> items)
  not-method-line(0): {
  not-method-line(0): foreach (string symbol in items)
  not-method-line(0): Add(symbol);
  not-method-line(0): }
  method(0): protected checked override void InsertItem(int index, string item)
        {
            if (Contains(item))
                return;
            base.InsertItem(index, item);
        }


    method-declaration(0): protected checked override void InsertItem(int index, string item)
      method-modifier(3): protected
      method-modifier(9): checked
      method-modifier(6): override
      type-name(0): void
      method-name(0): InsertItem
      method-params(0): int index, string item
        method-param(0): int index
          type-name(0): int
          param-name(0): index
        method-param(0): string item
          type-name(0): string
          param-name(0): item
    block(0):
        {
            if (Contains(item))
                return;
            base.InsertItem(index, item);
        }
```

```
  not-method-line(0): public SymbolListType ListType;
  not-method-line(0): }
  not-method-line(0): }
```

Output :

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Text;
using Ebnf.Language;

namespace Ebnf
{
    public enum SymbolListType
    {
        Inclusive,
        Esclusive
    }

    public class LanguageSymbolList : Collection<string>
    {
         public LanguageSymbolList()
         {
         }

        public void AddRange(EbnfSymbolCollection items)
        {
           //Codice di inizio metodo.
            foreach (EbnfSymbol symbol in items)
                Add(symbol.Name);
           //Codice di fine metodo.
        }

        public void AddRange(IList<string> items)
        {
            foreach (string symbol in items)
                Add(symbol);
        }

        protected override void InsertItem(int index, string item)
        {
           if (item == null)
              throw new NullPointerException();
           if (Contains(item))
                return;
            base.InsertItem(index, item);
       }

        public SymbolListType ListType;
    }
}
```