Federico Corrao, 523312

Programmazione Avanzata - Final Term - Simple Testing Framework

21/01/2015 - 01/02/2015

Il linguaggio scelto per lo svolgimento dell'elaborato è **C#** 4.0 (utilizzato con IDE Microsoft Visual C# 2010 Express).
Si allegano al presente documento due progetti distinti con, rispettivamente, il codice per gli esercizi 1-2-3 e 4.
Il codice è stato prodotto cercando di rispettare le convenzioni sui nomi descritte in **[1]** (*PascalCasing* per nomi di classi, membri, metodi pubblici; *camelCasing* per nomi di parametri).

Riepilogo

Le principali funzionalità di cui si richiede l'implementazione sono quattro:

1. Analizzare una stringa in input contenente una tabella codificata in HTML (mediante i tag **table**, **tr**, **td**)
2. Creare una rappresentazione della suddetta tabella basata sulle classi del Framework STF
3. Generare in modo automatico il codice sorgente di una classe C# che, una volta compilato, possa essere eseguito grazie al supporto del Framework STF
4. Eseguire il codice prodotto, ovvero completare una sessione di test e mostrarne i risultati.

Esercizio 1

La classe **Fixture** rappresenta una sessione di test. **ColumnFixture** rappresenta un tipo di **Fixture** in cui gli elementi da testare (*esempi*) sono rappresentanti tramite righe nella tabella. **Product** rappresenta una particolare **ColumnFixture** specializzata nell'eseguire il test sulle righe di una tabella *specifica* - è la classe da generare automaticamente.
Il testo dell'esercizio suggerisce due vincoli: *(1)* **Product** deve implementare un metodo **Check(Row)**, *(2)* **Product** deve derivare da una **ColumnFixture**. Sono state effettuate quindi le seguenti scelte:

Fixture - È stata definita come classe *astratta.* Definisce i metodi **Check(Row)** e **Execute(Table)**.
Non implementa alcun metodo (sarebbe possibile definirla in modo equivalente come interfaccia).

```
public abstract class Fixture
{
   public abstract bool Check(Row row);
   public abstract string Execute(Table table);
}
```

ColumnFixture - È stata anch'essa definita come classe astratta, ma implementa il metodo **Execute** ereditato da **Fixture**. Ciò significa che l'implementazione di **Check** è demandata alla classe **Product** (ovvero a qualsiasi sottoclasse di **ColumnFixture**). Notare come l'implementazione del metodo **Execute** faccia riferimento al metodo **Check** (ciò è possibile perché, essendo una classe astratta, **ColumnFixture** non può essere istanziata; l'invocazione concreta di **Execute** sarà fatta tramite **Product.Execute**, che utilizza **Product.Check**).

```
public abstract class ColumnFixture : Fixture
{
   public override string Execute(Table table)
   {
      List<bool> outcomes = new List<bool>();
      foreach (Row row in table)
         outcomes.Add(this.Check(row));
      return table.GetHTML(outcomes);
   }
}
```

Product - Come descritta nel testo. Si veda la soluzione all'**Esercizio 3** per il codice di esempio generato.

```
public class Product : ColumnFixture
{
   /* ... */
   public override bool Check(Row row) { /* ... */ }
}
```

Row e **Table** - La classe **Row** rappresenta una riga della tabella in input; contiene quindi i valori numerici presenti nelle rispettive celle. **Row** è una struttura eterogenea , poiché tali dati hanno un *tipo* associato (*int*, *float*, *double*, *string*) che varia da colonna a colonna e non è noto a priori. Si è scelto quindi di rappresentare la singola cella con il tipo

object (in un certo senso il più "generico") e di utilizzare **List<object>** come classe base per **Row**.
La classe **Table** è stata quindi definita come una **List<Row>**, ed esclude le prime tre righe della tabella in input (cui si attribuisce un signficato "speciale"), rappresentate come *attributi*. **Table** contiene inoltre il metodo **GenerateCode** (che "converte" la struttura della tabella in codice sorgente) e **GetHTML** (che ri-codifica la tabella in HTML aggiungendo informazioni sull'esito del test). Per l'implementazione, si veda la soluzione all'**Esercizio 3**.

```
public class Row : List<object> { }

public class Table : List<Row>
{
   public readonly string FixtureName;
   public readonly List<string> ArgNames = new List<string>();
   public readonly List<string> ArgTypes = new List<string>();

   public Table(string fixtureName, List<string> argNames, List<string> argTypes)
   {
      this.FixtureName = fixtureName;
      this.ArgNames = argNames;
      this.ArgTypes = argTypes;
   }
   public string GenerateCode() { /* ... */ }
   public string GetHTML(List<bool> outcomes) { /* ... */ }
}
```

Esercizio 2

Dato che il .NET Framework non fornisce alcun tokenizzatore (a differenza di **StringTokenizer** per Java), è stato implementato "a mano" un tokenizzatore banale (adatto esclusivamente agli scopi del progetto) che utilizza la classe **Regex** e i metodi **String.Split** e **String.Replace** - si veda il file *Lexer.cs* per l'implementazione.
La grammatica utilizzata per il parsing - in notazione BNF - è la seguente:

```
(1)     <Table>       ::=     TableBegin <RowList> TableEnd
(2)     <RowList>     ::=     <Row> <RowList> | <Row>
(3)     <Row>                 ::=     RowBegin <FieldList> RowEnd
(4)     <FieldList>   ::=     <Field> <FieldList> | <Field>
(5)     <Field>               ::=     FieldBegin <Text> FieldEnd
(6)     <Text>                ::=     Identifier | Number
```

Dove i terminali corrispondono alle stringhe "<table>", "</table>", "<tr>" etc; il simbolo non-terminale **<Table>** è l'assioma della grammatica. Una volta applicata la fattorizzazione sinistra alle regole *(2)* e *(4)*, la grammatica è LL(1). Tuttavia, è stata evitata l'introduzione di produzioni aggiuntive, modificando in modo oppotruno le procedure per il parsing di **<RowList>** e **<FieldList>**. Segue la definizione della **enum TokenType** e della classe **Token**, entrambe utilizzate durante il processo di analisi lessicale e sintattica:

```
[Flags]
public enum TokenType
{
   TableBegin = 1,        TableEnd = 2,
   RowBegin   = 4,        RowEnd = 8,
   FieldBegin = 16,       FieldEnd = 32,
   Identifier = 64,       Number = 128,
   Unrecognized = 256
}

public class Token
{
   public readonly TokenType Type;
   public readonly string Attribute;

   public Token(TokenType name, string attribute)
   {
      this.Type = name;
      this.Attribute = attribute;
   }
}
```

Il parser implementato svolge due compiti: *(1)* costruisce esplicitamente il *parse tree* tramite istanze di classi ausiliarie di

tipo **Node** (si veda il codice); *(2)* costruisce simultaneamente (per semplicità, onde evitare una ulteriore visita dell'albero sintattico) una "tabella" di **Token** (**List<List<Token>>**) da cui ricavare la **Table** vera e propria, tramite il metodo **GenerateTable**. Il codice è il seguente:

```
public class Parser {

/* ParseTree Node Classes */

private abstract class Node { }
private class NodeTable      : Node { public NodeRowList RowList; }
private class NodeRowList    : Node { public NodeRowList RowList; public NodeRow Row; }
private class NodeRow        : Node { public NodeFieldList FieldList; }
private class NodeFieldList  : Node { public NodeFieldList FieldList; public NodeField Field;}
private class NodeField      : Node { public NodeText Text; }
private class NodeText       : Node { public string Attribute; }
private class NodeIdentifier : NodeText { }
private class NodeNumber     : NodeText { }

/* Parse and GenerateTable Methods */

   private List<List<Token>> tempTable;
   private List<Token> tempRow;

   private List<Token> Tokens;
   private int Index;

   public Table Parse(string fixtureHTML)
   {
         this.Index = -1;
      this.Tokens = (new Lexer()).Tokenize(fixtureHTML);
      NodeTable parseTree = ParseTable();
      return GenerateTable();
   }

   private static Dictionary<string, Type> stringToType = new Dictionary<string, Type>()
      { { "int", typeof(int) }, { "float", typeof(float) } };

   private Table GenerateTable()
   {
         string fixtureName = tempTable[0][0].Attribute;
      List<string> argNames = new List<string>();
      List<string> argTypes = new List<string>();

         foreach (Token t in tempTable[1]) argNames.Add(t.Attribute);
         foreach (Token t in tempTable[2]) argTypes.Add(t.Attribute);
      tempTable.RemoveRange(0, 3);
      Table resultTable = new Table(fixtureName, argNames, argTypes);

      foreach (List<Token> l in tempTable)
      {
         Row r = new Row();
         for (int i = 0; i < argNames.Count; i++)
            r.Add(Convert.ChangeType(
               l[i].Attribute,
               stringToType[argTypes[i]],
               System.Globalization.CultureInfo.InvariantCulture));
         resultTable.Add(r);
      }
      return resultTable;
   }

/* Utilities */

   private void Match(TokenType expected)
   {
      Index++;
      if(!expected.HasFlag(Tokens[Index].Type))
```

```
            throw new Exception("Parser.Match: Expected " + expected.ToString() + " @ " + Index);
        }

        private bool Lookahead(TokenType expected)
        {
            return expected.HasFlag(Tokens[Index + 1].Type);
        }

        private Token GetToken() { return Tokens[Index]; }

    /* Non-Terminal Procedures */

        private NodeTable ParseTable()
        {
            tempTable = new List<List<Token>>();
            Match(TokenType.TableBegin);
            NodeTable tbl = new NodeTable { RowList = ParseRowList() };
            Match(TokenType.TableEnd);
            return tbl;
        }
        private NodeRowList ParseRowList()
        {
            NodeRowList rl = new NodeRowList { Row = ParseRow() };
            if (Lookahead(TokenType.TableEnd))
                rl.RowList = null;
            else if (Lookahead(TokenType.RowBegin))
                rl.RowList = ParseRowList();
            else throw new Exception("Parser.ParseRowList(): Expected TableEnd or RowBegin");
            return rl;
        }
        private NodeRow ParseRow()
        {
            tempRow = new List<Token>();
            Match(TokenType.RowBegin);
            NodeRow r = new NodeRow { FieldList = ParseFieldList() };
            Match(TokenType.RowEnd);
            tempTable.Add(tempRow);
            return r;
        }
        private NodeFieldList ParseFieldList()
        {
            NodeFieldList fl = new NodeFieldList { Field = ParseField() };
            if (Lookahead(TokenType.RowEnd))
                fl.FieldList = null;
            else if (Lookahead(TokenType.FieldBegin))
                fl.FieldList = ParseFieldList();
            else throw new Exception("Parser.ParseFieldList(): Expected RowEnd or FieldBegin");
            return fl;
        }
        private NodeField ParseField()
        {
            Match(TokenType.FieldBegin);
            NodeField f = new NodeField { Text = ParseText() };
            Match(TokenType.FieldEnd);
            return f;
        }
        private NodeText ParseText()
        {
            if(Lookahead(TokenType.Identifier | TokenType.Number))
            {
                Match(TokenType.Identifier | TokenType.Number);
                Token t = GetToken(); tempRow.Add(t);
                return (t.Type == TokenType.Identifier) ?
                    (NodeText)new NodeIdentifier { Attribute = t.Attribute }:
                    (NodeText)new NodeNumber     { Attribute = t.Attribute };
            }
            else throw new Exception("Parser.ParseText(): Invalid Identifier or Number");
```

```
    }
}
```

Da notare che eventuali vincoli sulla correttezza semantica della tabella sono stati omessi per brevità (si assume per convenzione che una ed una sola colonna della tabella abbia nome "result()"; che per ogni colonna sia definito correttamente uno e un solo nome; uno e un solo tipo; che tutte le righe abbiano stesso numero di campi; etc).

Esercizio 3

Come già mostrato, sia il generatore di codice C# che il generatore dell'output HTML sono implementati nei metodi **GenerateCode** e **GetHTML** della classe **Table**. Il test viene effettuato da **ColumnFixture.Execute**.

```
public class Table : List<Row>
{
        /* ... */
/* Code Generation */

   private const string codeTemplate =
      "using STF;\n\npublic class $FixtureName$ : ColumnFixture { \n" + "$Definitions$" +
      "\tpublic $ResultType$ result() { \n\t\t /* Insert code here */ \n\t} \n" +
      "\tpublic override bool Check(Row row) { \n$CheckBody$" +
      "\t\treturn (result() == ($ResultType$)row[$LastIndex$]);\n"+ "\t}\n} \n";

   public string GenerateCode()
   {
      string definitions = string.Empty;
      string checkBody = string.Empty;

      for (int i = 0; i < ArgNames.Count - 1; i++)
      {
         definitions += "\tpublic " + ArgTypes[i] + " " + ArgNames[i] + "; \n";
         checkBody += "\t\tthis." + ArgNames[i] + " = (" + ArgTypes[i] + ")row[" +
            i + "]; \n";
      }
      return codeTemplate
         .Replace("$FixtureName$", this.FixtureName)
         .Replace("$ResultType$", this.ArgTypes[this.ArgTypes.Count - 1])
         .Replace("$Definitions$", definitions)
         .Replace("$CheckBody$", checkBody)
         .Replace("$LastIndex$", (this.ArgNames.Count-1).ToString())
         .Replace("\n", "\r\n");
   }

/* HTML Generation */

   private const string htmlTemplate =
      "<table>\n\t<tr><td>$FixtureName$</td></tr>\n" +
      "\t<tr>$Names$</tr>\n\t<tr>$Types$</tr>\n$Rows$</table>\n";
   private const string styleTrue  = " style=\"background-color:lime;\"";
   private const string styleFalse = " style=\"background-color:red;\"";

   public string GetHTML(List<bool> outcomes)
   {
      string names = string.Empty;
      string types = string.Empty;
      string rows  = string.Empty;

      for (int i = 0; i < ArgNames.Count; i++)
      {
         names += "<td>" + ArgNames[i] + "</td> ";
         types += "<td>" + ArgTypes[i] + "</td> ";
      }
      int j = 0;
      foreach (Row r in this)
      {
         int i = 0; rows += "\t<tr>";
```

```
        foreach (object kv in r)
        {
           rows += "<td" + ((i == r.Count-1)? (outcomes[j]? styleTrue : styleFalse) : "") +
              ">" + kv.ToString() + "</td> ";
           i++;
        }
        j++; rows += "</tr>\n";
     }
     return htmlTemplate
        .Replace("$FixtureName$", this.FixtureName)
        .Replace("$Names$", names)
        .Replace("$Types$", types)
        .Replace("$Rows$", rows)
        .Replace(",", ".");
  }
}
```

Si riporta (a destra) il codice C# generato a partire dalla tabella di esempio (a sinistra), opportunamente modificato:

```
<table>
<tr><td>Product</td></tr>
<tr> <td>x</td> <td>y</td> <td>result()</td>
        </tr>
<tr>
   <td>float</td>
   <td>float</td>
   <td>float</td>
</tr>

<tr><td>1</td><td>2.4</td><td>3.4</td></tr
>

<tr><td>7.5</td> <td>42</td>
<td>315</td></tr>
<tr><td>42</td> <td>-7.5</td> <td>-
315</td></tr>
   <tr><td>28.7846</td> <td>3.14159</td>
<td>90.4283</td></tr>
</table>
```

```
using STF;

public class Product : ColumnFixture
{
   public float x;
   public float y;
   public float result()
   {
      /* Insert code here */
      return x * y;
   }
   public override bool Check(Row row)
   {
      this.x = (float)row[0];
      this.y = (float)row[1];
      return (result() == (float)row[2]);
   }
}
```

e l'output HTML (a sinistra) restituito da **Product.Execute**, visualizzato su un browser (a destra):

```
<table border="1" style="border-
collapse:collapse;"> <tr><td>Product</td></tr>
<tr><td>x</td> <td>y</td> <td>result()</td>
</tr>   <tr><td>float</td> <td>float</td>
<td>float</td> </tr>
        <tr><td>1</td> <td>2.4</td> <td
style="background-color:red;">3.4</td> </tr>
        <tr><td>7.5</td> <td>42</td> <td
style="background-color:lime;">315</td>
</tr>
        <tr><td>42</td> <td>-7.5</td> <td
style="background-color:lime;">-315</td>
</tr>
        <tr><td>28.7846</td>
<td>3.14159</td> <td style="background-
color:red;">90.4283</td> </tr></table>
```

Product		
x	y	result()
float	float	float
1	2.4	3.4
7.5	42	315
42	-7.5	-315
28.7846	3.14159	90.4283

Ecco un esempio di utilizzo del Framework STF:

```
static void Main(string[] args)
{
   string input = System.IO.File.ReadAllText("_input_fixture.txt");
   Table table = (new Parser()).Parse(input);
   string code = table.GenerateCode();
   // code viene salvato in un file .cs ed aggiunto al progetto, oppure può essere compilato
   // a run-time (e la classe Product direttamente istanziata) via Reflection
   // analogamente table può essere memorizzata e recuperata in un secondo momento

   Fixture p = new Product();
   string html_output = p.Execute(table);
}
```

Esercizio 4

Si richiede l'aggiunta di un *nuovo tipo* di **Fixture** (ovvero una sua specializzazione): a differenza di **ColumnFixture**, **ActionFixture** non prevede che venga effettuato un **Check** per ogni riga della tabella, poiché interpreta la tabella come insieme di istruzioni (l'esito del test è unico). È stato necessario apportare delle piccole modifiche al progetto originale (e alla grammatica, modificata per permettere la presenza di campi vuoti nella tabella).

Fixture e **ColumnFixture** - Il metodo **Check** è stato rimosso da **Fixture** (che altrimenti non potrebbe generalizzare **ActionFixture**) ed è stato inserito direttamente in **ColumnFixture**. Né la funzionalità di **ColumnFixture** né l'implementazione di **Execute** sono state alterate:

```
public abstract class Fixture
{
   public abstract string Execute(Table table);
}

public abstract class ColumnFixture : Fixture
{
   public abstract bool Check(Row r);
   public override string Execute(Table table) { /* ... */ }
}
```

ActionFixture - Così come **ColumnFixture** (che lascia l'implementazione di **Check** alla classe "concreta" **Product**), anche **ActionFixure** è una classe astratta, e in modo simile implementa il metodo **Execute** imponendo l'implementazione del codice di test (**Run**) alla classe generata (e.g. la classe di esempio **Action**). Si noti come la *signature* di **ActionFixture.Execute** (come pure la sua funzionalità) sia identica a quella di **ColumnFixture.Execute**.

```
public abstract class ActionFixture : Fixture
{
   public abstract bool Run();

   public override string Execute(Table table)
   {
      return table.GetHTML(new List<bool>() { this.Run() });
   }
}
```

Table, **ExampleTable** e **ActionTable** - La classe **Table** degli esercizi 1-2-3 è stata rinominata in **ExampleTable**. Sono quindi stati messi a fattore comune gli elementi generici nella "nuova" classe astratta **Table**.

```
public abstract class Table : List<Row>
{
   public readonly string FixtureName;

   public abstract string GenerateCode();
   public abstract string GetHTML(List<bool> outcomes);

   public Table(string fixtureName)
   {
```

```
        this.FixtureName = fixtureName;
    }
}

public class ExampleTable : Table
{
    public readonly List<string> ArgNames = new List<string>();
    public readonly List<string> ArgTypes = new List<string>();

    public ExampleTable(string fixtureName, List<string> argNames, List<string> argTypes)
        : base(fixtureName)
    {
        this.ArgNames = argNames;
        this.ArgTypes = argTypes;
    }

    /* ... */
}

public class ActionTable : Table
{
    public ActionTable(string fixtureName) : base(fixtureName) { }

    public override string GenerateCode() { /* ... */ }
    public override string GetHTML(List<bool> outcomes) { /* ... */ }
}
```

L'implmentazione di **GenerateCode** e **GetHTML** per **ActionTable** è molto simile a quella di **ExampleTable** (**Esercizio 3**) ed è stata omessa per brevità (si trova fra i file allegati). Di seguito il codice C# generato per **Action** (opportunamente arricchito a posteriori della classe *Accumulator* e delle funzioni *product*, *sqrt*). Da notare che l'identificatore _ è riservato alla memorizzazione dell'ultimo valore calcolato; e che il tipo di default per i valori numerici è **float**.

```
<table>

<tr><td>Action</td></tr>
<tr><td>start</td> <td>Accumulator</td>
<td>acc</td> <td></td> <td></td>
</tr>

<tr><td>call</td> <td></td><td>product</td>
<td>12</td> <td>12</td>  </tr>

<tr> <td>result</td> <td>acc</td>
<td>add</td> <td> </td> <td>
</td> </tr>

<tr> <td>call</td> <td></td>
<td>product</td> <td>7</td> <td>7</td>
</tr>
<tr>
<td>result</td> <td>acc</td> <td>add</td>
<td></td> <td></td>
</tr>

<tr> <td>result</td> <td></td> <td>sqrt</td>
<td></td><td></td>
</tr>
<tr> <td>check</td> <td>13.8924</td>
<td></td> <td></td> <td></td>
</tr>
</table>
```

```
using STF;
class Accumulator
{
    private float value;
    public float add(float v)
    {
        this.value += v;
        return this.value;
    }
}
public class Action : ActionFixture
{
    float product(float a, float b)
    { return a * b; }
    float sqrt(float x)
    { return (float)System.Math.Sqrt(x); }

    public override bool Run()
    {
        float _;
        Accumulator acc = new Accumulator();
        _ = product(12, 12);
        _ = acc.add(_);
        _ = product(7, 7);
        _ = acc.add(_);
        _ = sqrt(_);
        return (_ == 13.8924);
    }
```

```
}
```

Codice HTML generato da **ActionTable.GetHTML** e visualizzato in un browser (notare che il risultato *"13.8924439..."* è diverso da *"13.8924"* - in tabella dovrebbe essere presente una istruzione di arrotondamento affiché il test abbia esito positivo. Il codice di **Action** è stato testato per sqrt(3*3 + 4*4) == 5 con esito positivo):

```
<table border="1" style="border-collapse:collapse;"><tr><td>Action</td></tr>
<tr><td>start</td><td>Accumulator</td> <td >acc</td> <td ></td> <td ></td> </tr>
<tr><td>call</td><td ></td> <td >product</td> <td >12</td> <td >12</td> </tr>
<tr><td>result</td><td >acc</td> <td >add</td> <td ></td> <td ></td>
</tr><tr><td>call</td><td ></td> <td >product</td> <td >7</td> <td >7</td>
</tr><tr><td>result</td><td >acc</td> <td >add</td> <td ></td> <td ></td>
</tr><tr><td>result</td><td ></td> <td >sqrt</td> <td ></td> <td ></td> </tr>
<tr><td>check</td><td style="background-color:red;">13.8924</td> <td ></td> <td ></td>
<td ></td> </tr></table>
```

Action				
start	Accumulator	acc		
call		product	12	12
result	acc	add		
call		product	7	7
result	acc	add		
result		sqrt		
check	13.8924			

L'ultimo cambiamento degno di nota è dato dalla funzione **Parser.Parse**, che è stata resa generica (parametrica), e l'introduzione dei metodi **GenerateExampleTable** e **GenerateActionTable** che sostituiscono **GenerateTable**.

```
class Parser
{
   public Table Parse<T>(string fixtureHTML) where T : Fixture
   {
      /* ... */

      if (typeof(T) == typeof(ColumnFixture))
         return GenerateExampleTable();
      else if (typeof(T) == typeof(ActionFixture))
         return GenerateActionTable();
      else
         throw new Exception("Invalid fixture type");
   }
}
```

Esempio di utilizzo di STF con polimorfismo (**Table** al posto di **ActionTable**, **Fixture** al posto di **ActionFixture)**.

```
static void Main(string[] args)
{
   string input = System.IO.File.ReadAllText("_input_actionfixture.txt");
   Table table = (new Parser()).Parse<ActionFixture>(input);
   code = table.GenerateCode();

   Fixture a = new Action();
   string output_html = a.Execute(table);
}
```

Esercizio 5

Il design pattern Visitor consente di separare la definizione di un certo insieme di dati su cui operare e l'implementazione degli algoritmi che operano su tali dati. Come conseguenza, è possibile aggiungere delle operazioni ad una certa classe, senza dover modificare il codice di essa. Nel pattern Visitor, la classe contenente i dati è detta *Element*, mentre le operazioni sono effettuate da una classe *Visitor*. Una *Element* "accetta" un visitatore.

In termini di metodi e interfacce, una classe *Element* implementa l'interfaccia **IElement**, che specifica il metodo **Accept(IVisitor)**; mentre una classe *Visitor* implementa l'interfaccia **IVisitor**. Ogni classe "concreta" che supporta il pattern Visitor deve quindi implementare **IElement** con il proprio metodo **Accept**. L'interfaccia **IVisitor** specifica un metodo **Visit** per ogni tipo di *Element* che si desidera visitare. In questo modo un *Visitor* è in grado di visitare tutte le classi che supportano l'interfaccia **IElement**.

Un esempio di applicazione del pattern al framework STF è il seguente: si consideri la generazione del codice C# per ogni tabella (**ExampleTable**, **ActionTable**, ed altri eventuali sotto-tipi). Gli oggetti che si desidera visitare per generare il codice sono dunque le istanze delle classi relative alle tabelle, e le istanze delle classi relative alle righe (**ExampleRow**, **ActionRow**). Tali classi implementano quindi l'interfaccia **ITableElement**. Il generatore di codice (metodo **Table.GenerateCode**) è stato rimosso dalle classi delle tabelle e spostato in una classe **CodeGenerator** che implementa l'interfaccia **ITableVisitor**. Di seguito uno schema delle classi:

```
interface ITableElement
{
   void Accept(ITableVisitor tv);
}
abstract class Table : ITableElement { public abstract void Accept(ITableVisitor tv); }
abstract class Row   : ITableElement { public abstract void Accept(ITableVisitor tv); }

class ActionTable  : Table { public override void Accept(ITableVisitor tv) { tv.Visit(this); } }
class ExampleTable : Table { public override void Accept(ITableVisitor tv) { tv.Visit(this); } }
class ExampleRow   : Row   { public override void Accept(ITableVisitor tv) { tv.Visit(this); } }
class ActionRow    : Row   { public override void Accept(ITableVisitor tv) { tv.Visit(this); } }

interface ITableVisitor
{
   string Visit(ExampleTable et);
   string Visit(ActionTable at);
   string Visit(ExampleRow er);
   string Visit(ActionRow ar);
}
class CodeGenerator : ITableVisitor
{
   public string Visit(ActionTable at)
   {
      /* Generate code for ActionTable, exploiting ActionRow.Accept(this) */
   }
   public string Visit(ExampleTable et)
   {
      /* Generate code for ExampleTable, exploiting ExampleRow.Accept(this) */
   }
   public string Visit(ExampleRow er) { /* */ }
   public string Visit(ActionRow ar)  { /* */ }
}
```

Si noti come la visita di una tabella implichi la visita di ogni sua riga (per cui anche Row è un **ITableElement**).
In generale, è utile applicare il pattern Visitor in presenza di strutture da visitare, iterativamente o ricorsivamente.
Altre due possibili situazioni di applicazione del pattern sono: *(1)* nella generazione dell'output HTML (in modo analogo a quanto descritto sopra); *(2)* in una eventuale visita dell'albero sintattico dell'input, da effettuare ad esempio per la costruzione di una istanza di **Table**.

Durante lo svolgimento dell'esercizio, è stato fatto il tentativo di applicare il pattern Visitor anche alle classi **Fixture**, **ColumnFixture** e **ActionFixture** per quanto riguarda l'esecuzione della sessione di test (quindi in sostituzione di **Execute**). Tuttavia è stato notato come questa soluzione ponga dei problema progettuali non banali (brevemente: dato che parte del codice di test è generato automaticamente, ciò implica la generazione del codice di un Visitor e del metodo Visit relativo al particolare tipo della tabella che si vuole testare; ma l'implementazione di tutti gli altri metodi che il Visitor dovrebbe fornire - per ogni tipo di tabella - sarebbe assente).

Per quanto riguarda il *multiple-dispatch*: supponiamo di rimuovere dalle classi **ExampleTable**, **ActionTable**, **ExampleRow**, **ActionRow** l'implementazione concreta di **Accept**; e di far implementare **Accept** alle classi **Table** e **Row** (anziché alle sottoclassi). In questa situazione, durante la chiamata **tv.Visit(this)** il compilatore C# darà

errore, perché non sarà in grado di stabilire staticamente quale sia il metodo **Visit** corretto da chiamare ("the best overloaded method match has some invalid arguments").
Se il linguaggio supportasse il *dispatch* multiplo (o doppio, in questo caso) potrebbe stabilire dinamicamente a run-time quale metodo **Visit** di **CodeGenerator** chiamare, dopo aver valutato il tipo del primo parametro (non sarebbe più necessario quindi implementare **Visit** in ogni sottotipo, per ottenere il comportamento desiderato). In questo senso, il pattern Visitor può essere considerato come un modo per simulare il *multiple-dispatch* nei linguaggi che non lo supportano nativamente.

Note

Durante lo svolgimento degli esercizi 1-2-3-4 è stato dimenticato un particolare: i metodi **Execute** delle classi **ColumnFixture** e **ActionFixture** potrebbero essere sottoposti ad override, rispettivamente da **Product** e **Action**. Qualora non si desiderasse questo comportamento, è sufficiente dichiarare **Execute** con il modiicatore **sealed**.

Riferimenti

[1] "General Naming Conventions" https://msdn.microsoft.com/en-us/library/ms229045%28v=vs.110%29.aspx
[2] "C# Reference" https://msdn.microsoft.com/en-us/library/618ayhy6.aspx
[3] "Visitor Pattern" http://en.wikipedia.org/wiki/Visitor_pattern