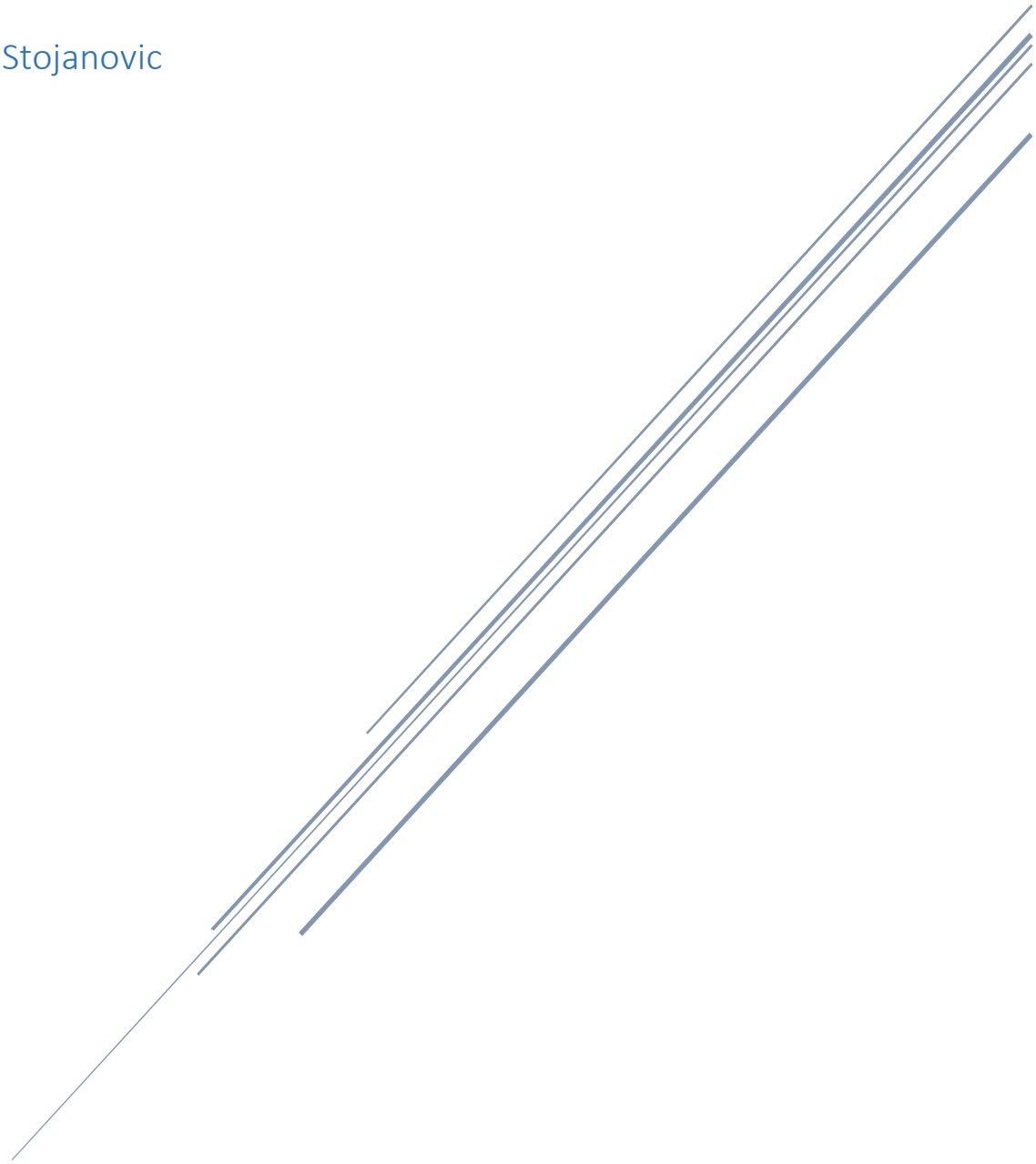
SIMPLE TESTING FRAMEWORK

Student Name: Luca Lovagni E-mail: lucalova91@gmail.com

*"I **visited** many places,*
Some of them quite
Exotic and far away,
But I always returned to myself."

— Dejan Stojanovic



University of Pisa
Advanced Programming

Nota per il lettore:

Oltre ai file riportati in questa relazione, sono stati scritti anche i *STFException.cs* dove vengono definite le eccezioni introdotte dal sistema e *MainProgram.cs* per l'esecuzionde del framework . Tuttavia, non sono riportati all'interno di questa relazione, in quanto non richiesti dagli esercizi del progetto. Inoltre l'insieme dei tipi utilizzabili in questa implementazione di STF è: **int, char, float, double, bool, String**.

Esercizio 1:

Dal momento che la generazione del codice HTML che avviene all'interno del metodo *execute(Table table)* definita in *ColumnFixture* nel file *Fixture.cs* fa utilizzo del visitor *HTMLGenerator*, si rimanda all'esercizio 3 per ulteriori dettagli su tale classe.

Table.cs:

```
public class Table
{
    public string name;
    public List<Row> rows;
    public Table(String name)
    {
        this.name = name;
        rows = new List<Row>();
    }
...
}
public class Row
{
    public List<Cell> columns;
    public Row()
    {
        columns = new List<Cell>();
    }
...
}
public enum Colors { WHITE , RED , GREEN};//used to color a cell
public class Cell
{
    public Colors color;
    public String value;
    public Cell(String value)
    {
        //by default a cell is white, it will be (eventually) colored by the fixture class (Column or
Action)
        this.color=Colors.WHITE;
        this.value=value;
    }
...
}
```

Fixture.cs:

```
public abstract class Fixture
{
    /// <summary>
    /// this method takesa a table and performs the test
    /// then a HTMLGenerator visitor is used in order to generate a table with colored cells
    /// finally, the generated HTML code is returned as a String
    /// </summary>
    /// <param name="table">the table used for the test(s)</param>
    /// <returns></returns>
    public abstract String execute(Table table);
}
public abstract class ColumnFixture : Fixture
{
    /// <summary>
    /// this method will be implemented by each (generated) class that extends ColumnFixture
    /// it takes a row and check that the value on the last cell is the same given by result()
    /// if the result is wrong then change the value with the one given by result()
    /// </summary>
    /// <param name="row">the row to be tested</param>
    /// <returns></returns>
    public abstract bool check (Row row);
    public override String execute (Table table)
    {
        HTMLGenerator visitor = new HTMLGenerator();
```

```
        for(int i=2;i<table.rows.Count;i++)
            if(check(table.rows[i]))//check in ColumnFixture is performed on each row
                table.rows[i].columns[table.rows[i].columns.Count-1].color = Colors.GREEN;
            else
                table.rows[i].columns[table.rows[i].columns.Count-1].color = Colors.RED;
        table.execute(visitor);
        return visitor.getHTML();
    }
}
```

Esercizio 2:

Token.cs:

```
public enum TokenType {OPEN_TABLE , CLOSED_TABLE , OPEN_TD , CLOSED_TD , OPEN_TR , CLOSED_TR , STRING ,
EOF};
public class Token
{
    public TokenType type;
    public string value;
    public Token (TokenType type)
    {
        this.type = type;
        this.value = null;
    }
    public Token(TokenType type, string value)
    {
        this.type = type;
        this.value = value;
    }
}
```

Scanner.cs:

```
public class Scanner
{
    static XmlTextReader file;//used to scan the HTML input file
    public Scanner(String filename)
    {
        file = new XmlTextReader(new StreamReader(filename));
    }
    /// <summary>
    /// iterators are used in order to start from where the last call ended
    /// so the element returned at each call is a token
    /// </summary>
    /// <returns></returns>
    public IEnumerator<Token> GetEnumerator()
    {
        while (file.Read())//until there are xml (html) tokens...
        {
            switch(file.NodeType)
            {
                case XmlNodeType.Element://open tag
                case XmlNodeType.EndElement://closed tag
                switch (file.Name)
                {
                    case "table":
                        if(file.NodeType.Equals(XmlNodeType.Element))
                            yield return new Token(TokenType.OPEN_TABLE);
                        else
                            yield return new Token(TokenType.CLOSED_TABLE);
                        break;
                    case "tr":
                        if(file.NodeType.Equals(XmlNodeType.Element))
                            yield return new Token(TokenType.OPEN_TR);
                        else
                            yield return new Token(TokenType.CLOSED_TR);
                        break;
                    case "td":
                        if(file.NodeType.Equals(XmlNodeType.Element))
                            yield return new Token(TokenType.OPEN_TD);
                        else
                            yield return new Token(TokenType.CLOSED_TD);
                        break;
```

```
                    default:
                        throw new STFScannerException("unknown tag: "+file.Name);
                }
                break;
                case XmlNodeType.Text://string between two tags
                    yield return new Token(TokenType.STRING,file.Value);
                    break;
                case XmlNodeType.Whitespace://ignore whitespaces outside of tags
                    break;
                default:
                    throw new STFScannerException("unkown node: "+file.NodeType.ToString());
            }
        }
        yield return new Token(TokenType.EOF);//last generated token is End Of File
    }
}
```

Parser.cs:

```
public class Parser
{
    //the scanner gives a token on demand to the parser through an iterator
    IEnumerator<Token> iterator;
    Token token;//the actual token
    public Parser(Scanner scanner)
    {
        this.iterator = scanner.GetEnumerator();
        this.token = nextToken();//initialize token
    }

    public Token nextToken()
    {
        //generate the next token
        iterator.MoveNext();//there is no need to test MoveNext()
        return iterator.Current;//get the generated token
    }
    public void expect(TokenType type)
    {
        if(type!=token.type)
            throw new STFParserException("expected "+type+" got "+token.type);
        token = nextToken();
    }
    //Table -> <table> <tr> <td> String </td> </tr> Rows </table>
    public Table ParseTable()
    {
        expect(TokenType.OPEN_TABLE);
        expect(TokenType.OPEN_TR);
        expect(TokenType.OPEN_TD);
        string name = token.value;
        expect(TokenType.STRING);
        Table table = new Table(name);
        expect(TokenType.CLOSED_TD);
        expect(TokenType.CLOSED_TR);
        table.rows = this.ParseRows();
        expect(TokenType.CLOSED_TABLE);
        expect(TokenType.EOF);
        return table;
    }
    //Rows -> <tr> Row </tr>  Rows | epsilon
    public List<Row> ParseRows()
    {
        List<Row> rows = new List<Row>();
        while(token.type!=TokenType.CLOSED_TABLE)//else Rows->epsilon and follow(Rows)={</table>}
        {
            expect(TokenType.OPEN_TR);
            rows.Add(this.ParseRow());
            expect(TokenType.CLOSED_TR);
        }
        return rows;
    }
    //Row -> <td> Cell </td> Row | epsilon
    public Row ParseRow()
    {
        Row row = new Row();
```

```
        row.columns = new List<Cell>();
        while(token.type!=TokenType.CLOSED_TR)//else Rows->epsilon and follow(Row)={</tr>}
        {
            expect(TokenType.OPEN_TD);
            row.columns.Add(Parsecell());
            expect(TokenType.CLOSED_TD);
        }
        return row;
    }
    //Cell -> String | epsilon
    public Cell Parsecell()
    {
        Cell cell;
        if(token.type==TokenType.CLOSED_TD)//Cell->epsilon and follow(Cell)={</td>}
            cell = new Cell("");
        else
        {
            cell = new Cell(token.value);
            expect(TokenType.STRING);
        }
        return cell;
    }
}
```

Esercizio 3:

Allo scopo di poter generare il codice della classe estensione di *ColumnFixture* e la generazione della tabella HTML, l'autore ha utilizzato il visitor pattern. Di seguito è riportato il codice aggiunto alle classi precedentemente definite, oltre il codice e le interfacce dei due Visitor qui introdotti.

IVisitable.cs:

```
public interface IVisitable
{void execute (IVisitor visitor);}
```

IVisitor.cs:

```
public interface IVisitor
{
    void visit (Cell cell);
    void visit (Table table);
    void visit (Row row);
}
```

Table.cs:

```
public class Table : IVisitable
{
...
    /// <summary>
    /// Visitor Pattern Method. Is used in order to explore the whole table
    /// First visit each row, then the table
    /// </summary>
    /// <param name="visitor">the visitor used</param>
    public void execute(IVisitor visitor)
    {
        foreach(Row row in rows)
            row.execute(visitor);
        visitor.visit(this);
    }
}
public class Row : IVisitable
{
...
    /// <summary>
    /// Visitor Pattern method. Is used to explore the entire row
    /// First visit each cell, then the row
    /// </summary>
    /// <param name="visitor"></param>
    public void execute(IVisitor visitor)
    {
        foreach(Cell cell in columns)
            cell.execute(visitor);
        visitor.visit(this);
    }
```

```
}
public class Cell : IVisitable
{
    ...
    //this set of methods implement an "ad-hoc" polymorphism in order to convert
    //the content of a cell (string) in one of the other 6 possible types used in STF
    public void convert (ref int number)
    { number = Convert.ToInt32(value);}
    public void convert (ref float number)
    { number = Convert.ToSingle(value);}
    public void convert (ref double number)
    { number = Convert.ToDouble(value);}
    public void convert (ref char character)
    { character = Convert.ToChar(value);}
    public void convert (ref bool boolean)
    { boolean = Convert.ToBoolean(value);}
    public void convert(ref String mystring)
    { mystring = value;}
    /// <summary>
    /// Visitor Pattern Method, visit the actual cell
    /// </summary>
    /// <param name="visitor"></param>
    public void execute(IVisitor visitor)
    {visitor.visit(this);}

}
```

ColumnFixtureGenerator.cs:

```
public class ColumnFixtureGenerator : IVisitor
{
    private int row;//number of row already visited
    private StringBuilder check,code;
    private List<String> types , variables;
    public ColumnFixtureGenerator()
    {
        row=0;
        check = new StringBuilder();//check will contain the code related to check()
        code = new StringBuilder();//code will contain the rest of the class (heading+field declaration)
        types = new List<String>();// set of types used in the test
        variables = new List<String>();// set of variables used in the test
    }
    public String getCode()
    {
        return code.ToString();
    }
    public void visit(Cell cell)
    {
        if(row>1)//we're not interested in data rows
            return;
        if(row==0)//first row: variables row
        //supposing that last cell is always result() (maybe with a different name)
            variables.Add(cell.value);
        else//second row: types row
            types.Add(cell.value);
    }
    public void visit(Row visitable)
    {
        row++;//row is used by visit(Cell) to understand if to add the content to variables or types
    }
    public void visit(Table visitable)
    {
        code.Insert(0,"namespace Simple_Testing_Framework {"+Environment.NewLine+"public class
"+visitable.name+" : ColumnFixture {"+Environment.NewLine);//initial part of class
        check.AppendLine("public override bool check(Row row){");//initial part of check method
        //if everything is fine, variables and types have same elements (ArrayOutOfBoundException
otherwise)
        int max = Math.Max(types.Count,variables.Count);
        for(int i=0;i<max;i++)//the last one is result()
        {
            if(i<max-1)//normal column
            {
                code.AppendLine("public "+types[i]+" "+variables[i]+";");//adding i-th field
```

```
                check.AppendLine("row.columns["+i+"].convert(ref "+variables[i]+");");
            }
            else//last column (the column of result)
            {
                //expected_value will contain the value which have to be tested
                code.AppendLine("public "+types[i]+" expected_value;");
                check.AppendLine("row.columns["+i+"].convert(ref expected_value);");
                //creating the skeleton of result()
                code.AppendLine("public "+types[i]+" "+variables[i]+" {"+Environment.NewLine+"}");
                //result_value will contain the value returned by result()
                check.AppendLine(types[i]+" result_value= "+variables[i]+";");
                check.AppendLine("if(result_value.Equals(expected_value))");
                check.AppendLine("return true;");
                check.AppendLine("else {");
                check.AppendLine("row.columns["+i+"].value= result_value.ToString();");
                check.AppendLine("return false;"+Environment.NewLine+"}"+Environment.NewLine);
            }
        }
        code.AppendLine(check.ToString()+"}"+Environment.NewLine+"}"+Environment.NewLine+"}");
    }
}
```

Codice generato per l'esempio iniziale:

```
public class Product : ColumnFixture
{
    public float x;
    public float y;
    public float expected_value;
    public float result()
    {
    }
    public override bool check(Row row)
    {
        row.columns[0].convert(ref x);
        row.columns[1].convert(ref y);
        row.columns[2].convert(ref expected_value);
        float result_value = result();
        if (result_value.Equals(expected_value))
            return true;
        else
        {
            row.columns[2].value = result_value.ToString();
            return false;
        }

    }
}
```

HTMLGenerator.cs:

```
public class HTMLGenerator : IVisitor
{
    StringBuilder code,line;
    public HTMLGenerator()
    {
        //the entire code generated until now
        code = new StringBuilder();
        //used to wrap the code of a single table line between <tr> </tr>
        line = new StringBuilder();
    }

    public String getHTML()
    {
        return code.ToString();
    }
    /// <summary>
    /// Visitor Pattern Method. Insert the table tags and the table name
    /// </summary>
    /// <param name="table">the table to be visited</param>
    public void visit (Table table)
    {

code.Insert(0,"<table>"+Environment.NewLine+"<tr><td>"+table.name+"</td></tr>"+Environment.NewLine);
        code.AppendLine("</table>");
```

```
    }
    /// <summary>
    /// Wrap the html code contained in "line" with row tags, then reset the code of the line
    /// </summary>
    /// <param name="row">the row to be visited</param>
    public void visit (Row row)
    {
        //line contains the code generated by each cell (since they are all already visited)
        code.AppendLine("<tr>"+line.ToString()+"</tr>");
        line.Clear();
    }
    /// <summary>
    /// if the cell is colored, then generates the HTML attribute of the relative color
    /// then wrap the cell value inside cell tags
    /// </summary>
    /// <param name="cell">the cell to be visited</param>
    public void visit (Cell cell)
    {
        line.Append("<td");
        if(cell.color.Equals(Colors.GREEN))
            line.Append(" bgcolor=#00FF00 ");
        else if(cell.color.Equals(Colors.RED))
            line.Append(" bgcolor=#FF0000 ");
        line.AppendLine("> "+cell.value+" </td>");
    }
}
```

Esercizio 4:

Anche in questo caso, la generazione della classe estensione di ActionFixture viene generate tramite un visitor dedicato.

Fixture.cs:

```
public abstract class ActionFixture : Fixture
{
    /// <summary>
    /// Same as ColumnFixture.check() but here we don't need any parameters
    /// </summary>
    /// <returns></returns>
    public abstract bool check ();
    public override string execute(Table table)
    {
        HTMLGenerator visitor = new HTMLGenerator();
        if(check())//check in ActionFixture is performed on the whole table
            table.rows[table.rows.Count-1].columns[1].color = Colors.GREEN;
        else
            table.rows[table.rows.Count-1].columns[1].color = Colors.RED;
        table.execute(visitor);
        return visitor.getHTML();
    }
}
```

ActionFixtureGenerator.cs:

```
public class ActionFixtureGenerator : IVisitor
{
    int column;//the first column is the command type,set and used by visit(Cell)
    String command;//set by visit(Cell) when we visit the cell in the first column, used by visit(Row)
    List<String> arguments;//list of single command arguments (cells value starting from the second
column)
    StringBuilder line , code;
    public ActionFixtureGenerator()
    {
        column=0;
        line = new StringBuilder();//chain of commands accumulated until now. It's reset with start/call
commands
        code = new StringBuilder();//total code generated
        arguments = new List<String>();
    }
    /// <summary>
    /// creates the string of the code generated until now
    /// </summary>
    /// <returns>the string containing the code generated</returns>
    public String getCode()
    {
```

```
        return code.ToString();
    }
    /// <summary>
    /// This method generates the code relative to a single method/function call and save it in the
command chain "line"
    /// </summary>
    public void call()
    {
        //arguments[0]: OPTIONAL object name arguments[1]: function/method name
        if(!arguments[0].Equals(""))//if true, then it's an object
            line.Append(arguments[0]+".");
        line.Append(arguments[1]+"(");//method/function name
        if(arguments.Count>=3)//at least one argument
            line.Append(arguments[2]);
        for(int i=3;i<arguments.Count&&arguments[i]!="";i++)//generating other arguments
            line.Append(","+arguments[i]);
        line.Append(")");
    }
    /// <summary>
    /// Visitor Pattern method used to visit a single cell.
    /// If it's the cell of the first column, then the value must be a command
    /// Otherwise it is a command argument
    /// </summary>
    /// <param name="cell">The cell to be visited</param>
    public void visit(Cell cell)
    {
        if(column==0)//first column: command name
            if(new[] {"start","call","result","check"}.Contains(cell.value))
                command = cell.value;
            else
                throw new FormatException("command \""+cell.value+"\" doesn't exist");
        else//otherwise command argument
            arguments.Add(cell.value);
        column++;
    }
    /// <summary>
    /// Visitor Pattern method used to visit a Row, after all the cell of this Row are already visited
    /// Now we have the necessary amount of information to generates the next instruction of check()
    /// </summary>
    /// <param name="row">The row to be visited</param>
    public void visit(Row row)
    {
        column = 0;//next cell visited will be the first of a new row
        //if the command is "start" or "call" generates the chain of commands "accumulated" until now (if
any)
        if (new[] { "start" , "call" }.Contains(command)&&line.Length!=0)
        {
            code.AppendLine(line.ToString()+";");
            line.Clear();
        }
        switch(command)
        {
            case "start"://arguments[0]: class name arguments[1]: object name
                code.AppendLine(arguments[0]+" "+arguments[1]+" = new "+arguments[0]+"();");
                break;
            case "call":
                call();//generates the code relative to this row
                break;
            case "result":
                string actualLine = line.ToString();//save the chain generated until now
                line.Clear();//call will save the code generated in "line"
                call();//generates the code relative to this row
                //use the chain generated until now as first argument
                line.Insert(line.ToString().IndexOf("(")+1,actualLine);
                break;
            case "check"://arguments[0]: value to check
                //suppposing that check is the last command of the table
                code.AppendLine("return result("+line.ToString()+");"+Environment.NewLine+"}");
                code.AppendLine("public bool result(object o){return o.Equals("+arguments[0]+");}");
                break;
        }
        arguments.Clear();
    }
```

```
    /// <summary>
    /// Visitor Pattern method used to visit a Table, after all the others element are already visited
    /// It generates the head of the class and the signature of check()
    /// </summary>
    /// <param name="table">The table to be visited</param>
    public void visit(Table table)
    {
        code.Insert(0,"public class "+table.name+" : ActionFixture {"+Environment.NewLine+"public
override bool check(){"+Environment.NewLine);
        code.AppendLine("}");
    }
}
```

Codice generato per l'esempio iniziale:

```
public class Action : ActionFixture
{
    public override bool check()
    {
        Accumulator acc = new Accumulator();
        acc.add(product(12, 12));
        return result(sqrt(acc.add(product(7, 7))));
    }
    public bool result(object o){return o.Equals(13.8924);}
}
```

Esercizio 5:

L'autore ha utilizzato [1] come testo di riferimento per ottenere *parte* delle informazioni necessarie allo svolgimento dell'esercizio. Il **Visitor Pattern** fa parte della famiglia dei **behavioral design pattern**, ovvero pattern dove viene descritto come oggettti diversi comunicano tra loro e come i passaggi di task differenti vengono divisi tra oggetti diversi.

L'implementazione del Visitor Pattern si basa sull'utilizzo delle interfacce fornite nei linguaggi di programmazione Object Oriented e può essere riassunta nello schema riportato in Figura 1.

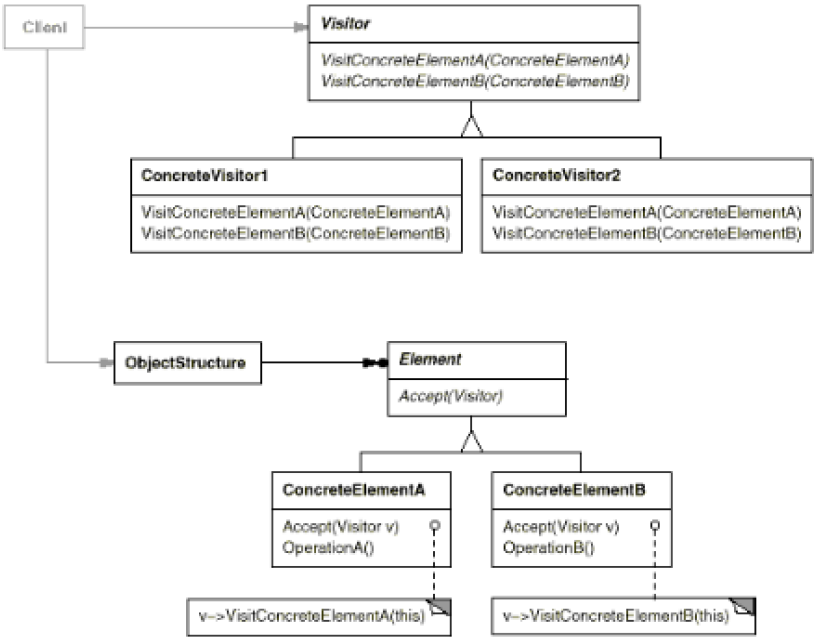


Figura 1: Visitor Pattern in UML

In particolare verranno definite due interfacce:

1. L'interfaccia **Element** che definisce il metodo *accpet(Visitor)*. Esso viene utilizzato per comunicare al *Visitor* quali elementi visitare all'interno dell'elemento in questione, o in altre parole quali elementi potrebbero essere di "interesse" per il Visitor e su cui devono essere effettuate delle operazioni (sfruttando l'overloading come vedremo in seguito).
2. L'interfaccia **Visitor** dichiara un operazione *visit(ConcreteElement)* per ogni classe *ConcreteElement* definita. Questo meccanismo permette al Visitor di eseguire la versione di *visit*() corretta a seconda dell'elemento utilizzato come parametro.
3. La **ConcreteElement** è la classe che implementa l'interfaccia *Element* e che dunque implementa il metodo *accpet(Visitor)*. Una tipica implementazione (ma che può variare a seconda della struttura di *ConcreteElement*) di *accept()* può essere:
   ```
   public void accept (Visitor visitor)
   { visitor.visit(this); }
   ```
4. Il **ConcreteVisitor** implementa l'interfaccia *Visitor* e dunque il metodo *visit()* per ciascun *ConcreteElement* che implementa l'interfaccia *Element*. Ciascuna *visit()* implementa una parte dell'algoritmo definito per la corrispondente classe nella struttura definita. Il ConcreteVisitor può accumulare uno stato. Il ConcreteVisitor può essere utilizzato anche per l'implementazione di traversal algorithm nel caso in cui la complessità dell'algoritmo in questione sia elevata (ad esempio possa dipendere dallo stato del ConcreteVisitor). Utilizzando la tipica implementazione del metodo *accept()* fornita nel punto precedente, il ConcreteVisitor è in grado di chiamare la corretta versione di visit a seconda del *ConcreteElement* che viene utilizzato come parametro. Questa strategia sottolinea l'utilizzo dell'overloading nel Visitor Pattern.

Il Visitor Pattern introduce una serie di vantaggi:

- Dalla precedente struttura si evince come il Visitor Pattern permetta di separare un algoritmo che opera su una certa struttura dalla struttura stessa. In questo modo, qualora volessimo aggiungere delle nuove operazioni sulle classi che abbiamo definito (o tali operazioni non fossero ben conosciute a priori), basterà definire un nuovo *ConcreteVisitor* che implementa tali operazioni.

- Tramite il Visitor pattern è possibile visitare oggetti in una struttura che non hanno una super classe in comune.
- Dallo schema precedente è chiaro come vi sia una divisione tra *traversal algorithms* (che si occupano di navigare all'interno della struttura) ed algoritmi che operano sulla struttura stessa, rendendo il codice più chiaro ed una maggiore "divisione delle responsabilità".
- Operazioni correlate tra loro appaiono tutte all'interno della stessa classe (il visitor), viceversa operazioni scorrelate appartengono a visitor diversi, rendendo l'implementazione di algoritmi più facile e chiara.
- Il meccanismo di overloading descritto in precedenza utilizzato dal Visitor permette di effettuare operazioni diverse tra oggetti eterogenei tra loro in base alla loro classe di appartenenza: si pensi al caso in cui abbiamo un insieme di oggetti appartenenti a classi differenti e volessimo effettuare operazioni diverse a seconda della loro classe di appartenenza.
- Un altro caratteristica estremamente importante del Visitor Pattern è l'implementazione del double-dispatch in linguaggi a single-dispatch.

In particolare l'ultimo punto merita un maggior approfondimento. Nei linguaggi a single-dispatch (come C++,Java o C#) l'operazione eseguita dipende da due fattori: il nome della richiesta ed il nome di colui che riceve la richiesta. Ad esempio, effettuare la richiesta (metodo) *emitSound* viene eseguita in modo diversa a seconda del ricevente, ad esempio *Human.emitSound()* oppure *Dog.emitSound()* etc.
In un linguaggio a double-dispatch l'operazione eseguita dipende da *due* riceventi.

Per capire meglio le limitazioni dei linguaggi a signle dispatch supponiamo di voler scrivere una funzione del tipo:
```
void process( virtual Base object1, virtual Base object2 )
```
La quale effettua operazioni diverse a seconda del tipo dei due oggetti utilizzati come argomenti, il tutto a run-time. Il problema in questo esempio è che la parola virtual *non* può essere utilizzata allo scopo di richiedere un binding dinamico degli oggetti passati come argomento. Una soluzione per rendere ciò possibile in un linguaggio single-dispatch è fare in modo che ciascun oggetto sia il ricevenete della chiamata di una funziona virtuale (esempio tratto da [2]). Ciò è esattamente ciò che avviene nei metodi di *accept()* e *visit()* utilizzati nel visitor pattern. Infatti nel visitor pattern le operazioni eseguite dipendono da due riceventi che vengono scelti a run-time, ovvero il tipo di visitor *ed* il tipo di elemento che esso visita (ciò risulta infatti il concetto fondamentale del visitor pattern). In un linguaggio dove il double-dispatch risulta nativo (come ad esempio Lisp) il visitor pattern risulta più facile (se non addirittura inutile) da implementare perchè non è necessario il meccanismo di metodi accept/visit ma un'unica operazione che dipende da due riceventi.

Tuttavia il Visitor Pattern è particolarmente sconsigliato quando la struttura delle classi (o l'insieme delle classi visitiabili) cambia con una certa frequenza. Infattti introdurre un nuovo *ConcreteElement* comporterebbe dover modificare l'interfaccia *Visitor* e tutti i *ConcreteVisitor* che la implementano.

In questa implementazione di STF l'autore ha utilizzato il Visitor Pattern in 3 occasioni:
- Generazione della classe che estende la *ColumnFixutre* (esercizio 3)
- Generazione della tabella HTML risultante dallo svolgimento dei test (esercizio 3)
- Generazione della classe che estende la *ActionFixture* (esercizio 4)

In particoalre, il Visitor Pattern si è rivelato particolarmente utile nel definire il generatore della classe estensione di *ActionFixture*: introdurre una nuova funzionalità nel sistema (o in altre parole estenderlo) ha richiesto semplicemente la definizione di un nuovo visitor che implementava le operazioni desiderate, senza modificare la struttura delle classi coinvolte o gli algoritmi di visita della tabella.

Bibliography

[1] E. Gamma, R. Helm, R. Johnson e J. Vissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1997.

[2] «Source Making, Visitor in Java: Double dispatch (within a single hierarchy),» [Online]. Available: http://sourcemaking.com/design_patterns/Visitor/java/2.