Advanced Programming

Antonio Mallia - 423458

mallia.antonio [at] gmail.com

**Esercizio 1**

Il codice seguente definisce il modello che rappresenta una generica tabella HTML. Per semplicità viene definito tutto public così da non dover scrivere un getter e un setter per ciascun campo.

Ciascun elemento di una tabella implementa un HTMLTag, interfaccia che contiene il metodo toHTML(IHTMLGenerator htmlGen) che servirà più avanti per la ricostruzione del codice HTML originale.

La classe Table include, inoltre, i metodi per generare il codice e per eseguire un test. Tali metodi non sono necessari per la risoluzione dell’esercizio, ma al fine di rendere l’elaborato più leggibile sono mostrati in anticipo.

**public** **interface** HTMLTag {

**public** String toHTML(IHTMLGenerator htmlGen);

}

**public** **class** Table **implements** HTMLTag {

**public** List<Row> rows = **new** ArrayList<Row>();

**public** String execute(Fixture f) {

**return** f.execute(**this**);

}

**public** String generateClass(CodeGenerator cg) {

**return** cg.generateClass(**this**);

}

**public** String generateTest(CodeGenerator cg) {

**return** cg.generateTest(**this**);

}

**public** String toHTML(IHTMLGenerator htmlGen) {

**return** htmlGen.toHTML(**this**);

}

}

**public** **class** Row **implements** HTMLTag {

**public** List<Cell> cells = **new** ArrayList<Cell>();

**public** String toHTML(IHTMLGenerator htmlGen) {

**return** htmlGen.toHTML(**this**);

}

}

La classe Cell comprende inoltre dei metodi per la converione del contenuto della cella, la stringa text, al tipo desiderato. È stato sfruttato l’overload del metodo convert(). Vedremo il fine di tale metodo nell’esercizio 3.

**public** **class** Cell **implements** HTMLTag {

**public** **enum** Colour {

***NONE***, ***GREEN***, ***RED***;

}

**public** String text;

**public** Colour colour;

**public** Cell(String text) {

**this**(text, Colour.***NONE***);

}

**public** Cell(String text, Colour colour) {

**this**.text = text;

**this**.colour = colour;

}

**public** String toHTML(IHTMLGenerator htmlGen) {

**return** htmlGen.toHTML(**this**);

}

**public** Integer convert(**int** i) {

**return** **new** Integer(text);

}

**public** Long convert(**long** l) {

**return** **new** Long(text);

}

**public** Double convert(**double** d) {

**return** **new** Double(text);

}

**public** Float convert(**float** f) {

**return** **new** Float(text);

}

**public** Boolean convert(**boolean** b) {

**return** **new** Boolean(text);

}

**public** Byte convert(**byte** b) {

**return** **new** Byte(text);

}

**public** Short convert(**short** s) {

**return** **new** Short(text);

}

**public** String convert(String s){

**return** text;

}

}

Fixture è pensata per essere un’interfaccia che espone un metodo astratto, execute(Table table), il quale, nella sua implementazione in ColumnFixture, avrà il compito di eseguire i test riga per riga e ritornare l’oggetto Table covertito in codice HTML.

**public** **interface** Fixture {

**public** String execute(Table table);

}

**public** **abstract** **class** ColumnFixture **implements** Fixture {

**private** **final** **static** **int** ***DATA\_START\_ROW*** = 3;

**public** **abstract** **boolean** check(Row row);

**public** String execute(Table table) {

**for** (**int** i = ***DATA\_START\_ROW***; i < table.rows.size(); i++) {

**if** (check(table.rows.get(i))) {

table.rows.get(i).cells.get(table.rows.get(i).cells.size() - 1).colour = Colour.***GREEN***;

} **else** {

table.rows.get(i).cells.get(table.rows.get(i).cells.size() - 1).colour = Colour.***RED***;

}

}

**return** table.toHTML(**new** HTMLGenerator());

}

}

**Esercizio 2**

Il parser del codice HTML è stato implementato traendo ispirazione da quanto visto a lezione. Uno scanner o tokenizer, che prende come imput una stringa contenente il codice HTML della tabella, si occupa della generazione dei Token. Questi ultimi, sono oggetti contenenti il tipo di token e, se presente, il suo valore.

La separazione dei token è stata fatta sfruttando lo StringTokenizer, dando come delimitatori le parentesi angolari che contraddistinguono un tag HTML e. ovviamente, i caratteri della formattazione.

**public** **class** Tokenizer {

**public** Token token;

**public** StringTokenizer st;

**public** Tokenizer(String input) {

st = **new** StringTokenizer(input, "<>\t\n\r");

}

**public** Token nextToken() {

String next = **new** String();

**if** (st.hasMoreTokens()) {

next = st.nextToken();

**switch** (next) {

**case** "table":

token = **new** Token(TokenType.***OPEN\_TABLE***);

**break**;

**case** "/table":

token = **new** Token(TokenType.***CLOSE\_TABLE***);

**break**;

**case** "tr":

token = **new** Token(TokenType.***OPEN\_ROW***);

**break**;

**case** "/tr":

token = **new** Token(TokenType.***CLOSE\_ROW***);

**break**;

**case** "td":

token = **new** Token(TokenType.***OPEN\_COLUMN***);

**break**;

**case** "/td":

token = **new** Token(TokenType.***CLOSE\_COLUMN***);

**break**;

**default**:

token = **new** Token(TokenType.***CELL***, next);

**break**;

}

} **else** {

token = **new** Token(TokenType.***EOF***);

}

**return** token;

}

}

**public** **class** Token {

**public** **enum** TokenType {

***OPEN\_TABLE***, ***CLOSE\_TABLE***, ***OPEN\_ROW***, ***CLOSE\_ROW***, ***OPEN\_COLUMN***, ***CLOSE\_COLUMN***, ***CELL***, ***EOF***;

}

**public** TokenType tokenType;

**public** String value;

**public** Token(TokenType tokenType, String value) {

**this**.tokenType = tokenType;

**this**.value = value;

}

**public** Token(TokenType tokenType) {

**this**.tokenType = tokenType;

**this**.value = **null**;

}

**public** **boolean** isType(TokenType tokenType) {

**return** **this**.tokenType.equals(tokenType);

}

}

Anche il parser rispecchia quello visto a lezione. Si presuppone la presenza di una sola tabella e niente altro. Ovviamente sia le righe che le colonne possono essere di numero indefinito, ma è stato previsto un controllo sul numero delle colonne visto che dalla seconda riga in poi questo deve essere uguale per ogni riga. Errori sintattici interrompono l’esecuzione sollevando una SyntaxException, definita a parte.

**public** **class** Parser {

**private** Tokenizer tokenizer;

**private** Token token;

**private** **int** columnNumber = 0;

**public** Parser(String s) **throws** Throwable {

tokenizer = **new** Tokenizer(s);

token = tokenizer.nextToken();

}

**public** Table doTable() **throws** Throwable {

Table table = **new** Table();

expect(TokenType.***OPEN\_TABLE***);

table.rows.add(doHeader());

**while** (token.isType(TokenType.***OPEN\_ROW***)) {

table.rows.add(doRow());

}

expect(TokenType.***CLOSE\_TABLE***);

expect(TokenType.***EOF***);

**return** table;

}

**private** Row doRow() **throws** Throwable {

Row row = **new** Row();

expect(TokenType.***OPEN\_ROW***);

**int** c = 0;

**while** (token.isType(TokenType.***OPEN\_COLUMN***)) {

row.cells.add(doCell());

c++;

}

**if** (columnNumber > 0) {

**if** (columnNumber != c) {

**throw** **new** SyntaxException("Il numero di celle non corrisponde");

}

} **else** {

columnNumber = c;

}

expect(TokenType.***CLOSE\_ROW***);

**return** row;

}

**private** Cell doCell() **throws** Throwable {

Cell cell;

expect(TokenType.***OPEN\_COLUMN***);

**if** (token.value != **null**) {

cell = **new** Cell(token.value);

expect(TokenType.***CELL***);

} **else** {

cell = **new** Cell("");

}

expect(TokenType.***CLOSE\_COLUMN***);

**return** cell;

}

**public** Row doHeader() **throws** Throwable {

Row row = **new** Row();

expect(TokenType.***OPEN\_ROW***);

expect(TokenType.***OPEN\_COLUMN***);

Cell cell = **new** Cell(token.value);

row.cells.add(cell);

expect(TokenType.***CELL***);

expect(TokenType.***CLOSE\_COLUMN***);

expect(TokenType.***CLOSE\_ROW***);

**return** row;

}

**private** **void** expect(TokenType tokenType) **throws** Throwable {

**if** (!token.isType(tokenType)) {

**throw** **new** SyntaxException("Expected: " + tokenType + " Obtained: "

+ token);

}

token = tokenizer.nextToken();

}

}

**public** **class** SyntaxException **extends** Exception {

**private** **static** **final** **long** ***serialVersionUID*** = -7866445943069603392L;

**public** SyntaxException(String s) {

**super**(s);

}

}

**Esercizio 3**

CodeGenerator è una classe astratta che implementa il generatore di test, una funzione che, presa una tabella come input, chiama su questa la execute passando come parametro il tipo di Fixture che quella tabella rappresenta.

Il generatore della classe skeleton, invece, è un’estenzione di CodeGenerator, specializzata per il tipo di tabelle che rappresentano una ColumnFixture.

**public** **abstract** **class** CodeGenerator {

**public** **abstract** String generateClass(Table table);

**public** String generateTest(Table table) {

StringBuffer stringBuffer = **new** StringBuffer();

stringBuffer.append("public static String test(Table table) {\n");

stringBuffer.append(String.*format*("\t%s %s = new %s();\n",

table.rows.get(0).cells.get(0).text,

table.rows.get(0).cells.get(0).text.toLowerCase(),

table.rows.get(0).cells.get(0).text));

stringBuffer.append(String.*format*("\t return table.execute(%s);\n",

table.rows.get(0).cells.get(0).text.toLowerCase()));

stringBuffer.append("}");

**return** stringBuffer.toString();

}

}

**public** **class** ColumnFixtureCodeGenerator **extends** CodeGenerator {

**public** String generateClass(Table table) {

StringBuffer stringBuffer = **new** StringBuffer();

stringBuffer.append(String.*format*(

"public class %s extends ColumnFixture {\n\t",

table.rows.get(0).cells.get(0).text));

**for** (**int** i = 0; i < table.rows.get(1).cells.size() - 1; i++) {

stringBuffer.append(String.*format*("public %s %s;\n\t",

table.rows.get(2).cells.get(i).text,

table.rows.get(1).cells.get(i).text));

}

stringBuffer

.append(String.*format*("public %s result;\n\t",

table.rows.get(2).cells.get(table.rows.get(1).cells

.size() - 1).text));

stringBuffer

.append(String.*format*("public %s %s {\n\t}\n\t",

table.rows.get(2).cells.get(table.rows.get(1).cells

.size() - 1).text, table.rows.get(1).cells

.get(table.rows.get(1).cells.size() - 1).text));

stringBuffer.append("public boolean check(Row row){\n\t\t");

**for** (**int** i = 0; i < table.rows.get(1).cells.size() - 1; i++) {

stringBuffer.append(String.*format*(

"%s = (%s) row.cells.get(%d).convert(%s);\n\t\t",

table.rows.get(1).cells.get(i).text,

table.rows.get(2).cells.get(i).text, i,

table.rows.get(1).cells.get(i).text));

}

stringBuffer

.append(String.*format*("%s result = %s;\n\t\t",

table.rows.get(2).cells.get(table.rows.get(1).cells

.size() - 1).text, table.rows.get(1).cells

.get(table.rows.get(1).cells.size() - 1).text));

stringBuffer

.append("boolean check = row.cells.get(row.cells.size() - 1).convert(result).equals(result);\n\t\t");

stringBuffer

.append("row.cells.get(row.cells.size() - 1).text= String.valueOf(result);\n\t\treturn check;");

stringBuffer.append("\n\t}\n}");

**return** stringBuffer.toString();

}

}

L’output dello skeleton relativo alla fixture Product è il seguente. Si fa notare che è necessario convertire i dati presenti in tabella con il tipo fornito, questo comporta che è necessario estendere il convertitore nel caso in cui si voglia creare una fixture con tipi non supportati. Per questa release è stato previsto il supporto per tutti i tipi primitivi di Java e per le stringhe.

Nella funzione check, oltre a verificare se la funzione da testare restituisce il valore corretto, viene anche aggiornata la cella di output della riga.

**public** **class** Product **extends** ColumnFixture {

**public** **float** x;

**public** **float** y;

**public** **float** result;

**public** **float** result() {

**return** x \* y;

}

**public** **boolean** check(Row row) {

x = (**float**) row.cells.get(0).convert(x);

y = (**float**) row.cells.get(1).convert(y);

**float** result = result();

**boolean** check = row.cells.get(row.cells.size() - 1).convert(result)

.equals(result);

row.cells.get(row.cells.size() - 1).text = String.*valueOf*(result);

**return** check;

}

}

Il codice per far partire il test è una funzione statica che il programmatore potrà invocare passando come parametro una tabella. La funzione ritorna l’output HTML opportunamente elaborato.

**public** **static** String test(Table table) {

Product product = **new** Product();

**return** table.execute(product);

}

Per la conversione da oggetto Table a stringa HTML si è usato il pattern Visitor Design, chiamando la funzione toHTML(…) su se stessi e ricorsivamente sugli elementi contenuti in ciascuno.

**public** **interface** IHTMLGenerator {

**public** String toHTML(Table table);

**public** String toHTML(Row row);

**public** String toHTML(Cell cell);

}

**public** **class** HTMLGenerator **implements** IHTMLGenerator {

**public** String toHTML(Table table) {

StringBuffer sb = **new** StringBuffer();

sb.append("<table>\n");

**for** (Row row : table.rows) {

sb.append(row.toHTML(**this**));

}

sb.append("</table>\n");

**return** sb.toString();

}

**public** String toHTML(Row row) {

StringBuffer sb = **new** StringBuffer();

sb.append("\t<tr>\n");

**for** (Cell cell : row.cells) {

sb.append(cell.toHTML(**this**));

}

sb.append("\t</tr>\n");

**return** sb.toString();

}

**public** String toHTML(Cell cell) {

StringBuffer sb = **new** StringBuffer();

**switch** (cell.colour) {

**case** ***GREEN***:

sb.append("\t\t<td bgcolor='green'>");

**break**;

**case** ***RED***:

sb.append("\t\t<td bgcolor='red'>");

**break**;

**default**:

sb.append("\t\t<td>");

**break**;

}

sb.append(cell.text);

sb.append("</td>\n");

**return** sb.toString();

}

}

Ulteriori test sono stati eseguiti, ad esempio, con una Fixture per l’esecuzione dell’operatore unario NOT fornendo come input tipi booleani, oppure dell’operazione di concatenzione fornendo stringhe.

**Esercizio 4**

Per estendere il framework per testare sequenze di invocazioni è stata introdotta la classe astratta ActionFixture, il cuo metodo execute(Table table) esegue il test e ritorna la tabella in formato HTML opportunamente colorata in base al risultato del metodo process().

**public** **abstract** **class** ActionFixture **implements** Fixture {

**public** **abstract** **boolean** process();

**public** String execute(Table table) {

**if**(process()){

table.rows.get(table.rows.size()-1).cells.get(1).colour=Colour.***GREEN***;

}**else**{

table.rows.get(table.rows.size()-1).cells.get(1).colour=Colour.***RED***;

}

**return** table.toHTML(**new** HTMLGenerator());

}

}

Il generatore ha quindi lo scopo di creare una classe il cui nome è fornito in tabella e che estende la classe ActionFixture.

Questa deve implemetare il metodo process(), eseguendo tutte le invocazioni presenti in tabella e ritornando il risultato della check().

Il generatore di codice presuppone che l’ordine delle esecuzioni sia corretto dal punto di vista della logica delle operazioni, non viene quindi eseguito alcun controllo.

**public** **class** ActionFixtureCodeGenerator **extends** CodeGenerator {

**public** String generateClass(Table table) {

StringBuffer sb = **new** StringBuffer();

String checkFunction = **new** String();

Stack<String> stack = **new** Stack<String>();

sb.append(String.*format*("public class %s extends ActionFixture {\n\t",

table.rows.get(0).cells.get(0).text));

sb.append("public boolean process() {\n");

**for** (Row row : table.rows) {

StringBuffer arguments = **new** StringBuffer();

**for** (**int** i = 3; i < row.cells.size(); i++) {

**if** (!row.cells.get(i).text.equals("")) {

arguments

.append(String.*format*("%s", row.cells.get(i).text));

**if** (i != row.cells.size() - 1) {

arguments.append(", ");

}

}

}

**switch** (row.cells.get(0).text) {

**case** "start":

stack.push(String.*format*("%s %s = new %s()",

row.cells.get(1).text, row.cells.get(2).text,

row.cells.get(1).text));

**break**;

**case** "call":

**if** (row.cells.get(1).text.equals("")) {

stack.push(String.*format*("%s(%s)", row.cells.get(2).text,

arguments.toString()));

} **else** {

stack.push(String.*format*("%s.%s(%s)",

row.cells.get(1).text, row.cells.get(2).text,

arguments.toString()));

}

**break**;

**case** "result":

**if** (!arguments.toString().equals("")) {

arguments.insert(0, String.*format*("%s, ", stack.pop()));

} **else** {

arguments.append(stack.pop());

}

**if** (row.cells.get(1).text.equals("")) {

stack.push(String.*format*("%s(%s)", row.cells.get(2).text,

arguments.toString()));

} **else** {

stack.push(String.*format*("%s.%s(%s)",

row.cells.get(1).text, row.cells.get(2).text,

arguments.toString()));

}

**break**;

**case** "check":

checkFunction = String

.*format*("private <T> boolean check(T t){\n\t\treturn t.equals(%s);\n\t}",

row.cells.get(1).text);

stack.push(String.*format*("return check(%s)", stack.pop()));

**break**;

**default**:

**break**;

}

}

List<String> reversed = **new** ArrayList<String>(stack);

**for** (String statement : reversed) {

sb.append("\t\t" + statement + ";\n");

}

sb.append("\t}\n\t");

sb.append(checkFunction);

sb.append("\n}");

**return** sb.toString();

}

}

Il codice seguente rapprenta il risultato della classe ottenuta dalla generazione per l’esempio fornito.

**public** **class** Action **extends** ActionFixture {

**public** **boolean** process() {

Accumulator acc = **new** Accumulator();

acc.add(product(12, 12));

**return** check(sqrt(acc.add(product(7, 7))));

}

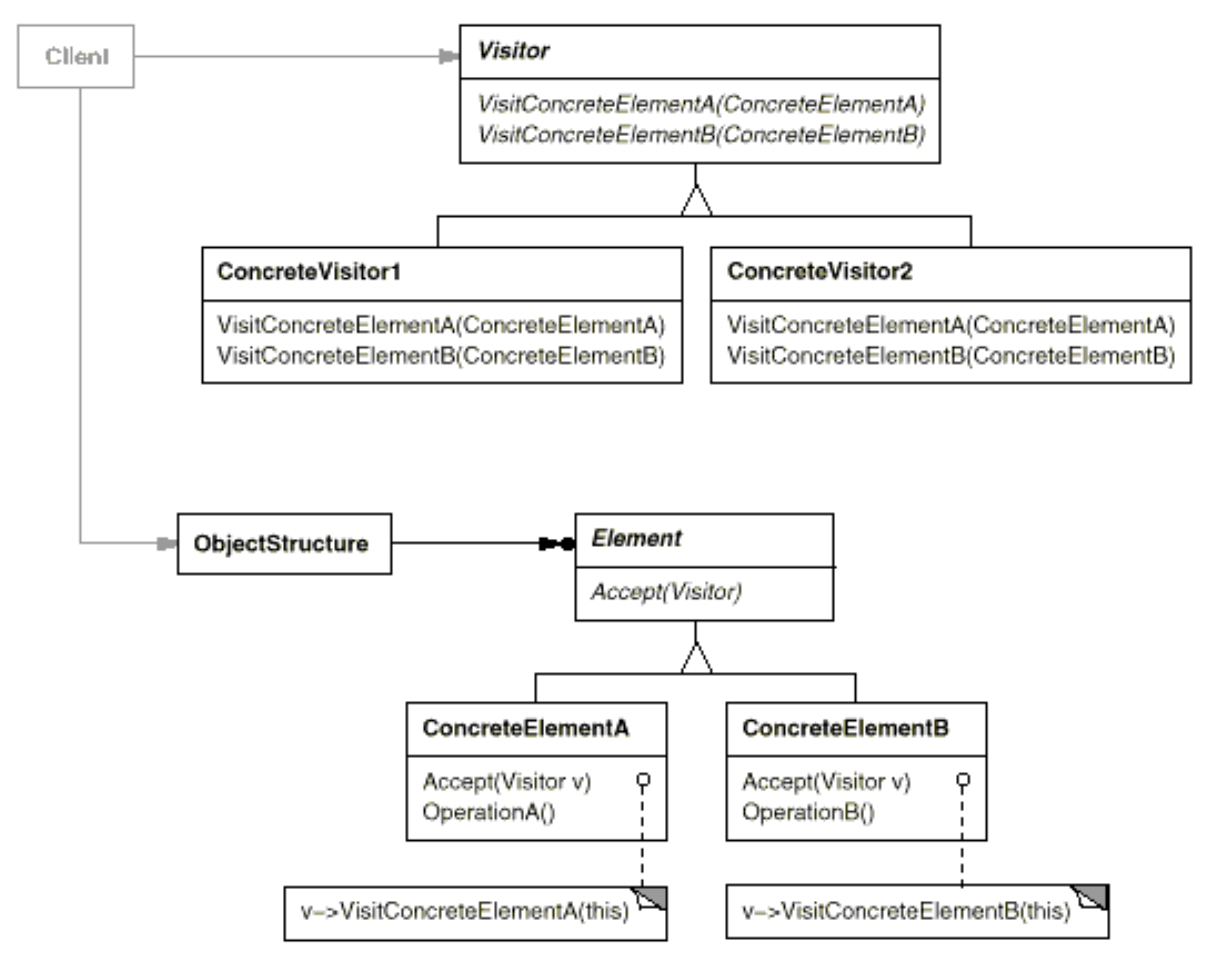
**private** <T> **boolean** check(T obj) {

**return** obj.equals(13.8924);

}

}

**Esercizio 5**

Il Visitor Design Pattern risolve la necessità di divedere gli oggetti dagli algoritmi ad essi applicati. Uno dei vantaggi principali di questa separazione è quello di poter consolidare il modello, il quale raramente cambia, e non doverlo modificare ogni qualvolta un nuovo metodo, ad esso applicabile, viene definito.

Volendo riportare quanto detto a questo progetto, possiamo notare che le tabelle in codice HTML sono rappresentate con la classe Table, costituita da Row (ovvero righe) esse stesse fatte di Cell. È facile capire che una tabella HTML sarà sempre rappresentabile in questo modo indipendentemente da come poi quella tabella sarà adoperata e da cosa effettivamente rappresenterà. Infatti che essa rappresenti una ColumnFixture, quindi una serie di test da eseguire su una operazione definita, oppure una Action, una serie di operazione di cui si vuole controllare il risultato finale, la sua struttura non varia, ma bensì cambia il modo di interpretarla.

Tutto questo conferisce, inoltre, maggiore eleganza nell’organizzazione del progetto, oltre che una vera e propria divisione dei ruoli nel caso di sviluppo in team. Molto spesso chi progetta il modello e gli algoritmi non sono la stessa persona. Oltretutto, il modello può essere condiviso da più progetti, la cui business logic si differenzia.

Questo pattern risulta utile, inoltre, nel caso in cui si voglia definere azioni differenti per una struttura costituita da oggetti di tipi diversi, come ad esempio è stato fatto per la generazione del codice HTML a partire da una Table, la cui funzione toHTML() produce un risultato consono al tipo di oggetto.

Il pattern si compone di classi Visitor ed Element, di solito astratte, implementate da ConcreteVisitor e ConcreteElement. Visitor contiene la dichiarazione di un metodo Visit per ogni singolo ConcreteVisitor, mentre Element definisce l’operazione Accept di un ConcreteVisitor. Il metodo Accept chiama il metodo Visit del ConcreteVisitor passando come argomento se stesso, quindi un ConcreteVisitor. Il metodo Visit a quel punto è l’incaricato dell’esecuzione degli algoritmi sull’elemento.

I linguaggi che permettono il multi-dispatch semplificano l’implementazione del Visitor Design Pattern, essendo la stessa Accept un’operazione di tipo multi-dispatch in quanto dipende dal tipo di Element e di Visitor. Il multi-dispatching permette ai Visitor di richiedere operazioni diverse su ogni Element. Invece di associare staticamente le operazioni agli Element, queste possono essere definite nel Visitor e quindi decise a run-time.

**References**

1. Erich Gamma, 1998. *Design Patterns CD: Elements of Reusable Object-Oriented Software (Professional Computing)*. 1 Edition. Addison-Wesley Professional.
2. Visitor pattern - Wikipedia, the free encyclopedia. 2015. *Visitor pattern - Wikipedia, the free encyclopedia*. [ONLINE] Available at:<http://en.wikipedia.org/wiki/Visitor_pattern>. [Accessed 30 January 2015].