23. Visitor

(GoF pag. 331)

23.1. Descrizione

Rappresenta una operazione da essere eseguita in una collezione di elementi di una struttura. L'operazione può essere modificata senza alterare le classi degli elementi dove opera.

23.2. Esempio

Si consideri una struttura che contiene un insieme eterogeneo di oggetti, su i quali bisogna applicare la stessa operazione, che però è implementata in modo diverso da ogni classe di oggetto. Questa operazione potrebbe semplicemente stampare qualche dato dell'oggetto, formattato in un modo particolare. Per esempio la collezione potrebbe essere un Vector che ha dentro di se oggetti String, Integer, Double o altri Vector. Si noti che se l'oggetto da stampare è un Vector, questo dovrà essere scandito per stampare gli oggetti trovati ai suoi interni. Si consideri anche che l'operazione ad applicare non è in principio implementata negli oggetti appartenenti alla collezione, e che questa operazione potrebbe essere ulteriormente ridefinita.

Un approccio possibile sarebbe creare un oggetto con un metodo adeguato per scandire collezioni o stampare i dati dell'oggetto:

```
public void printCollection(Collection collection) {
   Iterator iterator = collection.iterator()
   while (iterator.hasNext()) {
      Object o = iterator.next();
      if (o instanceof Collection)
         printCollection( (Collection) o );
      else if ( o instanceof String )
         System.out.println( "'"+o.toString()+"'" );
      else if (o instanceof Float)
         System.out.println( o.toString()+"f" );
      else
         System.out.println( o.toString() );
   }
}
```

Questo approccio va bene se si vuole lavorare con pochi tipi di dati, ma intanto questi aumentano il codice diventa una lunga collezione di `if…else.`

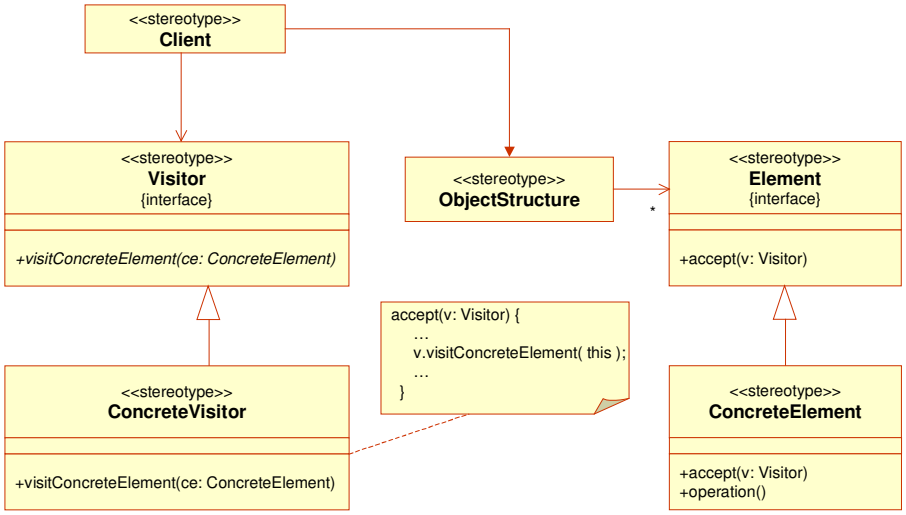
Il problema è trovare un modo di applicare questa operazione a tutti gli oggetti, senza includerla nel codice delle classi degli oggetti.

23.3. Descrizione della soluzione offerta dal pattern

La soluzione consiste nella creazione di un oggetto (ConcreteVisitor), che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Per agire in questo modo bisogna fare in modo che ogni oggetto della collezione aderisca ad un'interfaccia (Visitable), che consente al ConcreteVisitor di essere "accettato" da parte di ogni Element. Poi il Visitor, analizzando il tipo di

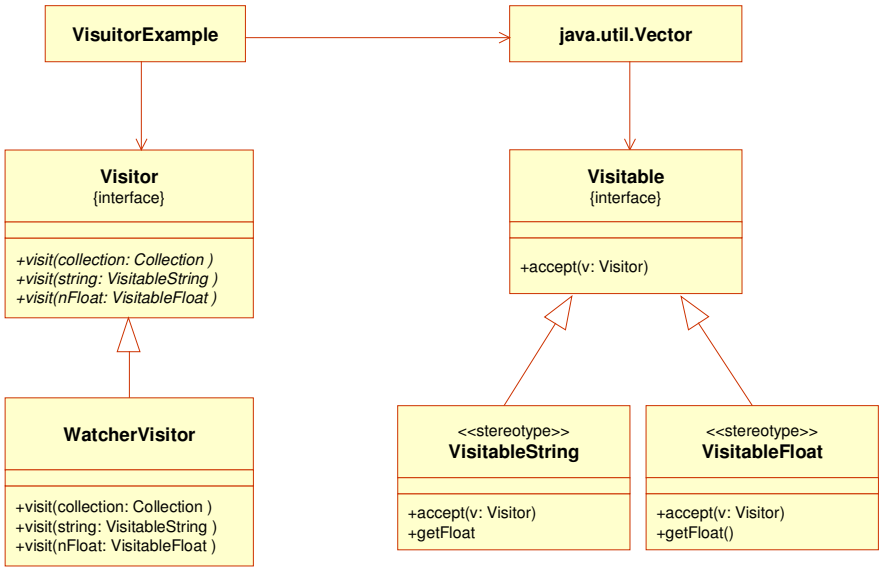
oggetto ricevuto, fa l'invocazione alla particolare operazione che in ogni caso si deve eseguire.

23.4. Struttura del Pattern



23.5. Applicazione del Pattern

Schema del modello



Partecipanti

- **Visitor**: interfaccia Visitor.
  - Specifica le operazioni di visita per ogni classe di **ConcreteElement**.
- **ConcreteVisitor**: interfaccia Visitor.
  - Specifica le operazioni di visita per ogni classe di **ConcreteElement**. La firma di ogni operazione identifica la classe che spedisce la richiesta di visita al **ConcreteVisitor**, e in questo modo il visitor determina la concreta classe da visitare. Finalmente il **ConcreteVisitor** accede agli elementi direttamente tramite la sua interfaccia.
- **Element**: interfaccia Visitable.
  - Dichiara l'operazione *accept* che riceve un riferimento a un **Visitor** come argomento.
- **ConcreteElement**: classi VisitableString e VisitableFloat.
  - Implementa l'interfaccia **Element**.
- **ObjectStructure**: classe Vector.
  - Offre la possibilità di accettare la visita dei suoi componenti.

Descrizione del codice

Per l'implementazione si definisce l'interfaccia `Visitable`, che dovrà essere implementata da ogni oggetto che accetti la visita di un Visitor:

```
public interface Visitable {

   public void accept( Visitor visitor ) ;

}
```

Due concreti oggetti visitabili sono `VisitableString` e `VisitableFloat`:

```
public class VisitableString implements Visitable {

   private String value;

   public VisitableString(String string) {
      value = string;
   }

   public String getString() {
      return value;
   }

   public void accept( Visitor visitor ) {
        visitor.visit( this );
   }

}

public class VisitableFloat implements Visitable {

   private Float value;

   public VisitableFloat(float f) {
      value = new Float( f );
```

```
    }

    public Float getFloat() {
       return value;
    }

    public void accept( Visitor visitor ) {
         visitor.visit( this );
    }

}
```

Si noti che in entrambi casi, oltre ai metodi particolari di ogni classe, c'è il metodo `accept`, dichiarato nell'interfaccia `Visitable`. Questo metodo soltanto riceve un riferimento ad un **Visitor**, e chiama la sua operazione di visita inviando se stesso come riferimento.

I **ConcreteVisitor** che sono in grado di scandire la collezione e i suoi oggetti, implementano l'interfaccia `Visitor` :

```
import java.util.Collection;
public interface Visitor {

   public void visit( Collection collection );
   public void visit( VisitableString string );
   public void visit( VisitableFloat nFloat );

}
```

Si presenta di seguito il codice della classe `WatcherVisitor`, che corrisponde ad un **ConcreteVisitor**:

```
import java.util.Collection;
import java.util.Iterator;

public class WatcherVisitor implements Visitor {

   public void visit(Collection collection) {
      Iterator iterator = collection.iterator();
      while (iterator.hasNext()) {
         Object o = iterator.next();
         if (o instanceof Visitable)
            ((Visitable)o).accept(this);
         else if (o instanceof Collection )
            visit( (Collection) o );
      }
   }

   public void visit(VisitableString vString) {
      System.out.println( "'"+vString.getString()+"'" );
   }

   public void visit(VisitableFloat vFloat) {
      System.out.println( vFloat.getFloat().toString()+"f" );
   }

}
```

Si noti come il `WatcherVisitor` isola le operazioni riguardanti ogni tipo di Element in un metodo particolare, essendo un approccio più chiaro rispetto di quello basato su `if…else` nidificati.

Ecco il codice di un'applicazione che gestisce una collezione eterogenea d'oggetti, e applica, tramite il `WatcherVisitor`, un'operazione di stampa su ogni elemento della collezione. Deve osservarsi che l'ultimo elemento aggiunto alla collezione corrisponde a un `Double`, che non implementa l'interfaccia `Visitable`:

```
import java.util.Vector;

public class VisitorExample {

   public static void main (String[] arg) {

   // Prepare a heterogeneous collection
   Vector untidyObjectCase = new Vector();
   untidyObjectCase.add( new VisitableString( "A string" ) );
   untidyObjectCase.add( new VisitableFloat( 1 ) );
   Vector aVector = new Vector();
   aVector.add( new VisitableString( "Another string" ) );
   aVector.add( new VisitableFloat( 2 )  );
   untidyObjectCase.add( aVector );
   untidyObjectCase.add( new VisitableFloat( 3 ) );
   untidyObjectCase.add( new Double( 4 ) );

   // Visit the collection
   Visitor browser = new WatcherVisitor();
   browser.visit( untidyObjectCase );

   }

}
```

Osservazioni sull'esempio

Si noti che dovuto a che tutti i **ConcreteElement** da visitare implementano il metodo `accept` allo stesso modo, un'implementazione alternativa potrebbe dichiarare `Visitable` come una classe astratta che implementa il metodo `accept`. In questo caso i **ConcreteElement** devono estendere questa classe astratta, ereditando l'`accept` implementato. Il problema di questo approccio riguarda il fatto che gli Element non potranno ereditare da nessuna altra classe.

Esecuzione dell'esempio

Si noti che l'ultimo elemento della lista (un `Double` con valore uguale a `4`), non implementa l'interfaccia Visitable, ed è trascurato nella scansione della collezione.

```
C:\Design Patterns\Behavioral\Visitor>java VisitorExample

'A string'
1.0f
'Another string'
2.0f
3.0f
```

23.6. Osservazioni sull'implementazione in Java

Blosser [3] propone una versione del Visitor in Java, diversa dalla specifica indicata dai GoF, che non richiede fornire nomi diversi per ogni metodo di visita, associato ad una particolare tipologia di oggetto da visitare (esempio, `visitString` o `visitFloat`). Tutti possono avere lo steso nome, dato che il metodo particolare da applicare viene identificato discriminato dal tipo di parametro specificato (esempio, `visit(VisitableString s)` o `visit(VisitableFloat f)`). Questa caratteristica, insieme alle funzionalità offerte da Java, viene sfruttata nella proposta alternativa del Visitor pattern, che si descrive di seguito.

Implementazione del visitor pattern basata sulla Reflection API

I problemi principali del Visitor Pattern, descritto nell'implementazione precedente sono due:

- Se si vuole aggiungere un nuovo tipo di **ConcreteElement** da visitare, per forza bisogna modificare l'interfaccia Visitor, e quindi, ricompilare tutte le classi coinvolte[13].
- I **ConcreteElement** devono implementare una particolare interfaccia per essere visitati (sebbene questo non è davvero un grosso problema).

Tramite la Reflection API di Java si ottiene un'implementazione più semplice e flessibile, che da soluzione a questi due inconvenienti. Per esemplificare ciò si presenta lo stesso caso dell'esempio anteriore.

Per primo, in questo approccio non è necessario di costringere i **ConcreteElement** da visitare debbano implementare una particolare interfaccia, rendendo innecesaria l'interfaccia `Visitable`. Per questa ragione la collezione da visitare in questo esempio, farà uso soltanto degli oggetti `String`, `Float`, `Double`, e altri forniti da Java.

L'interfaccia che i `ConcreteVisitor` basati nella Reflection devono implementare ha un unico metodo con la firma `visit(Object o)`:

```
import java.util.Collection;

public interface ReflectiveVisitor {

   public void visit( Object o );

}
```

Il **ConcreteVisitor** `ReflectiveWatcherVisitor`, che ha la stessa funzionalità del `WatcherVisitor` dell'esempio precedente, implementa l'interfaccia `ReflectiveVisitor`:

```
import java.util.Vector;
import java.util.Collection;
import java.util.Iterator;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;

public class ReflectiveWatcherVisitor implements ReflectiveVisitor {


   public void visit(Collection collection) {
      Iterator iterator = collection.iterator();
      while (iterator.hasNext()) {
         Object o = iterator.next();
         visit( o );
      }
   }

   public void visit(String vString) {
      System.out.println( "'" + vString + "'" );
   }

   public void visit(Float vFloat) {
```

[13] Si noti che le interfacce in Java "non possono crescere".

```
      System.out.println( vFloat.toString() + "f" );
   }

   public void defaultVisit (Object o) {
      if (o instanceof Collection )
         visit( (Collection) o );
      else
         System.out.println(o.toString());
   }


   public void visit(Object o) {


      try {
         System.out.println( o.getClass().getName() );
         Method m = getClass().getMethod( "visit",
                                          new Class[] { o.getClass() });

         m.invoke(this, new Object[] { o });

      } catch (NoSuchMethodException e) {
         // Do the default implementation
         defaultVisit(o);
      } catch (IllegalAccessException e) {
         // Do the default implementation
         defaultVisit(o);
      } catch (InvocationTargetException e) {
         // Do the default implementation
         defaultVisit(o);
      }
   }
}
```

Come già è stato detto, il metodo `visit(Object o)` riceve l'oggetto a visitare, e invoca il metodo `visit` corrispondente al particolare tipo di oggetto. Per fare ciò si prepara una istanza dell'oggetto `Method` che dovrà essere eseguito. La prima istruzione chiave è:

```
Method m = getClass().getMethod( "visit", new Class[] {o.getClass()});
```

In questa istruzione viene creato un oggetto `Method` che deve corrispondere ad un metodo della classe del **Visitor**. L'istruzione `getClass()` restituisce un riferimento ad un oggetto della classe `Class` corrispondente al `ReflectiveWatcherVisitor`. Su di questo oggetto viene invocato il metodo `getMethod` che restituisce un oggetto `Method`, il quale fa riferimento a un metodo appartenente alla classe `ReflectiveWatcherVisitor`. Per costruire l'oggetto `Method` bisogna specificare:

- il nome particolare del metodo, che in tutti i casi dell'esempio sarà "`visit`"; e
- un `array` di oggetti della classe `Class` contenenti oggetti della classe Class alla quale appartengono i parametri. In questo caso i metodi `visit` hanno un unico parametro, così che l'array avrà un unico oggetto, corrispondente al tipo del parametro. In questo caso l'array è creato tramite l'istruzione `new Class[] {o.getClass()}` (si noti che la variabile `o` ha un riferimento all'oggetto a trattate.

La seconda istruzione è soltanto l'invocazione al metodo:

```
m.invoke(this, new Object[] { o });
```

Questa istruzione riceve due parametri, un riferimento all'oggetto attuale, e un array con gli oggetti che corrispondono ai parametri.

Può capitare che il `ReflectiveWatcherVisitor` non abbia definito un metodo `visit` per trattare un particolare tipo di oggetto, come è il caso, in questo esempio, degli oggetti `Double`. In questo caso viene lanciata una `NoSuchMethodException` (perche il metodo `visit` che riceve il particolare tipo di parametro non è fornito dal `ReflectiveWatcherVisitor`), e l'eccezione viene intercettata, per eseguire una `defaultVisit`.

Si noti che le collezioni sono gestite prima dal `defaultVisit`, perché l'istruzione che costruisce l'oggetto `Method` anche genera una `NoSuchMethodException` quando tenta di costruire un oggetto con un tipo particolare di collezione. Questo significa che non riesce a fare il match tra, per esempio, la visita di un `Vector` (o un'altra particolare collezione, che implementi l'interfaccia `Collection`) e il metodo `visit(Collection collection)`. Questo match, invece, si riesce nell'invocazione diretta del metodo `defaultVisit`, con il previo casting del `Object` a `Collection`:

```
visit( (Collection) o );
```

Ecco il codice dell'esempio:

```
import java.util.Vector;

public class ReflectiveVisitorExample {

   public static void main (String[] arg) {

   // Prepare a heterogeneus collection
   Vector untidyObjectCase = new Vector();
   untidyObjectCase.add( "A string" );
   untidyObjectCase.add( new Float( 1 ) );
   Vector aVector = new Vector();
   aVector.add( "Another string" );
   aVector.add( new Float( 2 )  );
   untidyObjectCase.add( aVector );
   untidyObjectCase.add( new Float( 3 ) );
   untidyObjectCase.add( new Double( 4 ) );

   // Visit the collection
   ReflectiveVisitor browser = new ReflectiveWatcherVisitor();
   browser.visit( untidyObjectCase );

   }

}
```

Esecuzione dell'esempio

```
C:\Design Patterns\Behavioral\Visitor>java ReflectiveVisitorExample

'A string'
1.0f
'Another string'
2.0f
3.0f
4.0
```