Oron Peled and myself as part of the requirements in a course on Advance Topics in Java. IBM Haifa Research Lab provided the Java software.

References

[1] Apple Computer, Inc., Cupertino, CA. *Dylan: An object-oriented dynamic language*, 1992.

[2] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the 1st European Conference on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 151–170, Paris, France, June 15-17 1987. ECOOP'87, Springer Verlag.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[4] J. Y. Gil and D. H. Lorenz. Design patters and language design. *IEEE Computer*, 31(3):118–120, Mar. 1998. Object Technology.

[5] J. Hernandez, M. Papathomas, H. M. Murillo, and F. Sanchez. Coordinating concurrent objects: How to deal with the coordination aspect?, 1997.

[6] D. Holmes, J. Noble, and J. Potter. Aspects of synchronization, 1997.

[7] C. Houser. Manual and compiler for the terse and modular lagnuage DEM. *ACM SIGPLAN Notices*, 31(12):41–51, Dec. 1996.

[8] M. E. N. III. Default and extrinsic visitor. In Martin et al. [13], pages 105–124.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[10] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS-Kent Publishing, 1996.

[11] K. J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming using graph-based custom ization. *Commun. ACM*, 37(5):94–101, May 1994.

[12] D. H. Lorenz. Tiling design patterns - a case study using the interpreter pattern. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 206–217, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, Acm SIGPLAN Notices 32(10) Oct. 1997.

[13] R. Martin, D. Riehle, and F. Buschmann, editors. *Pattern Languages of Program Design 3*, Software Patterns. Addison-Wesley, 1998.

[14] R. C. Martin. Acyclic visitor. In Martin et al. [13], pages 93–104.

[15] K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-oriented programming workshop report, 1997.

[16] J. Vlissides. Pattern hatching: Visiting rights. *C++ Report*, 7(7), Sept. 1995.

ample, for the class java.lang.reflect.Method, three class names will flash: java.lang.Object—its superclass, java.lang.Member—its interface, and int—the type of its field. The figure shows the screen dump at the end of the execution, and therefore int is showing in the output box.

This example comprises of only *traversal* beans. Examples using *behavioral* beans (e.g., introducing counters, resets, increments, and counter reporters along the path to compute the number of classes instead of just flashing them) are skipped.

5 Conclusions

This paper submits that it is useful to develop aspect-oriented patterns for describing an aspect-oriented solution to a tangling problem in a particular context. We presented the *Visitor Beans* pattern, as an example of an aspect-oriented pattern, which solves the problem of tangling between reflection, navigation, and behavior.

The *Visitor Beans* pattern is a substantive variant of the VISITOR pattern. It implements the pattern in Java without confining to the working of the traditional implementation. *Visitor Beans* uses Java's new event model as an alternative to writing visit() methods, and Java Reflection as an alternative to writing accept() methods. Instead of sending a visitor to visit an element, the element is fired to the visitor as an event. Weaving is done via visual builder tools that support the JavaBeans technology.

5.1 VISITOR pattern lessons

Visitor Beans present a better way to compose visitors. Each visitor is packed as a Java bean. A visitor may register as an event listener with other visitors and send events to other visitors. Visual builder tools (like Java Studio™ and VisualAge™ for Java Professional) provide visual means to compose the visitors.

With *Visitor Beans*, we can extend class hierarchies that never anticipated extension. This capability was illustrated by extending java.lang.reflect itself, although java.lang.Class is a system class definition with no accept hooks, and its instances are pre-existing classes.

5.2 AOP lessons

The VISITOR BEANS pattern describes an aspect-oriented solution to a tangling problem in a particular context. Identifying the components, aspects, join-points, and aspect weaver in the *Visitor Beans* pattern, nonetheless, was not all that straightforward. Some observations are listed here.

- *Weaving-time weaving.* Beans are first compiled, then combined into an application via a builder. The weaving is done during the run-time of the builder, after compilation and before running. Hence this is neither compile-time weaving nor run-time weaving. Rather, this seams to be a new time frame for weaving, which may be called *weaving-time weaving*.
- *Reflection and AOP.* Reflection was used here intensively. It was the subject for extension and an aspect of the extension. It plays part in introspection during weaving-time weaving; and it is the means in which run-time weaving is achieved.
- *Visual representation of AOP.* To a limited degree, the graphical builder tool provides a visual representation of the component, aspects, and of the weaving. The lines connecting beans can be viewed as webs, connecting the beans' connectors, which it turn can be viewed as primitive aspects.
- *AOP and existing technology.* Patterns are essentially about achieving the desired from what exists. The *Visitor Beans* pattern demonstrated how the JavaBeans technology may be applied in AOP.
- *Decoupling of aspects.* Join points, like the accept() and visit() methods, create undesired coupling between aspects. Reflection decoupled those.
- *Strongly typed weaving.* Connectors of beans are an example of *typed* join points. The aspect weaver processes the aspect language in terms of the join points of components. A strongly typed aspect language can thus prevent senseless weaving.

6 Acknowledgments

I thank Karl Lieberherr for objecting to the statement about the VISITOR in [4, page 119]. My thanks also to Daniel Berry for his encouragement and for helpful comments on the manuscript. The example shown in Figure 4 was implemented by

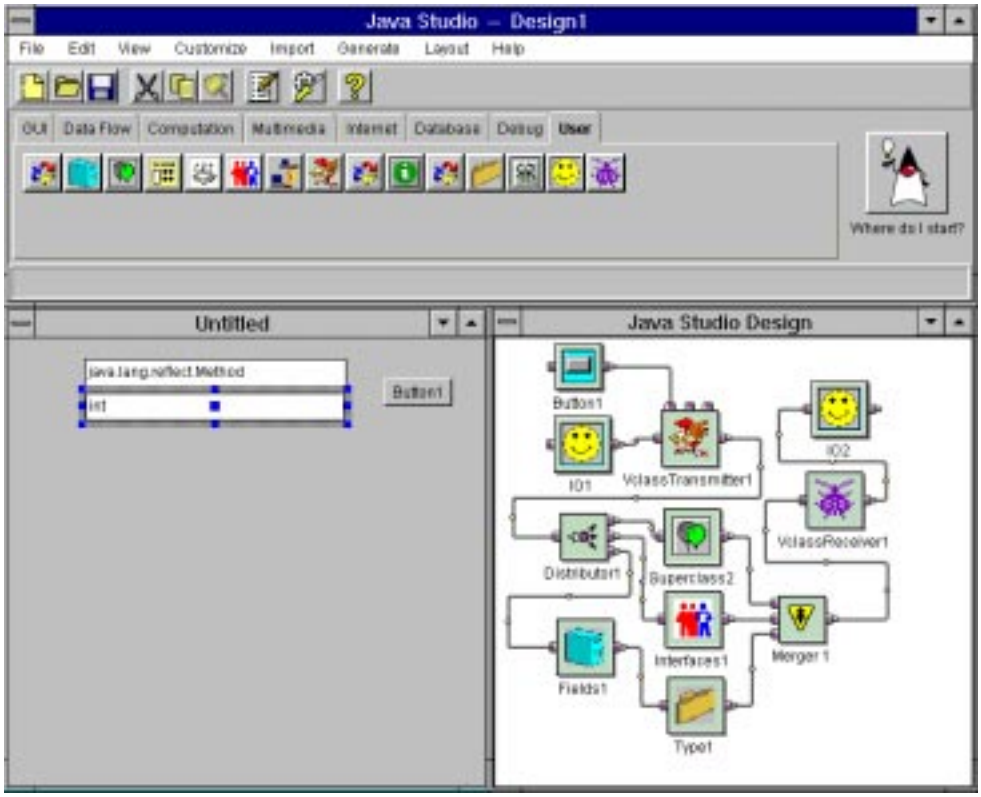

Figure 4: Weaving *Visitor Beans*

4.1 Weaving-time weaving

Figure 4 is an example of how *Visitor Beans* may be combined as reusable software components using any standard visual builder tool that supports JavaBeans technology. The figure is a screen dump of Java Studio™ builder. The top window is showing the user panel with 15 visitor beans that are currently available. The untitled window on the lower-left is the visual image of the application being built in the lower-right window entitled 'Java Studio Design'. The two field text lines are the visual appearance of primitive IO beans used for I/O. The upper one named IO1 is used for input, and the lower one named IO2 for output.

When text is entered to the input IO1 beans, TextEvents are fired to VclassTransmitter1. VclassTransmitter1 reacts by customizing itself to generate a ClassEvent with a class whose name is the string received from IO1. (VclassTransmitter1 can also be customized directly via a bean customizer.) Once it has been customized, the event is ready to be fired.

Button1 is connected to the method fire() of VclassTransmitter1 so that when it is pressed the VclassTransmitter1 sends out the awaiting ClassEvent. If the string obtained from IO1 is not a legal class name, no event is fired. For debugging and demonstration purposes, VclassTransmitter writes to System.out whether the string is legal or not. The Button, the Distributer and the Merger are GUI beans that come with Java Studio™.

Once Button1 is pressed and VclassTransmitter1 fires the ClassEvent, it is distributed to Superclass2, Interfaces1, and Fields1. As a result, Superclass2 triggers a ClassEvent with the superclass of the class. Interfaces1 triggers multiple ClassEvents, one for each interface of the class. Fields1 triggers multiple FieldEvents, one for each public field of the class. The FieldEvents are received by Type1 which fires a ClassEvent corresponding to the type of the filed. All the ClassEvents are then collected by Merger1, and sent, one by one, to VclassReceiver1. Finally, VclassReceiver1 generates a TextEvent, and IO2 displays the class names, one by one. In order not to miss any class name, IO2 pauses for about half a second after each name change.

In short, this example generates for a given class all the classes it immediately depend on. For ex-

```
package visitor.beans.event;

import java.util.EventObject;

public class VisitEvent extends EventObject {

  protected VisitEvent(Object source,int type,Object visited,Object[] data) {
    super(source);
    fieldType = type;
    fieldVisited = visited;
    fieldData = data;
  }

  private transient int fieldType = 0;
  private transient Object fieldVisited = null;
  private transient Object[] fieldData = null;

}
```

Figure 3: VisitEvent class

classed visitor. There is no filtering of visitations. The visit() methods is always called on a visitor regardless of whether the visitor actually handles those elements or not, thus the need for a DEFAULT VISITOR [8] or a NULL VISITOR. [12]

3.3 Visit events

Instead of using C++ templates (which do not yet exist in Java), we shall pack each visitor as a Java bean. Packing visitors as JavaBeans changes the ways visitors are used. Visitors may now report their visiting actions via *events* which are the natural inter-bean communication mechanism in Java. The direct implications of this approach are:

- Sending an event passes an *event object*. This object may carry information (like a reference to the visited object which the receiver of the event may access.)
- A visitor may register as an *event listener* with (possibly many) other visitors, and send events to (possibly many) other visitors. An arbitrary graph of event passing may be constructed, representing some complex visitor composition, without extra cost.
- Events are typed by the event objects sent. Since the event types form an inheritance hierarchy, visitors may register to each other in a typesafe (although not polymorphic) manner.

In order to supply the visitor with sufficient data, VisitEvent extends java.util.EventObject as given by the code of Figure 3. It consists of four parts. The first is the source of the event, like in any EventObject. Like java.awt.event.AWTEvent, the second part is the type of the event. We use the event type to pass control instruction, such as *reset*, *normal*, and *terminate*. The third part contains a reference to the *visitee*. It allows visitors to access the visited object. The fourth part contains an array of additional *data*.

Traversal visitors manipulate the visitee part and pass the data as is, while *behavioral visitors* pass the visitee untouched and manipulate only the data. All visitors mark themselves as the source of the event, for upwards compatibility with other beans.

4 Implemented Example

We do not need, in the reflection case, to create the class hierarchy since these classes already exist in java.lang and java.lang.reflect. Instead we create for each class $\tau \in \{\mathsf{Class}, \mathsf{Field}, \mathsf{Method}, \mathsf{Constructor}\}$, an event class named τEvent. For each of these events we need of course to define a corresponding Listener interface. A visitor that is interested in visiting only particular types of elements would register to receive notifications from only those types of events, and should declare itself as implementing the required interface.

```
public void visit_dispatch(Object o) {
  Class[] formal = new Class[1];
  formal[0] = o.getClass();
  try {
      Method m = getClass().getMethod("visit", formal);
      Object[] actual = new Object[1];
      actual[0] = o;
      m.invoke(this, actual);
  } catch (Exception e) {
      e.printStackTrace(System.err);
  }
}
```

Figure 2: Dispatching visit calls using reflection

java.lang.Class, which serves as the type of all those instances.

Adding to this that *reflection* is also a useful class to extend, makes this example possess all the right qualities. The classes are predefined; elements are pre-instantiated; and still, we wish to be able to visit elements (classes) and perform a new operation over them. We shall do so using run-time weaving by applying Java's reflection to itself.

3.1 Cross-cutting aspects

Consider a particular extension to Java's reflection: a class's dependency method. There are three orthogonal aspects to computing dependency of classes (instances) conforming to the grammar in Figure 1:

- *Reflection:* Ability to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts.
- *Navigation:* Traversing through the reflected information.
- *Behavior:* The actual computation carried out (along the path).

These are *aspects* that cut across system functionality. They are not simply functional decomposition because they affect each other's semantics. Traversal is defined in terms of the reflected structure and may depend on partial computation results. Similarly, the computation may depend both on the reflected information and on the traversal path. The *Visitor Beans* pattern tells you to implement these aspects as beans, and handle the dependencies during weaving.

3.2 Decoupling aspects

One adaptation to Java is to discard the original accept() methods and replace them with a general dispatching mechanism using Java reflection. Applying reflection, we need only a single accept() method, global to all visitors, that takes two arguments, the *element instance* and the *visitor instance*, as illustrated by the code in Figure 2. (In the figure, the concrete visitor is the hidden this parameter.) It then introspects the *visitor class* to locate a method named 'visit' which takes a single argument of type *element class*, and finally invokes the method found on the *element instance*.

The next adaptation is to find a better way for visitor composition and for combining visitors. The model for visitor combinations proposed in [12] is based on inheritance and templates. In order for a visitor to extend and process another visitor's visit() method, it must subclass that visitor and override either the visit() for that element or one of its related classes. Returning without calling the inherited accept() method consumes the visit. Otherwise the visit is propagated up the visitor hierarchy until the traversal completes.

While the above works fine for small tilings of simple visitors, it does not scale well for larger visitor combinations for the following reasons. The requirement to subclass a visitor in order to make any real use of its functionality is cumbersome. The inheritance-genericity model does not lend itself well to maintaining a clean separation between the traversal and behavioral because traversal code must be integrated statically into the sub-

3 A Case Study: Extending Reflection

Figure 1 describes a simplified grammar for Java's reflection. Following the guide lines of the INTERPRETER, it prescribes a definition of a class hierarchy. Left-hand-side variables are classes. Right-hand-side fields are instance variables. In the figure, fields have the general form '*label*:type', meaning that the class will have an instance variable named '*label*'.

The type of the instance variable can be a primitive type like in '*name*:String', or one of the left-hand-side variables, i.e., a reference to another class like in '*type*:Class'. A type surrounded in square brackets indicates that it is optional, as in '*superclass*:[Class]' which means that the *superclass* of an instance of Class may be *null*. Suffixing a type with a '[]' indicates that the instance variable is a container of values. For example, for '*interfaces*:Class[]', *interfaces* is the name of an instance variable of type 'Array of Classes'. If the label is missing, we assume the instance variable's name to be the plural of the component type. So 'Method[]' means that there is an instance variable named *methods* of type 'Array of Method'. The same goes for Constructor and Field.

The grammar displayed in Figure 1 is *not* an ordinary BNF specification, for the following reasons:

- It is a *simplified* grammar for reflection. Java's reflection includes information that is not revealed in the grammar in Figure 1, like Modifiers of Methods and Fields. These were excluded because their merit does not justify their weight in this exposition.

- It is an *abstract* grammar. It is worthless for parsing. Information about the concrete syntax of class, fields, or methods is missing. But this is all right because we do not intent to parse class declarations. On the contrary, much of the challenge lies in visiting pre-existing objects, that is, in this case, classes.

- It is a *semantic* description of class declarations. An instance of the hierarchy expressed by Figure 1 is a semantic net rather than a parse tree. For example, the *superclass* field of Class is not an instance variable of type *String* with the name of the superclass, but rather an instance of type Class which references the superclass itself (if one exists). As another example, Method knows its *declaringClass*, information that requires semantic analysis on top of parsing.

```
Class       ::= name:String
                superclass:[Class]
                interfaces:Class[]
                Field[]
                Method[]
                Constructor[];

Member      ::= Field | Method | Constructor;

Field       ::= declaringClass:Class
                name:String
                type:Class;

Method      ::= declaringClass:Class
                name:String
                returnType:Class
                parameterTypes:Class[];

Constructor ::= declaringClass:Class
                name:String
                parameterTypes:Class[];
```

Figure 1: A simplified grammar for reflection

Normally, the next stage involves the creation of the prescribed class hierarchy. Sometimes, however, we do not need to, do not wish to, or simply cannot produce the class hierarchy. When extending Java's *reflection* all three apply:

1. The class hierarchy already exists, and the designer did not leave accept() methods, nor accommodated our desire to extend this class. This example will show how we can extend an existing class hierarchy, something that the traditional VISITOR cannot.

2. Moreover, it is a system class whose classes are spread in more that one package. Even if we wanted to we cannot replace those classes with classes of our own, let alone change their code.

3. Instances of these classes already exist, and in abundance. Every class in the system is in fact an instance of this hierarchy. So, we cannot permit ourselves to change the class

aspect-oriented characteristics of the VISITOR are assimilated, making it difficult to tell them apart. Implemented in Java, however, the two aspects of the VISITOR, namely OOP and AOP, are more easily discerned. This is outlined in Section 3 by introducing a new variant of the VISITOR, the *Visitor Beans* pattern, and a case study of using it. Section 4 describes an implemented example. Finally, Section 5 concludes the AOP lessons learned in the process.

2 VISITOR as an Aspect-Oriented Pattern

The VISITOR pattern lets you add behavior to a class hierarchy without extending it. It localizes structure into a set of accept() methods, and behavior into a set of *visitor* objects. The details of implementation vary (e.g., [14, 8, 16]), of course. In the delicate balance between the purpose and the internal-working of the pattern, this section highlights the aspect-oriented *intent* of the VISITOR.

Although *structure* and *behavior* are not the best examples of aspects that cross-cut system functionality, the VISITOR does have basic AOP characteristics: without it the structure and behavior decisions are scattered throughout the tangled code instead of being dealt with separately. The VISITOR thus provides "a solution to an aspect-oriented problem in a context." We shall look at the VISITOR from this perspective for what it is, a case for an aspect-oriented pattern, and put aside the controversy on whether or not the VISITOR is "really" aspect-oriented in its full sense (whatever that may be.)

When you use the VISITOR for traversing an INTERPRETER [3] pattern, using accept() methods and performing the visiting tasks with visitor objects, you have the advantage that object structure is spelled out in the accept() methods which are reused in performing varies tasks. When the class structure changes, you need to update only once the accept() methods instead of changing the code for all the different tasks. Conversely, when new tasks are required, you need to implement only new visitors, hooking-up to the already existing accept() methods, without changing the class structure.

One can argue (or rather, be mis-understood [4]) that there is very little need for the VISITOR pattern if you use a multi-method object-oriented language such as CLOS [2] or Dylan. [1] While this is true for the object-oriented internal workings of the VISITOR (e.g., the single-dispatch "ping-pong" implementation in C++), it is not so for its aspect-oriented purpose.

In a way, such an argument would have said, for example, that *Adaptive Programming* [10], an aspect-oriented technique explicitly applying the VISITOR pattern, is not useful for CLOS, although the CLOS community developed a useful version of DEM [7] (a tiny version of Demeter [11]) in CLOS. The localization of structural information is thus also quite helpful for languages with multi-methods. It is the aspect-oriented ingredient of the VISITOR which prevails.

Implementing the VISITOR pattern in Java, which does not support multi-methods but provides other programming capabilities, introduces a dilemma. You can technically realize the C++-specific implementation of the VISITOR also in Java. Most applications of the VISITOR in Java do. However, you can better serve the aspect-oriented need of extending the behavior of a class hierarchy by exploiting other advanced features of Java. This results in a different implementation, an aspect-oriented variant of the VISITOR, which is named *Visitor Beans*.

2.1 *Visitor Beans* in a nutshell

The VISITOR patterns can be implemented in Java almost exactly as it is done in C++: defining in each element class an accept() method, and in each visitor class multiple visit() methods (one per element class.) However, new Java core APIs permit an implementation that is tailored for Java. Java Core Reflection Service allows an alternative to writing accept() methods; a traversal visitor applies reflection and handles all dispatches. Java 1.1 new event model allows an alternative to writing visit() methods: instead of sending a visitor to visit an element, the element is fired to the visitor as an event.

Visitors can chose which events they wish to listen to. Visitors also communicate by sending events. Visitors may then be wrapped as Java beans and combined in different ways using standard builder tools, allowing this way to combine primitive visitors into complex ones and keeping a clean separation between reflective, traversal and behavioral visitors.

Visitor Beans: An Aspect-Oriented Pattern

DAVID H. LORENZ

The Faculty of Computer Science,
Technion—Israel Institute of Technology,
Technion City, Haifa 32000, ISRAEL;
Email: david@cs.technion.ac.il

Abstract

It's only natural to assume that aspect-oriented patterns would one day play the role design patterns play today in the object-oriented technology. This paper strives to declare aspect-oriented the already known object-oriented VISITOR design pattern. The VISITOR describes an aspect-oriented solution to a tangling problem in a particular context. We preset a substantive variant of the VISITOR, a *Visitor Beans* pattern, which implements the VISITOR in Java without confining to the traditional VISITOR pattern operation. *Visitor Beans* weaving is done via visual builder tools that support the JavaBeans technology. With *Visitor Beans* it is possible to extend class hierarchies that never anticipated extension. To illustrate this, we extend java.lang.reflect. Lessons learned in aspect-oriented programming are reported.

1 Introduction

Aspect-oriented programming [9] (AOP) in its current state has been compared by its mentors Kiczales and colleagues to that of object-oriented programming (OOP) some twenty years ago. Like OOP then, the basic concepts are only beginning to emerge, based on existing research and experience. Yet they already show increasing promise and interest (e.g., the *Forum on New Research Directions* session, OOPSLA '97.)

Assuming AOP will indeed evolve similarly to OOP, it might be a good idea to examine from a twenty years perspective the milestones in OOP development. Extrapolating their corresponding turning points in AOP, might help in avoiding obstacles on one hand, and making right decisions on the other hand.

One evident breakthrough in OOP is the emergence of object-oriented design patterns [3]. Object-oriented patterns are considered by many to be one of the single most important advance in recent OOP. Surprisingly, however, in the first workshop on AOP held during the eleventh European Conference on OOP (ECOOP '97), a workshop whose main goal was to identify the "good questions" for exploring the idea of AOP, the question of *aspect-oriented patterns* was not raised (at least not in the workshop report. [15])

What should aspect-oriented patterns be like? No mainstream programming language is yet aspect-oriented. But you can design and write aspect-oriented programs, just like you can write object-oriented programs in almost any language. This is because AOP is more than a programming paradigm. It is a design framework for separation of concerns. In the absence of linguistic support, though, aspect-oriented patterns can provide the novice with *simple and elegant aspect oriented solutions to specific problems*. In fact, a few of the specific concerns raised in last year's workshop (e.g., those expressed in [6, 5]) are actually quests for aspect-oriented patterns.

This paper strives to declare *aspect-oriented* the already known *object-oriented* VISITOR [3] design pattern revisited from the point view of aspect-orientation. Section 2 ahead leads to the observation that the VISITOR not only describes an object-oriented pattern, but and perhaps even more importantly, it describes an aspect-oriented pattern, an aspect-oriented solution to a tangling problem in a particular context. As an aspect-oriented pattern it stands up to the *known-uses* measure: there are (at least two) real aspect-oriented related applications, the Demeter [11] system being the most famous, that apply the VISITOR pattern successfully for achieving separation of aspect-oriented concerns.

Implemented in C++, the *object-oriented* and the