

The Essence of the Visitor Pattern

Jens Palsberg¹ C. Barry Jay²

¹ Purdue University, Dept of Computer Science, W Lafayette, IN 47907, USA,
palsberg@cs.purdue.edu

² University of Technology, Sydney, School of Computing Sciences, P.O. Box 123
Broadway, 2007, Australia, cbj@socs.uts.edu.au

Abstract. For object-oriented programming, the Visitor pattern enables the definition of a new operation on an object structure without changing the classes of the objects. The price has been that the set of classes must be fixed in advance, and they must each have a so-called accept method. In this paper we demonstrate how to program visitors without relying on accept methods and without knowing all classes of the objects in advance. The idea, derived from related work on shape polymorphism in functional programming, is to separate (1) accessing subobjects, and (2) acting on them. In the object-oriented setting, reflection techniques support access to sub-objects, as demonstrated in our Java class, `Walkabout`. It supports all visitors as subclasses, and they can be programmed without any further use of reflection. Thus a program using the Visitor pattern can now be understood as a specialized version of a program using the `Walkabout` class.

1 Introduction

Design patterns [3] aim to make object-oriented systems more flexible. In particular, the Visitor pattern enables the definition of a new operation on an object structure without changing the classes of the objects. Recent examples of uses of the Visitor pattern include the Java Tree Builder (JTB) tool [13] and the JJTree tool [11], which are frontends for the Java Compiler Compiler (JavaCC) [11]. In both cases, the idea is that the user of the tool can write syntax-tree operations as so-called visitors rather than changing and recompiling the syntax-tree-node classes. The experience with the Visitor pattern is that many tasks can conveniently be written as visitors.

A basic assumption of the Visitor pattern is that one knows the classes of all objects to be visited. When the class structure changes, the visitors must be rewritten. For example, if one is writing a grammar and is using JTB or JJTree together with JavaCC, then changes to the grammar imply that the visitors must change as well. This is the case even when the changes are in a part of the grammar which is of no direct interest to a particular visitor.

Moreover, each of the classes must have a so-called accept method, which is used to pass the name of the class back to the visitor through dynamic binding.

In this paper we demonstrate how to program visitors without knowing all classes of the objects in advance, or relying on accept methods. The idea is to

separate (1) accessing subobjects, and (2) acting on them. We present the Java class `Walkabout` which uses reflection to access the structure of the objects it visits. All visitors are subclasses of `Walkabout`, and they can be programmed without any further use of reflection. We can now write visitors that will interact with arbitrary object structures. The classes of these objects need not change at all. The generic `Walkabout` class traverses the object structure without performing any actions. Actions appropriate to the different classes of objects can then be introduced in subclasses to produce specific visitors.

An inflexible alternative to the `Walkabout` class may be possible in cases where we are using the traditional Visitor pattern. From a description of the classes of the objects to be visited, we can generate a class with the same behavior as the `Walkabout` class. This approach is taken by the JTB tool. This is inflexible because when the class structure changes, a new `Walkabout`-like class has to be generated. With our new approach, the `Walkabout` class is the same for all applications.

The walkabout approach raises a whole new range of potential applications. For example, one could construct walkabouts that compute statistics associated to all office bearers in an organisation, without having to adapt the program to the current organisational structure. More generally, these techniques may underpin a query language for object-oriented data-bases that is stable with respect to changes in the underlying structure of the data-base. Thus, one could extract the names of all files which reference a particular individual, or update all occurrences of her address, without knowing the structure of the database itself. The goal of supporting such applications is shared with Lieberherr's adaptive programming [8, 9, 12], but this requires new language constructs whereas we are able to exploit the reflection capabilities of, say, Java 1.1.

Reflection introduces a significant performance penalty. One might say that this is the price of complete flexibility (which may or may not be worth paying). We are more optimistic, anticipating that future work will be able to automatically support specialised walkabouts, analogous to visitors, that combine efficiency in common cases without loss of generality. Some suggestions are considered in Section 5.

More generally, one can view programs constructed from walkabouts as executable specifications. Benchmarking can then be used to establish the chief sources of inefficiency, for which specialised code can be constructed.

The theoretical possibility of walkabouts was foreshadowed in [6] which was inspired by the emergence of *shape polymorphism* in functional programming [5, 1]. Shape polymorphism recognises that many common functions can be applied to a wide variety of data structures. Closest to the Visitor pattern is the higher-order function `map`. The high-level algorithm for `map f` is expressed as: (1) find every datum, and (2) apply `f` to it. The challenge is to find all the data using a single algorithm. The analogy extends further. Just as visitors can be seen as specialised forms of walkabouts, *polytypic programming* [4, 10] can be seen as a specialised form of shape polymorphism.

In the following section we review existing methods of visiting objects, and in

Section 3 we introduce walkabouts. In Section 4 we benchmark the performance of the `Walkabout` class, and in Section 5 we explore possible specialisations of walkabouts that relate them to the earlier techniques. Finally, in Section 6 we discuss links to shape polymorphism in functional programming, and in Section 7 we supply conclusions and directions for future work.

2 Visitors

The Visitor pattern describes a mechanism for interacting with the internal structure of composite objects that avoids frequent type casts or recompilation. Its advantages will be illustrated by means of a running Java example, summing an integer list.

2.1 First Approach: Instanceof and Type Casts.

The first attempt at summation in Figure 1 uses a simple list interface, with summation given by a loop which uses `instanceof` to check if a given `List`-object `l` is a `Nil`-object or a `Cons`-object. If it is a `Cons`-object, then the fields are accessed via type casts, and the loop is repeated.

```
interface List {}

class Nil implements List {}

class Cons implements List {
    int head;
    List tail;
}
.....

List l;      // The List-object we are working on.
int sum = 0; // Contains the sum after the loop.
boolean proceed = true;
while (proceed) {
    if (l instanceof Nil)
        proceed = false;
    else if (l instanceof Cons) {
        sum = sum + ((Cons) l).head; // Type cast!
        l = ((Cons) l).tail;        // Type cast!
    }
}
.....
```

Fig. 1. Type Casts

The advantage of this code is that it can be written without touching the classes `Nil` and `Cons`. The drawback is that the code constantly uses type casts and `instanceof` to determine what class of object it is considering.

2.2 Second Approach: Dedicated Methods.

One can dismiss the above piece of code by simply saying: it is not object-oriented! To access parts of an object, the classical approach is to use dedicated methods which both access and act on the subobjects. For our running example, we can insert `sum` methods into each of the classes in Figure 2.

```
interface List {
    int sum();
}

class Nil implements List {
    public int sum() {
        return 0;
    }
}

class Cons implements List {
    int head;
    List tail;
    public int sum() {
        return head + tail.sum();
    }
}
```

Fig. 2. Dedicated Methods

We can now compute the sum of all components of a given `List`-object `l` by writing `l.sum()`. The advantage of this code is that the type casts and `instanceof` operations have disappeared, and that the code can be written in a systematic way. The disadvantage is that every time we want to perform a new operation on `List`-objects, say, compute the product of all integer parts, then new dedicated methods have to be written for all the classes, and the classes must be recompiled.

2.3 Third Approach: The Visitor Pattern.

The Visitor pattern lets us define a new operation on an object structure without changing the classes of the objects on which it operates. Rather than writing dedicated methods for each programming task and afterwards recompiling, the

idea is to insert a so-called `accept` method in each class which passes control back to the visitor, which acts as a repository for the new methods. Code for the running example is given in Figure 3.

```
interface List {
    void accept(Visitor v);
}

class Nil implements List {
    public void accept(Visitor v) {
        v.visit(this);
    }
}

class Cons implements List {
    int head;
    List tail;
    public void accept(Visitor v) {
        v.visit(this);
    }
}

interface Visitor {
    void visit(Nil x);
    void visit(Cons x);
}

class SumVisitor implements Visitor {
    int sum = 0;
    public void visit(Nil x) {}
    public void visit(Cons x) {
        sum = sum + x.head;
        x.tail.accept(this);
    }
}

.....
    SumVisitor sv = new SumVisitor();
    l.accept(sv);
    System.out.println(sv.sum);
.....
```

Fig. 3. Visitors

Each `accept` method takes a visitor as argument. Its purpose is to inform the visitor of the object's class, which is used to determine the appropriate `visit` method. The interface `Visitor` declares a `visit` method for each of the basic

classes, which must be instantiated before use. Note that the `visit` methods describe both the action to be performed, e.g. `sum = sum + x.head`; and also the pattern of access, e.g. `x.tail.accept(this)`; accesses the tail of the list. The instance `sv` of a `SumVisitor` above shows how to compute and print the sum of all components of a given `List`-object `l`.

The advantage is that one can write code that manipulates objects of existing classes without recompiling those classes, provided that all objects must have an `accept` method.

In summary, the Visitor pattern combines the advantages of the two other approaches, as represented in the following table:

	Frequent type casts?	Frequent recompilation?
Instanceof and type casts	Yes	No
Dedicated methods	No	Yes
The Visitor pattern	No	No

3 Walkabouts

We now demonstrate how to program visitors without relying on `accept` methods. The chief conceptual difference from the visitors is that the default access pattern is supplied by a generic class `Walkabout`: only the actions performed at each object need to be supplied. This expresses the insight that visitors proceed by finding each datum, and then acting on it, much as a general mapping algorithm in functional programming does.

The default access pattern is determined by reflection, which is used to determine the internal structure of an object, specifically its fields, which are then visited in sequence. Reflection can also be used to determine the class of an object, which eliminates the need for `accept` methods.

The `Walkabout` class has just one method, `visit`, which takes an argument of type `Object`. Replacing the reflection code with pseudo-code yields the informal description of the class in Figure 4.

```
class Walkabout {
  void visit(Object v) {
    if ( v != null )
      if (this has a public visit method for the class of v)
        this.visit(v);
      else if (v is not of primitive type)
        for (each field f of v)
          this.visit(v.f);
  }
}
```

Fig. 4. Walkabout Pseudo-code

When the general `visit` method is invoked, it will first try to invoke a `visit` method for the class of the argument. If no such method is found, and the argument is an object, then the general `visit` method will be invoked on each of the fields of the argument.

The full Java code for the class is presented in Figure 5. All visitors are subclasses of `Walkabout` that extend it with `visit` methods for particular classes, with no need for further use of reflection.

```
class Walkabout {
    void visit(Object v) {
        if ( v != null ) {
            Object [] os = {v};
            Class vClass = v.getClass();
            Class [] cs = {vClass};
            try { this.getClass().getMethod("visit",cs).invoke(this,os); }
            catch (java.lang.NoSuchMethodException e) {
                if (!(v instanceof Number)      |
                    (v instanceof Byte)         | (v instanceof Short) |
                    (v instanceof Character) | (v instanceof Boolean))) {
                    java.lang.reflect.Field [] vFields = vClass.getFields();
                    for (int i=0; i<vFields.length; i++) {
                        try { this.visit(vFields[i].get(v)); }
                        catch (java.lang.IllegalAccessException e1) {}
                        // Cannot happen.
                    }
                }
            }
            catch (java.lang.Exception e) {} // Cannot happen.
        }
    }
}
```

Fig. 5. The `Walkabout` class

A walkabout for computing and printing the sum of all components of a given `List`-object `l` is given in Figure 6. Let us compare this walkabout with the corresponding visitor in figure 3.

- The customised `Visitor` interface has been replaced with the general `Walkabout` class.
- The basic classes (for lists) are here defined exactly as in the first approach since there are no `accept` methods to worry about.
- Even so, the customised walkabout actually is simpler than the corresponding visitor in two ways.
- First, there is no `visit` method for `Nil` in class `SumWalkabout`. Such a

- method is unnecessary because class `Walkabout` will determine that `Nil` has no fields.
- Second, and more fundamental, is that the indirection caused by `accept` in `x.tail.accept(this)`; has been replaced by the direct recursive call `this.visit(x.tail)`;

```
interface List {}

class Nil implements List {}

class Cons implements List {
    int head;
    List tail;
}

class SumWalkabout extends Walkabout {
    int sum = 0;
    public void visit(Cons x) {
        sum = sum + x.head;
        this.visit(x.tail);
    }
}

.....
SumWalkabout sw = new SumWalkabout();
sw.visit(l);
System.out.println(sw.sum);
.....
```

Fig. 6. SumWalkabout

4 Performance Measurements

The running example with Lists is a slightly simplified version of our benchmark program. The full program has a third implementation of `List`, namely a class `Link` which in its basic form looks like this:

```
class Link implements List {
    boolean color;
    List li;
}
```

To test the efficiency of the code produced by the various programming styles, we wrote a procedure which produces a list of length 2000 with alternating `Cons`

and `Link` cells. We still want to compute the sum of all the integer components. In each program we repeat the summation 100 times. The following run times were obtained on a Sun Ultra 1, model 170E, 167-MHz UltraSPARC, 2.1-Gbyte 7200 F/W SCSI-2 disk, and 128-Mbyte Memory. The compilation and execution of the Java code was done with Sun's JDK 1.1.3.

Approach	Run-time
Construction of list only	0.68 s
type casts	1.18 s
dedicated methods	0.99 s
Visitor	1.17 s
Walkabout	4 min 59.93 s

These results show that the use of reflection imposes a significant performance penalty which we will now discuss. In passing, notice that the extra control flow caused by the `Visitor` is no slower than the use of `instanceof` and type casts.

5 From Walkabout to Visitors

We now explain how a program using the Visitor pattern can be understood as a specialized version of a program using the `Walkabout` class. We will explore two related avenues: one leads to the Visitor pattern as described in Section 1, and another leads to a variant which is used, for example, in the Java Tree Builder tool [13]. The specialization of class `Walkabout` is in two steps.

Step 1: Which classes of objects will be visited?

Suppose that the object structure to be visited only contains objects of classes C_1, \dots, C_n and that C_i has fields f_{i1}, \dots, f_{im_i} . This knowledge can be used to specialize class `Walkabout` to the more efficient version in Figure 7.

Notice the use of type casts and `instanceof`. These play the role formerly played by `getFields()`, to allow access to fields even though no `visit` method is specified for the class of the current object. Thus the code contains fewer uses of reflection than the original `Walkabout` class, and is therefore faster. On the other hand, it is more rigid than the original `Walkabout` because it will ignore objects of unrecognised class.

Suppose next that the programmer of the subclass of `Walkabout` supplies a `visit` method for *all* of the classes C_1, \dots, C_n , so that no default action has to be supplied in the `walkabout` itself. Then none of the “`else`” branches in the `visit` method of class `Walkabout` will ever be executed, and so can be deleted to obtain the class in Figure 8.

Step 2: Accept methods to determine the class of the argument

Both outcomes from Step 1 hide a use of reflection in the description `this has a public visit method for the class of v`, as explained in Section 3. The

```

class Walkabout {
  void visit(Object v) {
    if ( v != null )
      if (this has a public visit method for the class of v)
        this.visit(v);
      else if (v instanceof C1) {
        this.visit(((C1) v).f11);
        ...
        this.visit(((C1) v).f1m1);
      }
      else ...
      else if (v instanceof Cn) {
        this.visit(((Cn) v).fn1);
        ...
        this.visit(((Cn) v).fnmn);
      }
    }
  }
}

```

Fig. 7. Walkabout for Fixed Classes

```

class Walkabout {
  void visit(Object v) {
    if ( v != null )
      if (this has a public visit method for the class of v)
        this.visit(v);
    }
  }
}

```

Fig. 8. Walkabout with no Default Action

problem is to invoke the `visit` method for the right class. Of course, if the compiler could determine the class then reflection would be unnecessary, but in general recursive calls to `visit` the compiler can only infer the class `Object`.

One way of getting rid of the use of reflection is to use type casts, just as in the long piece of code shown in Step 1. The elegant alternative is to have a method `accept` in each of the classes C_1, \dots, C_n that simply invokes the `visit` method, as illustrated in Section 2. Then `v.accept(this);` calls the visitor (`this`) on `v` with dynamic binding ensuring that the visitor knows the class of `v`.

Combining this with the specialisation of Step 1 leaves two possibilities.

A: In the former case, some subclasses of `Walkabout` may not have a `visit` method for some of the classes C_1, \dots, C_n so that some default behaviour is required, similar to the `else` clauses in Step 1. The class `Walkabout` then takes the form in Figure 9. Notice that the type casts and `instanceof` disappear because of the dynamic binding. Such a class is used in the Java Tree Builder tool [13].

B: In the latter case, all subclasses of `Walkabout` have a `visit` method for all of

the classes C_1, \dots, C_n which will be invoked directly by the accept methods. Thus, no code in class `Walkabout` is needed, and it can be written as an interface, like the interface named `Visitor` in Figure 3.

```
class Walkabout {
  void visit( $C_1$  v) {
    this.accept(v.f11);
    ...
    this.accept(v.f1m1);
  }
  ...
  void visit( $C_n$  v) {
    this.accept(v.fn1);
    ...
    this.accept(v.fnmn);
  }
}
```

Fig. 9. Walkabout with Accept Methods

Ideally, one would like to avoid reflection where specialised code is supplied, but retain the ability to invoke it in exceptional cases. Perhaps partial evaluation [7] could be used to eliminate the reflection where specialised code exists. We have not yet investigated this.

6 Shape Polymorphism

The `Walkabout` class is inspired by novel techniques for functional programming with recursive data types. In the standard approaches, each inductive data type, for example, list or tree, requires its own functions for traversal, typically described by pattern-matching. For example, to apply a function f to each entry in a list one applies `listmap f` to the whole list, where `listmap` is defined by

$$\begin{aligned} \text{listmap } f \text{ nil} &= \text{nil} \\ \text{listmap } f \text{ cons}(h, t) &= \text{cons}(f h, \text{listmap } f t) \end{aligned}$$

The mapping of f across a binary tree requires a different pattern-matching algorithm, with actions on leaves and nodes.

The first attempt to regularise this situation was made in `CHARITY` [2]. Each data type constructor F , used to construct, say, a new tree type, induced the compiler to construct code for a suite of combinators, such as `mapF` for the corresponding data structures, sparing the programmer from writing the pattern-matching definitions themselves, at the cost of writing the appropriate annotations for type constructors.

From this developed *shape polymorphic programming* [5, 1]. It shares the same goals as CHARITY, but uses *parametrically polymorphic* algorithms, instead of type-based specialisation. That is, the run-time code for, say, mapping is the same for lists and trees, instead of being specialised according to the type. The algorithms are not based on pattern-matching, but using information stored within the data structure to determine where the data is stored.

In shape polymorphic programming, the means of locating the data in a data structure, or shape, is separated from the description of the action to be performed on each datum found. Code reuse arises since the means of locating the data can be described in general terms that applies to arbitrary tree types, by including a little more information than usual at each node of the tree.

In another related development, *polytypic programming* [4, 10] has eliminated the need for constructor annotations in CHARITY by improved type inference techniques, on which code specialisation is based. That is, polytypic programming is a principled form of ad hoc polymorphism in that the compiler uses type inference to determine the appropriate choice of algorithm. The primary application domain appears to be compiler-construction techniques, many of whose algorithms are tree-based.

Of course, the specialised polytypic programs execute faster than the generic shape polymorphic ones, so the challenge is to automate the specialisation of shape polymorphic programs to their polytypic counterparts.

The striking analogy with visitors was first noted in [6], and is here put to good effect. Walkabouts correspond to shape polymorphic programs. In particular, the high level mapping algorithm

- find all data,
- act on each one.

is very similar to the high-level walkabout algorithm

- find all objects,
- act at each one.

The main difference is the ability to manipulate state in the latter. (But even this difference can be minimised using state monads.)

It is this observation that lead to the `Walkabout` class. More precisely, the `Walkabout` class corresponds to `map identity` as it does not perform any action on the objects encountered. Conversely, `map f` corresponds to a sub-class of `Walkabout` which only acts to update fields by `f`.

Like shape polymorphic programs, walkabouts are completely generic, but must extract a little more information from their arguments (using reflection) to succeed. Like polytypic programs, visitors use class (or type) information to specialise their code. The challenge in both domains is to automate the code specialisation, so as to combine the benefits of parametricity and efficiency.

7 Conclusion

The `Walkabout` class improves upon earlier implementations of the Visitor pattern as it is able to act on arbitrary object trees. This uniform action is achieved by reflection techniques which allow the class and fields of an arbitrary object to be determined. As Java 1.1 supports reflection, we have been able to implement `Walkabout` as a class of some 20 lines.

The inspiration for walkabouts was supplied by shape polymorphic (functional) programs, such as mapping.

Executing code that uses the `Walkabout` class is slow, but one can imagine a partial evaluation technique which automatically produces efficient Visitor code. Other objectives include extending `Walkabout` to handle array objects and object graphs.

Acknowledgment. We thank Luca Cardelli for suggesting the use of Java 1.1.

References

1. G. Bellé, C.B. Jay, and E. Moggi. Functorial ML. In *Proc. PLILP '96*, pages 32–46. Springer-Verlag (LNCS 1140), 1996.
2. J.R.B. Cockett and T. Fukushima. About Charity. Technical Report 92/480/18, University of Calgary, 1992.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
4. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Proc. POPL '97, 24th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, 1997.
5. C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
6. C.B. Jay and J. Noble. Shaping object-oriented programs. Technical Report 96–16, University of Technology, Sydney, 1996.
7. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.
8. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
9. Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.
10. L. Meertens. Calculate polytypically! In *Proc. PLILP '96*. Springer-Verlag (LNCS 1140), 1996.
11. Sun Microsystems. The java compiler compiler. www.suntest.com/Jack/, 1997.
12. Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.
13. Kevin Tao and Jens Palsberg. The Java tree builder. Purdue University, www.cs.purdue.edu/people/palsberg, 1997.

This article was processed using the L^AT_EX macro package with LLNCS style