Guiding Visitors: Separating Navigation from Computation

Martin Bravenboer
Eelco Visser

Institute of Information and Computing Sciences, University of Utrecht, P.O. Box 80.089, 3508 TB, Utrecht, The Netherlands
{mbravenb,visser}@cs.uu.nl

29th November 2001

Abstract

Traversals over the object structure are widely used in object-oriented programming, in particular in language processing applications. The visitor pattern separates computation from traversal by specifying the computations that should be performed at each object in a separate visitor class. This makes the implementation of different computations reusing the same traversal scheme possible. However, navigation through the object structure is fixed in the accept methods implemented by the objects that are traversed. This makes it difficult to use other navigation orders.

In this paper, we introduce the Guide pattern that describes the separation of navigation from computation *and* object structure using a double-dispatching iterator. The pattern makes it possible to implement a whole range of navigation schemes for an object-structure. Using a *self-dispatching* approach based on reflective method lookup such navigation schemes can be made reusable for whole classes of object-structures (implementing a common interface). The efficiency of this approach is provided by caching method lookups. We extend the approach to generic navigation through arbitrary object-structures using reflective field lookup. This results in a generalization of the Walkabout class of Palsberg and Jay with a huge performance improvement in Java, making the Walkabout usable in practice.

1 Introduction

The Visitor pattern is widely used in language processing systems that manipulate abstract syntax trees. The Visitor pattern enables the addition of new operations that traverse an object structure without changing the classes of the object structure.

Parser generator tools such as Java Tree Builder (JTB) [10], JJTree [8] and JJForester[6, 7] use the visitor pattern to provide traversals over syntax trees produced by parsers. The tools generate the classes for representing nodes in abstract syntax trees from a grammar file. The generated classes include accept methods for use by visitors. Operations on abstract syntax trees can then be defined separately from the abstract syntax classes in visitor classes that rely on the generated accept methods. The separation of operation and structure is crucial in this class of tools to avoid editing generated code.

In their original description of the Visitor pattern, Gamma et al. [5] raise the question where the navigation through the object structure should be defined and suggest three possibilities: in the object structure, in the visitor, or in a separate iterator object. Elaborations of the first two options are discussed in [5] and are commonly used in applications such as the tools mentioned above.

A restriction of these solutions is that navigation is closely coupled either to the object structure or to the operation on that structure. Many operations require other traversal orders over trees than the regular top-down traversal and/or do not need to traverse the complete abstract syntax tree. Accommodating such non-standard traversals requires extending the object structure to provide a different kind of accept or including navigation code in visitors. Moreover, operations often do not need to traverse the complete abstract syntax tree.

In this paper, we introduce the Guide pattern that describes the separation of navigation from computation *and* object structure. A guide is an object that captures a traversal scheme through an object structure. A visitor uses a guide to navigate the object structure, thus hiding the details of navigation from the visitor. Consequently visitors can be reused with different guides to achieve different effects. If we have for example defined a tree where the components can be selected, we can define a visitor that just selects items. The guide will determine what items are selected in the tree. Similarly, guides are not dependent on visitors and can be reused in many different visitors. For instance, the notion of top-down or bottom-up traversal can be captured by a guide and used in visitors for pretty-printing, collecting components by predicates, transforming a structure etc. When guides are defined for a fixed class hierarchy they can also be reused on extensions of this hierarchy.

In Section 2 we review the Visitor pattern and note how the coupling of navigation with the object structure or the computation reduces the flexibility and reusability of navigation. We propose to solve this problem by capturing a navigation scheme in a separate object that *guides* a visitor through the object structure. In Section 3 we discuss *consulting* guides that are called on by the visitor to direct it through the object structure. We illustrate the flexibility in navigation this provides with several implementations of traversal schemes. In Section 4 we present *commanding* guides that are in total control of navigation and call the visitor when it must perform a computation. Examples of commanding guides include a generalization of the Hierarchical Visitor pattern.

The use of guides enables the reuse of navigation. In Section 5 we consider the reuse of guides and visitors for specializations and extensions of an object structure. It turns out that the accept methods in the object structure, which are responsible for double-dispatching to the correct visit method, limit the extensibility of guides and visitors. We introduce *self-dispatching* visitors and guides, which take care of dispatching to the correct method themselves through reflective method lookup. Thus, a guide defined over a family of structures can be used for any visitor and any member of the family.

Finally, we notice that the Walkabout of Palsberg and Jay [9] is just a guide. In Section 6 we present a generalization of the Walkabout as a combination of a self-dispatching guide and a self-dispatching visitor. This makes it possible to define generic traversal schemes over arbitrary object structures without the need to provide double-dispatching accept methods. The problem of the original Walkabout is its poor performance. By caching reflective method and field lookups we have been able to improve the performance of the Walkabout such that it can almost reach the speed of a normal visitor without reflection. This makes the Walkabout now applicable in practice.

2 The Visitor Pattern

The Visitor pattern [5, 9] is used to define an operation over an object structure such as an abstract syntax tree or a collection without having to change the class structure itself. The Visitor pattern is useful in situations where many new operations will be added, but the class structure is stable.

2.1 Visitors

The Visitor pattern will be illustrated by a simple example: counting the number of leafs in a tree. Figure 1 shows the classes we use to represent a tree.

Note that we have used inner classes to represent the notion of a Tree as one Object and not a collection of Objects. The components of the Tree 'live' in the Tree in this way. Although we use this notion of an object structure encapsulated in an object in this paper, please keep in mind that this is absolutely not necessary if you want to apply Visitors and/or our Guides.

We have left out the implementation of the methods because they are not interesting. A `Node` will have a collection and returns a `List<Tree.Component>` in the `getChildren` method. Also note that we have used parameterized types. This feature is currently not yet available in Java but it is planned for the Java 2 SDK 1.5. We have used the early-access implementation of the compiler [1]. This implementation is much like the approach of GJ [3, 2]. Parameterised types allow us to parameterise a class with another class. This will prevent many type casts, especially when working with collections.

2.1.1 Monolithic Recursive Method

The first attempt to count the number of leafs is to write a method outside the object structure. The method is shown in Figure 2. The code looks quite clean because of the recursion but it contains instanceof to determine the type of a `Tree.Component`. This approach is maybe useful in many cases (and often it will be the smallest solution) but it is certainly not object-oriented, which does however not mean that this solution is always to be avoided.

2.1.2 Many Methods

The best known solution to computing values over an object structure is to add dedicated methods to the structure. This approach is shown in Figure 3. We have left out the methods that were already defined in Figure 1.

The advantages of this solution are that there are no type casts and instanceof operations. The key to this is the abstract method `sum` in `Tree.Component`. This solution looks more attractive from an object-oriented point of view and the addition of a type of `Tree.Component` to the `Tree` is

```
class Tree
{
   Component getRoot();

   abstract class Component
   {
      Node getParent();
   }

   class Node extends Component
   {
      List<Component> getChildren();
   }

   class Leaf extends Component
   {}
}
```

Figure 1: Simple tree structure

```
int countLeafs(Tree tree)
{
   return countLeafs(tree.getRoot());
}

int countLeafs(Tree.Component component)
{
   int result = 0;

   if(component instanceof Tree.Node)
   {
      Tree.Node node = (Tree.Node) component;
      Iterator<Tree.Component> children
               = component.getChildren().iterator();

      while(children.hasNext())
      {
         result += countLeafs(children.next());
      }
   }
   else if(component instanceof Tree.Leaf)
   {
      result = 1;
   }

   return result;
}
```

Figure 2: First attempt to count the number of leafs

quite easy. The algorithm can be extended in an easy, local way. The disadvantage of this solution is that the algorithm is spread over all components of the `Tree` and that the addition of a new operation requires rewriting all the

```
class Tree
{
   abstract class Component
   {
      abstract int countLeafs();
   }

   class Node extends Component
   {
      int countLeafs()
      {
         int result = 0;

         Iterator<Tree.Component> children
                 = component.getChildren().iterator();

         while(children.hasNext())
         {
            result += chilren.next().countLeafs();
         }

         return result;
      }
   }

   class Leaf extends Component
   {
      int countLeafs()
      {
         return 1;
      }
   }
}
```

Figure 3: Second attempt to count the number of leafs

components. Furthermore, operations that only need information from a small subset of classes in the object structure, still require dedicated methods to be defined for all components of that structure. This is already illustrated in the `Tree` object structure with only 2 types of objects. The nodes are in fact of no interest to the leaf counting operation.

2.1.3 Separate Algorithm from Object Structure

The solution is to apply the Visitor pattern. The Visitor pattern allows the operation to be defined in just one class separately from the object structure with a separate method for each kind of object in the object structure.

The interface for a `TreeVisitor` is shown in Figure 5. Implementations must have two `visit` methods, one for each type of component in the structure. All implementations of `Tree.Component` must define a method `acceptVisit` as is shown in the new `Tree` in Figure 4. This method plays the crucial role of selecting the right type in the Visitor pattern.

```
class Tree
{
   abstract class Component
   {
      abstract void acceptVisit(TreeVisitor visitor);
   }

   class Node extends Component
   {
      void acceptVisit(TreeVisitor visitor)
      {
         visitor.visit(this);

         Iterator<Tree.Component> children
                = getChildren().iterator();
         while(children.hasNext())
         {
            children.next().acceptVisit(visitor);
         }
      }
   }

   class Leaf extends Component
   {
      void acceptVisit(TreeVisitor visitor)
      {
         visitor.visit(this);
      }
   }
}
```

Figure 4: Tree with accept methods in charge of navigation

```
interface TreeVisitor
{
   void visit(Tree.Node node);
   void visit(Tree.Leaf leaf);
}
```

Figure 5: Interface of a Visitor of a Tree

The `acceptVisit` is necessary to dispatch a visit of an object in the object structure to the correct `visit` method. It is in this way a (better) replacement of the instanceof and type casts. In the common superclass of all objects in the tree (`Tree.Component`) we have defined the method `acceptVisit` as abstract. Therefore we can call the acceptVisit method on a `Tree.Component`. The `acceptVisit` method must now call a `visit` method in the `TreeVisitor` to execute a certain operation on the current location in the tree. In all extension of the `Tree.Component` this method is (and must be) implemented. Because of this, a call to the `acceptVisit` method of a `Tree.Component` will go via the real class of the object and thus `this` in that `acceptVisit` method will refer to an object of a real type (in this case a `Node` or a `Leaf`) in the tree. The

```
class CountLeafsVisitor implements TreeVisitor
{
   int result = 0;

   void visit(Tree.Node node)
   {
   }

   void visit(Tree.Leaf leaf)
   {
      result = result + 1;
   }
}
```

Figure 6: Third attempt: the Visitor pattern applied

`TreeVisitor` has `visit` methods in for all this kind of classes.

Because of this we can define operations, which are `visit` methods, on all real classes of the tree and call the correct `visit` method when we only now that we have a `Tree.Component`. This `acceptVisit` method is said to be a *double-dispatch* operation. This type of dispatching is called 'double' because the correct operation (`visit` method) will be executed depending on two values: the type of `Tree.Component` and the type of `Visitor`.

The Visitor pattern comes in two styles. In the common version navigation is controlled by the classes of the object structure. The `acceptVisit` method is not only responsible for dispatching to the correct `visit` method but also controls the navigation through the object structure. Class `Tree` in Figure 4 is an implementation in this style. The `CountLeafsVisitor` in Figure 6 is an example visitor in this style. The `visit` method for a `Node` does nothing. The `visit` method for a `Leaf` just adds one to the result.

In the second style the `acceptVisit` method is only responsible for double-dispatching. The navigation, i.e., calling the `acceptVisit` methods of the children, is done in the visitor itself. The code for the leaf counting visitor using this style is shown in Figure 7 and the new `Tree` is shown in Figure 8. The `acceptVisit` methods are now much smaller. The advantage of this style is that the Visitor is able to control navigation. It can for example choose to implement another traversal order over the object structure. This style is also preferable when the results of the visits to the children are needed in the visit method. Clearly, the disadvantage of this style is the increased complexity of the visit methods. This disadvantage can be mitigated by reusing navigation code through inheritance from a `TreeVisitor` class which defines a default navigation scheme in the visit methods. However, each override of a visit method will have to reimplement navigation.

2.2 Navigation Coupled with Computation

In the variants of the Visitor pattern discussed above navigation is strongly coupled with either the object structure or with the Visitor. If navigation is controlled by the `accept` methods of the object structure, the order of traversals is fixed, which entails that all visitors have to follow the same path over the

```
class CountLeafsVisitor implements TreeVisitor
{
   int result = 0;

   void visit(Tree.Node node)
   {
      Iterator<Tree.Component> children
               = node.getChildren().iterator();
      while(children.hasNext())
      {
         children.next().acceptVisit(visitor);
      }
   }

   void visit(Tree.Leaf leaf)
   {
      result = result + 1;
   }
}
```

Figure 7: Third attempt, second edition

structure. If navigation is controlled by the `visit` methods of the visitor, the encoding of traversals is a source of extra complexity for the programmer, and traversal schemes are not reusable. The parser generator tools JTB, JJTree and JJForester each make a different compromise in their design decisions.

The Java Tree Builder [10] puts the navigation control into the `visit` methods of the Visitor. JTB generates a default navigation behaviour in an implementation of a Visitor: `DepthFirstVisitor`. Visitors that want to use this route must extend `DepthFirstVisitor`. JJTree [8] is very similar to JTB in its style of generated `visit` and `accept` methods. JJTree also puts navigation in the Visitor. JJTree does not generate a default navigation behaviour. It does, however, provide a method `childrenAccept` in the object structure to facilitate the job of writing a Visitor. In contrast, JJForester [6, 7] puts navigation control in the object structure. To provide control over navigation, JJForester generates several (currently two) styles of `accept` methods for each kind of navigation in the object structure. `Visit` methods must decide what kind of `accept` method to invoke.

Each of the tools tries to offer the Visitor freedom of navigation. However, none of the tools offer complete freedom of navigation *and* easy reusability of navigation schemes. In the next section we will introduce the notion of guide to capture navigation schemes.

3 Capturing Navigation in Guides

Defining the navigation in the object structure or in the visitor reduces the flexibility and reusability of navigation. By capturing a navigation scheme in a separate object that *guides* a visitor through the object structure, navigation becomes first class and can be defined separately from object structure and visitor. Guides come in two flavours. A *consulting* guide is called on by the

```
class Tree
{
   abstract class Component
   {
      abstract void acceptVisit(TreeVisitor visitor);
   }

   class Node extends Component
   {
      List<Component> getChildren();

      void acceptVisit(TreeVisitor visitor)
      {
         visitor.visit(this);
      }
   }

   class Leaf extends Component
   {
      void acceptVisit(TreeVisitor visitor)
      {
         visitor.visit(this);
      }
   }
}
```

Figure 8: Tree with accept methods without navigation control

```
interface TreeGuide
{
   void guide(TreeVisitor visitor, Tree.Node node);
   void guide(TreeVisitor visitor, Tree.Leaf leaf);
}
```

Figure 9: Interface of a Guide for a Visitor of a Tree

visitor to direct it to the next object to visit. A *commanding* guide, on the other hand, is in complete control of the traversal and calls to the visitor when it may perform computations. In this section we present the notion of a consulting guide and illustrate it with several navigation schemes. In the next section we will discuss commanding guides.

We illustrate the Guide pattern with the `TreeVisitor` interface of Figure 5 and the `Tree` class of Figure 8, in which the `acceptVisit` methods have the function of double-dispatching to the right `visit` method and are not responsible for navigation. Note that in this setup the classic Visitor pattern (in the style where the visitor is responsible for navigation) can still be used. This makes the Guide pattern optional and not compulsory. When an object structure is generated by a parser generator it is therefore possible to generate just one accept method and let the user decide whether or not to use guides for navigation control.

A `TreeGuide` (Figure 9) has a `guide` method for each implementation of `Tree.Component`. A `guide` method is called by the visitor with the visitor

```
class TopDownTreeGuide implements TreeGuide
{
   void guide(TreeVisitor visitor, Tree.Node node)
   {
      Iterator<Tree.Component> children
         = node.getChildren().iterator();

      while(children.hasNext())
      {
         Tree.Component component = children.next();
         component.acceptVisit(visitor);
      }
   }

   void guide(TreeVisitor visitor, Tree.Leaf leaf)
   {}
}
```

Figure 10: Top-down implementation of a TreeGuide

```
abstract class GuidedTreeVisitor implements TreeVisitor
{
   protected TreeGuide guide;

   GuidedTreeVisitor(TreeGuide guide)
   {
      this.guide = guide;
   }

   abstract void visit(Tree.Node node);
   abstract void visit(Tree.Leaf leaf);
}
```

Figure 11: TreeVisitor with a TreeGuide

and the object currently visited and determines which objects to visit next. The double-dispatching `acceptVisit` method of the `Tree` classes is used to call the correct `visit` method of the visitor. A concrete example of a guide is the `TopDownTreeGuide` (Figure 10), which implements the common top-down navigation strategy in which the nodes of a tree are visited in a top-down, depth-first order. For a `Node` the `TopDownTreeGuide` leads the Visitor to the children of the `Node`. For a `Leaf` the `TopDownTreeGuide` does nothing.

A `GuidedTreeVisitor` (Figure 11) is a `TreeVisitor` with an associated `TreeGuide`. We can now reimplement the `CountLeafsVisitor` as a `GuidedTreeVisitor` (Figure 12). Instead of explicitly implementing navigation in the `visit` methods, the guide is called on for navigation. Thus, navigation is separated from the visitor and the object structure. A navigation scheme can be reused in different visitors and a visitor can be used with different navigation schemes. For instance, the `CountLeafsVisitor` can get a `TreeGuide` as a parameter at construction time. The default constructor (without arguments) constructs a visitor that uses the `TopDownTreeGuide`.

Other examples of guides that capture different navigation schemes are the

```
class CountLeafsVisitor extends GuidedTreeVisitor
{
   int result = 0;

   CountLeafsVisitor()
   {
      this(new TopDownTreeGuide());
   }

   CountLeafsVisitor(TreeGuide guide)
   {
      super(guide);
   }

   void visit(Tree.Node node)
   {
      guide.guide(this, node);
   }

   void visit(Tree.Leaf leaf)
   {
      result += 1;
      guide.guide(this, leaf);
   }
}
```

Figure 12: Counting leafs with Guide

ToRootGuide, the TopDownSpineGuide, and the BreadthFirstTreeGuide.

The ToRootGuide (Figure 15) guides the visitor from a node in a Tree to the root of that Tree. The visitor visits all nodes on the path to the root. In an experimental implementation of the Extract Method refactoring as an extension of the refactoring tool of Chris Seguin [11], this guide is used to collect the exceptions that are catched around a set of statements. If we had chosen to count nodes instead of leafs, we could use the ToRootGuide to count the depth of a node or leaf in the tree.

The TopDownSpineGuide is another example of a guide that does not visit all nodes in a tree, but instead finds a single path down a tree (the spine). The nodes along this path are visited by the visitor. By throwing a VisitException (a simple extension of Exception), the visitor indicates that a certain path was not the right one. The guide catches the exception and continues searching for a good path at the next child. When there are no children that can be visited by the Visitor the traversal stops. This TopDownSpineGuide can be used to find a node in a tree and collect information along the path leading to that node.

Note that the introduction of the VisitException slightly complicates matters, since the TreeVisitor must be able to throw a VisitException. The RiskyTreeVisitor interface (Figure 14) models visitors that can raise this exception. The Tree class must be extended with an additional acceptVisit method that can throw a VisitException. We do not want to use the RiskyTreeVisitor for all visitors to avoid having to catch exceptions where they are not used.

```
class TopDownSpineGuide implements RiskyTreeGuide
{
   void guide(RiskyTreeVisitor visitor, Tree.Node node)
   {
      Iterator<Tree.Component> children
                  = node.getChildren().iterator();

      tryNext(visitor, children);
   }

   void tryNext(  RiskyTreeVisitor visitor,
                  Iterator<Tree.Component> iterator)
   {
      try
      {
         if(iterator.hasNext())
         {
            iterator.next().acceptVisit(visitor);
         }
      }
      catch(VisitException exception)
      {
         tryNext(visitor, iterator);
      }
   }

   void guide(RiskyTreeVisitor visitor, Tree.Leaf leaf)
   {
   }
}
```

Figure 13: Guide that guides a Visitor over a spine

```
interface RiskyTreeVisitor
{
   void visit(Tree.Node node) throws VisitException;
   void visit(Tree.Leaf leaf) throws VisitException;
}
```

Figure 14: TreeVisitor with a chance of failure

As a final example, Figure 16 presents a guide for breadth-first traversal. This traversal visits an object structure layer by layer. All components of depth x will be visited before the components at depth $x + 1$ will be visited. This traversal is hard to implement in the classical visitor pattern. The default algorithm for `BreadthFirstTreeGuide` uses a `Queue` (`LinkedList` in Java). The components that must be visited are placed in the `Queue`.

To conclude, we have seen that the use of a separate guide for navigation control enables the reuse and adaption of navigation in combination with the Visitor pattern. More advanced traversal strategies are now possible without rewriting them again and again in the `visit` methods of the Visitor. In the next section we will examine commanding guides, which embody a more radical

```
class ToRootTreeGuide implements TreeGuide
{
   void guide(TreeVisitor visitor, Tree.Node node)
   {
      guideToParent(visitor, node);
   }

   void guide(TreeVisitor visitor, Tree.Leaf leaf)
   {
      guideToParent(visitor, leaf);
   }

   void guideToParent(TreeVisitor    visitor,
                      Tree.Component component)
   {
      Tree.Node parent = component.getParent();

      if(parent != null)
      {
         parent.acceptVisit(visitor);
      }
   }
}
```

Figure 15: To-root implementation of a TreeGuide

separation of navigation from computation.

4 Guides in Control

In the guided visitors in the previous section the visit method still has to decide when to call to the guide. For example, in the `TopDownTreeGuide` the visitor can do something before and after calling the guide for navigation. Therefore, there is no difference between a bottom-up and a top-down guide, i.e., the visitor still has some control over navigation. In this section we will present a more radical version of guides that have total control over navigation.

To take away control over navigation from the visitor, the guide should call the visitor instead of the other way around. We will illustrate this using the notions of bottom-up and top-down. The `TopDownTreeGuide` in Figure 17 implements a strict top-down traversal in which visit actions are performed *before* the children of a node are visited. That is, the `guide` method first (indirectly) calls the `TreeVisitor` and then guides itself to new locations. The `visit` methods of the visitor are now not allowed to call a guide or do any other navigation. A strict bottom-up guide is a variation on `TopDownTreeGuide` in which the visitor is called *after* navigation is performed.

In this approach the visitor for the leaf counting example remains the same as in Figure 6. However, that visitor was originally created for an object structure where the `acceptVisit` methods are also responsible for navigation. In the new implementation, navigation is now completely abstracted from the operation *and* from the object structure. Therefore, `Tree.Components` must have a new

```
class BreadthFirstTreeGuide implements TreeGuide
{
   LinkedList<Tree.Component> queue;

   void guide(TreeVisitor visitor, Tree.Node node)
   {
      Iterator<Tree.Component> children
                  = node.getChildren().iterator();

      while(children.hasNext())
      {
         Tree.Component component = children.next();
         queue.addLast(visitor);
      }

      toNext(visitor);
   }

   void toNext(TreeVisitor visitor)
   {
      if(!queque.isEmpty())
      {
         Tree.Component component
                         = queque.removeFirst();
         component.acceptVisit(visitor);
      }
   }

   void guide(TreeVisitor visitor, Tree.Leaf leaf)
   {
      toNext(visitor);
   }
}
```

Figure 16: TreeGuide providing a breadth-first traversal

method `acceptGuide` which has exactly the same double-dispatching function as the `acceptVisit` method. Figure 18 shows the complete `Tree` that is needed to use commanding guides.

Another example of a commanding guide is provided by the down-up traversal, a combined bottom-up and top-down traversal. This traversal is in fact a generalization of the Hierarchical Visitor Pattern [4]. A visitor making a down-up traversal must have two methods for each class in the object-structure. The first method, `discover`, is called when the visitor is going down and encounters the object for the first time. After the discovery of the object the Guide guides the visitor deeper into the structure. When the Visitor returns from this the `leave` method is called.

It is also possible to add an extra method `bypassing` to the visitor that is called every time the visitor returns to an object it has already discovered before, but which it is not going to leave yet. The new visitor interface has now three methods: `discover`, `bypassing` and `leave` (Figure 21). This `bypassing` method is essential in pretty-printing systems, for example.

```
class TopDownTreeGuide implements TreeGuide
{
   TreeVisitor visitor;

   void guide(Tree.Node node)
   {
      node.acceptVisit(visitor);

      Iterator<Tree.Component> children
            = node.getChildren().iterator();
      while(children.hasNext())
      {
         Tree.Component component = children.next();
         component.acceptGuide(this);
      }
   }

   void guide(Tree.Leaf leaf)
   {
      leaf.acceptVisit(visitor);
   }
}
```

Figure 17: Top-down Guide with total control

Class `WalkTreeGuide` in Figure 19 implements a guide that guides a visitor over a tree in a complete *Structure Walk*. In a structure walk it is extremely easy to keep track of the current depth in the object structure, for example.

As an example of the application of `WalkTreeGuide` consider the pretty-printing of trees. We assume that a `Leaf` has a value of type `Object` and that the method `getValue` returns this value. The tree is in fact flattened by the pretty-printer. A tree with one node and two leafs 1 and 2 will for example be pretty-printed to `(1,2)`. The `TreePrettyPrintVisitor` is shown in Figure 20.

5 Reuse of Guides and Visitors

The biggest advantage of our approach is that the navigation is extracted to a separate class. Therefore navigation and computation can be reused at several levels:

- Traversals for some object structure can be reused for different visitors.
- Traversals can be extended or adapted to create a new traversal over the same object structure.
- Traversals for object structure X can be re used for object structure Y when Y is a specialization of X, where specialization means that the object structure is extended but no new classes are added to the structure.
- Traversals for object structure X can be extended to create a new traversal order for object structure Y when Y is a extension of X. In this case new classes of objects can be added.

```
class Tree
{
   Component getRoot();

   abstract class Component
   {
      Node getParent();
      abstract void acceptVisit(TreeVisitor visitor);
      abstract void acceptGuide(TreeGuide guide);
   }

   class Node extends Component
   {
      List<Component> getChildren();

      void acceptVisit(TreeVisitor visitor)
      {
         visitor.visit(this);
      }

      void acceptGuide(TreeGuide guide)
      {
         guide.guide(this);
      }
   }

   class Leaf extends Component
   {
      void acceptVisit(TreeVisitor visitor)
      {
         visitor.visit(this);
      }

      void acceptGuide(TreeGuide guide)
      {
         guide.guide(this);
      }
   }
}
```

Figure 18: Tree for the second style Visitor-Guide

The first two options are clear. They are possible because of the separation of navigation into a separate class. We have not discussed examples of the last two options of reuse of guides. Also it is not very clear whether this reuse is easy to implement.

First of all we will give an example of a specialisation of `Tree`. When an object structure is specialised it is possible (and often needed) to reuse the existing visitors and guides of the super object structure.

A visitor that counts the number of leafs in `Tree` should still work correctly for specialisations of this `Tree`. Figure 22 shows an extension of `Tree` to `BinarySearchTree`. Because visitors and guides might want to use information

```
class WalkTreeGuide implements TreeGuide
{
   TreeVisitor visitor;

   void guide(Tree.Node node)
   {
      node.acceptDiscover(visitor);

      Iterator<Tree.Component> children
            = node.getChildren().iterator();
      while(children.hasNext())
      {
         Tree.Component component = children.next();
         component.acceptGuide(this);

         if(children.hasNext())
         {
            component.acceptBypassing(visitor)
         }
      }

      node.acceptLeave(visitor);
   }

   void guide(Tree.Leaf leaf)
   {
      node.acceptDiscover(visitor);
      node.acceptLeave(visitor);
   }
}
```

Figure 19: Walk Guide for a Tree

only available in the `BinarySearchTree` components and not just in the `Tree` components we also need to define new interfaces for the guide as well as for the visitor. The `BinarySearchTreeVisitor` must extend the `TreeVisitor` because we want a `TreeGuide` to be able to guide a `BinarySearchTreeVisitor`. Because of the type system the `BinarySearchTreeVisitor` has to define the visit method for the `Tree` components as well as the `BinarySearchTree` components. The visit methods for the Tree will most likely be forwarded to the visit methods of the `BinarySearchTree` components.

Also the components of the new `Tree` must have `accept` methods for all visitors and guides. Because the `accept` methods for the `TreeVisitors` and `TreeGuides` are already defined in the `Tree` structure they do not have to be defined again. However it is quite possible that we want a `BinarySearchTreeGuide` to guide a `TreeVisitor`. In this case the guide has to work with both visitor interfaces.

This kind of reuse is possible, but it requires some work because we want a `TreeGuide` to guide a `BinarySearchTreeVisitor` and a `BinarySearchTreeGuide` to guide a `TreeVisitor`. The strong object-oriented type system of Java is just not capable of handling this in a more easy way.

```
class TreePrettyPrintVisitor implements TreeVisitor
{
   StringBuffer result;

   void discover(Tree.Node node)
   {
      result.append('(');
   }

   void bypassing(Tree.Node node)
   {
      result.append(',');
   }

   void leave(Tree.Node node)
   {
      result.append(')');
   }

   void discover(Tree.Leaf leaf)
   {
      result.append(leaf.getValue().toString());
   }

   void bypassing(Tree.Leaf leaf){}
   void leave(Tree.Leaf leaf){}
}
```

Figure 20: Tree pretty-printer

```
interface TreeVisitor
{
   void discover( Tree.Node node);
   void bypassing(Tree.Node node);
   void leave(    Tree.Node node);

   void discover( Tree.Leaf leaf);
   void bypassing(Tree.Leaf leaf);
   void leave(    Tree.Leaf leaf);
}
```

Figure 21: Extended interface of a Visitor for a Tree

We can conclude that reuse of guides and visitor over object structures is not easy enough in this way. But we have found a solution to this problem. In fact all problems are caused by the `accept` methods in the object structure. Because of them we need to define complete interfaces of visitors and guides.

We have implemented the double-dispatching function of the `accept` methods in a visitor: the `SelfDispatchingVisitor`. This visitor uses a part of the implementation of the Walkabout [9] (which will be considered later). We will only show the implementation for the commanding guide, where the guide has complete control over the navigation. The `SelfDispatchingVisitor`

```
class BinarySearchTree extends Tree
{
   Tree.Component getRoot()
   Component getBinarySearchTreeRoot()

   abstract class Component extends Tree.Component
   {
      int getKey()
      Node getComponentParent()

      abstract void acceptVisit(BSTreeVisitor v);
      abstract void acceptGuide(BSTreeGuide g);
   }

   class Node extends Component extends Tree.Node
   {
      Component getLeftChild();
      Component getRightChild();
   }

   class Leaf extends Component implements Tree.Leaf
   {}
}
```

Figure 22: BinarySearchTree as an extension of Tree

```
interface BinarySearchTreeGuide
{
   void guide(BinarySearchTree.Node node);
   void guide(BinarySearchTree.Leaf leaf);
}
```

Figure 23: Guide of a BinarySearchTree

```
interface BinarySearchTreeVisitor extends TreeVisitor
{
   void visit(BinarySearchTree.Node node);
   void visit(BinarySearchTree.Leaf leaf);
}
```

Figure 24: Visitor of a BinarySearchTree

has the task of dispatching the object that the visitor (an extension of the `SelfDispatchingVisitor`), must visit currently to the correct `visit` method. The interface for a `SelfDispatchingVisitor` is shown in Figure 25. The interface for a `SelfDispatchingGuide` is shown in Figure 26.

The implementation of the `SelfDispatchingVisitor` can be chosen but all implementation will use reflection to dispatch the object to the correct `visit` method. Reflection enables method lookup and invocation of these methods at run-time. To lookup a method you must specify what parameters the method must have and what the name of the method is. Note that the reflection system of Java does not search for a (the best) method that *can* be invoked with the

```
interface SelfDispatchingVisitor
{
   void dispatch(Object object);
}
```

Figure 25: Visitor that is responsible for dispatching

```
interface SelfDispatchingGuide extends Guide
{
   void dispatch(Object object);
}
```

Figure 26: Guide responsible for dispatching

```
class ClassicSelfDispatchingVisitor
               implements SelfDispatchingVisitor
{
   void dispatch(Object object)
   {
      if(object != null)
      {
         try
         {
            Method m = MethodTools.getMethod(
               getClass(), "visit", object.getClass());
            method.invoke(this, new Object[]{object});
         }
         catch(NoSuchMethodException exc)
         {}
      }
   }
}
```

Figure 27: The classic implementation of a SelfDispatchingVisitor

specified parameters, but just looks for the exact method as specified by the parameter types. The simple method lookup mechanism that is used in the classic Walkabout just looks for a `visit` method with exact the same parameter type as the type of the object that must be dispatched. This is, however, not very useful because it disables the use of polymorphism in visitors. Therefore we have implemented a more advanced method lookup strategy, which also searches for methods with the implemented interfaces and superclasses of the object as the parameter type. The `ClassicSelfDispatchingVisitor` is the first try to implement a `SelfDispatchingVisitor`. The method `MethodTools.getMethod` takes care of the new method lookup mechanism but the implementation is not very interesting. It is available for download as part of the complete framework.

The guide can now also use a self-dispatching mechanism. The implementation is exactly the same as the `ClassicSelfDispatchingVisitor` but in this case we need to call a `guide` method.

Now we can implement visitors and guides as extensions of the `SelfDispatchingVisitor` or `SelfDispatchingGuide` and only implement the `guide` and `visit` methods we need. For other classes the guide

```
class TopDownTreeGuide
      extends ClassicSelfDispatchingGuide
      implements TreeGuide
{
   SelfDispatchingVisitor visitor;

   TopDownTreeGuide(SelfDispatchingVisitor visitor)
   {
      super();
      this.visitor = visitor;
   }

   void guide(Tree.Node node)
   {
      visitor.dispatch(node);

      Iterator<Tree.Component> children
            = node.getChildren().iterator();
      while(children.hasNext())
      {
         Tree.Component component = children.next();
         dispatch(component);
      }
   }

   void guide(Tree.Leaf leaf)
   {
      visitor.dispatch(leaf);
   }
}
```

Figure 28: Reusable TopDownTreeGuide

(or visitor) will do nothing. The parameters can be chosen because the `SelfDispatchingVisitor` will take care of the correct call to a `visit` method.

An example implementation of guide that is an extension of the `ClassicSelfDispatchingGuide` is the `TopDownTreeGuide` in Figure 28. Note that all calls to the various accept methods from Figure 17 are now replaced by calls to the dispatch methods of the visitor and the guide.

To illustrate this solution we have reimplemented leaf counting for Tree again. The implementation is shown in Figure 29. The self-dispatching mechanism makes sure that for every Leaf that is encountered on the traversal the method `visit(Tee.Leaf)` is called. Note that no implementation of `visit` for `Tree.Node` is needed. Also note that the result of the visitor depends on the chosen guide. This implementation can therefore even be considered to be generic in a certain way.

To conclude, with some help of the self-dispatching visitors and guides it is possible to reuse a guide or visitor on every object structure X that is a specialization of Y.

We have not yet considered our fourth point of reuse: what if object structure X extends object-structure Y? In this case the navigation scheme might have changed because new classes of objects are added to the structure. The

```
class CountLeafsVisitor
      extends ClassicSelfDispatchingVisitor
{
   int result = 0;

   void visit(Tree.Leaf leaf)
   {
      result = result + 1;
   }
}
```

Figure 29: Reusable CountLeafsVisitor

advantage of the self-dispatching of guides (and visitors) is that they will always make sure that the most specific method will be called. Thus, the existing guides can just be extended to cope with the new types in the structure.

The original designers of the Walkabout already stated that the Walkabout has a serious performance problem. This performance problem arises also in the fragment of the Walkabout we have used in our `SelfDispatchingVisitor`. We have however found a complete solution to this problem, which will be addressed in the next section. Therefore we have called the current implementation of the `SelfDispatchingVisitor` *classic*.

6 Generic Guides: Walkabout

The Walkabout of Palsberg and Jay [9] provides a mechanism for navigating over an object structure without specifying the navigation. The Walkabout examines the object structure at run-time and determines the places where the visitor should go. The Walkabout uses reflection to discover and visit the fields of a class.

When looking at the Walkabout through our Guide glasses, it becomes apparent that, in fact, the Walkabout can be seen as a combination of a `SelfDispatchingVisitor` and a (Walkabout)Guide. Before we introduce this implementation and show the dramatic performance gain we have been able to reach because of the new, clear design, we will first review the original version of the Walkabout proposed by Palsberg et al.

6.1 Classic Walkabout

The Walkabout described by Palsberg and Jay [9] is implemented as a visitor. The pseudo code of a slightly modified version of this Walkabout is shown in Figure 30.

The Walkabout does in fact do two things. First, it tries to dispatch the object that the visitor (an extension of the Walkabout), is currently visiting to a `visit` method. We have already seen this before in the `SelfDispatching-Visitor`. Second, the Walkabout examines the class of the object to find out what kind of fields it has. The Walkabout then recursively calls itself to visit those fields.

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

         if(object is not of primitive type)
         {
            foreach(field f of object)
            {
               this.visit(object.f);
            }
         }
      }
   }
}
```

Figure 30: Pseudo code of the classic Walkabout

Note that also in the Walkabout there is no need for `accept` methods in the object structure, because the Walkabout dispatches the method call to the visitor on its own with a primitive method lookup algorithm.

The real code of the classic Walkabout is shown in Figure 31. We have changed the Walkabout in three points. First, in the original Walkabout all navigation was originally done in the catch of the `NoSuchMethodException`. If there was a defined `visit` it should call the navigation. Second, we use the method `isPrimitive` instead of verifying a set of classes. Third, the original Walkabout visited static fields. In most applications this is however not useful. Note that this classic Walkabout provides a top-down traversal.

This Walkabout might look useful, the performance is however poor. For large structures the Walkabout will spend much more time than an ordinary visitor with implicit navigation control (in whatever place). We will discuss the performance issues later when we have introduced our version of the Walkabout.

6.2 Walkabout as a Guide

With the notion of guides in mind it is clear how to improve the Walkabout implementation. The classic Walkabout does two things in one method: dispatch the call to a visit method and control the navigation by visiting the fields of an object. The second task should be left to a guide! The guide paradigm would suggest splitting the Walkabout in two classes:

SelfDispatchingVisitor does the dispatching of a method call to the visitor. This self-dispatching mechanism can now be reused for other tasks and it can be changed independent of the Walkabout.

```
class Walkabout
{
   void visit(Object object)
   {
      if(object != null)
      {
         Object[] parameters = {object};
         Class objectClass = object.getClass();
         Class[] parameterClasses = {objectClass};

         try // to dispatch
         {
            Method method =
               getClass().getMethod("visit",
                                    parameterClasses);

            method.invoke(this, parameters);
         }
         catch(NoSuchMethodException exc)
         {}
         catch(Exception exc)
         { exc.printStackTrace(); }


         Field[] fields = objectClass.getFields();
         for(int i=0; i<fields.length; i++)
         {
            Class type = fields[i].getType();
            int modifiers = fields[i].getModifiers();

            if (!type.isPrimitive() &&
                !Modifier.isStatic(modifiers)))
            {
               try
               {
                  this.visit(fields[i].get(object));
               }
               catch(IllegalAccessException exc2)
               { exc2.printStackTrace(); }
            }
         }
      }
   }
}
```

Figure 31: Classic Walkabout implementation

WalkaboutGuide is in charge of the navigation control. It guides the visitor to new places in the object structure by examining the fields of an object.

We can however do a better job, still. The `WalkaboutGuide` will have one method `guide` with one parameter, i.e., the object that the visitor must visit. If we implement the `WalkaboutGuide` in this way there is not a good possibility to change to navigation for special cases in the object structure. It would be useful when we could specialize the WalkaboutGuide for these cases by extending the `WalkaboutGuide` and add specialized `guide` methods. Because we have already seen the `SelfDispatchingGuide` in the last section it is an obvious but perfect solution to use this class and make the `WalkaboutGuide` an extension of this guide.

But we can make the WalkaboutGuide even more attractive. The classic Walkabout provided a generic top-down traversal. Why not also provide a bottom-up, top-down spine etc traversal? But first of all we will implement the `TopDownWalkaboutGuide`.

The code of the `SelfDispatchingVisitor` is already shown in Figure 27. It is based on the classic `Walkabout` but it provides a more advanced method lookup. The method lookup of the classic `Walkabout` just looks for a visit method with as the parameter type exactly the class of the object at the current location. Superclasses and implemented interfaces are not used for method lookup and this is not done by Java reflection either. Therefore we have implemented a simple method lookup mechanism which also searches for methods with the implemented interfaces and superclasses of the object as the parameter type. type the implemented interfaces and superclasses of the object. This enables the use of *polymorphism* in the Walkabout. This method lookup is done in a method in a separate tool collection class: `MethodTools.getMethod(Class methodClass, String name, Class parameter)`. The `ClassicSelfDispatchingGuide` is very similar to the `ClassicSelfDispatchingVisitor`. The classic and the new method lookup mechanism both have a serious performance problem but we do not try to improve this yet.

The `ClassicWalkaboutGuide` (Figure 32) is an extension of the `ClassicSelfDispatchingGuide`. It has one `guide` method with an Object as a parameter. This method examines the fields of the current location and guides the visitor to these fields. This first try is also an artless version based on the classic `Walkabout` because this guide also has a performance problem.

The current implementation has a few advantages over the classic `Walkabout`:

1. Visitors can use polymorphism because of the extended method lookup mechanism. In this way it can for example have a `visit` method for superclass or interface of a collection of objects in the object structure. It does not have to define `visit` methods for all kinds of these objects.

2. Navigation can be specialized by extending the `ClassicWalkaboutGuide`.

3. Specialized `WalkaboutGuide`s can also use polymorphism.

4. The separation of concerns into the `WalkaboutGuide` and `SelfDispatchingVisitor` is clearer.

5. The method lookup mechanism in the `SelfDispatchingVisitor` can be changed independent of the `WalkaboutGuide`.

6. Other traversals such as bottom-up are easier to implement.

```
class ClassicWalkaboutGuide
             extends ClassicSelfDispatchingGuide
{
   SelfDispatchingVisitor visitor;

   void guide(Object currentLocation)
   {
      visitor.dispatch(currentLocation);

      Field[] fields = object.getClass().getFields();
      for(int i=0; i<fields.length; i++)
      {
         Class type = fields[i].getType();
         int modifiers = fields[i].getModifiers();

         if (!type.isPrimitive() &&
             !Modifiers.isStatic(modifiers)))
         {
            try
            {
               dispatch(fields[i].get(object));
            }
            catch(IllegalAccessException exc2)
            { exc2.printStackTrace(); }
         }
      }
   }
}
```

Figure 32: Classic implementation of a WalkaboutGuide

6.3 Tuning the Performance

As we have already mentioned several times the Walkabout of Palsberg had a serious performance problem because of the poor performance of reflection in Java. Palsberg reported unacceptable running times over a LinkedList of length 2000. In the current Java version the performance is more acceptable, but still poor compared to a run with a normal visitor. Because our Walkabout is using even more reflection because of the more advanced method lookup and extra method lookup at the guide, which allows us to specialize the navigation, we really need a performance improvement.

A surprisingly simple discovery led to an impressive performance improvement. We found out that looking for a method of a class with reflection is a very time consuming task, but invoking a method with reflection is extremely fast and even comparable to a normal invocation. With this interesting fact in mind, it is easy to see where the bottleneck in the Walkabout of Palsberg is. This Walkabout searches for a method every time it is visiting an object. Object structures however tend to be composed of just a few classes. What will happen when we cache the methods in a hashtable and thus search for a method of a certain class only once and use the cached method when we encounter an object of a certain class again? If the performance of this approach is better then this would enable more advanced method lookup mechanisms like ours.

```
interface StructureDiscoveries
{
   Iterator getFields(Class theClass);
   Method   getMethod(Class methodClass,
                      String methodName,
                      Class parameter);
}
```

Figure 33: StructureDiscoveries interface

The new `WalkaboutGuide` uses a `StructureDiscoveries` object to enable the caching of the methods and fields. The StructureDiscoveries interface (Figure 33) forces an implementation to have two methods: `getFields(Class)` and `getMethod`: `getMethod(Class, String, Class)`, which is used for method lookup in the `DefaultSelfDispatchingVisitor` and the `DefaultSelfDispatchingGuide`.

The artless implementation would just forward these calls to `MethodTools` or return the fields of a class. But the interface allows an implementation to do a better job. The methods and fields can be cached in Hashtables. The sourcecode of the DefaultStructureDiscoveries is not very interesting further and is therefore not shown here. The new implementations of the `WalkaboutGuide` and the `DefaultSelfDispatchingGuide` are shown in the Figures 34 and 35. We have omitted the `DefaultSelfDispatchingVisitor` because it is exactly the same as the `DefaultSelfDispatchingGuide` except for the method name and the absence of the visitor reference. Note that the `StructureDiscoveries` can be be even reused for second runs or can be shared between visitors and guides.

6.4 Performance Results

To measure the performance gain we have been able to reach with the new implementation of the Walkabout we use the same benchmark program as Palsberg et al. They have used the running example of LinkedList with three kind of the components: `Node`, `Link` and `Cons`. A `Cons` has an int value (`head`) and a `tail` which is a `List`. All components implement the interface `List`. A link has a Boolean value called `color`. We want to compute the sum of all the integer components. The code of the benchmark program is shown in Figure 36.

The benchmark has been run on a randomly generated list of length 2000. We have tried several implementations:

1. Monolithic recursive method
2. Many methods in the object structure
3. Traditional Visitor where the object structure controls the navigation
4. Traditional Visitor where the Visitor controls the navigation
5. Visitor with a consulting Guide and thus the Visitor still calls the Guide
6. Visitor with a commanding Guide
7. Our new implementation of the Classic Walkabout

```
class DefaultSelfDispatchingGuide
                     implements SelfDispatchingGuide
{
   StructureDiscoveries structureDiscoveries;

   public void dispatch(Object currentLocation)
   {
      Method method = structureDiscoveries.getMethod(
                           getClass(),
                           "guide",
                           currentLocation.getClass());
      if(method != null)
      {
         Object[] args = {currentLocation};

         try
         {
            method.invoke(this, args);
         }
         catch(Exception exc)
         { exc.printStackTrace(); }
      }
   }
}
```

Figure 34: DefaultSelfDispatchingGuide

8. WalkaboutGuide with caching of method and field lookups and reuse of the `StructureDiscoveries` between the runs.

9. WalkaboutGuide with all the features of the implementation before, but without the SelfDispatchingGuide. This disables specialization of navigation for special cases.

All implementations used the same list. The time needed to create the list is not included in the run-time. It is also interesting to see how deep an object structure is allowed to be in all implementations. When the depth exceeds this limit and `StackOverflowError` will be thrown. We have also tried this out for all implementations.

All benchmarks have been executed on an Intel Pentium II 300-MHz with 192-Mbyte of memory. The code is compiled and executed with Sun's JDK 1.3.0 and the Java Hotspot Client VM in mixed mode.

The benchmark results are shown in Table 1. The run-times of implementations with a smaller maximum depth are the run-times on a list of this depth and *not* on a list of depth 2000. A few things are clear immediately. First of all it appears that the use of Guides (without reflection) does not have a serious impact on the run-time because the run-time just increases from 40 to 90 milliseconds.

The implementation where the visitor calls the guide has, however, a serious impact on the allowed depth of the object structure. This is because the `visit` method calls the guide. Because of this all methods stay on the `Stack` resulting in a `StackOverFlowError` because the application recurses to deep.

```
class WalkaboutGuide
      extends DefaultSelfDispatchingGuide
{
   SelfDispatchingVisitor visitor;
   StructureDiscoveries structureDiscoveries;

   void guide(Object currentLocation)
   {
      Iterator fields = structureDiscovery.getFields(
                     currentLocation.getClass());

      while(fields.hasNext())
      {
         Field f = (Field)fields.next();

         try
         {
            visitor.dispatch(f.get(currentLocation));
         }
         catch(IllegalAccessException exc)
         { exc.printStackTrace(); }
      }
   }
}
```

Figure 35: WalkaboutGuide

```
interface List {}

class Link implements List
{  boolean color;
   List list;
}
class Cons implements List
{  int head;
   List tail;
}
class Nil implements List {}
```

Figure 36: Class structure used for the benchmarks

The traditional visitor with the navigating visit also has this problem.

It is also clear that the classic Walkabout is extremely slow. The allowed depth is however quite large. This is because there are no `accept` methods and the navigation is done from the end of the `visit(Object)` method.

The completely self-dispatching system with method lookup caching performs very well. The run-time has changed from minutes to seconds, which is however still quite slow compared to a run without reflection. The new problem with the WalkaboutGuide is sadly the allowed depth. When the Guide uses a self-dispatching mechanism to allow the specialization of navigation the allowed depth has dropped to 300. This is caused by the method invocations with reflection. The WalkaboutGuide calls from the `guide(Object)` its own

Table 1: Benchmark results of several implementations of a sum computation

Implementation	*Run-time*	*Max. depth*
Monolithic recursive method	180 ms	8200
Many methods	30 ms	8200
Traditional Visitor	40 ms	8300
Visitor with navigating visit	60 ms	3100
Visitor with consulting Guide	90 ms	1960
Visitor with commanding Guide	90 ms	3000
Classic Walkabout	3 min 11 sec 845 ms	> 5000
WalkaboutGuide with dispatch	2 sec 123 ms	300
WalkaboutGuide without dispatch	5 sec 779 ms	4200

`dispatch(Object)` method. But the `dispatch` method will also call the `guide` method. Because of this the complete order of `guide` methods stays on the `Stack`. In normal cases this should not be a problem (look for example at the allowed depth of 'many methods' implementation) but it looks like that the invocation of a method with reflection causes quite a lot of method calls to be put on the Stack. When we do not use self-dispatching in the Guide the allowed depth increases to 4200. The self-dispatching of the Visitor has no influence because these calls return before further navigation is done.

7 Concluding Remarks

The extraction of navigation into a separate class, called a Guide, enables the reuse of navigation and provides flexibility to use whatever traversal order is necessary. Many different traversals can be captured. We have shown some interesting guides like the top-down, bottom-up, top-down spine and the breadth-first guides. This collection already is evidence of the flexibility of navigation with guides. We have combined the guide approach with the existing Hierarchical Visitor Pattern. The resulting structure walk might be useful in many applications because it provides a clear environment to perform operations at specific locations in the structure walk.

The guide pattern arose in an implementation of the Extract Method refactoring in an experimental extension of the refactoring tool of Chris Sequin [11]. That application gave rise to many different traversal schemes for operations such as collecting local variabels from the extracted code and collecting all caught or thrown exceptions of the extracted code.

To make the reuse of navigation on other object structures possible we have introduced a high-performance and easy to use self-dispatching mechanism with advanced method lookup. This provides applications of the Guide pattern a very easy environment to extend and reuse existing traversal orders on new object structures.

Our generalization of the Walkabout makes it possible to define a wide range of completely generic traversals. Possible applications of these generic traversals include language processing operations on languages with a complex abstract syntax, and navigating third party object structures without having to change or extend any of the imported classes.

7.1 Related and Future Work

We have presented commanding as well as consulting guides. Further research and application will learn us which style is best usable in practice or maybe the two styles can live together in different applications.

Guides are similar to the iterator pattern where a method will ask the Iterator what the next item is. However in the Guide pattern the result will be dispatched to one of the methods capable of handling the result. This enables the use of more than one type in the structure.

JJForester [7] tries to offer different traversals by providing different accept methods in the object structure. The Guides can be applied in this parser and visitor generator tool in a perfect way.

Because of the complete seperation of navigation from the visitor, our Guides can be used perfectly with the visitor combinators of Joost Visser [15]. The visitor combinators compose new visitors from other visitors. Several visitor combinators are generic and therefore the combination with our new Walkabout is interesting. The visitor combinators also provide traversal control, which maybe should be translated to guides in our framework. Together with the visitor combinators, we see an upcoming area of research of object-oriented creation of visitors (computations or transformations) and guides (navigation) in terms of other visitors or guides. An interesting question is what should be a guide combinator and what should be a visitor combinator. The application of two visitors after each other is for example a guide combinator. The application of two visitors in parallel is however a visitor combinator.

Many of the traversal schemes and navigation extraction ideas in this papers have been inspired by the traversals in the Stratego Library [12, 13]. Stratego [14, 16] is a language for program transformation based on the paradigm of rewriting strategies. Stratego has a small set of primitive traversal operators, which can be combined into a wide range of full or partial generic tree traversals. For example, a generic top-down traversal applying the transformation `s` on all nodes of an abstract syntax tree is defined by `topdown(s) = rec x(s; all(x))`, where `all(x)` defines the application of the strategy `x` to all children of the current node. Future work on guides can maybe provide a similar set of operators that can be composed into guides. In contrast to Stratego, a weak point of object-oriented visitors and guides is that although visitors can be used to *collect* information from object-structures, their support for *transformation* of the object-structure is poor. Improving this aspect is an interesting area for further work.

Visitors and guides are similar to the polymorphic folds in functional programming. A functional fold gets an algebra of functions. The fold applies to each element in a data structure a function from the algebra. The guides have quite a similar behaviour: they apply to each object in the object structure a method ($\sim$ function) in the visitor ($\sim$ algebra). Note that the methods in the visitor do not do any navigation. They are just applied. The navigation is in the guide ($\sim$ fold). Folds however only provide a bottom-up traversal over a tree in which each node in the tree is touched. The results of nodes depend on the results of the children of the nodes.

Our results on the performance of the Walkabout and the complete separation of navigation from computation opens a new area of research because generic visitors can now be implemented. Examples of generic visitors that come to mind are for example the *DepthVisitor*, which keeps track of the depth in an object structure or the *MaxDepthVisitor*, which computes the depth of an object structure. Also it is now possible to implement a generic function to compute the size of an object structure. The current Walkabout implementation is maybe still not the best we can do. Further research on the performance and the maximum depth of the object structures is certainly necessary.

In this paper we have only shown examples on tree object structures. The guide system is however not limited to trees and can be applied in any object structure. We have also considered graphs, where there is the difficulty of cycles. There are two solutions to solve the termination problem that arises in graphs. The first is remembering in the guide where the visitor has already been in the object structure. This enables the use of more then one visitor at a time over an object structure. The second solution is to mark objects as visited. The problem with this solution is that the marks should be removed again from the graph.

References

[1] G. Bracha. *Prototype for JSR014: Adding Generics to the Java Programming Language*. Sun Microsystems. `http://developer.java.sun.com/developer/earlyAccess/adding_generics/`. 2.1

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. *GJ Home*. Pizza Group, Bell Labs, Lucent Technologies. `http://www.cs.bell-labs.com/who/wadler/pizza/gj/`. 2.1

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Adding genericity to the java programming language. In *Proceedings of OOPSLA '98, 13nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1998. 2.1

[4] R. D. Falco. *Hierarchical Visitor Pattern*. Portland pattern repository, 2001. `www.c2.com/cgi/wiki?HierarchicalVisitorPattern`. 4

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., 1994. 1, 2

[6] T. Kuipers and J. Visser. *JJForester Home*. CWI, Department SEN, Amsterdam, The Netherlands, 2000. `www.jjforester.org`. 1, 2.2

[7] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In M. van den Brand and D. Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. Proc. of Workshop on Language Descriptions, Tools and Applications (LDTA). 1, 2.2, 7.1

[8] Metamata and Sun Microsystems. *JavaCC Home*, 2000. `www.suntest.com/JavaCC/`. 1, 2.2

[9] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Proceedings of COMPSAC '98, 22nd Annual International Computer Software and Applications Conference*, pages 9–15, August 1998. 1, 2, 5, 6, 6.1

[10] J. Palsberg and K. Tao. *Java Tree Builder Home.* Purdue University, 1997. `www.cs.purdue.edu/jtb/`. 1, 2.2

[11] C. Seguin. *JRefactory Home.* ACM, 2000. `jrefactory.sourceforge.net/csrefactory.html`. 3, 7

[12] E. Visser. *The Stratego Library.* Institute of Information and Computing Sciences, Utrecht University, 0.5 edition, 1999-2001. Technical Documentation. [html] . 7.1

[13] E. Visser. Language independent traversals for program transformation. In J. Jeuring, editor, *Workshop on Generic Programming (WGP'00)*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Department of Information and Computing Sciences, Universiteit Utrecht. [ps.gz] . 7.1

[14] E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001. [pdf, ps.gz, bib, springer] . 7.1

[15] J. Visser. Visitor combination and traversal control. In *OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications.* To appear, 2001. 7.1

[16] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998. [ps.gz] . 7.1