DESIGN PATTERNS

The Visitor Design Pattern in Depth

Perform one or more operations on a collection of different data types without disrupting existing code.

by Ian Darwin

Suppose it's your first day at a new job at a midsize company. You'll probably be escorted around the building and introduced to every department of the organization. At each one you'll say "Glad to meet you" a few times and talk with the team there to discuss your common projects, and then you'll say "Nice to have met you." And you'll repeat this for each department. Congratulations! You have just implemented the Visitor design pattern in humanware.

The Pattern

Visitor is a useful pattern when you have many objects of different types in your data structure and you want to apply some operation to several or all of them. The pattern is helpful when you don't know ahead of time all the operations you will need; it gives you flexibility to add new operations without having to add them to each object type. The basic idea is that a `Visitor` object is taken around the nodes of a data structure by some kind of iterator, and each node "accepts" the visitor, allowing it access to that node object's internal data. When a new function is needed, only a new visitor needs to be written. The iteration is conceptually simple:

```
for (Node node : collection) {
    node.accept(visitor);
}
```

(There are two main code examples in this article; both can be found in my GitHub repository. Code from other articles in this series on design patterns can be found further up the trunk of that repository.)

The `Node` objects must know how to accept the `Visitor`, and they will usually call a method on the `Visitor` that is appropriate to the type of the node—for example:

```
class TextNode implements Node {
    void accept(Visitor v) {
        v.visitTextNode(this);
    }
    // other state and methods
}
```

Therefore, one consequence of this pattern is that the `Visitor` needs to know about all the node types it might encounter.

Double Dispatch

Many explanations of the Visitor pattern refer to it as *double dispatch*. This term sometimes makes readers think of a two-step dispatching process, as with a pointer to another pointer used in some languages. That's not what is meant. The term refers to the fact that both the type of the visitor and the type of the node (or "receiver") are used in sorting out which method winds up doing the work. You can see this in the `accept()` method above: there's the call to `accept()` and the call back to `visitTextNode()`.

Visiting the Text

Suppose I need to maintain a word processor that was written in Java. There are a few data types (text node, image node, and so on). Common operations, such as editing text, setting fonts, and setting colors, are taken care of. But there are many supplemental operations that need to be performed on the text, and new ones come along often as customers provide feedback. Here's what the text node's class started as:

```
public class TextNodeOld extends Node {
    private StringBuilder text = new StringBuilder(

    public TextNodeOld() {
        // empty
    }

    public TextNodeOld(String s) {
        // Here, you know the StringBuilder exists a
        text.append(s);
    }

    public String getText() {
        return text.toString();
    }

    public void setText(String text) {
        this.text.setLength(0);
        this.text.append(text);
    }

    // Lots of supplemental functionality methods he
    // that will be added below
}
```

It's becoming annoying that all the data types need to be modified every time somebody has an idea for a new function. I know from experience that I'm unlikely to be able to predict, at the start of the maintenance, all the remaining functionality that will be needed. So I'll introduce a Visitor pattern.

The basic data structure is still the `Node`, with subclasses `TextNode` and `ImageNode`. A real word processor would have more types of nodes, but I want to focus on the Visitor pattern, not compete with the well-known word processor that's out there. Therefore, `Node` is now an interface with just one method:

```
public interface Node {
    abstract void accept(Visitor v);
}
```

I was tempted to call this interface `Visitable` instead of `Node`. On one hand, `Visitable` is a more descriptive name for this version. On the other hand, most formal definitions of Visitor use the term `Node`. I know some of you will go to Wikipedia to get a second opinion after reading this, and I don't want to confuse anyone.

`Node` could alternatively be an abstract class, but that would force all the implementation classes to be related by inheritance, which may be an unnecessary restriction.

`Node` uses `Visitor` as a type, so the next step is to define `Visitor`:

```
public abstract class Visitor {
    public abstract void visitTextNode(TextNode text

    public abstract void visitImageNode(ImageNode in

    // And so on for TableNode, SectionNode, VideoN
}
```

Note that you could make all the methods be overloads of a single method called `visit()`, because the argument types are unique, but I think this way is clearer. It's a stylistic choice, so pick one way and try to be consistent.

At any rate, here you meet the one complication of the Visitor pattern: *Visitors need to know how to visit every main kind of node*.

The revised node classes themselves are not that interesting, so I didn't show their code—the `Text` node has a `Text` property; the `Image` node has a `FileName`, a width, a height, and an optional `Caption` (which is subclassed from `TextNode`); and so on.

The iteration doesn't need to be a for loop or even an iterator—any means of traversing all the nodes is fine.

With all that structure in place, it's time to start to write visitors. First, suppose there's a requirement to print a quick draft of the document, without trying to display the images (this capability was in the requirements from the days when graphics printers were expensive). The text stored in a `TextNode` might contain more characters than fit on a line, so I use an existing program called `Fmt` to crudely format lines to fit. `Fmt` wants its input as a stream (even though in this case it's only one string), so the `visitTextNode()` method wraps the current `TextNode` 's string in an array and streams that to the `format()` method of `Fmt`.

```
static Visitor draftPrinterVisitor = new Visitor()
        @Override
        public void visitTextNode(TextNode textNode
            String[] lines = { textNode.getText() }
            Fmt.format(Stream.of(lines), out);
        }

        @Override
        public void visitImageNode(ImageNode imageN
            String caption = imageNode.caption != nu
                imageNode.caption.getText() : "no ca
            System.out.printf("Image: name='%s',
                caption='%s'%n", imageNode.fileName
        }
};
```

The `Fmt` program requires a `PrintWriter` for output, so the code on the following page wraps `System.out` in a `PrintWriter` before passing the `draftPrinterVisitor` around to all the nodes.

The `visitImageNode()` method doesn't need to use `Fmt`, because image captions are assumed to be one line long. The method simply gets the text from the `ImageNode`'s caption (which is a subtype of `TextNode`, so it has a `getText()` method), defaulting to "no caption" if there is no caption, and prints the result to `System.out`.

The main demo program, `WordProcessorDemo`, creates a demo document and iterates over its `Node` instances like this:

```
out = new PrintWriter(System.out);
    for (Node n : nodes) {
        n.accept(draftPrinterVisitor);
    }
out.flush();
```

Note that the iteration doesn’t need to be a for loop or even an iterator—any means of traversing all the nodes is fine.

Suddenly, someone from marketing rushes in and says, “Gee, this draft format is neat. But the boss wants it to show the word count as well. Can you add a function to compute that too?”

“No problem,” you can say, turning back to the code. Soon the new code takes shape. The following `Visitor` counts the number of words in text nodes, and it even descends into `ImageNode`s to get the word count of the caption, if there is one.

```
public class WordCountVisitor extends Visitor {

    int wordCount = 0;

    public int getWordCount() {
        return wordCount;
    }

    @Override
    public void visitTextNode(TextNode textNode) {
        wordCount += wordCount(textNode.getText());
    }

    @Override
    public void visitImageNode(ImageNode imageNode)
        // You might say there's nothing to do, but
        if (imageNode.caption != null) {
            visitTextNode(imageNode.caption);
        }
    }

    /** Simplistic implementation of word counting 
    private int wordCount(String text) {
        // Replace all nonspace chars with nothing;
        // add one because "hello word" has one spac
        // but it is two words.
        return text.trim().
            replaceAll("[^\\s]", "").length() + 1;
    }
}
```

This code is plugged into `main()` in a similar fashion:

```
Visitor wordCountVisitor = new WordCountVisitor();
for (Node n : nodes) {
    n.accept(wordCountVisitor);
}
System.out.printf("The document has about %d words%
((WordCountVisitor) wordCountVisitor).getWordCount(
```

And when the code is run with the sample document, it prints this, which turns out to be the correct answer:

```
The document has about 78 words
```

Revisiting the JDK

Java 8 and later versions include two sets of visitor types: `ElementVisitor` or TypeVisitor, and `FileVisitor`. The `ElementVisitor` types are part of the package `javax.lang.model`, which bills itself as “Classes and hierarchies of packages used to model the Java programming language.” I don’t have room in this article to write my own Java compiler, so I’ll skip the language-modeling types. But I’ll note that the Visitor pattern is often explained in terms of a program language compiler visiting the nodes of an abstract syntax tree (AST), which is the output of the parsing phase.

> The Visitor pattern allows you to retain flexibility to add new methods at a slight cost: the reduction of encapsulation and the need for every visitor to know about all the different node types.

The `FileVisitor` and its solitary implementation class, `SimpleFileVisitor` in `java.nio`, are specialized for processing file hierarchies. They do follow the `visit`… naming pattern. A typical use is to subclass `SimpleFileVisitor`—overriding one or two of its four methods—and pass an instance of it to the `Files.walkFileTree()` method. The nodes you're visiting this time are actual file system nodes (represented by inodes in the UNIX/Linux sense). The `walkFileTree()` method performs the iteration, and it calls your `FileVisitor`'s visitation methods to "do something" at the beginning and end of each directory and for each file in each directory. A simple directory lister, for example, can be made with just the following visitor class (this example is in the `visitor.file` package in the GitHub repository):

```
public class TrivialListerVisitor extends SimpleFil

    @Override
    public FileVisitResult preVisitDirectory(Path di
        BasicFileAttributes attrs) throws IOExcepti
        System.out.println("Start directory " + dir
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path file,
        BasicFileAttributes attrs) throws IOExcepti
        System.out.println(file.getFileName());
        return FileVisitResult.CONTINUE;
    }
}
</path>
```

I also need to invoke the `walkFileTree()` method to do the iteration, using a `Path` object to describe the directory. This code is in the `main()` method of `FileVisitorDemo.java`:

```
// Set the starting path
Path startingPath = Paths.get(".");

// Instantiate the Visitor object
FileVisitor<path> visitor = new TrivialFileVisitor(

// Use the built-in walkFileTree client to
// visit all directory and file nodes
Files.walkFileTree(startingPath, visitor);
</path>
```

This code works, although it's obviously not a replacement for something like the UNIX/Linux/MacOS ls command, which sorts the entries and has a zillion options.

A slightly fancier version might indent one tab stop for each directory level. There's a start at making such a thing in the class `IndentingFileVisitor` in my GitHub repository, although it doesn't work superbly yet. To try it, just change the instantiation of the `FileVisitor` in the `main` method.

Conclusion

Besides the examples of the word processor add-on and directory navigation, are there other uses of the Visitor pattern? Certainly! Examples include its use in compilers (as mentioned earlier), report writing where different people need different reports, and graphics programs—in short, any application in which you need to add functionality across a hierarchy without disrupting (or even changing) the nodes in the hierarchy.

Visitor is like walking around a new company visiting all the teams and having them accept you (the introductions) and give you their impressions of your job (the visit). The Visitor pattern allows you to retain flexibility to add new methods at a slight cost: the reduction of encapsulation and the need for every visitor to know about all the different node types. It's not a one-size-fits-all pattern. It's optimal when the number of functionalities that you (might) have to add is significantly greater than the number of node types in your data structure. If the number of data types (node types) to be added is greater than the functions you'll need to add or you truly know that you won't need to add new functions very often, don't use this pattern, but for the other cases, you'll find Visitor to be an elegant solution.

This article was originally published in the September/October 2018 issue of *Java Magazine*.

Ian Darwin

Ian Darwin (@Ian_Darwin) is a Java Champion who has done all kinds of development, from mainframe applications and desktop publishing applications for UNIX and Windows, to a desktop database application in Java, to healthcare apps in Java for Android. He's the author of *Java Cookbook* and *Android Cookbook* (both from O'Reilly). He has also written a few courses and taught many at Learning Tree International.

Share this Page

Contact
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us
Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials
Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events
Acquisitions
Blogs
Events
Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices