

Applying the Visitor pattern using reflection

Inigo Surguy

Articles

[Java](#)
[Python](#)
[Delphi](#)
[XML and CSS](#)

About me

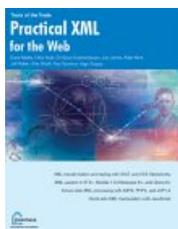
Contact me

About this site

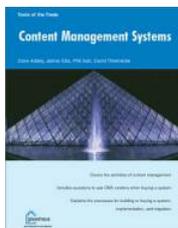
RSS

[Client / server-side](#)
[XSLT](#)

My books



"Love this book – as a seasoned web developer with heaps of experience I find this book a great reference tool, fantastic prompter when I'm struggling with XML technologies, all round very well written and did I mention already PRACTICAL." –
Hamish Fraser –
[amazon.com review](#)



"...tackles crucial technical issues that anyone involved in a CMS must face, but the pitch is accessible to most readers interested in the highly complex, and highly fascinating world of CMS" –
Paola Di Maio – [Content Wire](#)



About the Visitor pattern

The Visitor pattern, as defined in [Design Patterns](#), is great if you're acting on a structure of objects in several different ways, and it makes it easy to change between the actions that you're performing on each, without having to resort to instanceof's or changing the code of the objects you act upon.

A limitation to the standard implementation of the pattern is that it's not suitable when the classes defining the structure itself are frequently changing, rather than the actions to be applied to them. I describe here an implementation of the Visitor that will cope with a changing structure, by using reflection to select the narrow appropriate visit method to call at runtime.

A standard Java Visitor implementation

An example of a standard Java implementation of the Visitor pattern is:

```
class Tester {
    public static void main(String[] args) {
        Cheese cheese1 = new Wensleydale();
        Cheese cheese2 = new Gouda();
        Cheese cheese3 = new Brie();
        Visitor v = new VisitorImpl();
        cheese1.accept(v);
        cheese2.accept(v);
        cheese3.accept(v);
    }
}

interface Visitor {
    void visit(Wensleydale w);
    void visit(Gouda g);
    void visit(Brie b);
}

class VisitorImpl implements Visitor {
    public void visit(Wensleydale w) { System.out.println(w.wensleydaleName()); }
    public void visit(Gouda g) { System.out.println(g.goudaName()); }
    public void visit(Brie b) { System.out.println(b.brieName()); }
}

interface Cheese { void accept(Visitor v); }
abstract class BaseCheese implements Cheese { }
class Wensleydale extends BaseCheese {
    String wensleydaleName() { return "This is wensleydale"; }
    public void accept(Visitor v) { v.visit(this); }
}
class Gouda extends BaseCheese {
    String goudaName() { return "This is gouda"; }
    public void accept(Visitor v) { v.visit(this); }
}
class Brie extends BaseCheese {
    String brieName() { return "This is brie"; }
    public void accept(Visitor v) { v.visit(this); }
}
```

It's very easy to add another Visitor, to act on your structure of objects in a different way, but it's hard to add another type of cheese; you have to go through all of your existing Visitors adding a "visit(Gorgonzola g)" method.

So, the standard Visitor is very useful when you want to perform a number of different operations on one state of objects, but if the types of objects that you want to act on is changing, it's not so useful.

Overcoming the problems of the standard Visitor using reflection

But, you can solve this by using reflection, so the method to use is determined at runtime rather than at compile-time. This way, you can define your visitor to accept the broadest type, rather than all the narrow types and you can move your "accept" code into a superclass, rather than having it repeated in all of your implementations. New classes that you add will be handled by the default case in the Visitor, or by a case appropriate to the interface that they implement.

Here's an example:

```
import java.lang.reflect.*;

class Tester {
    public static void main(String[] args) throws Exception {
        Cheese cheese1 = new Wensleydale();
        Cheese cheese2 = new Gouda();
        Cheese cheese3 = new Brie();
        Cheese cheese4 = new Gorgonzola();
        Cheese cheese5 = new SomeOtherCheese();

        Visitor v = new VisitorImpl();
        cheese1.accept(v);
        cheese2.accept(v);
    }
}
```

```

        cheese3.accept(v);
        cheese4.accept(v);
        cheese5.accept(v);
    }
}
interface Visitor { void visit(Cheese c) throws Exception; }
class VisitorImpl implements Visitor {
    private Method getPolymorphicMethod(Cheese cheese) throws Exception {
        Class cl = cheese.getClass(); // the bottom-most class
        // Check through superclasses for matching method
        while(!cl.equals(Object.class)) {
            try {
                return this.getClass().getDeclaredMethod("visit", new Class[] { cl });
            } catch(NoSuchMethodException ex) {
                cl = cl.getSuperclass();
            }
        }
        // Check through interfaces for matching method
        Class[] interfaces = cheese.getClass().getInterfaces();
        for (int i=0; i<interfaces.length; i++) {
            try {
                return this.getClass().getDeclaredMethod("visit", new Class[] { interfaces[i] });
            } catch(NoSuchMethodException ex) {
            }
        }
        return null;
    }
}

public void visit(Cheese c) throws Exception {
    Method downPolymorphic = getPolymorphicMethod(c);
    if (downPolymorphic == null) {
        defaultVisit(c);
    } else {
        downPolymorphic.invoke(this, new Object[] {c});
    }
}

void defaultVisit(Cheese c) { System.out.println("A cheese"); }
void visit(Wensleydale w) { System.out.println(w.wensleydaleName()); }
void visit(Gouda g) { System.out.println(g.goudaName()); }
void visit(Brie b) { System.out.println(b.brieName()); }
void visit(AnotherCheese a) { System.out.println(a.otherCheeseName()); }
}

interface Cheese { void accept(Visitor v) throws Exception; }
interface AnotherCheese extends Cheese { String otherCheeseName(); }
abstract class BaseCheese implements Cheese {
    public void accept(Visitor v) throws Exception { v.visit(this); }
}
class Wensleydale extends BaseCheese { String wensleydaleName() { return "This is wensleydale"; } }
class Gouda extends BaseCheese { String goudaName() { return "This is gouda"; } }
class Brie extends BaseCheese { String brieName() { return "This is brie"; } }
class SomeOtherCheese extends BaseCheese implements AnotherCheese {
    public String otherCheeseName() { return "Different cheese "; }
}
class Gorgonzola extends BaseCheese { String gorgonzolaName() { return "This is gorgonzola"; } }
}

```

[Download this code](#)

When to use this implementation

Although this code is more flexible, it's not suitable for all situations. The downsides are:

- Reflection is slow – so you're trading flexibility for speed.
- The code's more complicated; but the long `getPolymorphicMethod` can be implemented only once in an abstract Visitor superclass, from which actual Visitor implementations are subclassed.

Note that in this sample code I'm throwing lots of exceptions; in practice you'd probably want to handle some of them.

References

I got the idea of using reflection to find the narrowest method to call from Dr. Heinz Kabutz's (heinz@javaspecialists.co.za) excellent [Java Specialists Newsletter](#) (see Issue 9, on Depth-first polymorphism). He's since written a newsletter called [Visiting your Collection's Elements](#) based on the ideas in this article. I recommend JavaSpecialists to anyone interested in advanced Java topics.

The Visitor design pattern is described in the "Gang of Four" [Design Patterns](#) book. There's information on implementing it in Java in [Bruce Eckel's Thinking in Patterns book](#).

Return to index

Return to the [index page](#).