Pattern Hatching

VISITOR in Frameworks

John Vlissides
C++ Report, November/December 1999

© 1999 by John Vlissides. All rights reserved.

> *I really like the design story until you get to the visitor—I'd just never use VISITOR for this in a real design. The VISITOR dependency limitation is just too drastic. I think it is a nice Gedankenexperiment, but I think it is almost "unethical" to suggest this as a solution in an article.... By the way, in my recent pattern talk I list my bottom-ten patterns, and VISITOR is at the very bottom.*

A grim assessment from none other than Erich Gamma. What's got his goat is the VISITOR pattern's role in TOOLED COMPOSITE, a compound design pattern I described last time.[1] TOOLED COMPOSITE supports direct-manipulation in applications as varied as music editors, schematic capture systems, user interface builders, and project management systems. It lets you define graphical objects (*shapes*) that users manipulate through an extensible set of *tools*, potentially with unique manipulation behavior for every shape-tool combination.

The "design story" that Erich mentions centers on a drawing editor I used to illustrate TOOLED COMPOSITE. I wanted a simple example of the pattern, and you might say a basic drawing editor is the "hello world" of direct manipulation. That's because it manipulates the most primitive of graphical objects—simple geometric shapes like lines, circles, and polygons, plus text—in the most prosaic ways: creating them, moving them, deforming them, and not much else. Applications like music editors and schematic capture systems define more specialized shapes and tools.

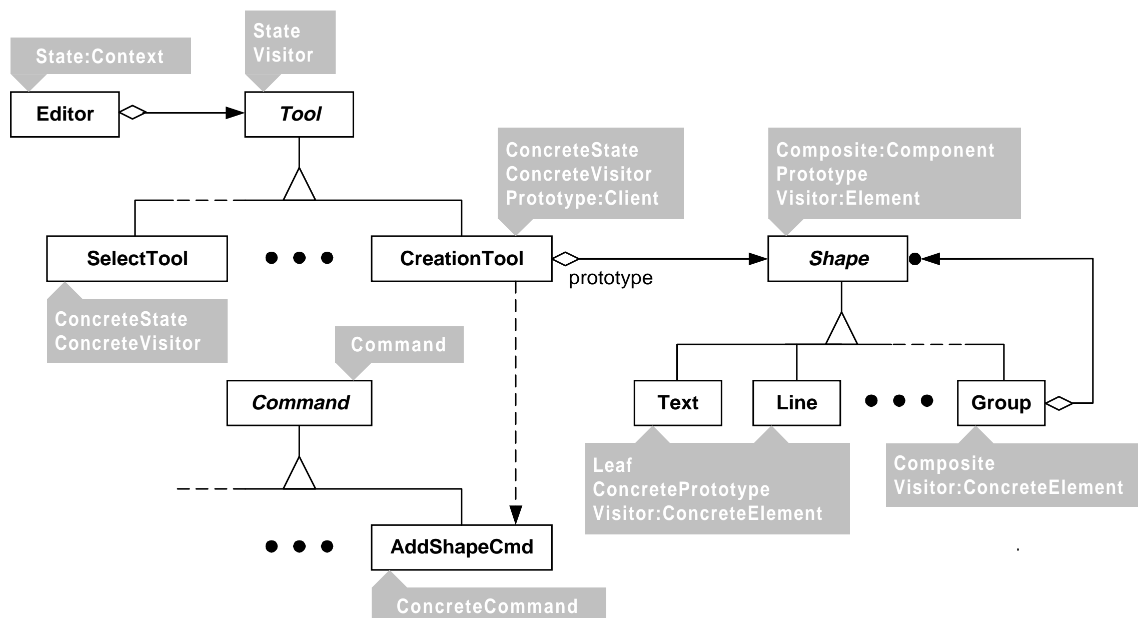

Figure 1: Summary of TOOLED COMPOSITE structure in drawing application

Figure 1 shows the structure of the drawing editor classes that arose from TOOL COMPOSITE and its constituent patterns:

- The COMPOSITE pattern furnished the recursive shape structure that users manipulate with tools;

- STATE keeps track of the current tool;
- PROTOTYPE lets one creation tool class create any kind of shape;
- COMMAND specifies the potentially undoable effects of wielding a tool; and
- VISITOR implements how tools affect shapes during direct manipulation.

Erich's beef

The bugaboo in this example is the application of VISITOR. TOOLED COMPOSITE uses VISITOR to address the "*m* x *n* problem"—*m* tools and *n* shapes may produce upwards of *m* x *n* different direct manipulation behaviors. If you're averse to defining a similar number of classes to house these behaviors (I am), then you've got this problem. Observe that the manipulation code to execute depends on two types: the type of tool, and the type of shape. This is the classic double-dispatch problem, and VISITOR is tailor-made to it.

The troubles start when we look at the C++ code that VISITOR yields. There's nothing extraordinary about the code except for a couple of application-specific tweaks: I renamed "`visit`" operations "`manipulate`," and I made them return a command that implements the undoable result of the manipulation. I also added a catch-all operation[2] to allow for extension should one define new `Shape` subclasses. All rather innocuous, it seemed to me.

But Erich thought otherwise. When an expert framework designer sees code like

```
class Tool {
    // ...

    virtual Command* manipulate (Shape*)   // catch-all
        { return 0; }

    virtual Command* manipulate (Text* t)
        { return manipulate((Shape*) t); }

    virtual Command* manipulate (Line* l)
        { return manipulate((Shape*) l); }

    // etc. for remaining Shape subclasses

};
```

an alarm goes off. Deep in said expert's brain is the sense that a framework interface shouldn't refer to concrete subclasses. He or she may not know it, but such references violate "The Dependency Inversion Principle" (DIP),[3] one of many useful aphorisms from our own Bob Martin. DIP states:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

DIP may sound like didactic hooey, but there's excellent basis for it, particularly in frameworks. If anything in a framework should be rock-stable, it's the interfaces it defines. Yet an interface that includes names of concrete classes will almost certainly need modification when someone defines new subclasses. And if there's one thing you do with framework interfaces, it's implement them in new subclasses.

A framework that violates DIP makes its designer look dippy indeed. So imagine the scorn to be heaped on the pattern writer whose handiwork incites such folly.

Erich's assessment seems pretty charitable now, *n'est-ce pas*?

"This ain't no steenking framework!"

In the interest of saving face, I could claim that my drawing editor—and by implication, *any* use of TOOLED COMPOSITE—isn't meant as a framework; it's just a one-off application. That may be true, but it's cold

comfort considering the severe limits it would impute to the pattern's applicability. Besides, the implementation would still violate DIP. If I were really desperate, I'd blame the problem on strong type checking and sing the praises of Smalltalk, where the problem goes away. You're reading *C++ Report*, however, so I don't think that's an option, desperate or no.

But that's all hypothetical, of course, because there's good justification for applying VISITOR: How else can we address the $m \times n$ behavior problem without proliferating classes? At the end of the day, double dispatch offers a solution that's hard to beat, at least in principle. The real issue lies in how we *implement* double dispatch (and thus VISITOR) in a single-dispatch language like C++.

The implementation described in *Design Patterns*[4] works fine if you never extend the Element hierarchy, which corresponds to the `Shape` hierarchy here. But that's hardly realistic even in one-off applications, let alone frameworks. That's the age-old problem with VISITOR: its brittleness in the face of a changing Element hierarchy.

Catch-all to the rescue—sort of

In *Pattern Hatching* I introduce the catch-all operation. I explain how it gives you a place to put code for new Element subclasses as they arise, without changing the Visitor interface.

Let's say I want to define three new `Shape` subclasses: `Star`, `Moon`, and `Heart`. I need to implement special manipulation behavior for each of them, but I don't want to change the Tool interface. Instead, I change the catch-all for any `Tool` subclasses that should manipulate these new subclasses specially. A good example of such a class is `CreationTool`, which if you recall is the tool for creating shapes. Here's its new catch-all implementation:

```
Command* CreationTool::manipulate (Shape* s) {
    Shape* newShape = 0;

    if (Star* star = dynamic_cast<Star*>(s)) {
        // code for creating a star by direct manipulation
        // and assigning it to newShape

    } else if (Moon* moon = dynamic_cast<Moon*>(s)) {
        // code for creating a moon by direct manipulation
        // and assigning it to newShape

    } else if (Heart* heart = dynamic_cast<Heart*>(s)) {
        // code for creating a heart by direct manipulation
        // and assigning it to newShape

    }

    return new AddShapeCmd(newShape);
}
```

Note that I've changed existing code here, namely the `CreationTool` class. Presumably its `manipulate(Shape*)` function didn't even exist before I added the new `Shape` subclasses (`CreationTool` could have inherited the default catch-all from `Tool`). If I *don't* want to change existing code, I can always subclass `CreationTool` and put the new catch-all behavior in the subclass.

So the catch-all saves us from changing the `Tool` interface to support stars, moons, and hearts. Great. But haven't we made a Faustian bargain in the process, what with all those dynamic casts and everything?

The answer is a qualified "yes." The catch-all approach remains viable only so long as we don't define new Element subclasses very often. That's a reasonable assumption, for example, in designs that favor composition over subclassing, which we should aspire to anyway.[5] When that's *not* a reasonable assumption, however, then we seem to have a compelling argument against VISITOR. To the framework designer who needs the double dispatch prowess of VISITOR but doesn't want to violate DIP in the deal, that's bad news.

Or is it?

Staggering VISITOR

As I've said, the challenge here isn't to eliminate VISITOR; it's to implement it more flexibly in C++.

The secret is in committing to an application-specific Visitor interface as deeply in the Visitor class hierarchy as possible. Vanilla VISITOR prescribes one and only one Visitor interface way up at the top of the hierarchy, in the base class. Instead, we will defer adding `visit` operations as long as possible.

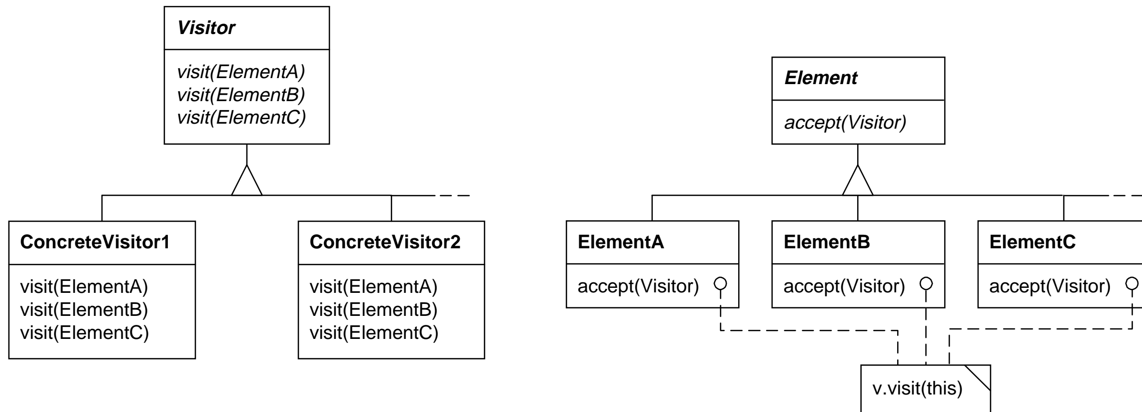

Figure 2: Vanilla VISITOR structure

Figure 2 depicts the standard VISITOR structure as described in *Design Patterns*. Notice how many different Visitor interfaces there are (i.e., just one) and where it's declared (i.e., up in the Visitor base class). Declaring the sole Visitor interface in the base class essentially bars that class from membership in a DIP-respecting framework.

What to do? See Figure 3, in which I've done two simple but crucial things:

1. All application-specific Visitor operations are moved out of the base class.
2. New AppVisitor and AppElement abstract classes bridge the gap between application-specific VISITOR classes and the framework's incarnation of the pattern.

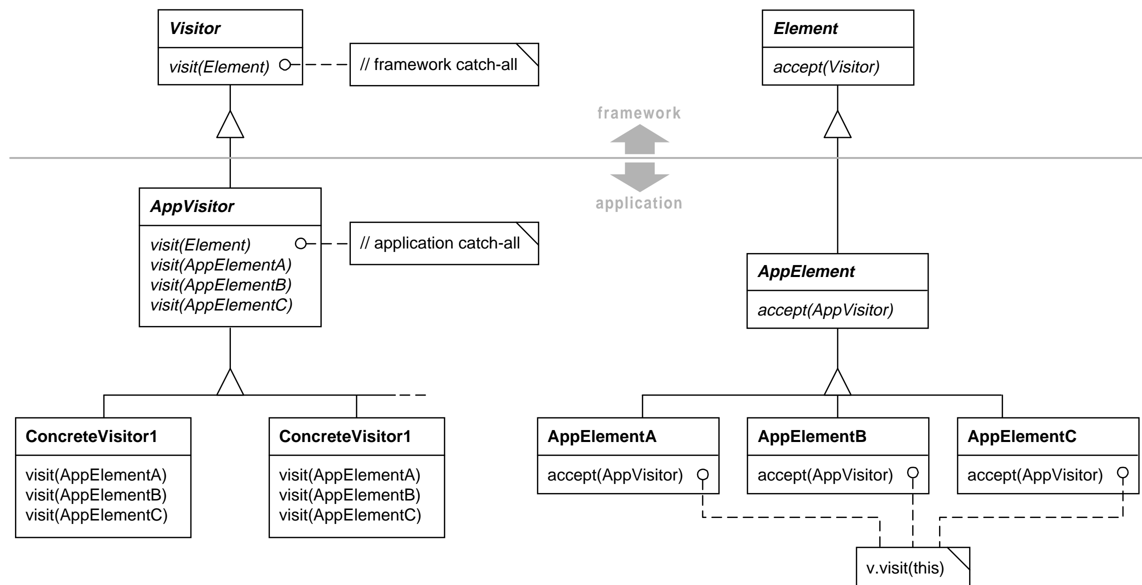

Figure 3: Staggered VISITOR structure

It's easy to see how I did #1, but what good are those new abstract classes? Basically, they allow the framework to do the VISITOR thing—namely, to include code like this:

```
Element* element;
Visitor* visitor;

// ...

element->accept(*visitor);
```

If we were to turn the drawing editor into a framework, then *any* client—whether an application or part of the framework itself—can use any tool on any shape, including yet-to-be-defined tools and shapes:

```
Shape* shape;
Tool* tool;

// ...

shape->accept(*tool);
```

The framework needn't refer to concrete classes in either its interfaces or its implementation, thus abiding by DIP. Moreover, application code needn't revert to tag-and-case-statement-style dispatch in the catch-all. Staggering application-specific Visitor interfaces down the class hierarchy greatly reduces our reliance on type tests.

To see how, look at a C++ implementation of the operation marked "application catch-all" in Figure 3:

```
void AppVisitor::visit (Element* e) {
    AppElement* ae;

    if (ae = dynamic_cast<AppElement*>(e)) {
        ae->accept(*this); // implemented in a concrete
                           // AppElement subclass
    }
}
```

Clearly, `AppVisitor`'s catch-all tests the type of its element argument only to see if its application-specific Visitor interface is appropriate for the element. If so, then `AppVisitor` puts that interface to work.

That's where we encounter the application-specific `AppElement::accept(AppVisitor&)` operation. It's intent is the same as a base-class `accept`, except that it's declared in `AppElement`, not the `Element` base class; and its parameter is a reference to an `AppVisitor`, not the base `Visitor` class. Hence `AppElement` doesn't override `Element::accept`; it introduces an entirely separate function with the same name. It is this `accept` that `AppElement`'s subclasses override.

Back to the drawing board

Let's see how staggering VISITOR might work in the drawing editor example. I can define `Shape` and `Tool` abstract base classes for the framework like so:

```
class Shape {
public:
    virtual ~Shape() = 0;

    // ...

    virtual Command* accept (Tool& t) { return t.manipulate(this); }
};

class Tool {
public:
    virtual ~Tool() = 0;

    virtual Command* manipulate (Shape*) { }; // framework catch-all
};
```

Were I elitist enough, I could stop there and relegate all other shape and tool implementations to the hapless application programmer. Fortunately, many graphical editors would benefit from a small set of framework-supplied shapes and tools—like for example the ones in our drawing editor: Line, Scribble, Rectangle, Ellipse, Polygon, and Text shapes; and Creation, Select, Rotate, Reshape, and Connect tools.

To accommodate these, let's equip the framework with predefined shapes and tools that follow the staggered approach. First the extended base classes:

```
class PredefinedShape : public Shape {
public:
    virtual ~PredefinedShape() = 0;

    virtual Command* accept (Tool& t) { return t.manipulate(this); }
    virtual Command* accept (PredefinedTool& pt)
        { return pt.manipulate(this); }
};

class PredefinedTool : public Tool {
public:
    virtual ~PredefinedTool() = 0;
    // ...

    virtual Command* manipulate(Shape*); // catch-all for
                                         // predefined shapes
    virtual Command* manipulate(Text*);
    virtual Command* manipulate(Line*);
    // etc. for all predefined shapes
};
```

The catch-all's implementation reflects the implementation of `AppVisitor::visit`:

```
Command* PredefinedTool::manipulate (Shape* s) {
    Command* cmd = 0;

    if (PredefinedShape* ps = dynamic_cast<PredefinedShape*>(s)) {
        cmd = ps->accept(*this);
    }

    return cmd;
}
```

All the concrete `Shape` and `Tool` classes of the example drawing editor can now be derived from `PredefinedShape` and `PredefinedTool`.

Where do I put default behavior?

Defining concrete subclasses of `PredefinedShape` and `PredefinedTool` is easy—just do what you'd do in a normal, brittle implementation of VISITOR. There are however a couple of interesting twists regarding default behavior.

Consider a `CreationTool` subclass for creating shapes by direct manipulation. This is one of those tools that most graphical editors need, so it would be nice if the framework came with one predefined. Here's one way to do it:

```
class CreationTool : public PredefinedTool {
public:
    // ...

    virtual Command* manipulate(Shape*);
```

```
    virtual Command* manipulate(Text*);
    virtual Command* manipulate(Line*);
    // etc. for all predefined shapes
};
```

Why am I redeclaring the catch-all? Because I'm going to extend it with special default behavior:

```
Command* CreationTool::manipulate (Shape* s) {
    Command* cmd = PredefinedTool::manipulate(s);

    if (!cmd) {  // implies s is not a PredefinedShape
        cmd = defaultManipulate(s);
    }
    return cmd;
}
```

I've introduced `CreationTool::defaultManipulate`, a protected helper function that encapsulates the default manipulation behavior. The default behavior can't be terribly specific to the shape, since we don't know what kind of shape we're dealing with at this point. But everybody knows all shapes can draw and translate themselves. So `defaultManipulate` could simply tell its `Shape*` argument to draw itself repeatedly, translating it to track the current mouse location, until the user up-clicks. Then `defaultManipulate` would copy the shape, wrap the copy in an `AddShapeCmd` command, and finally return that command.

Fine and dandy, but the astute among you may have noticed a rather nefarious problem here: What if `Shape* s` *is* in fact a pointer to a `PredefinedShape`, but it's a subclass thereof defined *after* `PredefinedTool`? In other words, what if someone extends the `PredefinedShape` subhierarchy after the fact?

This echoes our original extensibility problem with vanilla VISITOR, which gave rise to the vanilla catch-all approach. It's a bit more of a problem here, because we have the added danger of infinite recursion.

You heard me right. Consider the following sequence of operations:

1. A client calls `accept(Tool&)` on a newly derived subclass of `PredefinedShape`—a subclass that doesn't appear in `PredefinedTool`'s interface. Call this subclass `NewPredefinedShape`. The parameter passed is a `CreationTool` object.
2. `accept` dutifully calls `manipulate(this)` on the `CreationTool` object, which resolves to the `CreationTool::manipulate(Shape*)` operation I just described.
3. The first thing `CreationTool::manipulate` does is call the superclass operation `PredefinedTool::manipulate(Shape*)`, passing a pointer to the `NewPredefinedShape` instance.
4. If you look at the implementation of `PredefinedTool::manipulate(Shape*)` given earlier, you'll see that the first action of consequence is a dynamic cast to a `PredefinedShape*`. `NewPredefinedShape` is in fact a `PredefinedShape`, so this cast will succeed.
5. Having ascertained that the shape is a `PredefinedShape`, `PredefinedTool::manipulate` calls `accept` on the `PredefinedShape` with `*this` (i.e., the `CreationTool` itself) as a parameter.
6. Since there's no corresponding `manipulate(NewPredefinedShape*)` operation defined for `PredefinedTools`, the call to `manipulate` resolves statically to the catch-all `manipulate(Shape*)`, which will be called on the `CreationTool` object.

Notice how step #6 leaves us at step #3, hence the infinite recursion. Ouch.

Fortunately the solution is simple, and it too echoes what has gone before. What we're missing is a catch-all specific to `PredefinedShape`:

```
Command* PredefinedTool::manipulate (PredefinedShape* ps) {
    return defaultManipulate(ps);
}
```

This catch-all breaks the infinite recursion at step #6, because the call to `manipulate` will now resolve to this more specific function. I call it the "extension catch-all," and it works identically to the original catch-all described in *Pattern Hatching*: you override the extension catch-all to handle late-coming subclasses of `PredefinedShape`—that is, if there aren't too many of them. It also provides a place to put the functionality of `defaultManipulate`, thereby saving you a member function if you so choose.

Just as it's a good idea in vanilla VISITOR to include a conventional catch-all to handle extensions to the Element hierarchy, in the staggered approach it behooves one to include an extension catch-all for every *group* of extensions.

Application-specific shapes and tools

If an application defines entirely new shapes (like stars, moons, and hearts), and they should exhibit unique manipulation behavior during creation, then you apply the whole approach again. First, introduce new `AppShape` and `AppTool` subclasses from which to derive the new, application-specific shapes and the tools that work on them:

```
class AppShape : public Shape {
public:
    virtual ~AppShape() = 0;

    virtual Command* accept (Tool& t) { return t.manipulate(this); }
    virtual Command* accept (AppTool& at)
        { return at.manipulate(this); }
};

class AppTool : public Tool {
public:
    virtual ~AppTool() = 0;

    virtual Command* manipulate(Shape*);
    virtual Command* manipulate(AppShape*); // extension catch-all

    virtual Command* manipulate(Star*);
    virtual Command* manipulate(Moon*);
    virtual Command* manipulate(Heart*);
};
```

`AppTool`'s catch-all would look like this:

```
Command* AppTool::manipulate (Shape* s) {
    AppShape* as;
    Command* cmd = 0;

    if (as = dynamic_cast<AppShape*>(s)) {
        cmd = as->accept(*this);
    }

    return cmd;
}
```

A subclass of `AppTool` can now define different, specialized tool behavior for stars, moons, and hearts—nothing new about that. But in this case we're gonna be especially demanding: we want to extend the capabilities of a predefined tool, `CreationTool`. In other words, we want to *reuse* `CreationTool`.

This is one of those rare instances in which multiple implementation inheritance pulls its own weight:

```
class AppCreationTool : public AppTool, public CreationTool  {
public:
    // ...

    using AppTool::manipulate;
    using CreationTool::manipulate;
```

```
        virtual Command* manipulate(Shape*);

        virtual Command* manipulate(Star*);
        virtual Command* manipulate(Moon*);
        virtual Command* manipulate(Heart*);
    };
```

The `using` declarations bring the `AppTool` and `CreationTool` functions into `AppCreationTool`, saving us the chore of redefining them manually. The last three `manipulate` operations implement specialized manipulation behavior for creating stars, moons, and hearts, respectively.

That leaves the framework catch-all, which both superclasses define. And that means we have to disambiguate it in `AppCreationTool`.

As in most things object-oriented, specific behavior outranks general behavior. First the catch-all will do a dynamic cast akin to the one in `PredefinedTool`'s catch-all, except it'll be casting to `AppShape*`, not `PredefinedShape*`. If that cast fails, *then* it'll try casting to a `PredefinedShape*`. If that fails, `AppCreationTool` will finally revert to the default `CreationTool` behavior:

```
Command* AppCreationTool::manipulate (Shape* s) {
    Command* cmd = 0;

    if (AppShape* as = dynamic_cast<AppShape*>(s)) {
        cmd = as->accept(*this);

    } else if (PredefinedShape* ps = dynamic_cast<PredefinedShape*>(s)) {
        cmd = ps->accept(*this);

    } else {
        cmd = defaultManipulate(s);

    }

    return cmd;
}
```

Epilogue

Compared to the conventional VISITOR implementation, the overhead of the staggered approach amounts to twice the number of `accept` and `visit` operations plus one dynamic cast.* Contrast that with a conventional catch-all implementation, where you have an `accept`-`visit` pair plus dynamic casts proportional to the number of application-specific ConcreteElement classes. Staggering application-specific Visitor interfaces lets you extend the Element hierarchy in arbitrarily large chunks with only constant overhead.

These three implementation approaches define a spectrum of trade-offs between static type safety and ConcreteElement extensibility. Conventional VISITOR gives you full static type safety but no way to define new ConcreteElement subclasses. *Pattern Hatching*'s catch-all approach gives you full extensibility but no static type checking for new ConcreteElement subclasses. The staggered approach falls in-between, close to conventional VISITOR regarding type safety but close to the catch-all with respect to extensibility—the best of both worlds. But it's the most complex of the lot by a wide margin.†

---

* `AppCreationTool` adds another dynamic cast, the result of having combined two tools with multiple inheritance. Cute as that example may be, it probably isn't representative given the stigma attached to deriving from concrete classes.[6]

† Bob Martin describes an implementation variation called "Acyclic Visitor."[7] It is similar to the staggered approach except that dynamic casting occurs in `accept` rather than `visit` operations. Such placement has undesirable consequences on maintainability, as I describe in *Pattern Hatching*.[8]

This whole scenario demonstrates how important it is to keep the design pattern distinct from its implementation. VISITOR addresses three separate but related problems:

1. You want polymorphic behavior, but it must exist outside the class hierarchy to which it logically belongs.
2. You want to avoid scattering polymorphic behavior throughout a class hierarchy.
3. You need double dispatch semantics, as in the $m \times n$ problem addressed here.

These problems have importance on a conceptual level quite apart from how their solution is implemented. VISITOR characterizes them and offers a solution in terms of common object-oriented language facilities. In our context, the most relevant is the $m \times n$ problem. It's a pity that double dispatch is so tough to implement extensibly in single-dispatch languages, but that fact invalidates neither the problem nor the pattern.

The staggered VISITOR approach I've described takes us up a notch or two in the flexibility department, with incremental added cost. Still, the ultimate solution may lie outside C++. Programming language makes all the difference, generally speaking, and I'll bet implementing VISITOR in a language that supports double dispatch (such as CLOS) can simplify things a lot further, with even greater flexibility. That could make Erich's concerns vanishingly small indeed.

Have a safe and joyous Y2K!

Acknowledgments

Thanks to Andrei Alexandrescu, Brad Appleton, Dave Ehnebuske, Ralph Johnson, Scott Johnston, and Scott Meyers for the sanity checks.

References

[1] Vlissides, J. TOOLED COMPOSITE. *C++ Report*, September 1999, pp. 43–47.
[2] Vlissides, J. *Pattern Hatching: Design Patterns Applied*, Addison–Wesley, Reading, MA, 1998, pp. 36, 81–84.
[3] Martin, R. The Dependency Inversion Principle. *C++ Report*, June 1996, pp. 61–66.
[4] Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.
[5] *Ibid*, p. 20.
[6] Meyers, S. *More Effective C++*, Addison–Wesley, Reading, MA, 1996, pp. 258–270.
[7] Martin, R. "Acyclic Visitor," in *Pattern Languages of Program Design 3*, R. Martin, et al., eds., Addison–Wesley, Reading, MA, 1998, pp. 93–103.
[8] *Pattern Hatching*, pp. 83–84.