Pattern Hatching

TOOLED COMPOSITE

John Vlissides
C++ Report, September 1999

It's hard to believe (for me, anyway), but this is the fifth in a series of articles on compound design patterns—patterns that document synergies among other patterns. I find it hard to believe for two reasons. First, this is my longest-running series yet, having commenced well over a year ago. Where has the time gone, etc. Second, four prior articles might sound like a lot, but they haven't covered as much ground as I'd like:

1. The first article[1] introduced this category of patterns under a slightly different name—"*composite* design patterns," reflecting the prevailing nomenclature at the time. I sketched several examples in that article, promising to flesh them out in future installments.
2. Next came "PLUGGABLE FACTORY, Part I," an initial attempt at putting my money where my mouth is.[2] PLUGGABLE FACTORY is a compound of ABSTRACT FACTORY and PROTOTYPE. That article presented the first half of the pattern in standard GoF format.[3] I also took the liberty of proposing "*compound* design patterns" as an alternative name, to ward off confusion with the COMPOSITE pattern. With no complaints or threats to deter me, I've used "compound" ever since.
3. "PLUGGABLE FACTORY, Part II" picked up where part one left off, completing the first and only fully fleshed-out compound pattern in the series so far.[4]
4. In the most recent installment, "Compounding COMMAND," Richard Helm and I looked at several applications of a compound comprising COMMAND, COMPOSITE, and DECORATOR.[5] This unnamed threesome produced open-ended functionality with just a handful of classes, after TEMPLATE METHOD fell short in that capacity.

Regrettably, we never got around to writing up that threesome as a compound. It would have taken two more articles at least, not to mention a toll in blood, sweat, and tears. Rather than exacting the latter, I decided to change the subject in my last column and talk about Extreme Programming. Actually, Kent Beck did most of the talking.[6] Interviewing Kent turned out to be rather more fun than wrestling a pattern, and Richard, into submission.

Fresh from that interlude, I return to the compound pattern theme this month with a brand new combo. I also have the decency to give it a name: TOOLED COMPOSITE. While space constraints once again keep me from describing it in all its glory (i.e., in GoF form), there should be enough detail here to promote implementation. If you're adventurous and would like to flesh it out yourself, I'm here to help—but you have to promise to submit your work to a PLoP.[*]

[*] See http://hillside.net/patterns/conferences/.

A context

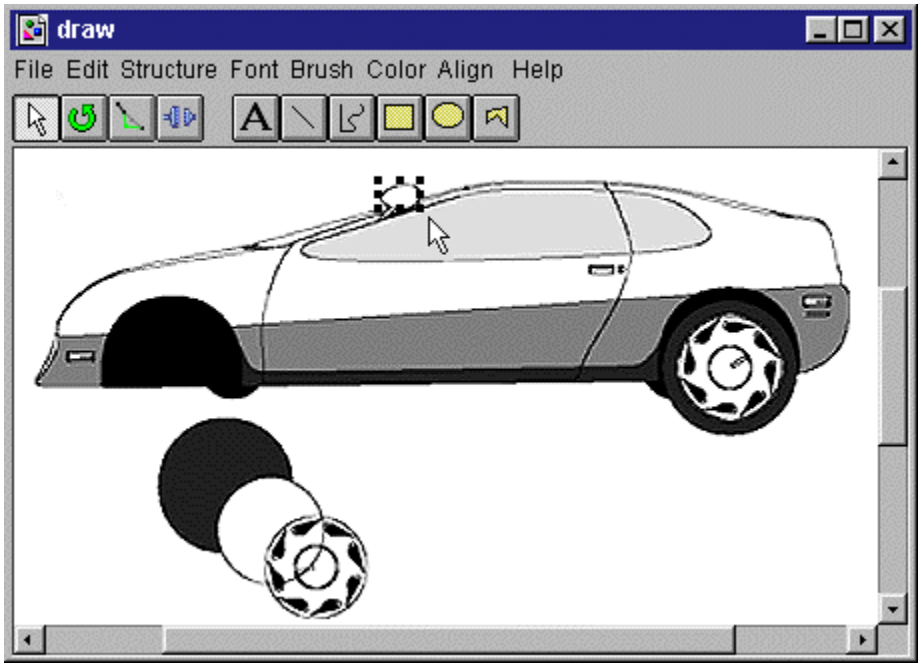

Figure 1: A simple drawing editor

You're building a simple drawing editor like the one in Figure 1. The editor lets you create drawings by *direct manipulation*—you "pick up" a drawing tool and "wield" it to create shapes, move them around, rotate them, and generally assemble them into complete drawings.

There are two kinds of tools in this editor. *Creation tools* let you produce new shapes and put them into your drawing. Buttons that represent these tools are grouped on the right of the toolbar atop the editor window. Presently there are text, line, scribble, rectangle, ellipse, and polygon creation tools. I call the other kind of tools *modification tools*, because they let you change existing objects. In Figure 1, the modification tool buttons are grouped on the left of the toolbar. These include selection, rotate, reshape, and connect tools.

To pick up a tool, you press the corresponding button on the toolbar. To start wielding the tool, you click in the drawing area below the toolbar, which is where you do your drawing. What happens after the initial *downclick* (i.e., before you release the mouse button) depends on the tool you're wielding and what, if anything, you downclicked on.

Suppose you're wielding the Line tool. To create a line with it, first you downclick in the drawing area wherever you'd like the first endpoint. To place the other endpoint, you release the mouse button (i.e., *upclick*). Between the downclick and the upclick, the editor will draw an animated line between the first endpoint and the current mouse position to indicate the line you're about to create. The interval between the downclick and the upclick is known as *dragging*. Such dynamic control and feedback are the essence of direct manipulation.

Other tools behave differently, often radically so. Creating an ellipse or a rectangle involves the same downclick/drag/upclick interaction, but the dragging animation is different in each case (an animated ellipse or rectangle versus an animated line). The Polygon tool differs more markedly in that it requires multiple clicks, one for each vertex of the polygon. You double-click to complete the polygon.

Modification tools tend to have even more specialized behaviors. The Select tool (the button with the mouse cursor on it) lets you single out a shape for further manipulation. Click on a shape with it, and eight small squares called *handles* appear around the shape, indicating that the shape is now selected. A down-

click/drag on a handle lets you deform the shape in one or two dimensions. Clicking anywhere else on a selected shape lets you drag it to a new location. Even a downclick on nothing in particular does something, producing an animated rectangle with which to select multiple shapes at once. Just drag to reshape the rectangle until it surrounds the shapes you're interested in; then release the mouse to select them. When finally you're happy with your selection(s), you can perform menu operations on them—operations that don't involve direct manipulation, like cut, duplicate, group, and change color.

The problems

That simple user metaphor is full of juicy design and implementation problems. TOOLED COMPOSITE tackles several of them:

1. *How do you represent shapes?* Every drawing needs to be stored in memory somehow, and that "somehow" defines the editor's internal representation of drawings. This representation is key to nearly everything. It lets a drawing appear in the editing window. It lets a user manipulate and print the drawing. And it is this representation that is saved and retrieved from disk in one format or another.

   But the representation should capture more than just every shape in the drawing. It should also record how shapes are grouped together, since grouping is an essential feature of today's drawing editors. It should allow unlimited nesting (groups of groups), too.

2. *How do you represent tools?* The drawing is not the only thing that needs internal representation. Tools have state and behavior that must live in the implementation somewhere.

3. *How do tools and shapes interact?* As the narrative suggests, the same tool can behave one way when used on one shape and a different way on another. If we have *m* tools and *n* shapes in our editor, the number of different interactions approaches *m* x *n*. It may even exceed that product if a tool has behavior that's independent of any shape, as the Select tool does when you click on nothing in particular.

4. *How do you enhance the editor with new shapes and tools?* To paraphrase Dirty Harry, you've got to know your limitations. That maxim applies here in many ways, but it's particularly important to recognize that we won't foresee every kind of shape that a user might need. The same goes for every kind of tool. We can demonstrate our humility by making it easy to add new shapes and tools to the application.

Now, I can assure you these problems aren't limited to drawing editors. A music editor, for example, may apply the same metaphor toward composing musical scores instead of arbitrary drawings. A graphical user interface (GUI) builder lets developers assemble buttons, scroll bars, and menus into dialog boxes and application windows with little or no programming. Likewise for network configuration, project management, and mapmaking applications. The solutions we work out here will apply to most any program that employs a direct manipulation metaphor.

The tactics

I'll depart from the pedagogical devices we GoF use elsewhere. In Chapter 2 of *Design Patterns*, for example, we look at seven specific problems in the design of a document editor. The primary goal there is to teach the design pattern concept to the uninitiated. We solve each design problem the hard way, essentially rediscovering a pattern in the process, before taking a brief look at the pattern itself.

I stand that approach on its head in Chapter 2 of *Pattern Hatching*.[7] There I assume the reader already knows what patterns are and that he or she wants to learn how to use them effectively. The emphasis is on finding, applying, and even rejecting certain patterns—not on rediscovering them.

The approach here will be different again. This being a compound pattern, I want to focus on its intent and how the constituent patterns play to it. We don't have to find the right patterns, nor do we have to redis-

cover them. So I don't mind telling you up-front that we'll be using COMPOSITE, STATE, PROTOTYPE, VISITOR, and COMMAND.

Representing shapes

The first design problem is a gimme. Which pattern addresses it? Well, if the name of this compound isn't enough of a hint, then consider what we're trying to do: represent drawings made up of shapes, groups of shapes, groups of groups of shapes, etc. If that's not recursive composition, I don't know what is. And recursive composition means one thing—COMPOSITE.

Applying the pattern is straightforward. The Shape class will play the role of the Component participant; Leaf classes include Line, Ellipse, Text, and so forth. Oddly enough, we'll call the Composite class "Group." Figure 2 shows the resulting class structure. The only operation we'll bother declaring is `draw`, which all shapes are required to do for themselves.

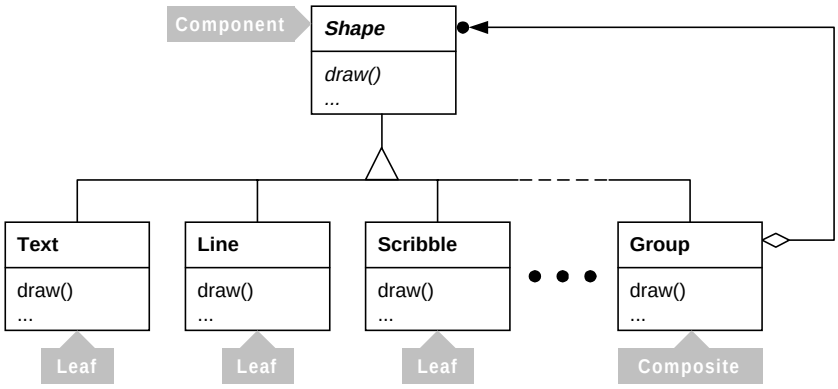

Figure 2: Shape class structure

Representing tools

So much for representing drawings. What about tools?

Lo and behold the following passage from *Design Patterns*:[8]

> *Most popular interactive drawing programs provide "tools" for performing operations by direct manipulation. For example, a line-drawing tool lets a user click and drag to create a new line. A selection tool lets the user select shapes. There's usually a palette of such tools to choose from. The user thinks of this activity as picking up a tool and wielding it, but in reality the editor's behavior changes with the current tool: When a drawing tool is active we create shapes; when the selection tool is active we select shapes; and so forth. We can use the STATE pattern to change the editor's behavior depending on the current tool.*

We'll do exactly that. A Tool class will play the role of the State participant, and ConcreteState subclasses will implement the tools we need. The Context role will be played by an Editor class, which could also implement the main window of the application (see Figure 3).

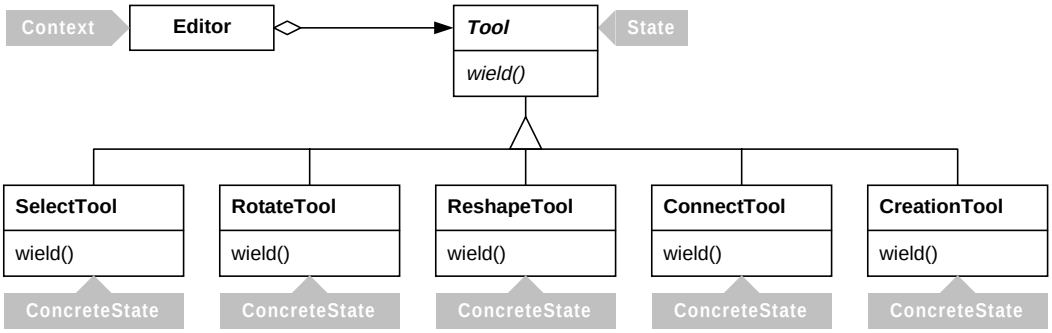

Figure 3: Tool class structure

Where are all the creation tool classes, you ask? Currently there's just one, CreationTool. And it's all we need.

From 100,000 feet, all creation tools appear to do the same thing: they instantiate shapes. Up close, their differences lie solely in the kinds of shapes they produce and how they produce them. These differences are clear to users. When a user wields the Line tool, for example, it's because he wants to draw a line, not a polygon. But that doesn't mean he's aware of the *classes* that implement creation tools.

We simply don't need a class for every creation tool—not as long as we can capture the distinctions among creation tools with the PROTOTYPE pattern. Its Motivation section explains how to do that in the context of a music editor.[9] There, instances of a GraphicTool class are configured with prototypes of different graphical objects like staves and half notes. When a user wields a GraphicTool instance, the tool merely copies its prototype and inserts it into the score.

That approach would work just as well here were it not for one thing. Yes, our CreationTool can keep a reference to a prototypical shape, and yes, it can clone that shape when it is wielded. But there's an issue that PROTOTYPE's Motivation conveniently ignores: Different shapes entail different direct manipulation behavior during creation. As we saw earlier, creating a polygon is pretty different from creating a line—each involves different mouse inputs and animated feedback. And creating a text object is different again.

Where are these disparate behaviors implemented? Seems we need multiple creation tool classes after all.

Direct manipulation behavior

This problem is really quite profound. It goes beyond shape creation, cutting to the heart of how shapes and tools interact. If we view our CreationTool as interacting not with an existing shape but with its prototype, it becomes just another contributor to the $m \times n$ possible interactions between m tools and n shapes. But never fear; we're not about to define $m \times n$ classes to implement these interactions. We're not going to define n creation tool classes either. We are going to solve this problem using—surprise!—VISITOR.

It's not very surprising, actually. VISITOR is, after all, a way of implementing double dispatch in single-dispatch languages.[10] Whereas single dispatch selects the code to execute based on an operation name and *one* object type (that of the receiver), double dispatch adds a second object type to the selection criteria. In other words, the run-time behavior of a double-dispatch call depends on the operation name and *two* receivers.[†]

Anyhow, our situation with tools and shapes cries out for double dispatch. The direct manipulation behavior we want depends on two types: the tool being wielded, and the shape it's wielded on. In the case of CreationTool, the shape in question is the CreationTool's prototype.

[†] Sadly, the "receiver" metaphor doesn't work as well for the double dispatch case. I find it more natural to think of the dispatch objects as special parameters to a function call, parameters whose run-time types specify the function's implementation.

So VISITOR gives us double dispatch in a C++ setting. Now, do the classes we've defined so far play any roles in the pattern? Or do we need new VISITOR-specific classes?

Actually not. We have a natural candidate for the Element participant—Shape. That means we should define an `accept` operation on the Shape interface. But what does it take as a parameter? Who plays the Visitor role? There's really only one candidate: Tool.

I hear gasps. "Tool as a *Visitor*?" Well, why not? The metaphor works quite well once we sack a cherished name or two. Specifically, instead of calling them `visit` operations, we'll use something closer to what tools actually do:

```
class Tool {
    // ...

    virtual Command* manipulate (Shape*)
        { return 0; } // catch-all[11]

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

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

    // etc. for remaining Shape subclasses

    virtual Command* wield() = 0;
};
```

Your typical `accept` implementation now looks like,

```
Command* accept (Tool& tool)
    { return tool.manipulate(this); }
```

which reads better with the renamed Visitor operations. In fact, one might wonder whether it's worthwhile to rename `accept` as well. At least I did. But even with concerted effort, I couldn't come up with anything clearly better. The best my thesaurus and I could muster were "undergo," "sustain," and "withstand." All seem rather graceless:

```
Command* undergo (Tool& tool)          // ?
    { return tool.manipulate(this); }

Command* sustain (Tool& tool)          // ?
    { return tool.manipulate(this); }

Command* withstand (Tool& tool)        // ?
    { return tool.manipulate(this); }
```

"`manipulate(this)`" sounds, ahem, abusive enough without the help of these alternatives. I think I'll stick with "`accept`." It's as appropriate as anything else, and it offers the added benefit of hinting at the VISITOR relationship between shapes and tools.

Whatever the names, we are now in a position to implement direct manipulation behavior for each and every combination of tool and shape, without generating a cross-product of classes. Consider how one might implement `wield`—the operation that sets tools in motion—for CreationTool:

```
Command* CreationTool::wield () {
    return getPrototype()->accept(*this);
}
```

If the prototype is, say, a Line instance, the corresponding `manipulate` operation might look something like this:

```
Command* CreationTool::manipulate (Line*) {
    Mouse& mouse = getMouse();
    DrawingArea& dwgArea = getDrawingArea();

    Point& p0 = mouse.getPosition();
    Point& p1 = p0;

    for (;;) {
        MouseEvent& event = mouse.nextEvent();

        if (event.buttonUp()) break;

        dwgArea.eraseLine(p0, p1);
        p1 = mouse.getPosition();
        dwgArea.drawLine(p0, p1);
    }

    dwgArea.eraseLine(p0, p1);

    return new AddShapeCmd(new Line(p0, p1));
}
```

Result of manipulation

I know. You're wondering about those `Command*` return values. What are they there for? Two things:

1. *Multilevel undo/redo.* As the name of the return type suggests, these are Command objects from the COMMAND pattern.[12] They're there to support unlimited undo and redo, something no modern application can (un)do without.

2. *Separation of concerns.* It's reasonable to ask why we're introducing new classes in support of the COMMAND pattern. Thus far we've preferred to add new roles to existing classes with each pattern application. You'll recall that when we applied VISITOR, we didn't introduce an independent Visitor hierarchy; we simply made Tool play the Visitor role. So why not have Tool take on the Command role as well?

   The short answer is, "To maintain a separation of concerns." Tools implement direct manipulation behavior, which is distinct from the ultimate effect of that behavior. The unit of undoable work is generally independent of direct manipulation. For example, undoing the creation of a Line object involves removing the line from the drawing, period; it doesn't re-animate the creation process in reverse. Separating the undoable unit of work from the semantics of direct manipulation lets us implement and vary the two independently. It also keeps tools from being schizophrenic—part something you wield, part unit of undoable work.

Once the client who called `wield` (say, the Editor object) has a command, it can execute it and put it on a history list. For details on multilevel undo/redo, see items 2 and 3 of COMMAND's Implementation section,[13] or check out Buschmann, et al.'s COMMAND PROCESSOR pattern.[14]

Summary

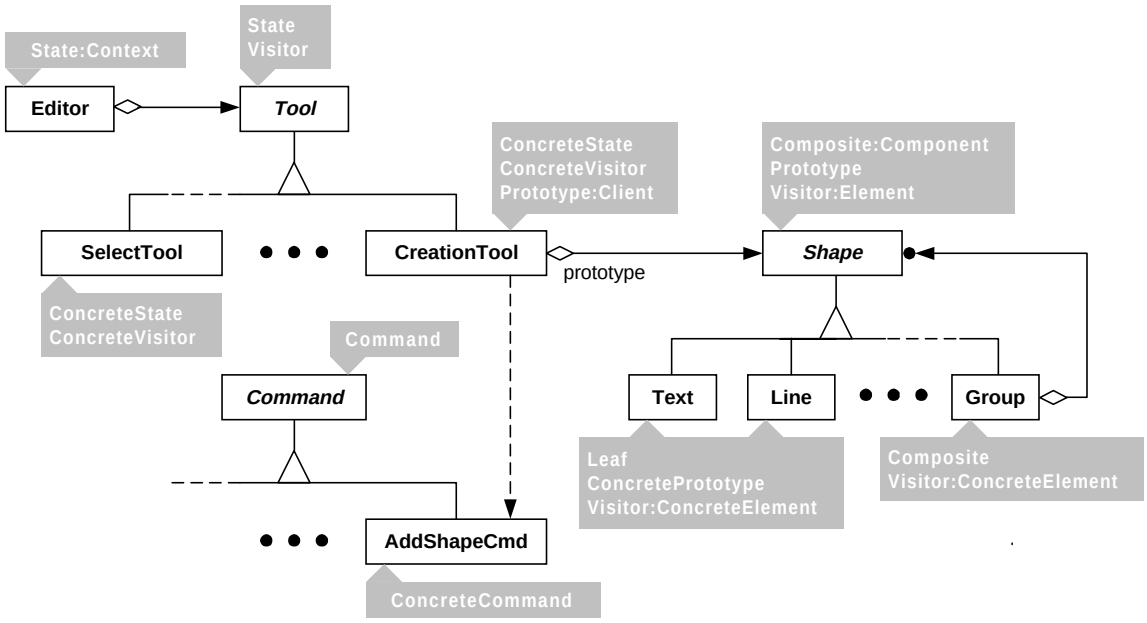

Figure 4: TOOLED COMPOSITE structure in this application

Figure 4 shows the classes, relations, and constituent patterns in this application of TOOLED COMPOSITE. In sum, this compound pattern combines

- COMPOSITE, to define the recursive structure you manipulate directly with tools;
- STATE, to keep track of the current tool;
- PROTOTYPE, to create any kind of shape with just one creation tool class;
- VISITOR, to implement how tools affect shapes during direct manipulation; and
- COMMAND, to specify the potentially undoable effects of wielding a tool.

Remember that this design isn't limited to your garden-variety drawing editor. TOOLED COMPOSITE is applicable wherever you'd like a user to manipulate familiar objects with a pointing device. It's a design that's near and dear to me, having discovered it as part of my thesis research.[15] That work is over ten years old now—an eternity in this business—but it seems just as relevant as ever.

Hey, what about Problem #4?

I skipped it on purpose. It makes such a nice segue.

How does TOOLED COMPOSITE promote extension of tools and shapes? Adding new tools is easy—just define new Tool subclasses. What's troublesome is adding new Shape subclasses after the fact.

Everything works fine when we can anticipate every shape in our application. But when we can't, we run up against the age-old problem with VISITOR: its brittleness in the face of a changing Element hierarchy.[16] The problem is especially acute when we want to use TOOLED COMPOSITE in a framework. Applications based on the framework will almost certainly want to extend the Composite hierarchy.

We'll tackle the general problem that VISITOR poses in frameworks next time. *Aloha!*

Acknowledgments

A torrent of thanks to Brad Appleton, Richard Helm, Ralph Johnson, and Dirk Riehle.

References

[1] Vlissides, J. Composite design patterns (they aren't what you think). *C++ Report*, June 1998, pp. 45–47, 53.
[2] Vlissides, J. PLUGGABLE FACTORY, part I. *C++ Report*, November/December 1998, pp. 52–56, 68.
[3] Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.
[4] Vlissides, J. PLUGGABLE FACTORY, part II. *C++ Report*, February 1999, pp. 51–57.
[5] Vlissides, J. and R. Helm. Compounding COMMAND. *C++ Report*, April 1999, pp. 47–52.
[6] Vlissides, J. XP. *C++ Report*, June 1999, pp. 44–52, 69.
[7] Vlissides, J. *Pattern Hatching: Design Patterns Applied*, Addison–Wesley, Reading, MA, 1998.
[8] *Design Patterns*, p. 313.
[9] *Ibid*, pp. 117–118.
[10] *Ibid*, p. 339.
[11] *Pattern Hatching*, pp. 36, 81–84.
[12] *Design Patterns*, pp. 233–242.
[13] *Ibid*, pp. 238–239.
[14] Buschmann, et al. *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, Chichester, 1996.
[15] Vlissides, J. *Generalized Graphical Object Editing*, Ph.D. thesis, Stanford University, 1990.
[16] *Design Patterns*, p. 336.