Pattern Hatching

Compounding COMMAND

John Vlissides and Richard Helm

C++ Report, April 1999

This month I'm pleased to have the inimitable Richard Helm join me as co-columnist. Together we examine the COMMAND pattern and some nascent compound patterns it's central to. If the past is any indicator, our combined insights will be more astute than anything I could muster alone. (By the way, any bugs you find here are *his* fault.)

COMMAND, you'll recall, is all about encapsulating a request—and how to fulfill it—in an object.[1] The idea is to hide a logical unit of work behind a fixed interface. Clients use this interface to commission the work without knowing how it'll get done, or even *what* will get done. Indeed, the interface hides things so well that a client generally can't tell one unit of work from any other—which is precisely the point.

Figure 1 is the obligatory structure diagram from *Design Patterns*. The Command abstract class defines the interface for carrying out the unit of work in a ConcreteCommand object. A Command subclass may delegate the work to one or more Receiver objects, or it may do the work entirely on its own, or anything in-between those extremes. Oddly enough, the Client in this diagram isn't the object that actually gets the work started; that's the Invoker's responsibility. The Client acts as midwife and matchmaker, creating the command and initializing it with references to the receivers it depends on, if any.*

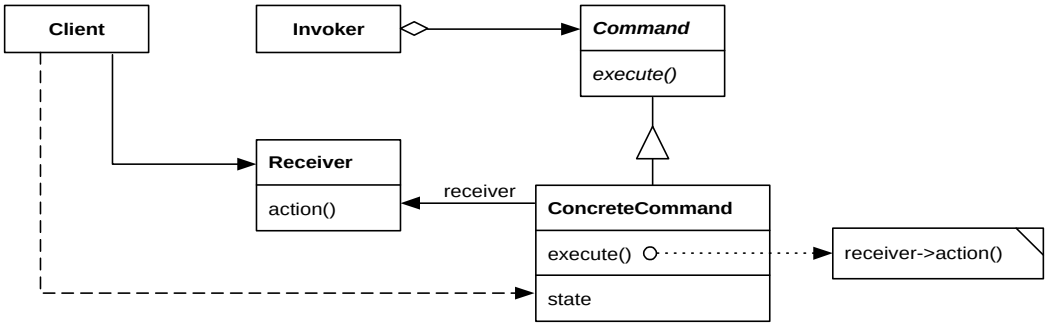

Figure 1: COMMAND Structure

Proliferating commands

That's the gist of the COMMAND pattern, and it's fine as far as it goes. Fortunately for us columnists that's not the end of the story, because a naïve implementation of Figure 1 can lead to a proliferation of ConcreteCommand classes, lots of duplicated code, or both.

Suppose your project needs undoable operations for managing customer information, and you've been charged with their design and implementation. "No problem," you tell yourself. "I'll just make a subclass of

* While both Client and Invoker are clients in a general sense, only one of them is identified by that name. If we had to do it over again, we should probably avoid "Client" altogether, to head off confusion.

Command for each operation." And so you do. When the dust settles, you've implemented about a dozen classes. Figure 2 shows a representative number of them and their place in the hierarchy of commands.[†]

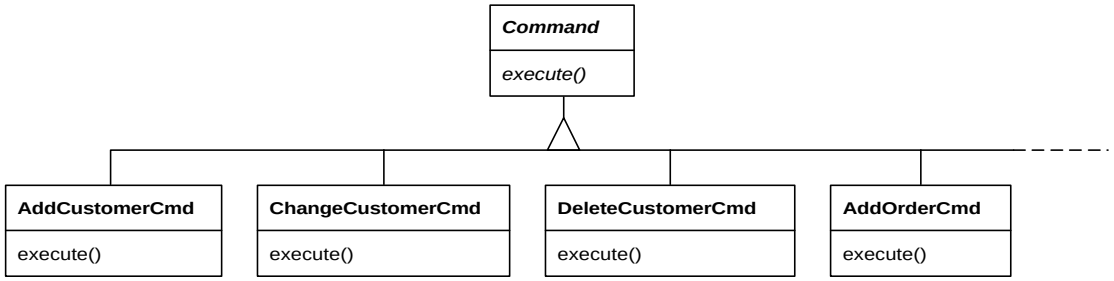

Figure 2: Application-specific ConcreteCommands

"Piece of cake," you muse. "If only the rest of the design were this easy."

Confident that you've reached a milestone, you treat yourself to a leisurely coffee break. As you watch the dregs of last night's pot fill the thickly stained mug, an unpleasant thought enters your mind, and yet it's not about getting sick from this swill. "What if someone tries to delete a customer that's being changed by someone else?"

Sigh. *Why* didn't you think of that earlier? A sinking feeling sets in.

Instinctively, you start groping for reasons why it could never be.

> *We're targeting single-user applications. Delete and edit at the same time? It ain't gonna happen…*
>
> *Besides, doing it right would require some kind of locking, which means overhead—lots of overhead. It'll kill performance. Programs that don't need locking shouldn't have to pay for it…*
>
> *Hey, users who are silly enough to delete and edit the same customer get what they deserve … right?*

Right. Now you understand why you never became a trial lawyer.

Okay, so the issue can't be swept under the rug. But you're not about to throw away stuff that already works. You resolve therefore to offer programmers a choice: They won't pay for locking if they don't need it, but those that do will get the best locking money can buy.

You know just how to do it too, because in your project, everything these commands deal with—all the data about customers, orders, whatever—*lives in a database*. The database provides not just locking but full-blown transactions. All that's needed is a companion set of commands that do exactly what your existing commands do, plus a teenie bit more.

Figure 3 shows the logical result. A preliminary investigation of the database API revealed a simple `beginTransaction/endTransaction` model. All you did was define "Atomic" subclasses of your commands, each of which implements its `execute` operation the same way: by sandwiching the base class operation between a pair of `beginTransaction/endTransaction` calls. Whatever the base class does is thus guaranteed to run either atomically or not at all.

[†] If you're wondering where the undo support is, we left it out to stay focused on the bigger picture. For details on implementing undoable commands, see item 2 in the pattern's Implementation section.

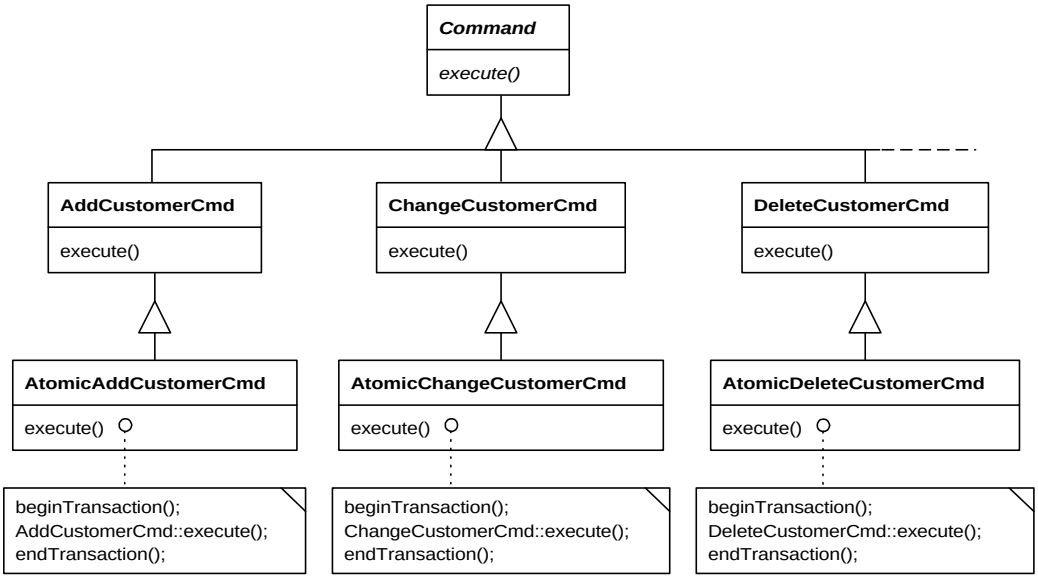

Figure 3: Transaction-savvy subclasses

A workable approach, no doubt, but there are some disquieting things about it. The new subclasses don't seem to do much, and there sure are a lot of them. It doesn't sit well that you've doubled the number of classes merely to encompass transactions. These subclasses are just not pulling their weight.

What's more, as your familiarity with the database API grows, you realize that the `beginTransaction/endTransaction` implementation is rather simple-minded. You learn that for robustness' sake, each subclass should implement `execute` more like this:

```
virtual void execute () {
    // some setup code

    try {
        beginTransaction();
        // call to parent class' execute()
        endTransaction();

    } catch (AbortedTransaction& e) {
        // (assume endTransaction has been called)
        // cleanup code for abnormal termination
        return;

    } catch (...) {
        // (can't assume anything)
        endTransaction();
        // cleanup code for abnormal termination
        throw;
    }

    // normal cleanup code
}
```

Robust though this code may be, you don't want it repeated in every `execute` operation. But more than that, you shouldn't expect everyone to appreciate its subtleties. It's a good bet many programmers will share your initial unfamiliarity with the database API and will therefore have the same learning curve to climb. Some will be slow to surmount it, and some, alas, never will. You can expect as many programmers to get it wrong as will get it right. Give them all the help you can, short of encouraging cut-and-paste reuse.

COMMAND + TEMPLATE METHOD

The TEMPLATE METHOD pattern offers a solution. It can capture the common aspects of a transaction-savvy implementation while leaving particulars to subclasses.

Here's one approach to leveraging the pattern. First you define a protected and *non*virtual `atomicExecute` operation in the Command base class. It captures the invariant aspects of the robust transaction implementation given earlier:

```
void Command::atomicExecute () {
    doSetup();

    try {
        beginTransaction();
        doExecute();
        endTransaction();

    } catch (AbortedTransaction& e) {
        // (assume endTransaction has been called)
        doAbnormalCleanup();
        return;

    } catch (...) {
        // (can't assume anything)
        endTransaction();
        doAbnormalCleanup();
        throw;
    }

    doNormalCleanup();
}
```

`atomicExecute` is a template method in that it defines the skeleton of a transaction-based `execute` operation, and subclasses can't override it (it's protected but nonvirtual). It is also a helper function in that subclasses may or may not avail themselves of its services. Those that do can implement `execute` simply by delegating to the template method,

```
void AtomicAddCustomerCmd::execute () { atomicExecute(); }
```

and then appropriately overriding the primitive operations, all of which are prefixed by `do-`: `doSetup`, `doExecute`, `doNormalCleanup`, and `doAbnormalCleanup`. Aspiring subclassers of Command can focus on implementing these responsibilities without the travails of robust transactions. TEMPLATE METHOD has reduced both complexity and code duplication in one fell swoop.

We're still a ways from nirvana, however, because we have yet to do something about the excessive number of classes. We've got just as many as ever: a ConcreteCommand class for every undoable operation plus an accompanying "Atomic-" subclass.

Whenever you find yourself with too many classes, some soul-searching is in order. Say to yourself, "Self, have you gone bananas with inheritance?" Put another way, judge for yourself whether you've encoded too much information in the types of your objects. Is there a common attribute that can be factored out of several classes and made an object in its own right?

In this case, the answer to both questions is an unequivocal "yes"; we *have* gone crazy with inheritance, and there *is* an attribute common to fully half of our classes: atomicity. We've been associating atomicity with concrete command classes for no good reason.

Think about what it would mean to make atomicity a first-class object. Imagine putting the code that currently lives in the `atomicExecute` operation into a separate class—call it "Atomic." Now consider how the code in that class could be melded into the basic, non-atomic ConcreteCommand subclasses you defined at the outset. One possible melding mechanism is multiple inheritance—that is, using Atomic as a mix-in to each ConcreteCommand class. But a moment's reflection will persuade you that this won't reduce the class

count. In fact, it *increases* the count by one mix-in class. Multiple inheritance does nothing to reduce our reliance on inheritance, much to no one's surprise.

Equally unsurprising (if you've gleaned anything from *Design Patterns*) would be recourse to the primary object-oriented alternative to inheritance, namely *composition*. We're supposed to favor it over inheritance, remember.[2] What should we use composition for here? To add functionality to existing classes, natch.

By now all but the greenest of GoF aficionados know where we're headed with this…

COMMAND + DECORATOR

The DECORATOR pattern is the classic, compositional, pay-as-you-go approach to imparting functionality without changing existing classes. Applying the pattern in all its generality yields the class diagram of Figure 4.‡

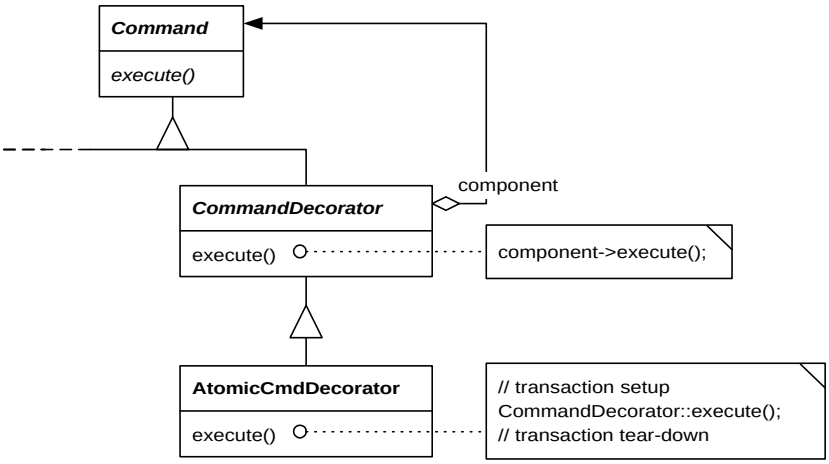

Figure 4: Class hierarchy post-DECORATOR

How does one use this stuff? To make a command atomic, merely wrap an AtomicCmdDecorator around it:

```
Command* cmd;

// …

cmd = new AtomicCmdDecorator(cmd);
cmd->execute();     // command executes atomically
```

We can go a bit beyond the vanilla implementation by adding `setComponent` and `getComponent` operations to CommandDecorator, which merely set and get the decorator's component. Now a client who knows it has an AtomicCmdDecorator can opt to forgo transaction overhead whenever it wants:

```
AtomicCmdDecorator* acd;

if (acd = dynamic_cast<AtomicCmdDecorator*>(cmd)) {
    acd->getComponent()->execute();
}
```

‡ Comments hide the details of the transaction code. If you'd like a variety of transaction implementations, simply define a CommandDecorator subclass for each. Or introduce a template method in AtomicCmdDecorator itself and vary it in subclasses.

In sum, DECORATOR gives you remarkable flexibility even as it consolidates redundant code and keeps subclassing under control. There is, of course, some fine print:

1. *Increased overhead.* The decorator costs you space; it's one more object to instantiate. The decorator also costs you execution speed, since it introduces a level of indirection that might not have been there otherwise.
2. *Interface occlusion.* A decorator will hide any extensions to the basic Command interface that a ConcreteCommand subclass introduces. Clients that need to get at the extended interface must either downcast the decorator's component (requiring a `getComponent` operation on the decorator), or you'll need a specialized Decorator class for the extended interface. Because most of the DECORATOR pattern's benefits accrue from interface uniformity, specialized Decorator classes tend to complicate the pattern's implementation and erode its benefits.

COMMAND + COMPOSITE

There's one more COMMAND compound worth mentioning here, the one involving COMPOSITE. COMMAND alludes to it in several places.[3] Basically, a MacroCommand is a command that also plays the role of Composite in the COMPOSITE pattern, and as such it can assemble and execute a series of other commands (or "subcommands"). Here's a sketch of one possible implementation:

```
class MacroCommand : public Command {
public:
    MacroCommand(
        Command* = 0, Command* = 0, Command* = 0, Command* = 0
    );
    virtual ~MacroCommand();

    virtual void add(Command*);
    virtual void insert(Command*, int index);
    virtual void remove(Command*, int index);

    virtual Command* getCmd(int index);
    virtual int size();

    virtual void execute(Command*);

private:
    vector<Command*> _cmds;
};
```

You can instantiate `MacroCommand` with a sequence of up to four subcommands, the usefulness of which we'll see in a minute. Reference to subcommands is by integer index, 0 referring to the first subcommand. If you think of a MacroCommand object (or simply a "macro" for short) as a rudimentary program, the index corresponds to the program counter. `add`, `insert`, and `remove` let you "edit" the macro. `getCmd` returns the subcommand at the specified index, and `size` returns one more than the index of the last subcommand in the macro.

Which brings us to `execute`, whose job it is to execute the subcommands one after the other:

```
void MacroCommand::execute () {
    for (int i = 0; i < _cmds.size(); ++i) {
        Command* subcmd = getCmd(i);
        if (subcmd) subcmd.execute();
    }
}
```

Now let's see how we might use macros. Suppose it's common for people to add a new customer to the database along with an order for that customer. Instead of hard-wiring an AddCustomerAndOrderCmd to that effect, we merely compose two existing commands:

```
MacroCommand cmd;

cmd.add(new AddCustomerCmd("Jane Blane"));
cmd.add(new AddOrderCmd("ISBN 0201633612", 100));

cmd.execute();
```

When the number of commands in the macro is small, as is often the case in many applications, you can build up the macro entirely in its constructor:

```
MacroCommand cmd(
    new AddCustomerCmd("Jane Blane"),
    new AddOrderCmd("ISBN 0201633612", 100)
);

cmd.execute();
```

If you want to be failsafe about it, you'll want to use that nifty AtomicCmdDecorator here. What you don't want to do is wrap a decorator around each command, which is easy enough:

```
MacroCommand cmd(
    new AtomicCmdDecorator(AddCustomerCmd("Jane Blane")),
    new AtomicCmdDecorator(AddOrderCmd("ISBN 0201633612", 100))
);

cmd.execute();
```

What's so bad about this? Nothing—as long as you're comfortable paying for two transactions when one will suffice. Transaction overhead can become a problem for macros that contain more than a few subcommands, or even for small macros that get run a lot. We keep the overhead to a minimum by wrapping atomic decorators around the biggest units of work we can get away with:

```
AtomicCmdDecorator cmd(
    new MacroCommand(
        new AddCustomerCmd("Jane Blane"),
        new AddOrderCmd("ISBN 0201633612", 100)
    )
);

cmd.execute();
```

Adding a customer and an order now executes atomically with just one transaction. Composition does it again, courtesy the COMPOSITE and DECORATOR patterns.

The sky's the limit

The possibilities of the COMMAND-DECORATOR-COMPOSITE troika are endless. Because MacroCommand objects are themselves commands, you can compose macros of macros of macros, ad infinitum. Macros become analogous to subroutines. You can build up composite command structures of arbitrary size and complexity just as in any other application of the COMPOSITE pattern.

And you're not limited to one MacroCommand class. You can define Composite–ConcreteCommand classes that go far beyond simple subroutine semantics, supporting a variety of control structures like for, while, and do loops; if–then–else statements; even exception handlers and closures! Moreover, you can control the granularity of transactions by placing decorators at strategic points in the composite structure. The nearer to the root an atomic decorator is, the coarser the locking granularity and, incidentally, the lower the concurrency among database clients.

The composite structure lends itself to expressing *nested* transactions too, simply by wrapping atomic decorators around composite structures containing atomic decorators of their own.[§] Here too, you aren't limited to a lone AtomicCmdDecorator class. Imagine another kind of decorator that arranges for its component command to run in a separate thread. Or one that makes its component execute on a remote machine. Or in shared memory. Like we said—the possibilities are endless.

Indeed, it's quite conceivable that you take to programming entirely in commands, decorators, and composites. At which point you should say to yourself, "Self, have you gone bananas with *composition*?"

Acknowledgments

Many thanks to Brad Appleton, Jim Coplien, and Dirk Riehle for their faithful feedback.

References

[1] Gamma, E., R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[2] Ibid, p. 20.

[3] Ibid, pp. 234, 235, 237, 241, 242.

[§] We're assuming here that the database supports nested transactions and that `beginTransaction`/`endTransaction` calls nest.