DESIGN PATTERNS

The Command Pattern in Depth

Packaging commands as objects and sending them to a receiver enables a clean, loosely coupled design that's easy to maintain.

by Ian Darwin

Orders. Commands. All developers are familiar with them in real life: one person's *request* or *demand* that another person perform (or not perform) some action is *transmitted* to another person or persons. It works the same in software: one component's request is transmitted to another in the Command pattern. In this article, I explain how this pattern works and illustrate it with several examples. I also demonstrate how it can be introduced when adding new functionality and when cleaning up existing code.

A Familiar Example of the Command Pattern

The Command pattern is one of about two dozen patterns popularized in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides—known more concisely as the "Gang of Four" book or even just the "GoF." (Incidentally, a less academic, more memorable read is *Head-First Design Patterns* by Bates, Sierra, Freeman, and Robson. One other reference worthy of study is *Refactoring to Patterns* by Joshua Kerievsky.)

The Command pattern is not simply a method call (or "message" in the sense that Java's founders used that term). The request is *packaged* in some way, like putting a letter into an envelope and getting the (old school) post office or courier to *deliver* it. In software, the request can be packaged simply as executable code to be performed, it can be a string in some "little language" devised for that purpose, or it can be anything that gets the message across.

Perhaps the most familiar example to Java developers is the `ActionListener` interface used in Swing or the JavaServer Faces action handler bound to a submit button. Some code, which is often loosely called the *handler*, is packaged up and associated with the `JButton` or other control, to be acted upon when the user chooses to click the button.

In this pattern (**Figure 1**), the button is called the *invoker*. The `ActionListener` implementation is the Command pattern; it consists of the command or code that the application has sent to the button. The object to act upon is called the *receiver*, because it receives the action. The receiver may be passed as a constructor argument to the Command, or it may be implicit in the case of a smaller application using a field in the main class as the receiver.

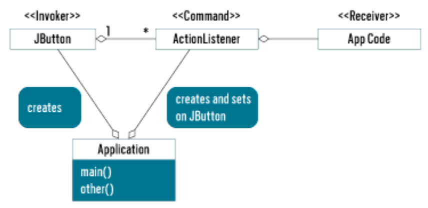

Figure 1. Key players in the Command pattern, illustrated with ActionListener

A Remote Sending Example

As another example, if you want to package some arbitrary code for execution in a different VM—perhaps on a rebel spaceship far, far away—you could package it into an instance of `Runnable`. The `Runnable` interface was designed for use in threading, but it's a perfectly fine interface to use as a Command interface: it has one method, no arguments, and a `void` return type. As my colleague Chris Mawata says, "Use standard interfaces where they serve."

To run a "hello, world" command on another VM, you could package it this way:

```
Runnable command = new Runnable() {
    public void run() {
        System.out.println("Hello, world.");
    }
};
```

Nowadays, I'd probably write that as a lambda, like this:

```
Runnable command = () -> System.out.println("Hello,
```

Then, assuming all the network plumbing has been set up, there might be a method such as `submit()` to send a command to the server:

```
remoteConnection.submit(myCommand);
```

To make the code clearer, you could define `Command` as an interface that extends `Runnable`:

```
public interface Command extends Runnable, Serializ
    // empty
}
```

`Serializable` is needed for some of the networking transports that might be used, such as remote method invocation (RMI), and it costs nothing anyway. I'd simplify the code by instantiating the lambda inline, as in the following:

```
remoteConnection.submit( () -> System.out.println("
```

The code on the other end—the "server"—could implement this method in a simple fashion:

```
public void submit(Command c) {
    c.run();
}
```

Or, the server could put the command into a batch queue, run it in a thread pool (see `java.util.concurrent.Executor`), or use any of several options. Either way, on the client side, you don't know and shouldn't care. Of course, the `Command` interface could be changed to have arguments and a return type that is not `void`, and it could be an abstract class instead of an interface. I used `Runnable` as a base just to get started.

The code in this `Command` object could perform arbitrary (and possibly malicious) actions on the server, so the server should provide a `SecurityManager` and a policy file to control the imported code.

One use of the Command pattern is packaging Java code to tell a remote process what to do.

All the code for this remote sending example is available in my GitHub site under remotecommand. There are two subdirectories, server and client, with Maven and Eclipse files and a README file showing how to build and run each. If you're not up to speed on RMI, you might want to read my Java RMI tutorial.

An Auction House Example

The "Gang of Four" book describes the `Command` object as holding a reference to the receiver; that is, the object on which the work will be done. In a word processor, the receiver might be a `Document` object. In an online auction house, it might be a `Listing` or `Auction` object. My demo implementation of the auction house scenario, called bidpay, is in my patterns-demos GitHub repository. My scenario is so simplified from real life that you can bet it will forever be outbid on eBay, but it's developed enough to show some interesting aspects of the Command pattern.

In that implementation, `Command` is a top-level interface, and two implementing classes with a `Receiver` field (the `Auction` object) are given: `BidCommand` and `CancelCommand`. Here is the former:

```
public class BidCommand implements Command {
    Auction receiver;
    double amount;
    Client bidder;
    public void execute() {
        receiver.bid(amount, bidder);
    }
    // Obvious three-argument constructor not shown
}
```

The intent is that the main program, `BidPaySite`, doesn't need to know or care what the clients are sending it. As long as the objects implement `Command`, it will be happy, and the `Auction` class will receive what's sent and process it.

```
public class BidPaySite {
    public void submitCommand(Command command) {
        // These could go into a queue to serialize
        // or you could make sure that all methods 
        // to the Command are thread-safe.
        // For now, just let the command do its thi
        command.execute();
    }
    ...
}
```

There are times when you want multiple commands to execute as a single command (for example, something like database transactions, or batching, or reducing network traffic on a remote connection). You could create a `CompositeCommand`, which is created with an array or List of commands. The execute method of a List implementation could be something like this:

```
class CompositeCommand implements Command {
    List<Command> commands;
    public void execute() {
        commands.forEach(Command::execute);
    }
    // Obvious one-argument constructor omitted
}
```

An Undo Stack Example

I've shown that one use of the Command pattern is packaging Java code to tell a remote process what to do. A different use would be the undo stack in an editor or word processor. When you request an operation such as "insert," "move," or "delete," the editor program could create a Command object representing the operation to be performed. This object would then be passed to a "perform" method in the editor and, upon successful completion, it would be added to the undo stack.

The stack could be implemented as a simple push-down stack of objects of the `EditorCommand` type. When you request an undo operation, the top element is popped off the stack, and it is passed to an "unperform" method in the editor, which removes the inserted text if the operation was an insertion, reinserts the deleted text if the operation was a deletion, and so on. In a full implementation, you wouldn't actually pop the undoable action and drop it after use; you would keep it there for use by a redo command.

In adding base undo functionality into a simple line-editor called edj (also on my GitHub site), I took a slightly simpler approach. To provide a degree of separation between the main code and the "model" (here, the in-memory buffer-handling code), I built the editor from the start with an interface called `BufferPrims` between the main code and the operations on the buffer. These are primitive operations such as "add lines," "delete lines," and so on.

There are two versions of the code: `BufferPrimsNoUndo` and `BufferPrimsWithUndo`. In real life, you probably don't need these, so you might not even need the interface, but having them both makes it easier to compare them to see all the changes. In the first version of the code, there was no undo operation. So the first step was refactoring to include the undo capability in the interface, and then have the no-undo implementation, shown next, just print a message:

```
public interface BufferPrims {
    void addLines(int start, List<String> newLines)
    void deleteLines(int start, int end);
    /** Print one or more lines */
    void printLines(int i, int j);
    /** Undo the most recent operation */
    void undo();
}
public class BufferPrimsNoUndo extends AbstractBuffe
    public void undo() {
        System.err.println("?Undo not written yet")
    }
    ...
}
```

Then, instead of writing code to decipher and reverse each command, I have the "with undo" version of each low-level modify operation create and push an `UndoableCommand` object that contains the exact code to undo the operation. For diagnostic purposes, I associate a String with each `Undoable`, so the `Undoable` looks like this:

```
class UndoableCommand {
    public UndoableCommand(String name, Runnable r)
        this.name = name;
        this.r = r;
    }
    String name;
    protected Runnable r;
}
```

The two constructor arguments provide all the information you could want, because the undo actions can be simple (the undo of inserting a number of lines is just to delete the inserted range of lines) or complex (the undo of deleting some lines must include all the text of the deleted lines). For example, here is a slightly simplified look at `addLines()`:

```
public void addLines(int startLnum, List<String> nev

    buffer.addAll(startLnum, newLines);
    current += newLines.size();
    pushUndo("add " + newLines.size() + " lines",
        () -> deleteLines(startLnum, startLnum + nev
}
```

The `pushUndo()` method is simply a convenience routine that creates the `UndoableCommand` and pushes it on the stack:

```
private void pushUndo(String name, Runnable r) {
    undoables.push(new UndoableCommand(name, r));
}
```

Now the undo implementation becomes trivial (error handling is omitted):

```
public void undo() {
    UndoableCommand undoable = undoables.pop();
    undoable.r.run();
}
```

Here is an example of the edj editor in action:

1. I run edj, telling it to start with the sample three-line file included with the source code.
2. The `,p` command prints all the lines in memory; it's short for *1,Np* where *N* is the number of lines in the buffer.
3. The `2d` command deletes the second line.
4. I print the whole thing again to show that the deletion worked.
5. I invoke the newly added undo feature using the `u` command.
6. I print the buffer again to show that line 2 was miraculously restored by the `u` command.
7. I use the command `q` to quit.

```
$ edj 3lines.txt       // 1
3L, 26C
,p                     // 2
Line One
Line Two
Line Three
2d                     // 3
,p                     // 4
Line One
Line Three
u                      // 5
,p                     // 6
Line One
Line Two
Line Three
q                      // 7
$
```

At this point, the undo operation in edj worked nicely. I had refactored the bottom layer made of buffer primitives. But when I went to hook this code into the main line code of the editor, I was reminded that that code is large and hoary. The main loop was something like this:

```
while ((line = in.readLine())  != null) {
    if (line.startsWith("e")) {
        // code to edit a new file
    } else if (line.startsWith("f") {
        // code to print or set filename
        } ...
    }
    // many more if/else statements, one per command
}
```

The book *Refactoring to Patterns* calls such code a *conditional dispatcher*, because it uses a conditional statement (a long chain of if statements, but a `switch` is also common). There's nothing inherently wrong with writing code this way, but it can lead to really long methods that are hard to read. You could extract each bit of code into a named method, but that leads to a lot of method names. Ideally, for a couple of reasons, conditional dispatcher code is refactored to use the Command pattern. One reason is if the code requires more flexibility. Another, as the book says, is the following:

"Some conditional dispatchers become enormous and unwieldy as they evolve to handle new requests or as their handler logic becomes ever more complex with new responsibilities."

That is exactly a description of the line editor's main loop: as more commands are implemented, the size of the code in the `if-else` chain or `switch` statement will grow larger without bound.

So I replaced the main loop with a table of Command implementations: an array, indexed by the first letter of each command, is nice and simple. This approach also forced me to provide standardized parsing of the input lines, which up to now was done on demand in the various sections. I introduced the `ParsedLine` class to hold the information about the input line and, in fact, it is a form of Command object, because it describes what to do (but not how, and the receiver is still implicitly `this`
).

```
public class ParsedLine {
    char cmdLetter; // 'a' for append, 'd' for dele
    boolean startFound, commaFound, endFound;
    int startNum, endNum;
    String operands;  // The rest of the line
    public String toString() {
        return String.format("%d,%d%c%s", startNum,
            operands == null ? "" : (' ' + operands
    }
}
```

The `toString()` method is used in this version for debugging, but in a GUI editor, it would appear in the Undo menu item.

The executable Command objects—the actual code—are defined by the interface `EditCommand`:

```
public interface EditCommand {
    void execute(ParsedLine pl);
}
```

With that structure, I was able to trim the main loop to look like this (error checking omitted):

```
while ((line = in.readLine())  != null) {
    ParsedLine pl = LineParser.parse(line, buffHandl
    EditCommand c = commands[pl.cmdLetter];
```

```
        c.execute(pl);
    }
```

That is, I parse the line into a `ParsedLine` structure, use the command code from that to find the executable `EditCommand` object, and invoke that. The array of `EditCommand` objects named `commands` is initialized in a static block using assignments like this:

```
    // d - delete lines
    commands['d'] = pl -> {
        buffHandler.deleteLines(pl.startNum, pl.endNum)
    };
```

In other words, each `EditCommand` is constructed as a lambda, passing the `ParsedLine` as a parameter to the `execute` method. As before, the receiver is implicitly the buffer handler.

I've described two uses of Command in my line editor. But most people don't use line editors anymore; they use screen-based editors. And the Swing UI framework already has support for undo operations. I have a simple notepad-style editor called TinyPad that uses this feature. There isn't room to dissect it here, but if you want to look at its code, check out this GitHub repository. In the "before" version, a `Document Listener` was attached to the main (and only) document, so that when the `TextArea` made any changes to the model, I'd be notified, and an `unsavedChanges` boolean would be set to prompt for unsaved changes when exiting.

In the "after" version, I use Swing's `UndoableEditListener` and `UndoManager`. To see how all those pieces fit together, look at the code starting at `// Set up Undo/Redo actions` and the Command objects `UndoAction` and `RedoAction`.

The GoF book says this: "A command can have a wide range of abilities. At one extreme, it merely defines a binding between a receiver and the actions that carry out the request. At the other extreme, it implements everything itself without delegating to a receiver at all…[in between] are commands that have enough knowledge to find their receiver dynamically."

In bidpay, the command has an explicit receiver and is little more than that binding. In edj, there's only one source file, so the document is available to all code and does not need to be passed with the command. In TinyPad, the command—when coupled with the undo manager—is smart enough to know its associated document internally.

Conclusion

The Command pattern isn't just for undo stacks, of course. It's good for remote execution (as you saw in my first example) and for journaling in database-like systems and file systems to be re-executed after a crash. A composite version can be used to implement database-style transactions and batch processing.

The Command pattern is a good example of a general-purpose design pattern that has many uses and, when applied properly, it will clarify your code and make it more readable and maintainable. And that's largely what this patterns business is all about.

This article was originally published in the May/June 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.